
Filière électronique - Projet Thématique
Semestre 8

# Détection et lecture de codes QR

---

# RAPPORT DE PROJET

---

BEN-MOUMEN Tahar
RODRIGUEZ-MAZOYER Roland
2018 - 2019

*Encadrant projet :* DONIAS M.

# Sommaire

Les QR codes sont des visuels carrés, contenant des informations dans les deux dimensions du plan, par opposition aux codes barres par exemple, ne contenant des informations que dans une seule dimension. La technologie du QR code a tout d'abord été développé pour des applications industrielles ; ces types de codes, pouvant contenir une grande quantité d'informations, se sont ensuite massivement développés pour une utilisation pour le public, notamment grâce à des techniques de lecture d'image implémentées dans les ordiphones possédant un appareil photo.


FIGURE 1 – Exemple de QR code dans l'espace public - exemple d'image testée

La détection d'information des QR codes se découpe en deux phases : une première phase de traitement d'image, donnant lieu à un QR code binaire sous forme de matrice ; et une seconde phase de lecture des informations depuis la matrice.

La spécification de lecture du QR code matriciel est entièrement définie dans la norme internationale spécifique, trouvable à cette adresse :

https://www.swisseduc.ch/informatik/theoretische_informatik/qr_codes/docs/qr_standard.pdf [3]

Nous avons utilisé la première séance pour découvrir globalement le fonctionnement d'un QR code, notamment grâce à la spécification standard internationale donnée ci-dessus. Nous nous sommes ensuite répartis les tâches en s'occupant chacun d'une partie par implémentation sous Matlab.

La première partie de ce rapport se concentrera donc sur le traitement d'image permettant d'extraire la matrice des pixels binaires, et la seconde se concentrera sur le traitement de cette matrice pour en extraire les informations contenues.

# Première partie

# Détection d'un code QR

---

## 1 Déroulé des séances et avancements

Nous nous sommes mis d'accord sur le fait que cette partie ne retournerait qu'une matrice contenant les pixels binaires du QR code ; cette partie de manipulation d'images ne s'occupe donc pas de la partie "Traitement des données binaires", traitée dans la Seconde partie. La partie Traitement d'Image peut ensuite être scindée en trois sous-traitements généraux :

- La détection de quatre points remarquables du QR code, voulus invariants selon la taille du QR code ;
- Le redressement de l'image contenant uniquement le QR code, pour la disposer dans une image carrée ;
- La binarisation, détection de la taille et placement de chaque bit dans une matrice de la taille du QR code initial.

Chaque traitement se voudra le plus automatique possible.

La première partie implémentée s'est concentrée autour du redressement d'images, suivie de la binarisation. La détection automatiques des 4 points remarquables, plus longue et complexe, a été implémentée en dernier.

## 2 Théorie

Avant de pouvoir implémenter les différentes parties *via* des fonctions, il a fallu se servir des bases mathématiques, pour ensuite les traduire dans Matlab. Nous allons donc décrire dans les pages suivantes les éléments théoriques utilisés

## 2.1 Détection des ancres

### 2.1.1 Algorithme non-fonctionnel : détection de rotationnels

#### Description de l'algorithme

L'algorithme développé dans cette partie part d'une réflexion d'analyse vectorielle. Cette approche se base sur le fait que les ancres peuvent être l'interprétation carrée/pixelisée d'un disque avec des alternances concentriques noires et blanches. De fait, il est normal de penser que la détection d'un rotationnel permettrait de repérer un tel motif.

Puisque nous travaillons en 2D, mais que les intensités des vecteurs ne peuvent réellement s'exprimer que sur le coefficient adressé à chaque pixel, il est nécessaire de ne considérer que la composante du rotationnel normale au plan de l'image, d'où la formule :

$$\boxed{rot(\vec{t}) = div(\vec{n}) = \frac{\partial n_x}{\partial x} + \frac{\partial n_y}{\partial y}} \tag{2.1}$$

Pour calculer $|\vec{n}|$, c'est-à-dire $\begin{pmatrix} n_x \\ n_y \end{pmatrix}$, plusieurs étapes sont nécessaires, et demandent d'utiliser la notion de tenseurs.

Nous devons donc tout d'abord calculer les coefficients du tenseur. Pour cela, nous calculons ses trois composantes :

$$\begin{aligned} T_{xx} &= G_{\sigma,lf} * \left(\frac{\partial I}{\partial x}^2\right) \\ T_{xy} &= G_{\sigma,lf} * \left(\frac{\partial I}{\partial x}\frac{\partial I}{\partial y}\right) \\ T_{xx} &= G_{\sigma,lf} * \left(\frac{\partial I}{\partial y}^2\right) \end{aligned}$$

En réalité, nous pouvons scinder cette étape en deux sous-étapes : la première consiste à calculer le gradient en x et en y de l'image, puis de multiplier entre eux les résultats obtenus (cf. figure 2.1) :

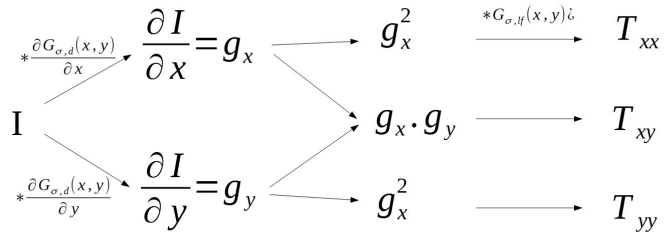



Figure 2.1 – Principe de calcul du tenseur

Remarquons que, puisque l'image est échantillonnée en niveaux de gris, il est indispensable de ne pas appliquer une dérivée droite, mais plutôt une dérivée gaussienne, beaucoup moins bruitée. Au point (i,j), le calcul de la dérivée s'effectue donc en convoluant l'image avec le filtre figure 2.2.

Nous obtenons donc un tenseur pour chaque point de l'image. Ce tenseur est de la forme :

$$\boxed{T = \begin{pmatrix} T_{xx} & T_{xy} \\ T_{xy} & T_{yy} \end{pmatrix}} \tag{2.2}$$

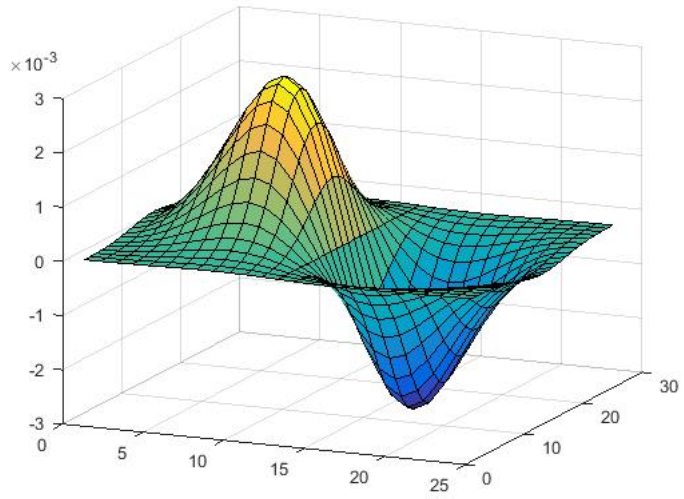



FIGURE 2.2 – Filtre utilisé pour calculer les dérivées en un point

On sait ensuite qu'on peut trouver $|\vec{n}|$ grâce aux coefficients de V trouvables grâce aux formules suivantes :

$$T = V^T.D.V$$
$$\Leftrightarrow \begin{pmatrix} T_{xx} & T_{xy} \\ T_{xy} & T_{yy} \end{pmatrix} = \begin{pmatrix} V_x \\ V_y \end{pmatrix} \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} \begin{pmatrix} V_x & V_y \end{pmatrix}$$

$$\Leftarrow \boxed{\begin{cases} V_x = & -T_{xy} \\ V_y = & T_{xx} - \lambda_1 \\ \lambda_1 = & \lambda_2 \\ \lambda_1 = & \frac{T_{xx}+T_{yy}+\sqrt{(T_{xx}+T_{yy})^2-4(T_{xx}T_{yy}-T_{xy}^2)}}{2} \end{cases}} \tag{2.3}$$

On récupère ensuite $|\vec{n}|$ *via* la formule suivante :

$$\boxed{\vec{n} = \begin{pmatrix} \frac{V_x}{\sqrt{V_x^2+V_y^2}} \\ \frac{V_y}{\sqrt{V_x^2+V_y^2}} \end{pmatrix}} \tag{2.4}$$

Il suffit ensuite de dériver l'image selon x et y, pour trouver la figure voulue.

En combinant ces équations encadrées, dans Matlab et pour chaque point de l'image d'origine, nous obtenons donc une image résultante, comme celle présente dans la figure 2.3

### Problèmes et limitations

Le problème de cette approche est qu'elle détecte les courbures ; ainsi, les angles droits des ancres sont détectés, mais pas l'ancre dans son ensemble, la figure de détection des ancres ressemble donc à celle présente figure 2.3. Cependant ce ne sont pas les seuls éléments à être détectés : en effet, le contenu d'un QR code étant une somme de pixels noirs et blancs avec une disposition particulière, un certain nombre d'angles droits sont présents au sein de ce QR code. On peut donc remarquer que la détection est très inefficace.

Pour pouvoir détecter ces ancres, il a donc fallu utiliser une autre méthode, telle que nous allons voir dans les paragraphes suivants.

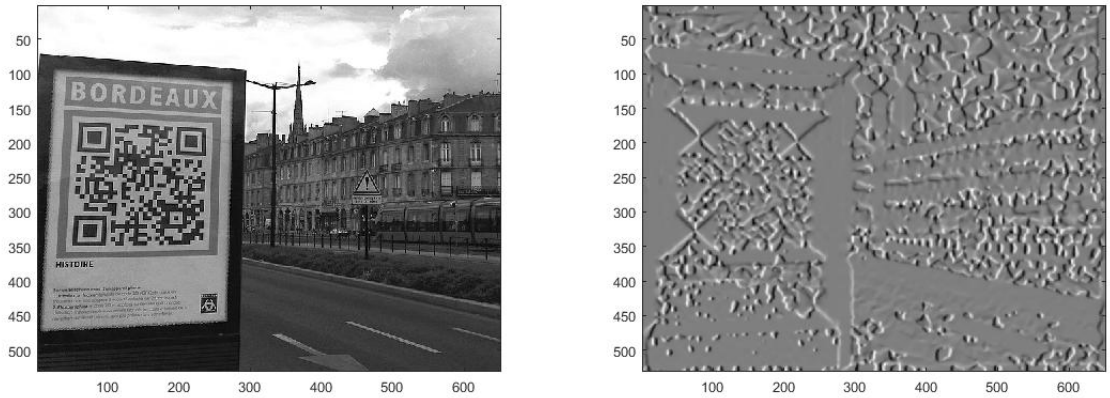



Figure 2.3 – Image résultante de l'algorithme développé

### 2.1.2 Algorithme non-fonctionnel par détection des bords et coloration additionnelle

Face au constat que l'algorithme précédent ne fonctionnait pas comme prévu, une autre approche a été temporairement mise en place.

#### Description de l'algorithme

Cet algorithme se base sur le fait que les ancres peuvent être détectées par 6 contours en les parcourant horizontalement et veritcalement ; ces contours sont de plus disposés selon un motif très précis, facilement repérables.

La première étape consiste à détecter ces contours. Pour cela, deux approches ont été testées : une approche en appliquant une dérivée à l'image, et une approche en utilisant les contours de Canny. Une fois cette étape effectuée, nous pouvons détecter les motifs des contours. Ces motifs sont arrangés de cette manière : 3 contours espacés régulièrement (un espace étant de x pixels), trois espaces de x pixels sans contours, puis de nouveau trois contours. Ils faut quand-même évidemment prendre en compte quelques pixels d'erreur dans la détection des longueurs, entre chaque contour.

Une fois que le motif a été détecté, l'algorithme colore la ligne (i.e : ajoute une quantité de 1 à chaque pixel de la ligne), entre les deux extrémités du motif détecté. Il effectue cette action sur toute la ligne, pour toute les lignes, puis sur toute la colonne pour toutes les colonnes.

Un filtre passe-bas est ensuite appliqué, pour pouvoir mettre en valeur les zones larges et de valeur forte. Une binarisation sera alors effectée, pour mettre en valeur les quelques pics principaux.

Un exemple d'application de cet algorithme se trouve en figure 2.4.

#### Problèmes et limitations

Cette approche n'est pas très rigoureuse : dans une image, un certain nombre d'éléments extérieurs au QR code (contour du QR code, bâtiments ou objets extérieurs au QR code présents dans l'image, motifs répétitifs simples...) peuvent être pris en compte par l'algorithme, ce qui fausse l'estimation. Ensuite, après passe-bas, puisqu'un certain nombre d'éléments ont été détectés, il est difficile de fixer un seuil de détection qui sélectionnerait uniquement les 3 pics les plus hauts (qui, de plus, ne correspondent pas forcément à une ancre). Enfin, l'approche par l'algorithme de Canny donne de meilleurs résultats, mais est très dépendante de la décision, automatique, que prend l'algorithme, puisqu'il effectue en interne une binarisation dépendante d'un seuil. Remarquons enfin que les contours ne sont pas parfaitement dessinés pour les ancres, ce qui ne permettra pas à l'algorithme de détecter le motif. Il est donc très probable que cet algorithme, grâce à l'approche de Canny, crée des erreurs de manière très fréquente, et donc ne permette pas de passer à l'étape suivante.

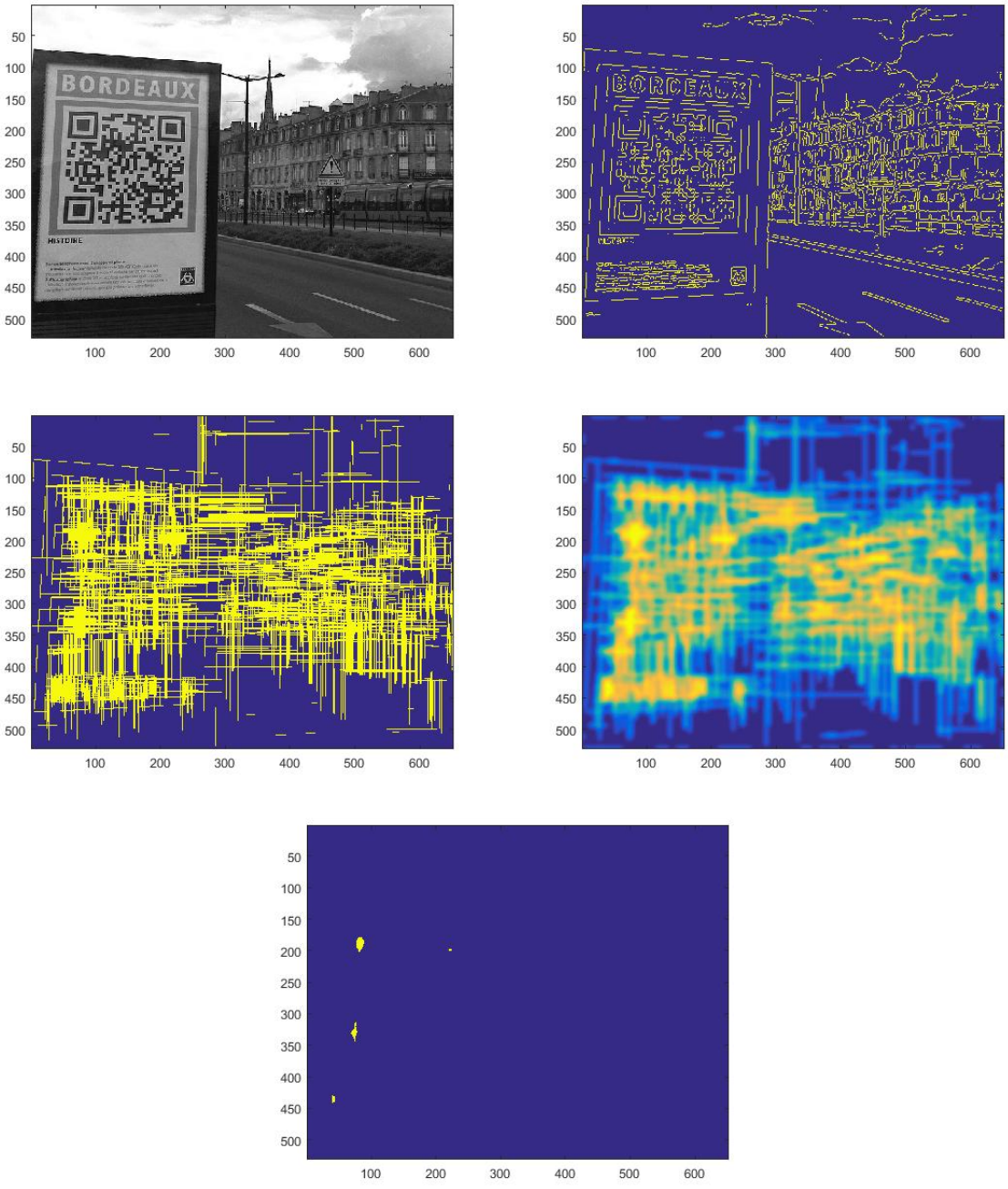



FIGURE 2.4 – Image résultante de l'algorithme développé, en utilisant ici les contours de Canny et un seuil à 99% du maximum de l'image filtrée

Cette approche a donc purement et simplement été abandonnée.

### 2.1.3 Algorithme fonctionnel : détection par convolution

#### Hypothèse

Nous supposerons que la projection subie par l'image du QR code n'affecte que très peu la géométrie globale des ancres, de sorte que ces ancres ressemblent très fortement visuellement à des ancres non projetées.

### Description de l'algorithme

Ici, l'approche se base sur la convolution d'un motif 1D prédéfini (correspondant au motif des ancres) avec l'image. En repixelisant successivement l'image (de manière à l'agrandir ou la réduire), ce motif va plus ou moins bien se superposer avec des lignes/colonnes de l'image, c'est-à-dire là où les ancres se trouvent. Remarquons que ce motif 1D existe quel que soit la rotation et la projection subie par le QR code, dans les directions x et y de l'image.

En effet, grâce à la convolution suivant x et y, nous allons voir des pics d'intensité (correspondant au centre de l'ancre) se matérialiser. En détectant successivement ces pics d'intensité, nous pouvons alors récupérer les centres des 3 ancres, comme visible sur la figure 2.5.

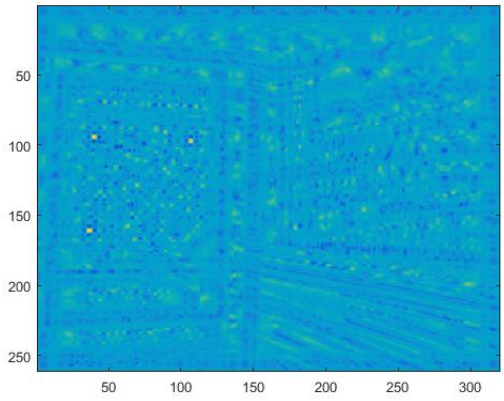



FIGURE 2.5 – Pics apparents lors de la convolution de l'image avec le motif, selon x et y

Remarquons que, une fois un pic détecté, et puisque la taille des ancres est souvent différente suivant sa place dans l'image (à cause des déformations), il est nécessaire de détecter séparément les 3 ancres. Pour cela, une fois une ancre détectée, toute la région de l'ancre détectée est mise à 0, avant de répéter l'opération pour détecter le pic suivant (cf. figure 2.6).

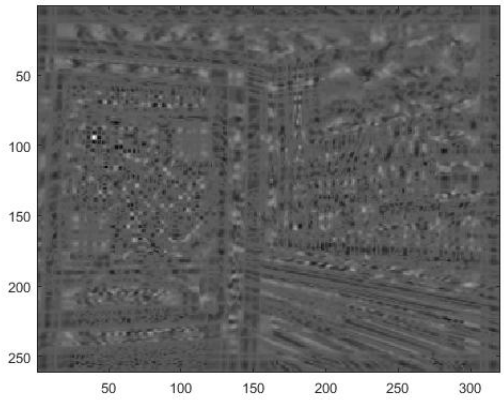



FIGURE 2.6 – Extinction des deux premières ancres de l'image détectées

### Limitations

Cet algorithme devient non fonctionnel dès lors qu'il y a de grandes différences d'exposition au sein de l'image, ou qu'il y a beaucoup de bruit. Ainsi, les photos prises avec un portable très haute résolution mais générant du bruit, sont susceptibles de mettre vraiment en défaut l'algorithme, qui placera alors les ancres quelque part aléatoirement dans l'image.

### 2.1.4 Algorithme non fonctionnel : Placement des points dans les angles du QR code grâce à la détection de contour

Une fois le centre des 3 ancres détectées, nous pouvons déterminer un quatrième point, correspondant au coin n'ayant pas d'ancre, par un autre moyen. Nous verrons dans la partie "Déroulé de l'algorithme final" comment ce problème est temporairement résolu. Considérons donc que le quatrième point est correctement placé dans le quatrième angle du QR code.

Afin de correctement placer les points dans les angles du QR code depuis le centre des ancres, une approche vectorielle a été utilisée, combinée avec l'utilisation des contours.

Il est tout d'abord nécessaire de trier les points de manière à ne pas faire un quadrilatère croisé. Pour cela, un algorithme faisant intervenir les angles entre les vecteurs définis entre 3 points et un point de référence a été utilisé, et il en ressort la liste des points triés dans le sens horaire.

Ensuite, puisque le quatrième point est correctement placé, nous avons seulement besoin de placer correctement les 3 autres points correspondants aux ancres. De plus, ces points étant initialement placés au centre des ancres, il est très facile de les déplacer dans le coin du QR code (qui est également le coin de l'ancre) en suivant les diagonales du QR code.

Ces diagonales peuvent être définies grâce à des vecteurs, en utilisant les points opposés deux à deux. Il suffit alors de suivre ces vecteurs, puis de placer le point réel sur le troisième pic d'intensité du contour. Sur la figure 2.7, nous pouvons voir un exemple graphique de cet algorithme : suite à la détection des quatre points (en bleu) et au tri de ceux-ci, nous utilisons le vecteur calculé (en rouge), et suivons sa direction pour détecter des pics d'intensité de contours, pour finalement placer le point sur le troisième pic (à l'extrémité des tirets). On renouvelle ensuite l'opération pour les deux autres points (diagonale avec tirets ; les point finaux se trouveront à l'extrémité de la ligne).

Figure 2.7 – Figure explicative de l'utilisation des diagonales et des contours

Cet algorithme a l'avantage de ne pas s'appuyer sur l'information de taille du QR code. Même si cette information est évidemment présente à l'intérieur du QR code, il est *a priori* plus difficile de la détecter à cette étape. De plus, puisqu'à ce stade du développement, un algorithme prenant en compte les quatre coins du QR code avait déjà été développée, il était beaucoup plus facile de l'utiliser plutôt que le redéfinir. Une autre approche est cependant expliquée dans la partie 4, "Axes d'améliorations".

#### Problèmes et limitations

Le problème majeur de cette approche est qu'elle n'est pas robuste. En effet, les QR codes pris en photo ne sont pas parfaits, et une dégradations de contour est très probable, notamment si une partie du QR code est volontairement abîmée ; il est donc relativement délicat d'utiliser cette information.

### 2.1.5 Algorithme non robuste : Placement des points dans les angles du QR code grâce à la convolution

#### Hypothèses

Nous utilisons la propriété de projection suivante : tout ensemble de points (alignés) formant une droite ou un segment de droite dans l'image non déformée, forme aussi une droite dans la même image projetée (et réciprocité). Ainsi, la déformation du QR code n'affecte pas les deux diagonales du QR code, correspondant aussi à une des diagonale des ancres (i.e. les ancres opposées ont leur coins alignés, cf. figure 2.7).

Même si l'image est bruitée ou comporte des zone dégradées, les bords des ancres dans les angles du QR code sont suffisamment visibles et très peu dégradés.

#### Description de l'algorithme

De même que précédemment, nous allons dans cet algorithme exploiter les diagonales définies par deux points opposés telles que définies dans la partie précédente ; cependant, nous n'allons pas utiliser les informations de contour, mais bien celles de l'image en elle-même, en réutilisant le principe de l'algorithme par convolution développé pour trouver le centre des ancres.

Ainsi, en extrayant des points sur le vecteur défini (ces points étant de plus en plus espacés, i.e. en effectuant un échantillonnage strict), autour du centre de l'ancre considérée, puis en convoluant cette ligne de points avec un motif prédéfini inhérent à l'ancre, nous obtenons une taille qui correspond à la distance entre le centre de l'ancre et son coin. Nous pouvons alors utiliser le vecteur pour placer correctement le point final dans le coin du QR code, et recommencer l'opération pour les deux autres coins.

#### Problèmes et limitations

Cet algorithme serait fonctionnel si le point était placé exactement au centre de l'ancre, puis que la recherche de distance par échantillonnage de la diagonale n'induisait pas d'erreurs d'arrondis des différents points extraits. Ces deux facteurs conduisent à une mauvaise estimation du maximum dans la convolution, donc à une mauvaise taille de vecteur, donc à un placement du point final incorrect également.

## 2.2 Redressement de l'image

#### Hypothèses

Le redressement de l'image s'appuie sur plusieurs hypothèses de travail :

- Les quatre points définissant les angles du QR code existent et ne forment pas un quadrilatère se résumant à un point ou un segment ;
- Le quadrilatère ABCD n'est pas un quadrilatère croisé, et donc les points A, B, C, D sont triés (sens horaire) ;
- L'image redressée est une image carrée de côté $M \in \mathbb{N}^*$, dont les angles correspondent aux quatre points cités précédemment ;
- L'algorithme doit pouvoir être exécuté dans un temps raisonnable

Nous nommerons les deux espaces utilisés : l'image d'entrée initiale se trouve dans l'espace $\mathcal{E}$, et l'image redressée de sortie se trouve dans l'espace $\mathcal{S}$.

Toutes les solutions développées sont explicitées ci-après.

### 2.2.1 Algorithme non-fonctionnel naïf : Redressement linéaire

Le premier algorithme développé repose sur une réflexion linéaire. Le principe est de calculer la position de chaque pixel de l'image d'arrivée ($\mathcal{S}$) dans les coordonnées de l'image de départ ($\mathcal{E}$), grâce aux formules ci-dessous :

$$\begin{cases} x_{\mathcal{S}\to\mathcal{E}}(j,i) &= \left(\frac{(x_C-x_D)(i-1)}{M-1}+x_D\right)\frac{j-1}{M-1}+\left(\frac{(x_B-x_A)(i-1)}{M-1}+x_A\right)\frac{M-j}{M-1} \\ y_{\mathcal{S}\to\mathcal{E}}(j,i) &= \left(\frac{(y_C-y_B)(j-1)}{M-1}+y_B\right)\frac{i-1}{M-1}+\left(\frac{(y_D-y_A)(j-1)}{M-1}+y_A\right)\frac{M-i}{M-1} \end{cases} \quad \forall j \in y_{\mathcal{S}}, \forall i \in x_{\mathcal{S}}$$

Une fois les pixels d'arrivée mappés dans les coordonnées de départ, on applique la transformée bilinéaire (source : [1]) telle que définie par la formule 2.5 ci-dessous :

$$\begin{aligned} y &= \lfloor y_{\mathcal{S}\to\mathcal{E}}(j,i) \rfloor \\ x &= \lfloor x_{\mathcal{S}\to\mathcal{E}}(j,i) \rfloor \\ b &= y_{\mathcal{S}\to\mathcal{E}}(j,i)) - \lfloor y_{\mathcal{S}\to\mathcal{E}}(j,i) \rfloor \\ a &= x_{\mathcal{S}\to\mathcal{E}}(j,i) - \lfloor y_{\mathcal{S}\to\mathcal{E}}(j,i) \rfloor \end{aligned}$$

$$\boxed{S(i,j) = b(1-a)E(y+1,x) + baE(y+1,x+1) + (1-b)(1-a)E(y,x) + a(1-b)E(y,x+1)} \tag{2.5}$$

$$\begin{aligned} E(i,j) &\in \mathcal{E} \\ S(i,j) &\in \mathcal{S} \\ j &\in y_{\mathcal{S}} \\ i &\in x_{\mathcal{S}} \end{aligned}$$

#### Problèmes et limitations

Cet algorithme donne de bonnes performances si l'image est simplement déformée par des transformations linéaires (cisaillement, rotation, etc... dans le plan), cependant il n'est pas du tout efficace et donne de grosses erreurs sur le mappage des pixels de $\mathcal{S}$ dans $\mathcal{E}$. La figure 2.8 donne un exemple de QR code à redresser et son redressement correct (cf. algorithme fonctionnel partie suivante), et la figure 2.9 dévoile ce que l'on obtient avec l'algorithme développé.

On constate donc que l'algorithme développé étire le QR code, ce qui est un frein pour déchiffrer correctement son contenu dans l'algorithme de l'étape suivante. De plus, ces déformations ne sont pas contrôlées, il est donc préférable de développer un algorithme plus robuste.

Il faut donc écrire un algorithme qui prend en compte les déformations dans l'espace 3D. Pour cela, nous pouvons faire appel à une matrice d'homographie, et appliquer la méthode homographique qui lui est liée.

### 2.2.2 Algorithme fonctionnel : Homographie

Pour redresser correctement le QR code, nous allons utiliser le principe des matrices d'homographie, dont les coefficients seront estimés automatiquement. Le principe de fonctionnement desdites matrices est résumé dans la source [2] en bibliographie, reprenant les principes de projections 3D sur un espace 2D.

FIGURE 2.8 – Exemple de QR code et son redressement correct

FIGURE 2.9 – QR code redressé avec l'algorithme non-fonctionnel

**Principe de fonctionnement**

La matrice d'homographie est telle que

$$\boxed{\begin{array}{lccc} \lambda & X_{\mathcal{S}\to\mathcal{E}} & = & H & X_{\mathcal{S}} \\ \\ \Leftrightarrow \lambda & \begin{pmatrix} x_{\mathcal{S}\to\mathcal{E}} \\ y_{\mathcal{S}\to\mathcal{E}} \\ 1 \end{pmatrix} & = & \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} & \begin{pmatrix} x_{\mathcal{S}} \\ y_{\mathcal{S}} \\ 1 \end{pmatrix} \\ \Leftrightarrow & \begin{pmatrix} x_{\mathcal{S}\to\mathcal{E}} \\ y_{\mathcal{S}\to\mathcal{E}} \\ 1 \end{pmatrix} & = & \frac{1}{\lambda} \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} & \begin{pmatrix} x_{\mathcal{S}} \\ y_{\mathcal{S}} \\ 1 \end{pmatrix} \end{array}} \tag{2.6}$$

Grâce à l'équation 2.6, il est donc possible de calculer n'importe quel point $(x_{\mathcal{S}}, y_{\mathcal{S}})$ dans l'image d'entrée, en prenant en compte le coefficient lambda. On peut ensuite appliquer la méthode bilinéaire telle que présentée à l'équation 2.5, selon le même principe.

Pour déterminer les coefficients de la matrice d'homographie, nous devons utiliser les quatre points

$A_\mathcal{E}, B_\mathcal{E}, C_\mathcal{E}, D_\mathcal{E}$ de départ, et les quatre points $A_\mathcal{S}, B_\mathcal{S}, C_\mathcal{S}, D_\mathcal{S}$ d'arrivée, tels que

$$\boxed{\begin{array}{ccccc} H_C & A & = & X_{\mathcal{S},\mathcal{C}} \\ \Leftrightarrow H_C & = & X_{\mathcal{S},\mathcal{C}} & A^{-1} \\ \\ \Leftrightarrow \begin{pmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{pmatrix} & = & \begin{pmatrix} x_{A,\mathcal{S}} \\ y_{A,\mathcal{S}} \\ x_{B,\mathcal{S}} \\ y_{B,\mathcal{S}} \\ x_{C,\mathcal{S}} \\ y_{C,\mathcal{S}} \\ x_{D,\mathcal{S}} \\ y_{D,\mathcal{S}} \end{pmatrix} & A^{-1} \end{array}} \tag{2.7}$$

Pour avoir le bon nombre d'équations vis-à-vis des inconnues, on prendra $h_{33} = 1$.

Pour déterminer A, on sait que

$$\begin{cases} x_\mathcal{S} &= \frac{h_{11}x_\mathcal{E} + h_{12}y_\mathcal{E} + h_{13}}{h_{31}x_\mathcal{E} + h_{32}y_\mathcal{E} + h_{33}} \\ y_\mathcal{S} &= \frac{h_{21}x_\mathcal{E} + h_{22}y_\mathcal{E} + h_{23}}{h_{31}x_\mathcal{E} + h_{32}y_\mathcal{E} + h_{33}} \end{cases}$$

Sachant que $h_{33} = 1$, on peut facilement en déduire

$$\begin{cases} x_\mathcal{S} &= h_{11}x_\mathcal{E} + h_{12}y_\mathcal{E} + h_{13} - h_{31}x_\mathcal{E}x_\mathcal{S} - h_{32}y_\mathcal{E}x_\mathcal{S} \\ y_\mathcal{S} &= h_{21}x_\mathcal{E} + h_{22}y_\mathcal{E} + h_{23} - h_{31}x_\mathcal{E}y_\mathcal{S} - h_{32}y_\mathcal{E}y_\mathcal{S} \end{cases}$$

D'après l'équation 2.7, on peut donc déduire les coefficients de A :

$$\boxed{A = \begin{pmatrix} x_{A,\mathcal{E}} & y_{A,\mathcal{E}} & 1 & 0 & 0 & 0 & -x_{A,\mathcal{E}}x_{A,\mathcal{S}} & -y_{A,\mathcal{E}}x_{A,\mathcal{S}} \\ 0 & 0 & 0 & x_{A,\mathcal{E}} & y_{A,\mathcal{E}} & 1 & -x_{A,\mathcal{E}}y_{A,\mathcal{S}} & -y_{A,\mathcal{E}}y_{A,\mathcal{S}} \\ x_{B,\mathcal{E}} & y_{B,\mathcal{E}} & 1 & 0 & 0 & 0 & -x_{B,\mathcal{E}}x_{B,\mathcal{S}} & -y_{B,\mathcal{E}}x_{B,\mathcal{S}} \\ & & & & \vdots & & & \\ 0 & 0 & 0 & x_{D,\mathcal{E}} & y_{D,\mathcal{E}} & 1 & -x_{D,\mathcal{E}}y_{D,\mathcal{S}} & -y_{D,\mathcal{E}}y_{D,\mathcal{S}} \end{pmatrix}} \tag{2.8}$$

En utilisant les équations 2.8 et 2.7, on détermine ainsi les coefficients de la matrice d'homographie. Puis, pour chaque point $(x_\mathcal{S}, y_\mathcal{S})$ de l'image de sortie, on calcule ses coordonnées dans l'image d'entrée grâce à l'équation 2.6, avant d'appliquer la transformation bilinéaire.

## 2.3 Binarisation et déchiffrement des pixels

### Hypothèses

Nous supposerons dans cette partie que l'image a été correctement redressée, et résulte en une image M*M pixels ne contenant que l'image du QR code. L'image redressée est constituée d'une somme de carrés sombres et d'autres plus clairs. Cependant, des zones entières de l'image peuvent être plus contrastées que d'autres, ou juste plus sombres/plus claires que d'autres. Nous supposerons donc que l'image est uniformément éclairée, et si ce n'est pas le cas, que les zones de l'image plus éclairée/plus sombre ne crée pas d'erreurs d'interprétation de l'algorithme(i.e. que les carrés clairs de la partie sombre sont plus clairs que les carrés sombres de la partie claire, cf. illusion d'optique de l'échiquier à éviter).

### 2.3.1 Première partie : Binarisation

La première partie de l'algorithme consiste à calculer l'histogramme de l'image redressée, prenant en compte les 256 valeurs que peut prendre chaque pixel. Grâce à la spécification du QR code, nous savons que les pixels blancs et noir du QR code sont censés être presque équirépartis (± 1 pixels puisque les QR codes comptent un nombre impair de pixels). On utilie donc cette propriété pour exploiter l'histogramme des QR codes, qui peuvent se présenter comme dans la figure 2.10.

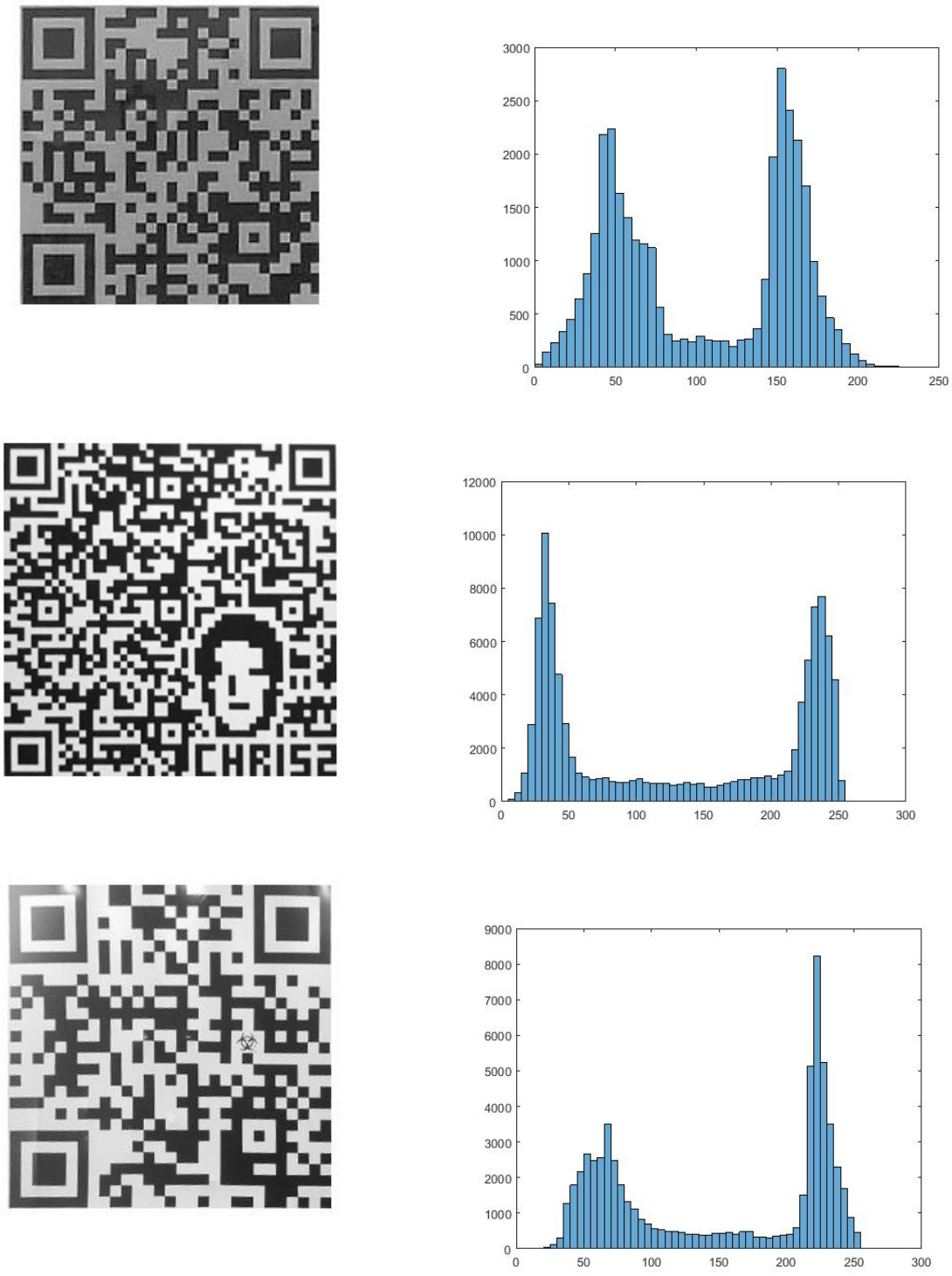



FIGURE 2.10 – Histogrammes des images (en luminance)

Ainsi, quelle que soit l'image choisie, l'histogramme présente deux dômes représentant les pixels clairs

et les pixels sombres, mais la hauteur de ces dômes varie. Pour pouvoir estimer au mieux la limite entre pixels sombres et pixels clairs, pour ensuite binariser l'image, deux algorithmes ont été mis au point. Nous allons les détailler ci-après.

### Binarisation – Détection de seuil : algorithme non robuste par intégrale

Après avoir calculé l'histogramme de l'image redressée, grâce à la caractéristique des histogrammes d'images de QR codes, cet algorithme exploite l'équirépartition des valeurs de l'image pour estimer le seuil de binarisation : il suffit de trouver la valeurs limite, où la moitié des valeurs se trouve au-dessous de cette valeur, et l'autre moitié en dessous, c'est-à-dire où on peut séparer deux populations, de taille $\frac{M \times M}{2}$ valeurs. Ainsi, en intégrant le nombre de pixels avec une certaine valeur, sur toutes les valeurs de l'image, il en sort le résultat décrit par le graphique représentatif sur la figure 2.11.

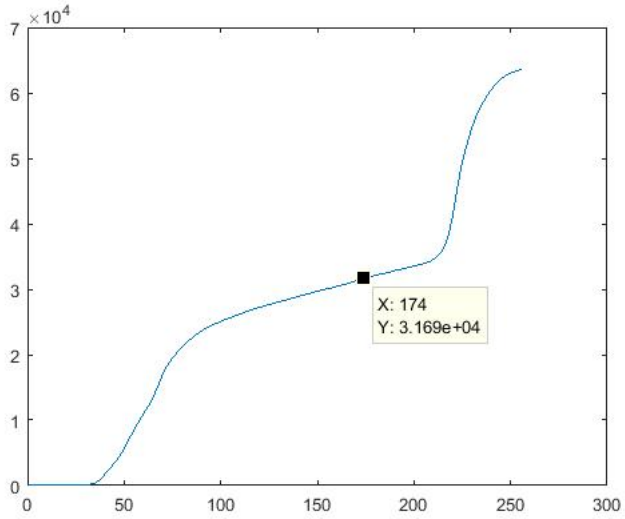



FIGURE 2.11 – Interprétation graphique de la détection du seuil par intégrale

Grâce à ce seuil, on peut ainsi appliquer l'algorithme de binarisation : tout pixel ayant une valeur en-dessous de cette valeur seuil sera mis à 0, et tout pixel ayant une valeur au-dessus sera mis à 1.

### Problèmes et limitations

Cet algorithme présente un désavantage majeur : si la population sombre paraît plus importante dans l'histogramme, ou si l'image est très peu contrastée (les deux dômes de l'histogramme sont très proches, voire se chevauchent), l'algorithme aura tendance à utiliser une valeur seuil non appropriée, comme on peut le voir sur la figure 2.12.

L'algorithme final utilisé ne repose donc pas sur le calcul d'une intégrale, mais plutôt sur un algorithme plus performant : l'algorithme d'Otsu (source : [4]).

### Binarisation – Détection du seuil par l'algorithme d'Otsu

L'algorithme d'Otsu se base, quant à lui, sur un calcul de probabilités. Pour chaque valeur, on calcule alors la probabilité de chaque pixels d'avoir une valeur inférieure ou égale à la valeur calculée, ainsi que la moyenne, comme visible dans l'équation 2.9.

$$\forall n \in [0;255], \boxed{\omega(n) = \sum_{i=0}^{n} \frac{Histogramme(i))}{NbPixels}, \mu(n) = \sum_{i=0}^{n} \frac{i \times Histogramme(i)}{NbPixels}} \tag{2.9}$$

FIGURE 2.12 – Exemple de résultat de mauvaise estimtion du seuil par intégrale

Les qualités du niveau de seuillage possibles sont ensuite calculées grâce à la formule suivante :

$$\boxed{crit(n) = \omega(n) \times (\mu(255) - \mu(n))^2 + (1 - \omega(n)) \times (\mu(n))^2} \tag{2.10}$$

On utilisera alors ce tableau de valeur pour faire correspondre son maximum à l'intensité à laquelle seuiller : la valeur seuil se trouve au maximum du tableau calculé, comme le montre la figure 2.13

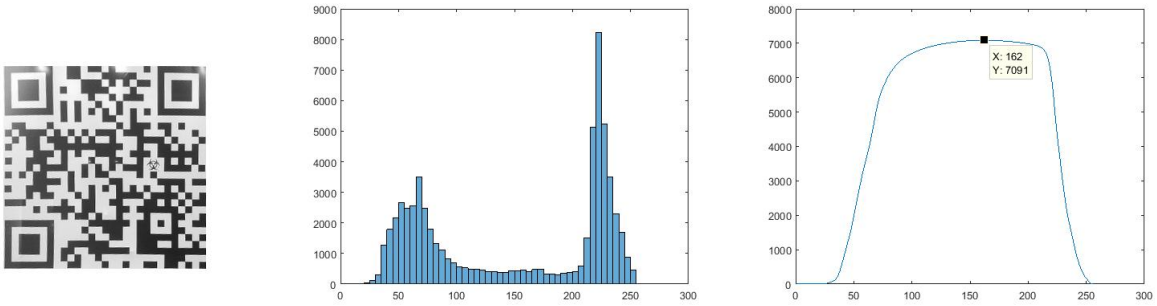



FIGURE 2.13 – Détermination du seuil, nécessaire à la binarisation

Une fois cette valeur trouvée, on peut ensuite appliquer la binarisation telle qu'expliquée dans le paragraphe précédent. Remarquons que les ancres, en bord d'image, sont souvent mal binarisées, sans pour autant dégrader le contenu du QR code, comme on peut le voir sur la figure 2.14.

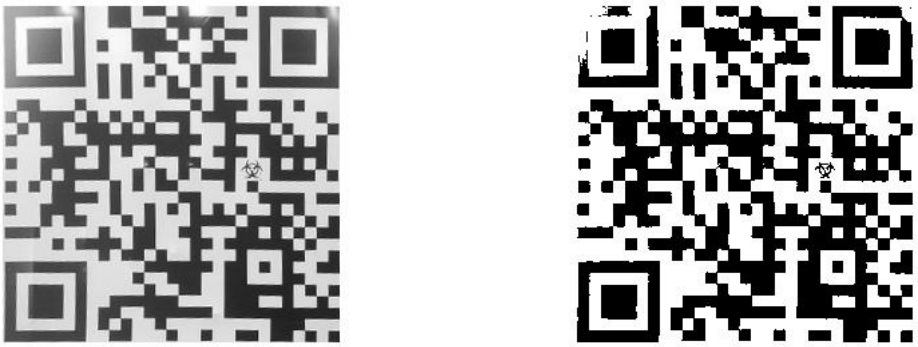


FIGURE 2.14 – Binarisation grâce à l'algorithme de Otsu

### 2.3.2 Deuxième partie : Détection des alternances du QR code

Dans la spécification, peut constater que, si les ancres se trouvent en bas à gauche, en haut à gauche et en haut à droite, la colonne 7 et la ligne 7 est uniquement composée d'un motif alternant les pixels noirs et les pixels blancs, comme visible sur la figure 2.15.

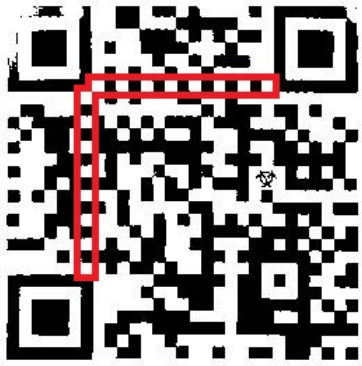


FIGURE 2.15 – Mise en évidence des alternances sur le QR code binarisé ci-avant

D'après la spécification, il y a 40 tailles de QR codes ; plus spécifiquement, le type 1 est un QR code de taille 21*21 pixels ; le type 40 est de taille 177*177 pixels ; et il y a 4 pixels de différence entre chaque type.

On en déduit que la taille d'un côté d'un QR code, en nombre de pixels, se calcule ainsi :

$$T = 17 + (4 \times type)$$

On peut ainsi trouver très facilement si le motif correspond à la taille, en comparant le motif de taille T (dont les 7 pixels de début et de fin sont noirs), avec la 7e ligne (ou 7e colonne) facilement identifiable par le type et la taille précédemment calculée. Si les deux motifs ne se correspondent pas, il faut passer au type suivant. Les motifs de l'image sont détectés en prenant le milieu de chaque carré de couleur composant le QR code, ceci pour éviter les mauvaises estimations et maximiser les chances de détection correcte si l'image redressée était amenée à être rognée sur un ou plusieurs des bords dans les étapes précédentes.

Remarquons qu'il est important de prendre en compte un pixel d'erreur : puisque nous prenons en compte un pixel et ses voisins directs pour pouvoir déterminer si l'endroit analysé est blanc ou noir, et puisque la binarisation n'est pas parfaite, l'algorithme tolère 1 pixel de différence avec la théorie. Il est aussi important de prendre en compte le fait que le motif se retrouve sur une ligne ET sur une colonne. Enfin, le motif peut être tourné de 90°, 180° ou 270°, une fois que l'algorithme a déterminé que le motif ne correspond pas, il est important de considérer les 3 autres possibilités, et donc de recommencer la comparaison en ayant pivoté l'image de 90°.

### 2.3.3 Troisième partie : Récupération des pixels

Maintenant que nous avons le type du QR code, et donc la taille de ce dernier, nous pouvons créer une matrice de taille $taille \times taille$, que nous allons remplir avec le milieu de chaque carré détecté. Les seules erreurs à corriger a priori, du fait du traitement de l'image, peuvent être les ancres. Puisque leur position est connue (du fait des rotations précédentes), nous savons que ces ancres se trouvent en haut à gauche, en bas à gauche, et en haut à droite, nous pouvons donc les forcer en les remplissant correctement une fois la détection faite.

On obtient donc la matrice de la figure 2.16

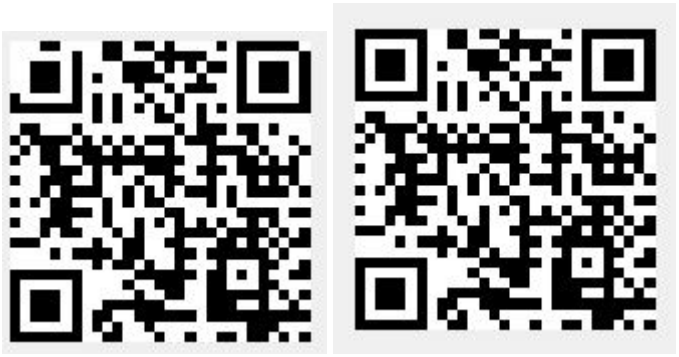


Figure 2.16 – Matrice contenant le QR code décodé, et matrice avec ancres corrigée

Maintenant que toutes les étapes théoriques ont été décrites, le déroulé de l'algorithme final est décrit dans la partie suivante.

# 3 Déroulé de l'algorithme final

Pour que le code produit fonctionne sans dysfonctionnements, il est nécessaire de respecter les différentes hypothèses formulées dans la partie théorique, i.e :

- L'image est suffisamment peu bruitée et peu dégradée pour permettre une détection optimale, notamment les ancres et les angles du QR code ;
- L'image est d'une taille raisonnable pour permettre un traitement en temps raisonnable ;
- Le QR code est projetable (i.e : la projection subie par le QR code dans l'image est suffisemment faible pour permettre une bonne reconnaissance de tous les pixels mis en jeu)
- Si le QR code est contrasté, les carrés clairs de la partie sombre sont plus clairs que les carrés sombres de la partie claire.

La première étape consiste à trouver les quatre points, extrémités du QR code. Pour cela, nous utilisons l'algorithme de détection des ancres par convolution (partie 2.1.3), puis le quatrième point est demandé à l'utilisateur, qui se charge de le placer correctement, c'est-à-dire parfaitement dans l'angle sans ancre. Les points corrects sont ensuite placés suivant l'algorithme utilisant la convolution avec un motif et les vecteurs entre les points (partie 2.1.5).

La deuxième étape est celle du redressement de l'image. Pour être sûr d'avoir la bonne suite de points (et donc pour former une image non-croisée), nous trions les points obtenus (grâce à l'algorithme de tri expliqué en partie 2.1.4). Nous calculons ensuite la plus grande longueur de côté : cette longueur deviendra M, la longueur de l'image redressée.

Nous calculons ensuite la matrice d'homographie, puis nous l'appliquons sur les coordonnées de notre image de sortie, tel qu'expliqué en partie 2.2.2. Puis nous utilisons la méthode bilinéaire pour recalculer les intensités de l'image d'entrée dans la nouvelle image.

La troisième étape est alors effectuée. Nous effectuons tout d'abord la binarisation de l'image grâce à la méthode d'Otsu (partie 2.3.1, second paragraphe). Nous détectons ensuite le motif grâce à l'algorithme explicité en 2.3.2, ce qui nous donne le type de QR code. À partir de cette sous-étape, si l'algorithme n'a pas réussi à détecter la taille du QR code, il renvoie -1 ; sinon, l'algorithme continue.

Nous pouvons alors effectuer la synthèse du QR code : dans une matrice de taille $17 + 4 * type$, nous récupérons les pixels grâce à l'algorithme expliqué en partie 2.3.3.

La partie décodage de l'information prend donc une matrice, contenant soit un QR code bien orienté,

soit -1.

# 4 Axes d'améliorations

Un certain nombre de points ont pu être modifiés et améliorés durant ce projet, mais certains aspects de l'algorithme global du traitement d'image restent encore à améliorer.

Tout d'abord, la détection des ancres n'est pas suffisamment robuste, puisqu'une part non négligeable des images traitées n'aboutit pas au placement correct des points. De plus, les axes x et y sont traités de manière globale, or une image très déformée aura tendance à être aplatie dans un sens (x ou y), ce qui peut fausser la détection et le placement correct du point. Pour remédier à ce problème, au niveau de la détection du centre des ancres, pour chaque convolution entre l'image rééchantillonnée au ième tour de boucle et le motif défini, selon x, il faudrait imbriquer une autre boucle d'échantillonnage et de convolution selon y. Le problème de cette approche, est qu'elle requiert un temps de calcul démesuré par rapport à l'application voulue. La librairie OpenCV base la détection des ancres de QR codes sur des contours intrieurs et extérieurs (6 au total), permettant dans le même temps de détecter l'orientation du QR code [1]. Il faudrait aussi robustifier la détection en appliquant des passe-bas en fonction de la taille de l'image pour éliminer les fréquences parasites, ou encore en utilisant un moyenneur des pixels au niveau de l'échantillonnage des diagonales pour placer correctement les points aux coins du QR code.

Ensuite, l'algorithme ne se sert pas (ou plutôt, pas suffisemment) de l'information de gradients donnée par la structure même du QR code ; cette information aurait pu être utilisée pour détecter le quatrième point.

La détection du seuil de binarisation dépend fortement de l'éclairage du QR code ; un algorithme plus robuste aurait pu fragmenter la zone en sous-zones, et appliquer l'algorithme d'Otsu sur cette zone. Remarquons cependant que ce dernier pourra être moins efficace puisque l'histogramme sera basé sur moins de points par zone.

Enfin, l'algorithme complet en lui-même est très lourd et prend beaucoup de temps de calcul et de ressources matérielles (mis à part le fait que le code est exécuté sous Matlab). Une toute autre approche serait de détecter les ancres, puis d'essayer de détecter le type de QR code directement en testant successivement les 40 possibilités. On éviterait ainsi le calcul complet de la matrice de passage grâce à l'homographie sur l'ensemble des points du QR code (seuls quelques points au centre des carrés de couleur seraient calculés) ; On pourrait aussi essayer de détecter le quatrième coin grâce à un lancé aléatoire d'un point sur l'image (dans une zone sélectionnée), suivie d'un ré-échantillonnage, ce qui éviterait des temps de calcul considérables. Notons tout de même que, la projection étant présente de toute manière, la matrice d'homographie serait tout de même à calculer.

1. Principe de détection des ancres dans OpenCV : `https://dsynflo.blogspot.com/2014/10/opencv-qr-code-detection-and-extraction.html` ; `http://www.ucsp.edu.pe/sibgrapi2013/eproceedings/wtd/114652.pdf`

# Deuxième partie

# Lecture d'un code QR

---

Après la détection du code QR dans l'image, on commence la lecture ce dernier.

## 5 Lecture d'un QR code sous forme de matrice de 0 et de 1

Le décodage d'un code QR se fait sur plusieurs étapes, on décrira par la suite chaque étape. La figure 5.1 ci-dessous présente les étapes de lecture d'un QR code [6].

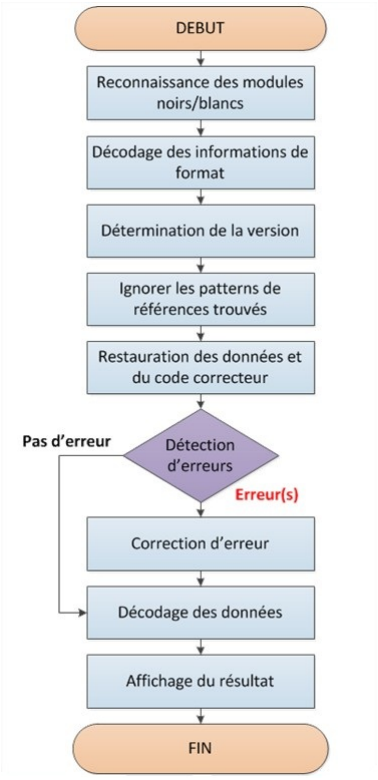



Figure 5.1 – Déroulé de la lecture d'un QR code

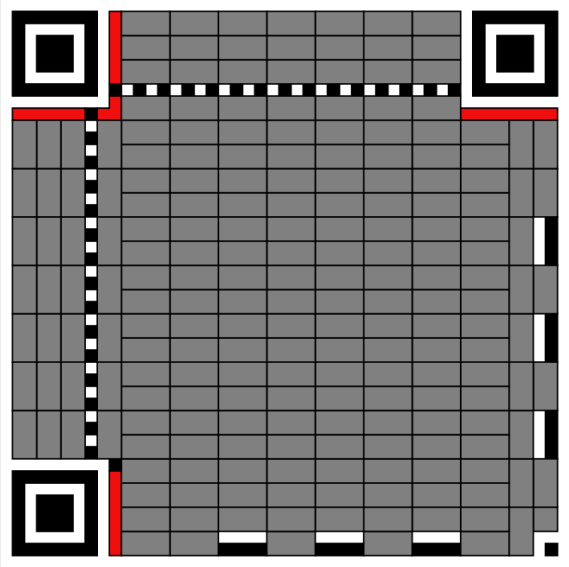


FIGURE 5.2 – Informations de format

## 5.1 Les informations de format

Les informations de format se trouvent dans une séquence de 15 bits [5] qui contient deux informations :
— Le niveau de correction d'erreur (2 bits)
— Le motif de masquage (3 bits)
Cette suite de bits se trouve en deux positions dans le QR code, verticale et horizontale. C'est une information redondante, soit verticale, soit horizontale, près des ancres du QR code. C'est la partie rouge visible dans la figure 5.2

Après le prélèvement de cette suite de bits , il faut le démasquer. Ainsi, on effectue un XOR de cette autre suite de bits, indépendante du QR code :

101010000010010

C'est le même flux qu'on utilise pour tous les QR code.

Ce flux peut être divisé en 3 parties :
— Les deux premiers bits représentent le niveau de correction d'erreur, du niveau L qui corrige le moins d'erreurs au niveau H qui corrige encore plus (Les lettres signifient Low,Medium,Quality et High) ;
— Du bit 3 au bit 5, c'est le masque ;
— Du bit 6 au bit 15, c'est un flux de bit qui se fait calculer à partir du code BCH (le code BCH est un code correcteur utilisé pour corriger des erreurs aléatoires).
Nous allons nous intéresser par la suite aux deux premières parties, après avoir expliqué l'intérêt du masque.

## 5.2 Démasquage

Le masquage des bits inverse certains modules de la zone de données en fonction des formules détaillées dans le tableau 5.3. Les variables i et j sont les numéros de ligne et de colonne de base (i,j)= (0,0).

L'objectif du masquage est de briser de grands blocs solides noirs ou blancs et d'éviter les motifs qui ressemblent de façon déroutante aux marques de repère. Il existe 8 modèles de masque différents (cf. figure 5.4 disponibles pour les codes QR standards).

| Masque | Équation |
|---|---|
| 000 | (i + j) mod 2 = 0 |
| 001 | i mod 2 = 0 |
| 010 | j mod 3 = 0 |
| 011 | (i + j) mod 3 = 0 |
| 100 | ((i div 2) + (j div 3)) mod 3 = 0 |
| 101 | (i*j) mod 2 + (i*j) mod 3 = 0 |
| 110 | ((i*j) mod 2 + (i*j) mod 3) mod 2 = 0 |
| 111 | ((i*j) mod 3 + (i+j) mod 2) mod 2 = 0 |

FIGURE 5.3 – Tableau récapitulatif des formules à utiliser

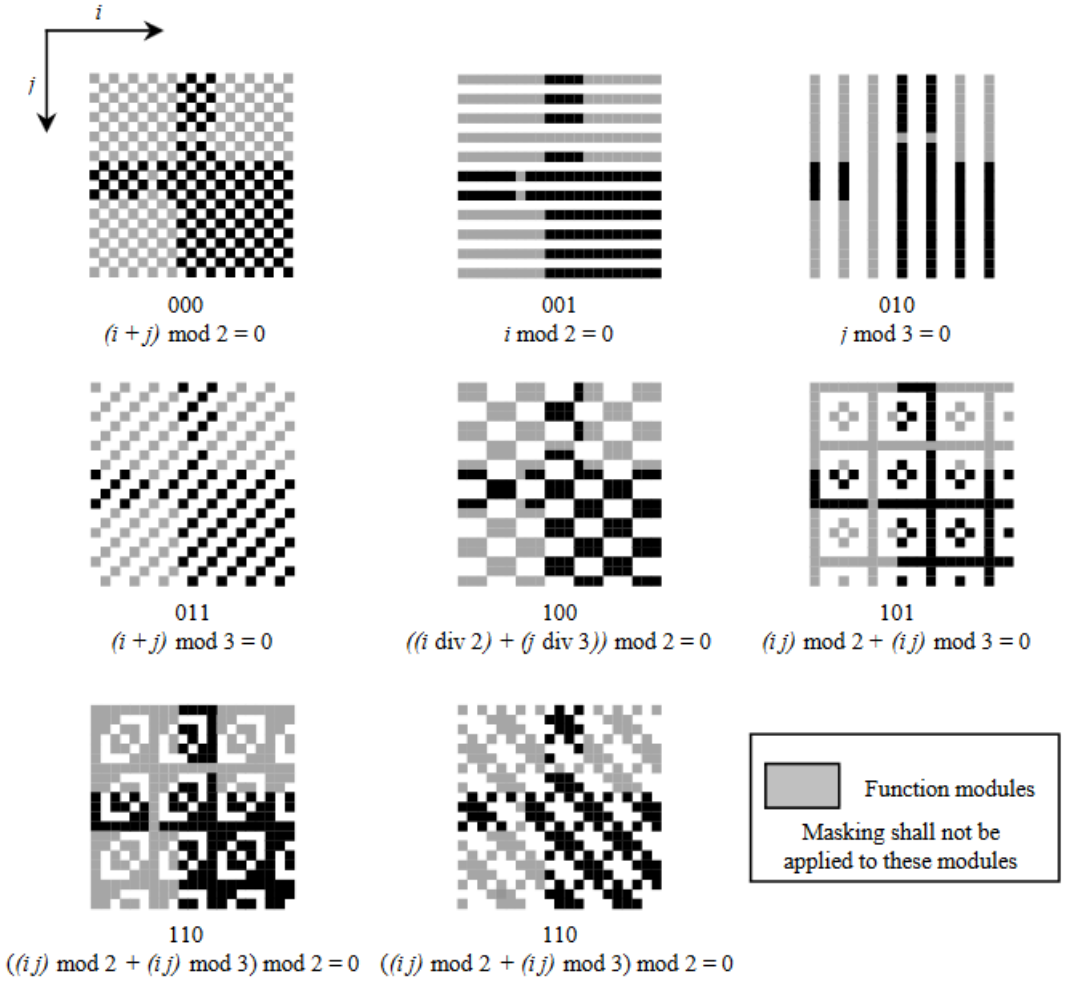



FIGURE 5.4 – Les 8 types de masques à appliquer

Pour démasquer le QR code, on crée une nouvelle matrice avec la même taille que celle de notre QR code, on commence à la remplir pixels par pixels en regardant la couleur des pixels dans les mêmes cordonnées de QR code et du masque qu'on va utiliser. Ainsi, on a :

$$Noir + noir = blanc \text{ et } Noir + blanc = noir.$$

Nous avons donc une image contenant notre QR code démasqué, on peut donc passer à l'étape suivante.

## 5.3 Extraction des données binaires

L'intérêt de cette étape est de transformer notre QR code en une suite de bits. Pour cela, et comme le montre la figure 5.5, on découpe notre QR code en rectangles, et chaque rectangle contient une suite de 8 bits (horizontal haut, horizontal bas, vertical haut, vertical bas). Nous avons donc créé une fonction qui prend en arguments les coordonnées du premier bit dans le rectangle et le type de rectangle, et qui retourne une suite de bits. On parcourt ainsi tout le QR code jusqu'à ce qu'on aie une suite de bits finale qui contient notre message en entier.

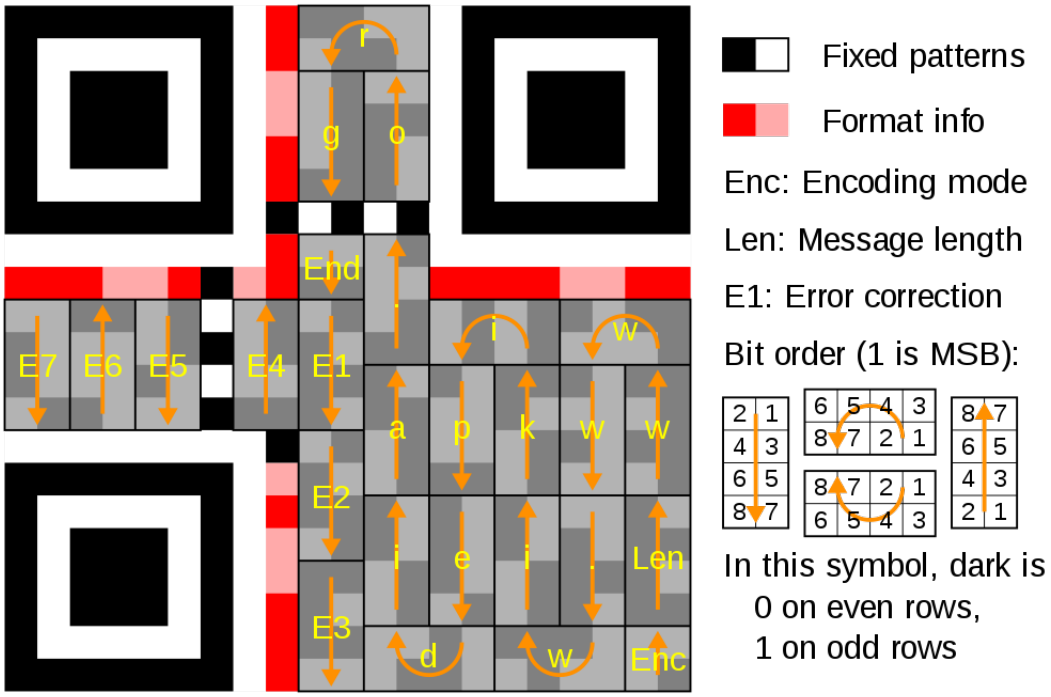



FIGURE 5.5 – Méthode de lecture du QR code

# 6 Décodage final

Le décodage de cette suite de bits se fait par plusieurs méthodes et le choix de la méthode dépends des 4 premiers bits de cette suite ; le tableau suivant figure 6.1 montre les différentes méthodes et leurs indicateurs correspondant.

On s'intéresse par la suite à 3 méthodes ( Numeric, Alphanumeric, Byte) car ce sont les méthodes les plus utilisées.

### 6.0.1 Numérique

Cette méthode ne permet de stocker que des nombres entiers positifs. A partir du 5eme bit, les premiers 10 bits représentent la longueur du message, on ne prend donc de cette suite que les bits dont on aura besoin pour trouver notre message. Ensuite, chaque 10 bits représentent 3 chiffres.

Que se passe-t-il si le nombre de chiffres n'est pas un multiple de 3 ? Si le nombre contient $3n+1$ chiffres, le dernier est codé sur 4 bits et s'il en contient $3n+2$, les deux derniers sont codés sur 7 bits.

| Indicator | Meaning |
|---|---|
| 0001 | Numeric encoding (10 bits per 3 digits) |
| 0010 | Alphanumeric encoding (11 bits per 2 characters) |
| 0100 | Byte encoding (8 bits per character) |
| 1000 | Kanji encoding (13 bits per character) |
| 0011 | Structured append (used to split a message across multiple QR symbols) |
| 0111 | Extended Channel Interpretation (select alternate character set or encoding) |
| 0101 | FNC1 in first position (see Code 128 for more information) |
| 1001 | FNC1 in second position |
| 0000 | End of message (Terminator) |

FIGURE 6.1 – Méthode de lecture suivant l'indicateur

### 6.0.2 Exemple

Apres l'extraction des données on aura une suite de bits qui ressemble à ça par exemple :

00010000 00100000 00001100 01010110 01100001 10000000

On découpe cette suite de la manière suivante :

0001 / 0000001000 / 0000001100 / 0101011001 / 1000011 / 0000

0001 : L'indicateur a 4 bits et les 4 derniers bits 0000 indiquent la fin du message.

Ensuite en transforme les autres blocs en décimal on aura :

0000001000 = 8, c'est la longueur du message. On a $\frac{8}{3} = 2 \times 10 = 20$ bits ; et $8 mod 3 = 2$ donc 7 bits de plus ; ainsi on aura besoin par la suite de 27 bits.

0000001100 = 012.

0101011001 = 345.

1000011 = 67.

D'où le message '01234567'.

### 6.0.3 Byte

Cette méthode est la plus facile, à partir du 5e bit dans la suite des bits, on découpe le reste en 8 bits et on le converti en décimal. Le premier nombre trouvé correspond à la longueur du message, on multiplie donc ce nombre par 8 pour connaître le nombre de bit dont on aura besoin. Ensuite, on transforme chaque nombre par le caractère qui le code dans le tableau ASCII, et on trouvera ainsi notre message.

### 6.0.4 Exemple

0100 / 00000011 / 01100001 / 01100010 / 01100011 / 0000

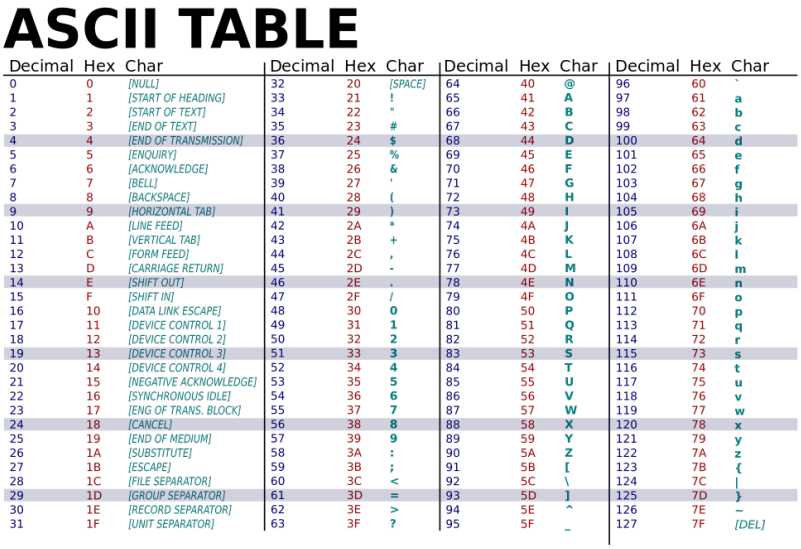
00000011 = 3 , la longueur du message.

01100001 = 97 , dans le tableau ASCII c'est la lettre a.

01100010 = 98 , dans le tableau ASCII c'est la lettre b.

01100011 = 99 , dans le tableau ASCII c'est la lettre c.

0000 = fin du message.



# ASCII TABLE

| Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | [NULL] | 32 | 20 | [SPACE] | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 1 | [START OF HEADING] | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 2 | [START OF TEXT] | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 3 | [END OF TEXT] | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 4 | [END OF TRANSMISSION] | 36 | 24 | $ | 68 | 44 | D | 100 | 64 | d |
| 5 | 5 | [ENQUIRY] | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 6 | [ACKNOWLEDGE] | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 7 | [BELL] | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 8 | [BACKSPACE] | 40 | 28 | ( | 72 | 48 | H | 104 | 68 | h |
| 9 | 9 | [HORIZONTAL TAB] | 41 | 29 | ) | 73 | 49 | I | 105 | 69 | i |
| 10 | A | [LINE FEED] | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | B | [VERTICAL TAB] | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | C | [FORM FEED] | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | D | [CARRIAGE RETURN] | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | E | [SHIFT OUT] | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | F | [SHIFT IN] | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | [DATA LINK ESCAPE] | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | [DEVICE CONTROL 1] | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | [DEVICE CONTROL 2] | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | [DEVICE CONTROL 3] | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | [DEVICE CONTROL 4] | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | [NEGATIVE ACKNOWLEDGE] | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | [SYNCHRONOUS IDLE] | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | [ENG OF TRANS. BLOCK] | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | [CANCEL] | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | [END OF MEDIUM] | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | [SUBSTITUTE] | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | [ESCAPE] | 59 | 3B | ; | 91 | 5B | [ | 123 | 7B | { |
| 28 | 1C | [FILE SEPARATOR] | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | \| |
| 29 | 1D | [GROUP SEPARATOR] | 61 | 3D | = | 93 | 5D | ] | 125 | 7D | } |
| 30 | 1E | [RECORD SEPARATOR] | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | [UNIT SEPARATOR] | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | [DEL] |

FIGURE 6.2 – Utilisation de la table ASCII pour décoder le message

### 6.0.5 Alphanumérique

Le format alphanumérique permet d'écrire des URL ou du texte simple. Il contient 45 symboles : de 0 à 9, les chiffres, de 10 à 35, les lettres et enfin neuf symboles supplémentaires de 36 à 44. Le tout forme la chaîne de caractères "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ $%*+-./ :"(l'espace après le Z est compris dans cette suite).

**Alphanumeric character codes**

| Code | Character | Code | Character | Code | Character | Code | Character | Code | Character |
|---|---|---|---|---|---|---|---|---|---|
| 00 | 0 | 09 | 9 | 18 | I | 27 | R | 36 | Space |
| 01 | 1 | 10 | A | 19 | J | 28 | S | 37 | $ |
| 02 | 2 | 11 | B | 20 | K | 29 | T | 38 | % |
| 03 | 3 | 12 | C | 21 | L | 30 | U | 39 | * |
| 04 | 4 | 13 | D | 22 | M | 31 | V | 40 | + |
| 05 | 5 | 14 | E | 23 | N | 32 | W | 41 | – |
| 06 | 6 | 15 | F | 24 | O | 33 | X | 42 | . |
| 07 | 7 | 16 | G | 25 | P | 34 | Y | 43 | / |
| 08 | 8 | 17 | H | 26 | Q | 35 | Z | 44 | : |

FIGURE 6.3 – Utilisation de la table Alphanuérique pour décoder le message

A partir du 5e bit, les 11 premiers bits représentent la longueur, ensuite chaque 11 bits représentent deux symboles : le quotient et le reste de la division par 45. Si le nombre de symboles est impair, le dernier est codé sur 6 bits.

### 6.0.6 Exemple

0010 /000000101/ 00111001110 /11100111001 / 000010/ 0000

000000101 = 5

00111001110 = 492 et on a 492 = 10*45 +12 donc (10,12)

11100111001 = 1849 et on a 1849 = 41*45 + 4 donc (41,4)

000010 = 2 donc (2)

D'où le message (10,12,41,4,2) : on remplace ces nombres avec les caractères qui leur convient à partir du tableau. On trouve message 'AC-42'

# Troisième partie

# Interfaçage des deux parties, résultats et conclusions

---

L'interfaçage s'effectue grâce à la matrice passerelle de 0 et de 1 : la partie de traitement d'images renvoie cette matrice, et la partie de décodage des bits utilise cette matrice résultat. La matrice est censée être l'image exacte du QR code.

Le code développé n'est évidemment pas parfait et comporte des imperfections. Ainsi, Si l'image contient des fréquences parasites ou une projection trop importante, des bits risquent de ne pas être décodés correctement, et la matrice passerelle sera alors erronnée. Plus particulièrement, la détection des coins est perfectible et est encore très sensible, que ce soit pour la détection des ancres, ou sur le placement des coins à partir du centre des ancres. Le code donné est donc pensé pour mettre de côté cette détection et placer les points à la main, si la détection des coins échoue.

De plus, la lecture ne peut être effectuée que sur des QR codes de type 1 (taille 21x21 pixels), et ne contient pas encore la détection et la correction des erreurs.

Cependant le travail effectué nous a permis de comprendre des aspects fondamentaux du traitement d'images et de la lecture de données, notamment la détection de motifs par différentes méthodes, le redressement d'images, le choix de critères pour les différentes étapes du traitement ou la lecture d'un flux. Si nous avions eu plus de temps, le code aurait pu être revu et robustifié, et nous aurions aussi pu étendre la lecture pour d'autres types de QR codes et pour d'autres tailles.

# Bibliographie


[1] Y. Berthoumieu. Introduction au traitement de l'image. Cours E2, Enseirb-Matmeca, Bordeaux INP, 2019.

[2] M. Donias. Initiation au traitement des images. pages 156–164. Cours 2A, Ensc, Bordeaux INP, 2019. http ://donias.vvv.enseirb-matmeca.fr/iti/ensc_iti_c_1p.pdf.

[3] International Standard ISO. Iso 18004 : Information technology—automatic identification and datacapture techniques—bar code symbology—qr code. `https://www.swisseduc.ch/informatik/theoretische_informatik/qr_codes/docs/qr_standard.pdf`, 2006. [En ligne ; dernier accès le 25 avril 2019].

[4] Wikipédia. Méthode d'otsu, 2019. [En ligne ; dernier accès le 25 avril 2019].

[5] Wikipédia. Qr code. https ://en.wikipedia.org/wiki/QR_code, 2019. [En ligne ; dernier accès le 15 mai 2019].

[6] Wikipédia. Qr code en action. http ://www-igm.univ-mlv.fr/ dr/XPOSE2011/QRCode/workflows.html#2, 2019. [En ligne ; dernier accès le 15 mai 2019].