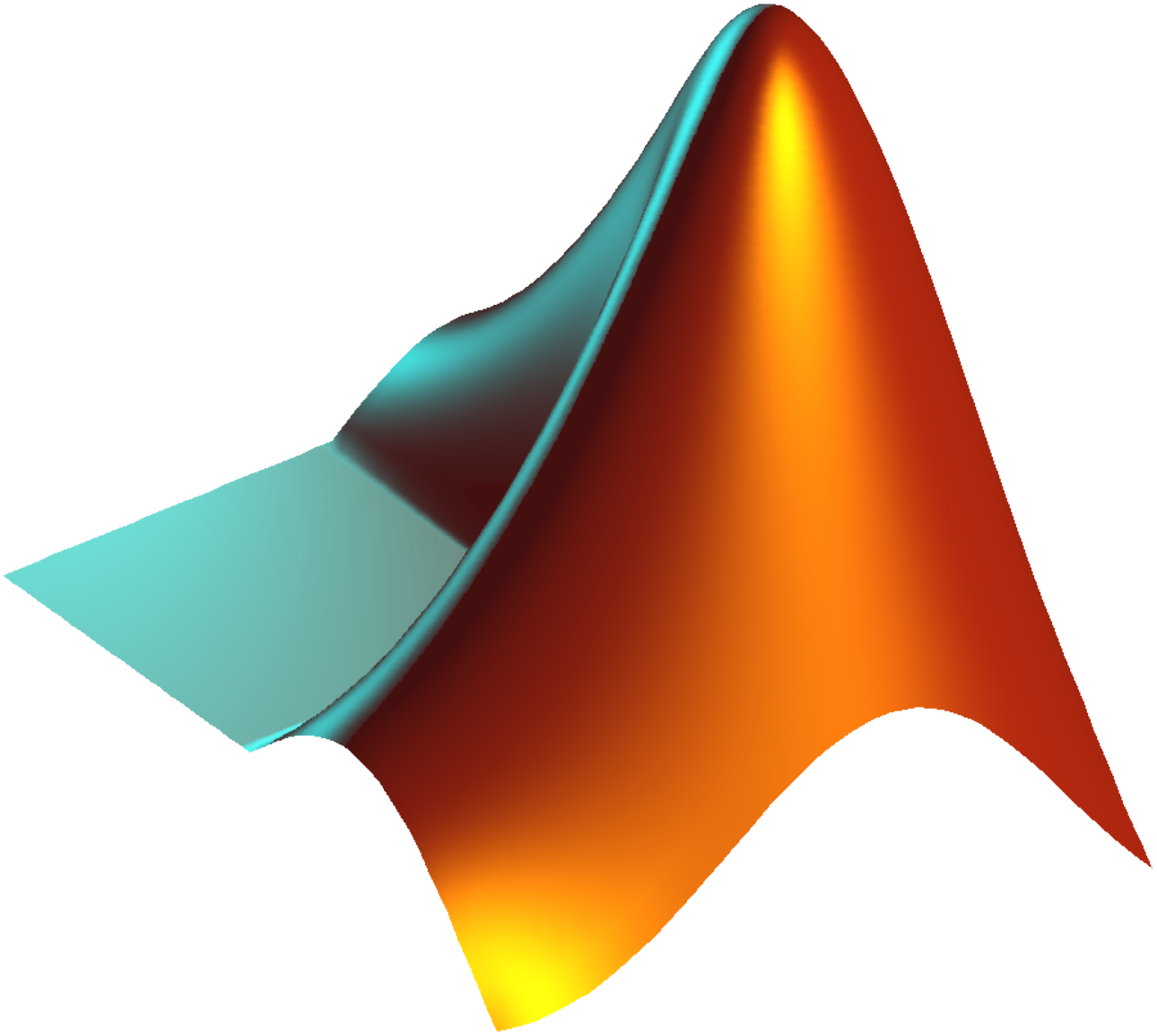
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN NORMANDIE

Projet de Physique P6
STPI/P6/2019 - 002

Traitement d'images sur MATLAB



Étudiants :
Amina El Bachari
Auriane Maymard
Emma Guilbault
Gauthier Imbert
Gawein Le Goff
Thomas Halipre

Enseignant-responsable du projet :
Anas Ayoub

Date de remise du rapport : 17/06/2019

Référence du projet : STPI/P6/2019 – 002

Intitulé du projet : Traitement d'images sur MATLAB

Type de projet : Bibliographie et TP

Objectifs du projet :
Mots-clés du projet : MATLAB, traitement d'images, code QR, développement

Institut National des Sciences Appliquées de Rouen
Département Sciences et Techniques Pour l'Ingénieur
685 Avenue de l'Université BP 08 - 76801 Saint-Étienne-du-Rouvray
Tél : +33 2 32 95 66 21 - Fax : +33 2 32 95 66 31

Table des matières

Notations et Acronymes

MATLAB : Matrix Laboratory
GUI : Graphic User Interface
QR Code (ou code QR en français) : Quick Response Code
RGB : Red Green Blue
Pixel : Picture Element
TF : Transformée de Fourier
FFT : La Fast Fourier Transformation inventée par J.Tukey et J.Cooley en 1965 est un algorithme qui réduit le nombre d'opérations nécessaires pour calculer la TF à un ordre $O(n \log^2(n))$
GUIDE : Graphical User Interface Development Environment

Introduction

Dans le cadre de notre projet P6 nous nous sommes intéressés au traitement d'images sur MATLAB. Ce projet aboutira la réalisation d'un rapport, à une soutenance orale et à la création d'une affiche. Outre l'apprentissage du travail en groupe, l'objectif était d'étendre nos connaissances scientifiques et technologiques.

Comme nous avons pu nous en rendre compte, le traitement d'images associe l'informatique et les mathématiques. Il permet de modifier une image ou d'en extraire de l'information.

Pour l'ensemble du groupe, il s'agissait d'une complète découverte du logiciel et du traitement d'images.

Notre premier objectif fut de découvrir l'outil MATLAB et ainsi de comprendre ce qu'est une image numérique. Ensuite notre professeur référent nous a distribué une feuille d'exercices à réaliser. L'objectif final de notre projet était donc de réutiliser cette feuille d'exercices afin d'implémenter différentes fonctions. Nous avons alors travaillé sur trois sous-projets en parallèle. D'un côté l'approfondissement du traitement d'images, d'un autre la lecture d'un code QR. Enfin, le troisième sous-projet synthétise l'ensemble des travaux du groupe en réalisant une interface graphique.

Nous avons donc exploré les deux principaux aspects du traitement d'images : la transformation (fonctions de traitement d'images) et l'extraction d'informations (lecture d'un code QR).

Organisation du travail

Lors de la première séance, sur les conseils de M. Ayoub, nous avons commencé par découvrir MATLAB : comment ouvrir une image, modifier les valeurs de sa matrice...

Les séances suivantes, nous nous sommes penchés sur une série d'exercices dans le but de nous familiariser avec le logiciel et son fonctionnement. Nous avons formé un groupe de 3 personnes, débutantes et les 3 autres, plus expérimentées, ont préféré travailler individuellement. Ces exercices consistaient à changer le format d'une image, la faire pivoter de 90 degrés, la convertir en noir et blanc...

Après avoir réalisé ces exercices, nous nous sommes réunis afin de faire un point sur l'avancement du travail, mettre au point un planning et définir les objectifs pour la suite.

Enfin nous nous sommes répartis en groupes de 2, le premier travaillant sur l'interface graphique, le second sur le code QR et le dernier sur les fonctions utilisées dans l'interface. Enfin, chaque groupe a rédigé sa partie du rapport.

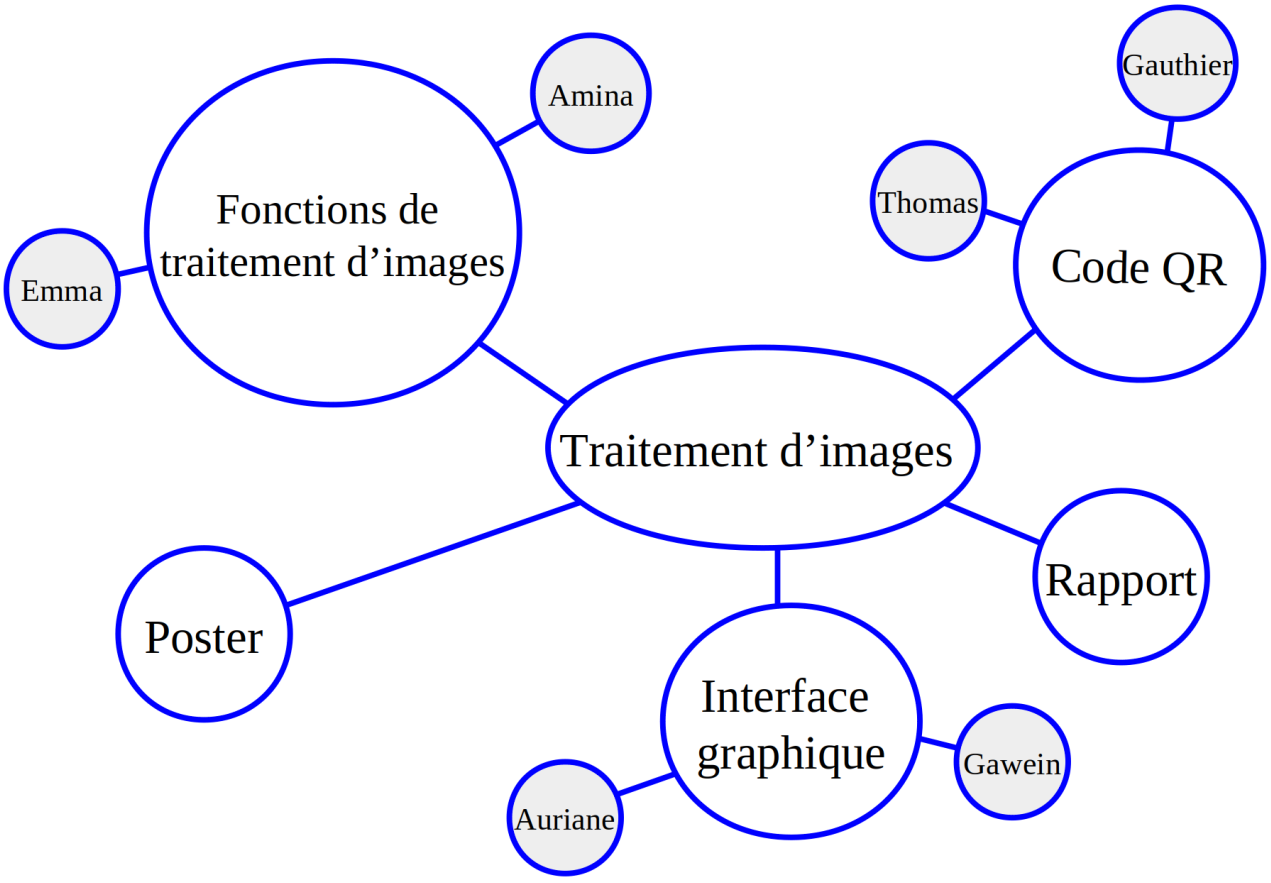


Figure 1 – Organigramme des tâches

Chapitre 1

Le Traitement d'Images

1.1 Une image

1.1.1 Qu'est-ce qu'une image ?

Tout d'abord il faut distinguer les images analogiques des images numériques. Une image numérique est représentée par un ensemble fini de valeurs. Si on possède l'ensemble de ces valeurs, il est possible de la reproduire à l'identique. En ce qui concerne les images analogiques, elles existent grâce à un support matériel (exemple une toile pour une peinture) et ne sont pas reproduisibles à l'identique. Pour les images numériques, il s'agit de valeurs discrètes contrairement aux images analogiques dont les valeurs sont continues. Nous allons nous intéresser uniquement aux images numériques.

Elles sont composées d'un nombre fini de pixels auxquels est associé une couleur ou une nuance de gris. La résolution d'une image, quant à elle, est liée au nombre de pixels qui la constituent. Plus il y aura de pixels, plus la résolution sera bonne. Plus il y aura pixels plus on pourra zoomer également (dans la limite de la résolution de l'écran sur lequel nous affichons l'image).

Ce que nous appelons couramment une image en noir et blanc est en fait une image en nuances de gris. Une image en noir est blanc est une sous-catégorie des images en nuances de gris avec seulement 2 nuances différentes (0 pour le noir et 1 ou 255 pour le blanc). Autrement dit, pour les images en noir et blanc les pixels ne peuvent prendre que deux valeurs, c'est une image binaire. De plus, plus il y a de nuances de gris, plus la qualité de l'image est bonne. On peut le constater avec l'exemple ci-dessous (figure 1.1), l'image a été traitée par des fonctions que nous présenterons par la suite.

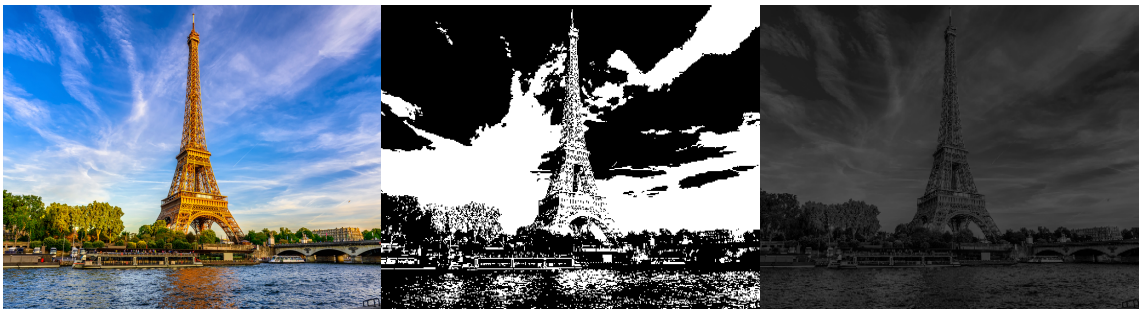


FIGURE 1.1 – Image couleur / noir et blanc / nuances de gris

Pour les images en couleur il y a en fait 3 couleurs pour chaque pixel. Il existe 256 nuances de ces 3 couleurs : le rouge, le vert et le bleu. Cela fait plus de 16 millions de couleurs différentes. Une image numérique est représentée par un tableau ou une matrice contenant les différentes valeurs de chaque pixel. Pour les images en couleur il s'agit d'une matrice 3D de dimensions (m,n,3), il y a donc une matrice pour chacune des 3 couleurs citées précédemment.

1.1.2 Le format

Il existe différents formats d'image dans MATLAB, parmi lesquels *uint8* et *double*. Ces formats définissent le type des valeurs que peuvent prendre les pixels (réels ou entiers) et l'intervalle des ces valeurs $[0;1]$ ou

$\{0;\ldots;255\}$). Le format *uint8* permet d'utiliser des valeurs entières comprises entre 0 et 255, stockées sur 8 bits. Cependant il y a perte d'informations lorsque l'on effectue certaines opérations puisque les valeurs sont entières. Le format *double* permet d'utiliser des valeurs réelles comprises entre 0 et 1, stockées sous forme de flottants sur 64 bits. Ce format prend donc plus de place en mémoire mais réduit la perte d'informations.

1.2 Les fonctions de traitement d'images

1.2.1 Changements de format et rotations

Comme nous l'avons expliqué précédemment, il est parfois plus facile de manipuler des images en format *double* qu'en format *uint8*. C'est pour cela que nous avons mis au point deux fonctions afin de pouvoir convertir les images que nous manipulons.

La première fonction (A.1.1.1) permet de passer d'un intervalle $[0,1]$ à un intervalle $\{0;\ldots;255\}$ en multipliant par 255 et en utilisant la fonction *floor* pour obtenir des valeurs entières .

La seconde fonction (A.1.1.2) permet au contraire de passer d'un intervalle $\{0;\ldots;255\}$ à un intervalle $[0,1]$ en divisant par 255. Comme le format double accepte les valeurs réelles il n'y a pas besoin d'utiliser la fonction *floor*.

Nous avons également mis au point 2 fonctions permettant de faire pivoter l'image de 180° (A.1.1.3) et 90° (A.1.1.4).

Concernant la rotation à 180 degrés, il suffit d'utiliser la formule suivante sur chaque pixel : $f(i,j) = (n+1-i, m+1-j)$ où $n \times m$ est la dimension de l'image (de sa matrice). La seconde fonction permet de pivoter une image de 90 degrés. On utilise la formule suivante : $f(i,j) = (m-j+1, i)$. On peut noter que ces fonctions peuvent être utilisées sur des images en couleurs.

1.2.2 Nuances de gris / noir et blanc

La fonction Nuances de gris (A.1.1.5) permet de transformer une image en couleur en image en nuances de gris. Pour cela on prend, pour chaque pixel, les valeurs de l'intensité de rouge, de bleu et de vert et on effectue une moyenne pondérée et empirique (nous avons choisi les coefficients expérimentalement). Autrement dit on passe d'une image définie par 3 matrices à une image définie par une seule matrice.

La fonction Noir et Blanc (A.1.1.6) quant à elle prend en entrée une image en nuances de gris et renvoie une image en noir et blanc. Pour cela il suffit de choisir un seuil au-dessus duquel chaque pixel prend la valeur 1 et en-dessous la valeur 0. Ainsi on obtient une image binaire (avec que des 0 et des 1 en format double). Le seuil que nous avons choisi se base sur la médiane de toutes les intensités des pixels. Ce seuil dépend du résultat attendu. Si on souhaite, par exemple, conserver les détails les plus clairs il faut choisir un seuil élevé.

Nous avons déjà observé les effets de ces deux fonctions dans la partie "Qu'est-ce qu'une image ?".

Nous avons également créé une fonction Négatif (A.1.1.7), qui renvoie le négatif d'une image, c'est à dire, qui renvoie la valeur opposée (modulo 256) de chaque pixel. Par exemple un pixel ayant la valeur 22 prendra la valeur 233. Voyez comme c'est joli.

FIGURE 1.2 – Négatif de l'image en couleur

1.2.3 L'expansion dynamique

Cette fonction (A.1.2.1) consiste à étaler les intensités d'une image, comprises dans l'intervalle $\{a;\ldots;b\}$ sur l'intervalle $\{0;\ldots;255\}$. Ainsi l'image contient tous les niveaux de gris, ce qui améliore le contraste. L'expansion dynamique permet de discerner des détails non visibles sur l'image originale mais n'a presque aucun effet sur les

images dont l'histogramme occupe presque toute la plage des niveaux de gris. Voici un exemple d'application de cette fonction.

FIGURE 1.3 – Nuances de gris avant et après l'expansion dynamique

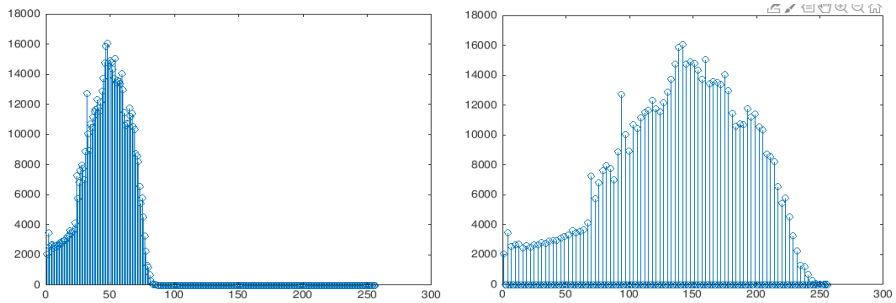


FIGURE 1.4 – Histogramme avant et après l'expansion dynamique

On voit bien l'étalement des valeurs et donc l'amélioration du contraste.

1.2.4 L'égalisation de l'histogramme [2]

Le but de cette fonction (A.1.2.2) est d'égaliser l'histogramme d'une image en nuances de gris. L'égalisation permet d'améliorer le contraste de l'image en rendant les différents niveaux de gris plus équiprobables. Pour cela nous avons besoin de l'histogramme cumulé. Nous créons une nouvelle matrice dans laquelle chaque pixel prend la valeur de 255 $\times$ le pourcentage de pixels ayant la valeur correspondante dans l'image originale.

Quant à l'histogramme cumulé, il permet de savoir combien de pixels ont une valeur en-dessous d'un niveau de gris fixé. Par exemple, le nombre de pixels d'une image dont la valeur est inférieure à 156 (en *uint8*). Une fois divisé par le nombre total de pixels dans l'image, on obtient un taux qui nous sera utile dans la fonction égalisation.

Voyons le résultat de l'égalisation de l'image en nuances de gris.

FIGURE 1.5 – Avant et après l'égalisation

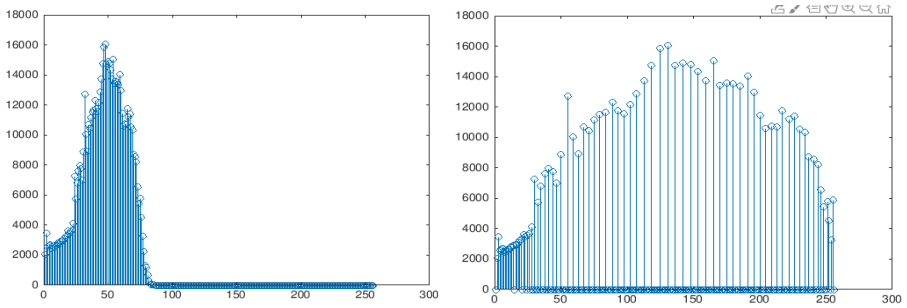


FIGURE 1.6 – Histogramme avant et après l'égalisation

On voit que le contraste de l'image est nettement amélioré. Les niveaux de gris sont mieux répartis qu'avec une simple expansion dynamique. Enfin on remarque que l'égalisation de l'histogramme effectue également un étalement des valeurs.

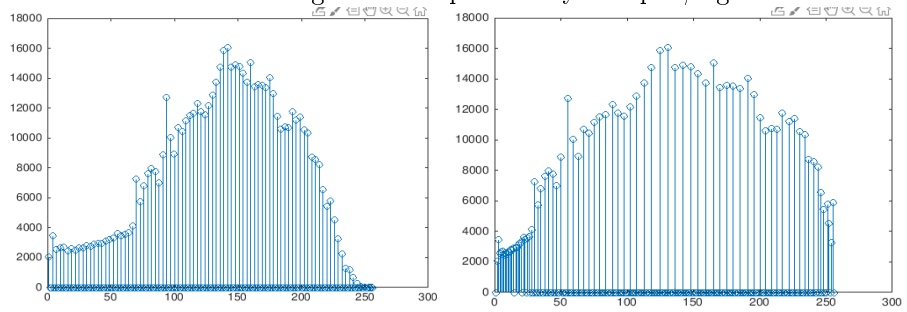
1.2.5 L'expansion dynamique et l'égalisation

L'égalisation de l'histogramme est plus efficace que l'expansion dynamique comme on peut le voir ci-dessous.

FIGURE 1.7 – Expansion dynamique / égalisation

On voit que le contraste est plus marqué. Ceci est dû au fait que l'égalisation va non seulement étaler l'histogramme mais aussi l'homogénéiser.

FIGURE 1.8 – Histogrammes expansion dynamique / égalisation



On constate que le "creux" entre 0 et environ 65 est réduit avec une égalisation (accentuation des nuances de gris proches du noir). De même, on remarque que la répartition des nuances de gris de 150 à 250 est beaucoup plus homogène (accentuation du clair). Ceci explique l'amélioration du contraste.

1.2.6 Le flou

1.2.6.1 Généralités et convolution [3]

Pour flouter une image il suffit de créer un masque, autrement dit une matrice carrée, que l'on remplit de 1. On divise ensuite les coefficients (des 1 donc) par leur somme totale. Ensuite nous réalisons un produit de convolution 2D entre notre masque et la matrice de notre image. Notre masque va alors "se déplacer" sur notre

matrice et chaque pixel va prendre la moyenne pondérée de sa valeur et des valeurs des pixels voisins. Nous l'avons illustrée dans l'exemple ci-dessous :

Soit I la matrice de notre image en nuances de gris, l'intensité du pixel de coordonnées (i, j) est donné par $I_{i,j}$. Prenons un masque : $H_{n,n}$ avec par exemple par exemple $n = 3$, alors

$$H = \begin{pmatrix} h_{1,1} & h_{1,2} & h_{1,3} \\ h_{2,1} & h_{2,2} & h_{2,3} \\ h_{3,1} & h_{3,2} & h_{3,3} \end{pmatrix}$$

H est appelée masque de convolution, ce masque va se déplacer sur l'ensemble de la matrice et appliquer à chaque pixel la formule suivante :

$$\begin{aligned} I_{i,j} = {} & h_{1,1}I_{i-1,j-1} + h_{2,1}I_{i,j-1} + h_{3,1}I_{i+1,j-1} \\ & + h_{1,2}I_{i-1,j} + h_{2,2}I_{i,j} + h_{2,3}I_{i+1,j} \\ & + h_{3,1}I_{i-1,j+1} + h_{3,2}I_{i,j+1} + h_{3,3}I_{i+1,j+1} \end{aligned}$$

Autrement dit chaque pixel prend la valeur de la moyenne (car on divise par la somme des coefficients du masque) des pixels voisins. Ceci explique pourquoi le flou ne s'applique pas sur les bords de l'image, là où les pixels n'ont pas de voisins.

Plus le masque est grand, plus la moyenne effectuée prend en compte un grand nombre de pixels voisins. Le flou sera donc plus marqué.

1.2.6.2 Utilisation de la gaussienne

Pour réaliser des flous plus sophistiqués nous pouvons remplacer les 1 du masque de convolution par d'autres coefficients.

Dans la première version du flou nous avons ainsi utilisé une fonction gaussienne, en deux dimensions. En effet, cette fonction est très utile car beaucoup de phénomènes naturels ou physiques suivent une distribution gaussienne.

Elle est de la forme :

$$f(x, y) = Ae^{-\left(\frac{(x-x_0)^2}{2\sigma_x^2} + \frac{(y-y_0)^2}{2\sigma_y^2}\right)}$$

Dans notre projet nous avons pris $x_0 = y_0 = \frac{n}{2}$ où n est la dimension de notre masque et une amplitude $A = 1$. Cette fonction nous permet de créer un filtre de Gauss (filtre passe-bas), obéissant à la loi normale.

Tout d'abord nous créons notre gaussienne. σ correspond à l'écart-type, c'est à dire à la largeur de la cloche gaussienne. Plus σ est grand, plus le flou est marqué.

Ensuite nous réalisons le produit de convolution. Le problème de ce floutage est que nous ne savons pas quel taux entrer, il faut donc faire plusieurs essais pour obtenir une image floue mais certains taux provoquent une saturation de l'image.

1.2.6.3 La transformée de Fourier [4]

Cette fonction *blurFourier* (A.1.2.5) a pour but de transformer une image en nuances de gris nette en image en nuances de gris floue. Pour comprendre cette transformation il convient tout d'abord d'expliquer comment fonctionne la transformée de Fourier.

D'un point de vu mathématique la TF est définie sur des fonctions continues de $-\infty$ à $+\infty$. Elle permet de passer d'une représentation spatiale ou temporelle à une représentation dans le domaine fréquentiel. Il existe 2 types de TF, la transformée de Fourier continue et la transformée de Fourier discrète. Une image en nuances de gris est une répartition d'intensités lumineuses dans un plan, c'est donc un signal à 2 dimensions qui peut être représenté par une fonction $f(x, y)$ continue ou discrète. Cependant les images numériques que nous voulons traiter sont représentées par des matrices de nombres ou chaque élément correspond à l'intensité lumineuse de chaque pixel, donc les images numériques sont discrètes. Alors nous allons utiliser la TF discrète.

En physique nous pouvons observer un phénomène concret de la transformation de Fourier : la décomposition de la lumière blanche par un prisme. Une onde lumineuse est un signal décomposable en somme (finie ou infinie) de sinus et cosinus. C'est l'analyse de Fourier. La TF permet de déterminer les sinusoïdes, ainsi que déterminer les fréquences correspondant à chaque couleur.

Soit $f(x, y)$ une fonction de 2 variables représentant l'intensité lumineuse du pixel de coordonnée $(x; y)$ d'une image F de taille $m \times n$. La transformée de Fourier discrète est :

$$TF[f(x,y)] = F(v_x; v_y) = \sum_{x=0}^{n-1} \sum_{y=0}^{m-1} f(x,y) e^{-i2\pi(\frac{v_x \times x}{n} + \frac{v_y \times y}{m})}$$

avec v_x et v_y les coordonnées spectrales.

On obtient finalement un spectre de Fourier des différentes fréquences et amplitudes de notre image. Afin de flouter notre image il faut, après avoir appliqué la TF, appliquer un filtre passe bas pour garder les basses fréquences, c'est a dire les pixels qui ont des intensités lumineuses proches (en dessous d'une fréquence de coupure notée f_c). Les hautes fréquences sont supprimées par le filtre et prennent une valeur moyenne des basses fréquences proches. Les hautes fréquences de notre image correspondent à un changement brusque d'intensité lumineuse, donc aux contours de notre image. Flouter notre image revient à supprimer ces contours.

Coder cette fonction sur MATLAB nécessite l'utilisation d'un masque de la taille de notre image qui va nous permettre de créer notre filtre passe bas. Suivant les valeurs de notre matrice les effets sur notre images seront différents. Nous créons donc une matrice avec des 1 au centre et des 0 autour, c'est le masque.

Procédons maintenant à l'explication du code *blurFourier*. La fonction *fft2* nous permet dans un premier temps d'effectuer la TF de notre image. Une matrice complexe de la même taille que notre image nous est renvoyée contenant les différents coefficients de la TF. Ensuite la commande *fftshift* permet de placer la fréquence nulle au centre de l'image afin de mieux visualiser les propriétés de symétrie de notre image à valeur réelle. Nous pouvons maintenant appliquer le masque pour supprimer les hautes fréquences et garder les basse fréquences situées au centre de la matrice. Nous sommes passés à un espace fréquentiel pour calculer la TF plus simplement. Il nous faut maintenant revenir à notre espace de départ pour pouvoir visualiser notre image. Pour cela nous utilisons la TF inverse avec la fonction *ifft2*.

Chapitre 2

Le code QR

Le code QR, de l'anglais Quick Response, est une version améliorée des codes barres, dont le but est de représenter une donnée numérique. Il est formé de modules noirs, disposés sur un fond blanc, qui définissent donc l'information que transmet le code.

Le premier code barre est apparu en 1952, par le biais des travaux de deux étudiants : Norman Joseph Woodland et Bernard Silver. Le principe est donc de scanner le code avec de la lumière pour traduire les barres verticales en informations. Les codes-barres ont changé les circuits de la grande distribution, grâce à une réduction du temps de passage en caisse, une baisse des prix pour le consommateur, l'arrivée des cartes de fidélité sur le marché, et d'autres avantages.

Quant au code QR, il est créé en 1994, par l'entreprise japonaise Denso Wave. Le contenu du code peut être déchiffré par téléphone ou webcam, et est donc plus facile à utiliser qu'un code barre. De plus, le code QR a la capacité de stocker plus d'informations que le code à barres (7089 caractères numériques contre 10 à 13 caractères pour le code barre). Il peut renvoyer divers types de contenus tels que des URL ou des numéros de téléphone. C'est pour cela qu'au cours de ce projet nous avons cherché à déchiffrer un code QR, afin de comprendre leur fonctionnement.

FIGURE 2.1 – Exemple de code QR contenant *https ://www.insa-rouen.fr/*

2.1 Description et norme du code QR

Afin de mener à bien notre projet, c'est à dire de décoder un code QR, nous devions impérativement savoir ce que nous allions devoir décoder. Notre premier réflexe a été d'aller chercher sur internet. Or, les sources n'étaient que très rarement fiable, et elles pouvaient même parfois se contredirent. Nous nous sommes alors dirigés vers le document de référence du code QR : le norme ISO/IEC 18004 de 2015 [1]. On fait ici une brève description des éléments dont nous aurons besoin.

2.1.1 Les versions du code QR

Il existe 40 versions de code QR (versions 1, 2, 3, ..., 40). Plus la version est élevée, plus le code QR peut contenir de l'information. Par exemple, un code QR de version 1 peut stocker au maximum 25 caractères tandis que la place dans un QR code de version 40 peut aller jusqu'à plus de 4 000 caractères. Lorsque l'on veut générer un code QR, il faut donc au préalable savoir quelle quantité d'information on veut stocker. La plupart du temps les codes QR contiennent une URL, nécessitant au maximum un code QR de version 4.

Dans un code QR, une petite unité noire ou blanche est appelée « module ». Pour obtenir la version d'un code QR, il suffit d'appliquer l'opération suivante sur la taille du code : $(\text{taille} - 17)/4$, où la taille correspond à la longueur d'un côté du code QR. Par exemple, un code de taille 29×29 correspond à un code QR de version $(29 - 17)/4 = 3$.

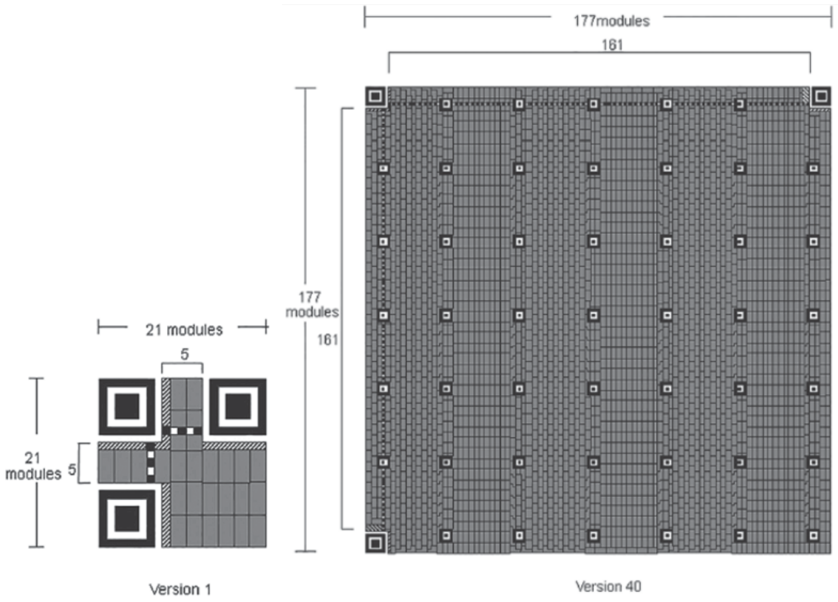


FIGURE 2.2 – Schéma de code QR version 1 et 40

2.1.2 Les différents types de données

Les codes QR supportent 4 types de données : des caractères numériques, alphanumériques, des binaires et enfin des Kanji et des Kana (caractères de l'alphabet chinois et japonais). Il est également possible de stocker deux types de données différents sur un même code QR. La capacité de stockage des codes QR peut varier selon le type de données codées. Le code QR est à l'heure actuelle l'un des seul moyen de stocker des caractères Kanji ou Kana, c'était d'ailleurs leur objectif premier lors de leur création.

2.1.3 La correction d'erreur

Un gros avantage apporté par les codes QR est qu'ils permettent de compenser une perte de données. Ainsi, un code QR partiellement déchiré ou effacé peut toujours être lu car il peut contenir jusqu'à 30% de redondance, c'est à dire que même si 30% des modules du code QR sont effacés ou illisibles, il sera toujours possible de lire son contenu. Il y a 4 niveaux de correction des erreurs :

— L (low) qui permet environ 7% de redondance
— M (medium) : 15%
— Q : 25%
— H (high) : 30%

On utilise le code Reed-Solomon pour la correction des erreurs. Ce code consiste à construire un polynôme contenant les données ainsi que d'autres paramètres tels que des symboles de contrôles permettant de reconstruire l'intégralité du code à partir de morceaux de celui-ci.

2.1.4 Les masques

Lors de la création d'un code QR, il est possible d'obtenir des formes plutôt atypiques tels que des grandes zones blanches ou encore une forme similaire aux formes utilisées pour le calibrage.

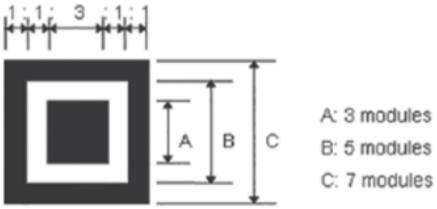


FIGURE 2.3 – Structure d'un motif de repérage

Pour pallier à ce problème, on applique des masques aux données. Appliquer un masque consiste à effectuer une opération *xor* entre les données et le masque. Il existe 8 types de masques.

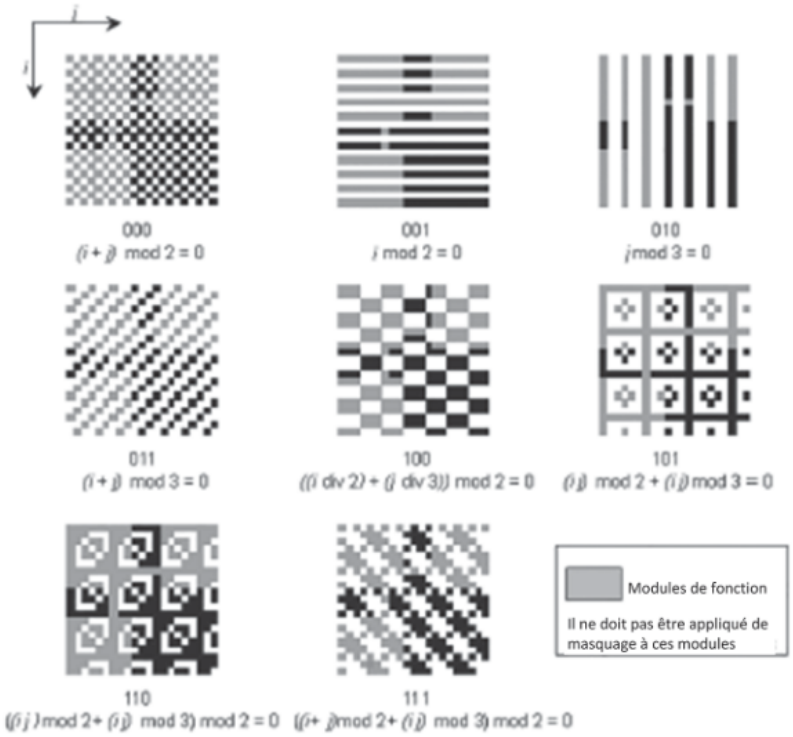


FIGURE 2.4 – Motifs de masquage de données pour la version 1

2.2 Notre projet

Pour notre projet, nous nous sommes limités à la lecture de certains codes QR. En effet, au vu de la complexité de certains algorithmes, il nous aurait été impossible de terminer le projet à temps. C'est pourquoi nous nous sommes limités au déchiffrage de codes QR de version 1, intact et sur fond blanc. Nous ne traitons pas la partie reconnaissance de code QR depuis une image réelle (une photo), ni la reconstruction d'information par la correction d'erreur. De même, nous nous sommes limités au mode *binaire*, les autres modes ne présentant pas d'intérêt supplémentaire pour le projet.

Par ailleurs, même si notre programme ne fonctionne que sur ce type de code QR, il est important de préciser que nous avons essayé de faire les algorithmes les plus généraux possibles, afin de permettre l'implémentation d'autres types de code QR.

Afin de décoder ce type de code QR, nous avons implémenté une suite d'algorithme réalisant chacun une petite tâche.

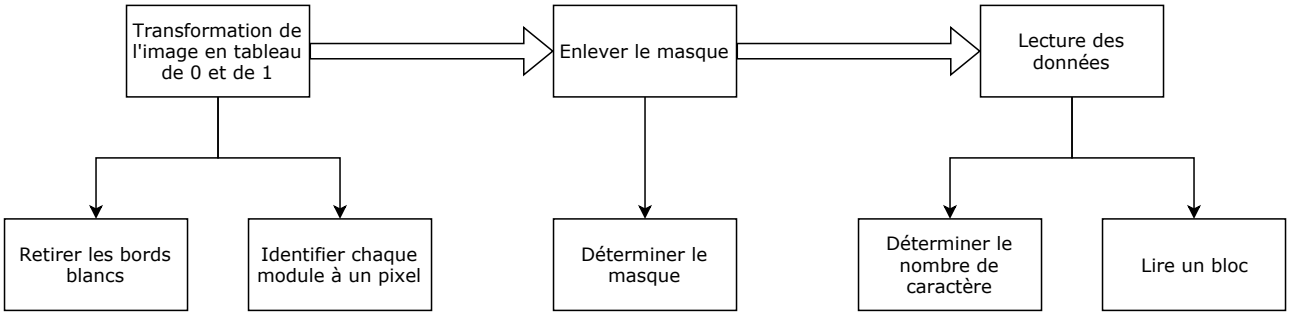


FIGURE 2.5 – Décodage d'un code QR

2.2.1 Transformation d'une image en un tableau de donnée

Un QR code se présente toujours de la même manière. Pour que nous puissions le décoder, il est impératif de transformer cette image, en retirant d'abord les bords blancs (obligatoire suivant la norme, de la taille de 2 à 4 modules), puis en transformant chaque groupe de pixel noir ou blanc (chaque module) en un unique pixel.

2.2.1.1 Retirer bords blancs (A.2.1.1)

Cette étape est extrêmement simple, on récupère simplement les coordonnées minimales et maximales des pixels noir : x_{min}, x_{max}, y_{min} et y_{max}, puis on extrait de l'image le carré ABCD de coordonnée $A\,(x_{min}; y_{min})$, $B\,(x_{max}, y_{min})$, $C\,(x_{max}, y_{max})$ et $D\,(x_{min}, y_{max})$ contenant le code QR.

2.2.1.2 Identifier chaque module à un pixel (A.2.1.2)

On souhaite maintenant réduire notre code QR à une taille de 21×21. Pour ce faire, on suppose que notre code QR est bien carré, et que chaque module représente lui même un carré. On sait que le module en haut à gauche est forcément noir, et que ce module est suivi par un module blanc sur la diagonale. On calcule alors la longueur r d'un module en regardant un à un les pixels de la diagonale. Au premier pixel blanc rencontré, on sait que la longueur de cette diagonale correspond à la longueur et à la largeur d'un module. On récupère alors le pixel en haut à gauche de chaque module que l'on stocke dans un nouveau tableau de 21×21, où chaque module correspond bien à un pixel.

On doit maintenant transformer la valeur des données. En effet, sur une image en nuance de gris, un pixel blanc correspond à une valeur de 255 et un pixel noir correspond à une valeur de 0. Sur un code QR, un pixel noir correspond à une valeur de 1 et un pixel blanc une valeur de 0. On applique alors simplement cette transformation. Il est important de noter qu'à ce stade, MATLAB n'est plus capable d'afficher correctement notre code QR, car elle ne correspond plus à un type de d'image en nuance de gris.

2.2.2 Masque

Comme nous avons pu le voir auparavant, un masque est toujours appliqué à un code QR afin d'empêcher l'apparition de motifs réservés, ou de zone monochrome trop grande. Cependant, le décodage du masque ajoute une complexité supplémentaire au programme. Il faut déterminer le numéro du masque, puis l'enlever.

2.2.2.1 Déterminer le masque (A.2.2.1)

Autour des motifs de repérage se trouvent 15 modules particuliers, représentant des informations du code QR, comme le niveau de correction d'erreur, ou le masque. À ces 15 bits est appliqué un masque constant (10101 00000 10010). Les 3 bits qui nous intéressent sont les bits 12 à 10, il faut donc appliquer un *xor* avec 101. Ces pixels sont redondants (au moins à deux endroits dans le code QR), mais comme on considère que notre code QR est complet, on peut les lire toujours au même endroit, c'est à dire sur les pixels ((3, 9), (4, 9), (5, 9)).

2.2.2.2 Enlever le masque (A.2.2.2)

Maintenant que l'on possède le nombre binaire représentant le masque, on peut calculer le masque et l'appliquer à tout notre code QR. Pour chaque pixel de coordonnée (i, j), si la formule est vérifiée le pixel est noir. Sinon, il est blanc.

#Masque	Formule en (i, j)	#Masque	Formule en (i, j)
000 (0)	$(i + j) \bmod 2 = 0$	100 (4)	$((i \text{ div } 2) + (j \text{ div } 3)) \bmod 2 = 0$
001 (1)	$i \bmod 2 = 0$	101 (5)	$(ij) \bmod 2 + (ij) \bmod 3 = 0$
010 (2)	$j \bmod 3 = 0$	110 (6)	$((ij) \bmod 2 + (ij) \bmod 3) \bmod 2 = 0$
011 (3)	$(i + j) \bmod 3 = 0$	111 (7)	$((i + j) \bmod 2 + (ij) \bmod 3) \bmod 2 = 0$

TABLE 2.1 – Formule des masque du code QR

Un point reste cependant important à prendre en compte. Le masque ne doit pas être appliqué sur les zones réservées. Pour pallier à ce problème, nous avons généré les 8 masques possibles et remplacé les pixels des zones réservées par l'élément nul du *xor*, c'est à dire le 0. Nous stockons alors ces masques sous forme d'image que nous chargeons lorsque l'on doit retirer un masque.

2.2.3 Lecture des données

Maintenant que nos données sont lisibles, il faut les décoder. Tout d'abord, il faut lire l'un des 4 modes possibles. Celui-ci est représenté par les 4 modules en bas à droite. C'est le module binaire qui nous intéresse, on vérifie donc que l'on trouve bien le nombre binaire 0100 (A.2.3.2).

2.2.3.1 Lecture d'un bloc

Les données d'un code QR sont réparties en « blocs » de 8 modules. Chaque bloc sera interprété comme un entier sur 8 bits. Il existe un ordre entre les modules de chacun des blocs : chaque module correspond à une puissance de 2 bien précise. Si le module est noir, alors sa valeur est comptée, alors que s'il est blanc, on le compte comme 0. De ce fait, chaque bloc peut être interprété comme un nombre binaire formé sur 8 bits.

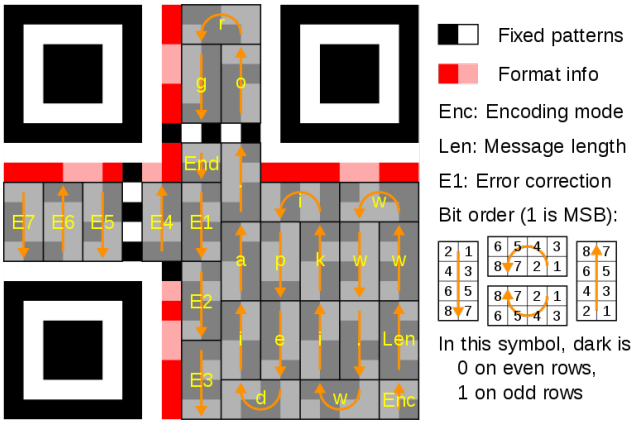


FIGURE 2.6 – Ordre de lecture d'un code QR version 1

La somme dans un bloc est donc un entier entre 0 et 255. Pour le premier bloc, ce nombre va représenter la longueur de la chaîne de caractère totale (A.2.3.3). Les blocs suivants sont associés à un caractère correspondant à la table ASCII (B.1). Ainsi, lire un bloc revient à calculer une somme, puis transformer le nombre obtenu en caractère ().

2.2.3.2 Lecture de la chaîne de caractère (A.2.3.1)

Pour former la chaîne de caractère, on lit les blocs un par un en suivant un motif variant selon la version du code QR (on peut voir la division en bloc sur la figure 2.2). Afin de lire les blocs dans l'ordre, nous avons eu recours à un masque. Ce masque est en réalité une image en nuance de gris où chaque bloc possède une valeur distincte croissante, correspondant à l'ordre de lecture des blocs. On peut alors utiliser la fonction *find* de MATLAB pour lire les blocs un par un dans le bon ordre.

Pour ce qui est de l'orientation de chaque bloc, nous avons remarqué qu'un changement dans la taille du bloc par rapport au bloc précédent correspond à un changement de sens. Ces changements de sens s'effectue toujours dans le même ordre. Pour l'algorithme, nous avons donc seulement détecté ce changement de sens et incrémenté le sens de lecture dans un bloc.

Chapitre 3

L'interface graphique MATLAB

3.1 La conception

Afin de rendre l'utilisation des fonctions de traitement d'image plus facile, nous avons choisi de créer une interface graphique interactive MATLAB, navigable à la souris ou au clavier. Celle-ci est basée sur GUIDE, qui est l'outil de développement d'interfaces utilisateurs graphiques inclus avec MATLAB. Il offre l'avantage de pouvoir facilement intégrer des fonctions MATLAB.

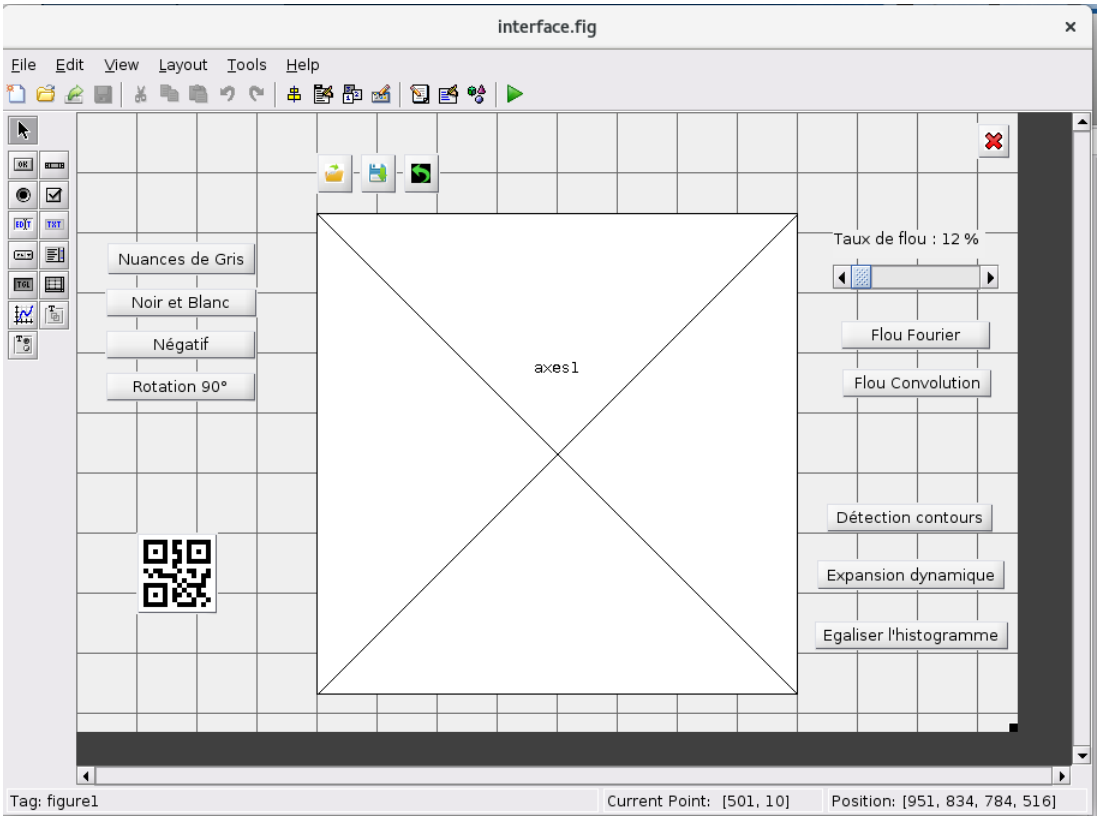


FIGURE 3.1 – L'outil GUIDE

Pour modifier l'apparence d'un élément, on utilise l'Inspecteur de Propriétés de GUIDE, qui permet comme son nom l'indique de modifier les propriétés graphiques d'un élément, tels que le texte, l'icône associée si celle-ci est présente, ou encore l'infobulle qui apparaît lorsqu'on passe la souris dessus.

GUIDE génère également automatiquement une base de code lorsqu'on ajoute un élément (boutons, curseurs...) sur l'éditeur graphique, qu'il faut ensuite compléter manuellement pour lui ajouter les fonctionnalités désirées. Les variables ne sont par défaut pas sauvegardées entre ces fonctions, c'est à dire qu'on ne peut à priori pas accéder à l'image modifiée par une fonction dans une autre, il faut donc explicitement dire à MATLAB d'enregistrer et de charger ces variables pour ne pas perdre d'informations entre les fonctions. Cela s'effectue

grâce à la fonction guiData qui prend en arguments l'élément d'interface appelant la fonction, usuellement nommé hObject, ainsi que la structure de données dans laquelle on va enregistrer ou charger les variables, qui dans notre cas se nomme handles.

Ainsi, une fonction d'interface typique va appeler une première fois guiData pour charger les variables partagées (notamment l'image actuelle), enregistrer cette image dans l'historique des modifications pour pouvoir éventuellement revenir en arrière, puis appeler les fonctions de traitement nécessaires sur l'image ayant été récupérée de handles, appeler la fonction afficheImage pour actualiser l'affichage et enfin réenregistrer l'image modifiée dans handles pour pouvoir lui apporter des modifications supplémentaires. Par exemple, le code associé au bouton Noir et Blanc est le suivant :

.8

```
function pushbutton3_Callback(hObject, eventdata, handles)
    % hObject handle to pushbutton3 (see GCBO)
    % eventdata reserved - to be defined in a future version of MATLAB
    % handles structure with handles and user data (see GUIDATA) guidata(hObject, handles);
        ajouterImageListe(hObject, handles, handles.I);
        handles.I = toBlackAndWhite(handles.I);
        afficheImage(handles);
        guidata(hObject, handles);
end
```

On notera le nom de la fonction ainsi que les commentaires générés automatiquement par MATLAB, contrairement au corps de la fonction qui a entièrement été conçu manuellement. On remarquera que l'image modifiée par la fonction de Noir et Blanc est directement stockée dans la structure partagée handles, et qu'il est nécessaire de rappeler la fonction d'affichage (dont le code se situe ci-dessous) pour pouvoir afficher les changements effectués par l'utilisateur.

.8

```
function afficheImage(handles)
    axes(handles.axes1); %Indique qu'il faut effectuer le imshow sur l'axe 1
    imshow(handles.I);
    drawnow(); %Actualise l'interface
    end
```

Pour implémenter la fonction de "retour arrière", présente de nos jours sur presque tous les logiciels permettant de modifier un fichier, il a fallu créer nous même un tableau dynamique, car en effet contrairement à la grande majorité des langages de programmation, MATLAB ne propose pas de listes à taille variable par défaut. Nous avons donc créé une classe nommée dynamicArray, comportant un tableau classique MATLAB d'une taille très élevée définie à l'avance (ici 1000), ainsi qu'une variable stockant la taille réelle actuellement utilisée de ce tableau. Elle est dotée de méthodes permettant d'ajouter, d'accéder et de supprimer des éléments de ce tableau en fonction de cette variable, ainsi il suffit par exemple pour ajouter un élément au tableau d'appeler monTableauDynamique.Add(unElement) et la méthode Add se charge d'insérer l'élément à la bonne position et de modifier la variable taille en conséquence. Le code de cette classe est le suivant :

```
classdef dynamicArray < handle
        properties (Constant)
                MAX_SIZE = 1000
        end
        properties
                data
                size
        end
        methods
                function obj = DynamicArray(x)
                        obj.data = cell(1000)
                        obj.size = 0;
                        disp(obj.size)
                end
                function Init(obj)
                        obj.data = cell(1000);
```

```
                bj.size = 0;
            end
            function Add(obj, toAdd)
                ind = obj.size + 1;
        obj.data{ind} = toAdd;
                obj.size = obj.size + 1;
            end
            function out = Get(obj, pos)
                out = obj.data{pos};
            end
            function Remove(obj, pos)
        obj.data(pos) = [];
                obj.size = obj.size - 1;
            end
            function RemoveLast(obj)
                obj.Remove(obj.size);
            end
            function out = GetLast(obj)
                out = obj.Get(obj.size);
            end
            function out = Pop(obj)
                out = obj.GetLast();
        obj.RemoveLast();
             end
    end
end
```

Le “retour arrière” se contente donc de récupérer l’image à la dernière position du tableau, de remplacer l’image actuelle dans handles par cette image, et de supprimer la dernière image de ce tableau.

3.2 Le résultat

Voici le résultat après l’utilisation de la fonction noir et blanc :

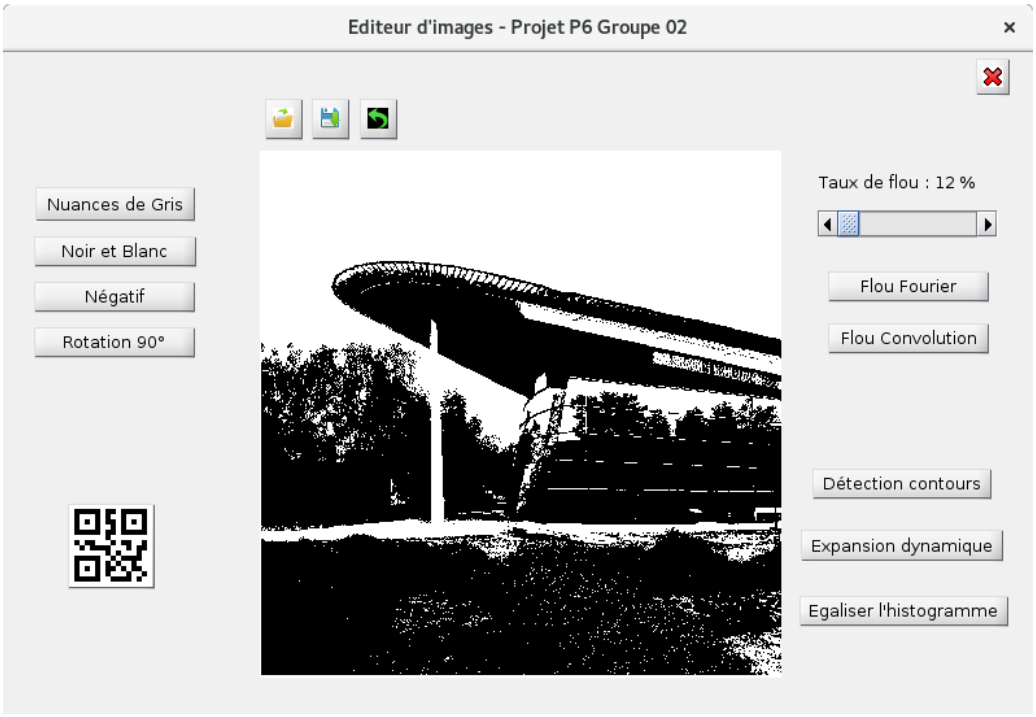


FIGURE 3.2 – Notre interface graphique MATLAB

Conclusion et perspectives

Pour terminer, nous aimerions tirer des conclusions du travail que nous avons réalisé. Nous avons produit des fonctions variées. Certaines concernent les couleurs de l'image ou sa qualité, d'autres la position des pixels et d'autres encore son format. Nous avons donc appris à modifier différents aspects d'une même image. Nous avons également appris à décoder un code QR de version 1 et implémenté du code pour automatiser ce décodage. Ce code, même s'il n'est fonctionnel que sur un certain type de codes QR, a été conçu pour que ses différentes fonctions soient générales et adaptables. Enfin, afin de mettre en exergue l'ensemble de nos réalisations et de pouvoir les présenter, nous avons créé notre propre interface graphique et dynamique. C'est une interface épurée, facile d'utilisation et intuitive. Son défaut, cependant, est de ne pas être portable.

Sur le plan personnel nous avons découvert l'univers très vaste du traitement d'images et nous avons ainsi pu entrevoir toutes les possibilités et la diversité des choses qu'il permet de faire. Nous avons pu toucher du doigt quelque chose de très utilisé aujourd'hui. De plus, nous avons découvert un nouvel outil mathématique et informatique, ainsi qu'un nouveau langage de programmation, et renforcé nos compétences en codage.

Au-delà de l'aspect intellectuel, l'aspect humain est aussi important dans ce projet. Il constitue une de nos premières expériences de gestion de projet, gestion du temps et d'équipe qui sont des choses fondamentales pour un futur ingénieur.

Nous sommes satisfaits de notre projet, néanmoins nous avons conscience de ses limites et des perspectives pour sa poursuite. Nous pourrions ainsi prévoir d'étendre l'ensemble de nos fonctions aux images en couleur, de rendre notre interface graphique portable, d'implémenter le déchiffrage de codes QR plus complexes et d'approfondir l'utilisation des filtres.

Bibliographie

[1] Nf *ISO/IEC* 18004 : Technologies de l'information - technologie d'identification automatique et de capture des données - spécification de la symbologie de code à barres code qr, 2015. 2.1

[2] Arnaud Calmettes. Introduction à la vision par ordinateur. https ://openclassrooms.com/fr/courses/1490316-introduction-a-la-vision-par-ordinateur/1491061-etirement-et-egalisation-dhistogramme. (document), 1.2.4

[3] Frédéric Legrand. Scilab : Filtrage d'une image par convolution. http ://www.f-legrand.fr/scidoc/docimg/image/filtrage/convolution/convolution.html. 1.2.6.1

[4] Telecom Paritech. La transformée de fourier. http ://www.tsi.telecom-paristech.fr/pages/enseignement/ressources/beti/ondelettes-2g/francais/Fourier/TF2Dimage.htm. 1.2.6.3

Annexe A

Code

A.1 Fonctions traitement d'image

A.1.1 Fonctions élémentaires

A.1.1.1 doubleToUint8

```
function [output] = doubleToUint8(input)
% cette fonction prend en entrée une image et renvoie la même image en format uint8
        output = uint8(floor(255*input));
end

function [output] = uint8ToDouble(input)
% cette fonction prend en entrée une image et renvoie la même image en format double
        output = double(input) / 255;
end
```

A.1.1.2 uint8ToDouble

```
function [output] = uint8ToDouble(input)
    output = double(input) / 255;
end
```

A.1.1.3 pivot180

```
function [out] = pivot180(in)
        dim = size(in);
        out = zeros(dim(1), dim(2), 'uint8');
        for i = 1 : dim(1)
                for j = 1 : dim(2)
                        out(dim(1)+1-i, dim(2)+1-j) = in(i, j);
                end
        end
end
```

A.1.1.4 pivot90

```
function [out] = pivot90(in)
    dim = size(in);
    dim2 = size(dim)
    if (dim2(2) == 3) % Cas ou l'image est en couleur
        out = zeros(dim(2), dim(1), dim(3),'uint8');
        for z = 1 : dim(3)
            for i = 1 : dim(1)
                for j = 1 : dim(2)
                    out(dim(2) - j+1, i, z) = in(i, j, z);
```

```
                end
            end
        end
    else
        out = zeros(dim(2), dim(1),'uint8');
        for i = 1 : dim(1)
                for j = 1 : dim(2)
                    out(dim(2) - j+1, i) = in(i, j);
                end
        end
    end
end
```

A.1.1.5 nuancesDeGris

```
function imb = nuancesDeGris(im)
    sizeim = size(im)
    size2 = size(sizeim)
    if (size2(2) == 3) %Si l'image est en couleurs
        imb = ((im(:,:,1)*0.5 + im(:,:,2)*0.25 + im(:,:,3)*0.25) / 3);
    else
        imb = im
    end;
end
```

A.1.1.6 toBlackAndWhite

Fonction pour mettre une image en nuances de gris en noir et blanc.

```
function [outNB] = toBlackAndWhite(inGS)
        dim = size(inGS);     outNB = zeros(dim(1), dim(2), 'uint8');
        m = median(median(inGS));  %médiane des valeurs
        for i = 1 : dim(1)
                for j = 1 : dim(2)
                        if inGS(i, j) < m+20  % seuil = médiane + 20
                                outNB(i, j) = 0;
                        else
                                outNB(i, j) = 255;
                        end
                end
        end
end
```

A.1.1.7 negative

```
function [out] = negative(in)
    out = 255 - in;
end
```

A.1.2 Modification de l'image

A.1.2.1 expansionDynamique

```
function [output] = expansionDynamique(input)
    image = uint8ToDouble(input);
    M = max(max(image));
    m = min(min(image));
    dim = size(image);
    temp = zeros(dim(1),dim(2));
        for j = 1:dim(2)
```

```
            for i = 1:dim(1)
                temp(i,j) = (image(i,j)-m)/(M-m);
            end
        end
    output = doubleToUint8(temp);
end
```

A.1.2.2 histogrammeCumule2

```
function [output] = histogrammeCumule2(input)
        hist = histogramme(input);  % nous avons recodé la fonction histogramme
        output = zeros(256,1);
        temp = 0;
        for i=1:256
                temp = temp + hist(i,1);
                output(i,1) = temp;
        end
        %stem(output) si on veut afficher l'histogramme cumulé
end

function [output] = egalisation(input)
        histc = histogrammeCumule2(input);
        pourcentage = histc ./ max(max(histc));
        dim = size(input);
        temp = zeros(dim(1),dim(2),'uint8');
        for i = 1:dim(1)
                for j = 1:dim(2)
                        temp(i,j) = pourcentage(input(i,j)+1) * 255;
                end
        end
        output = temp;
end
```

A.1.2.3 gaussian

Calcul de la Gaussienne pour le flou gaussien.

```
function [output] = gaussian (n,sigma)
        % n = taille d'un carré
        % sigma = variation standard ou écart type de la loi normale
        sigma_sqrt = 2 * sigma * sigma;
        temp = zeros(n); % temp = matrice de n par n
        for i = 1:n
                for j = 1:n
                        x = i - n/2; % x - x0
                        y = j - n/2; % y - y0
                        temp(i,j) = exp(-(x*x + y*y)/sigma_sqrt);
             end
        end
        s = sum(sum(temp));
        temp = temp ./ s;
        output = temp;
end
```

A.1.2.4 floue

Produit de convolution entre l'image et le masque gaussien.

```
function [output] = floue(input, taux)
        H = gaussian(10,2000);
```

```
        output = conv2(input,H/(10000*(1 - taux)));
        % convolution 2D de l'image et de la fonction gaussienne
end
```

A.1.2.5 blurFourier

Code du flou par transformée de Fourier.

```
function [output] = blurFourier(input, r)
        dim = size(input);
    % on créé un filtre de la taille de la photo
        mask = zeros(dim);
    % on met des 1 dans un carré centre au milieu du filtre
        rX = round(dim(1)/r);
        rY = round(dim(2)/r);
        xCenter = round(dim(1)/2);
        yCenter = round(dim(2)/2);
        mask(xCenter - rX : xCenter + rX, yCenter - rY : yCenter + rY) = 1;
    % traitement de la photo
    output = abs(ifft2(fftshift(fft2(input)).*mask)); end
```

A.2 Code QR

On utilise cette fonction pour décoder un code QR.

```
function text = lire_qr_code(qr_image)
% fonction prenant une image de qr code de type "byte" version 1 et renvoie
% le texte décodé.
    qr = retirer_bord_blanc(qr_image);
    qr = reduire(qr);
    masque = determiner_masque(qr);
    qr = enleverMask(qr, masque);
    text = decode(qr);
end
```

A.2.1 Transformation de l'image en tableau

A.2.1.1 retirer_bord_blanc

```
function qr_code_sans_blanc = retirer_bord_blanc(qr_code)
% Retire les bords blancs autour d'un QR code.
% La matrice qr_code est composé de 0 et de 1 :
%   Un 0 représente un pixel blanc
%   Un 1 représente un pixel noir
    [x, y] = find(qr_code == 1);
% les coordonnées des pixels noirs

% les coordonnées du coin haut gauche du motif de repérage
    xmin = min(x);
    ymin = min(y);

% les coordonnées du coin bas droite du qr code
    xmax = max(x);
    ymax = max(y);

    qr_code_sans_blanc = qr_code(xmin : xmax, ymin : ymax);
end
```

A.2.1.2 reduire

```
function qr_code_reduit = reduire(qr_code)
% Reduit le qr code de manière à ce que un module corresponde à un pixel
% La matrice qr_code est composé de 0 et de 1 :
%   Un 0 représente un pixel blanc
%   Un 1 représente un pixel noir
% Le pixel en haut à gauche correspond au premier pixel du qr code
    dim_grand_qr_code = size(qr_code);
         % On cherche la taille en pixel du premier module
    % r va représenter la taille de ce module
    r = 1;
    while qr_code(r, r) == 1
        r = r + 1;
    end
    r = r - 1;
         % on créer une nouvelle image où un pixel correspond à un module
    qr_code_reduit = zeros(floor(dim_grand_qr_code/r));
         % on remplit alors l'image avec les modules du grand qr code
    for i = 1 : (floor(dim_grand_qr_code(1) / r))
        for j = 1 : (floor(dim_grand_qr_code(2) / r))
            qr_code_reduit(i,j) = qr_code(i * r, j * r);
        end
    end
end
```

A.2.2 Gestion du masque

A.2.2.1 determiner_masque

```
function valeur_masque = determiner_masque(qr_code)
% Renvoie la valeur de masque du qr_code dont un module correspond à un % pixel.
    % on récupère la valeur du masque sur le qr_code et on applique un xor
    % avec la matrice [1, 0, 1] pour trouver la vrai valeure du masque
    masque = xor(qr_code(9, 3 : 5), [1, 0, 1]);
         % puis, on calcule la valeur du masque
    valeur_masque = masque(1, 1) * 4 + masque(1, 2) * 2 + masque(1, 3);
end
```

A.2.2.2 enlever_masque

```
function out = enelverMask(img, nMask)
% prend en entrée le qr code ainsi que numéro de masque associé pour
% renvoyer l'image sans le masque. Pour ce faire, une image contenant
% exactement le masque préétablie.
        switch nMask
                case 0
                        mask = imread('res/mask000.bmp')/255;
                case 1
                        mask = imread('res/mask001.bmp')/255;
                case 2
                        mask = imread('res/mask010.bmp')/255;
                case 3
                        mask = imread('res/mask011.bmp')/255;
                case 4
                        mask = imread('res/mask100.bmp')/255;
                case 5
                        mask = imread('res/mask101.bmp')/255;
                case 6
                        mask = imread('res/mask110.bmp')/255;
```

```
                case 7
                        mask = imread('res/mask111.bmp')/255;
        end
        out = xor(img, mask);
end
```

A.2.3 Lecture de la chaîne de caractère

A.2.3.1 decode

```
function txt = decode(qrImg)
% prend en entrée un qr code sans le masque, vérifie que le mode est bien
% "byte", et décode le qr code
        % contient la position des différents blocks
        ordre = rgb2gray(imread('res/ordreBlock.png'));
    % on cherche le mode du qrcode et on vérifie que c'est bien le mode
    % byte
    [x, y] = find(ordre == 0);
    a = estModeByte(qrImg, x, y);
        if a == 1 % on est dans le bon mode
        % on commence par trouver la longueur du texte
        [x, y] = find(ordre == 1);
        lon = trouverLongueur(qrImg(unique(x), unique(y)));
        sens = 1;
        txt = '';
                % pour savoir quand il faut changer de sens de lecture, nous allons
        % comparer la taille du block précédent avec la taille du block
        % actuel. On garde alors toujours une copie du block précédent
        blockPrecedent = zeros(4, 2);
        for i = 1 : lon
            % on commence par trouver le bon block
            [x, y] = find(ordre == i + 1);
            block = qrImg(unique(x), unique(y));
            % on vérfie si on doit changer le sens de lecture
            sens = trouverSens(block, blockPrecedent, sens);
            % on décode le block et on ajoute l'élément au texte final
            txt = strcat(txt, blockToLettre(block, sens));
            % le block actuel devient le block précédent
            blockPrecedent = block;
        end
    else % cas où le mode n'est pas "byte",
          % on affiche seulement un message d'erreur
        "Erreur, le qr code n'est pas dans le bon mode"
    end
end
```

A.2.3.2 estModeByte

```
function estByte = estModeByte(qrImg, x, y)
% renvoie vrai si le mode est bien "byte", c'est à dire 0100
        estByte = 1;
        if (qrImg(x(1), y(1)) ~= 0) || (qrImg(x(2), y(2)) ~= 1) || (qrImg(x(3), y(3)) ~= 0)
        ↪  || (qrImg(x(4), y(4)) ~= 0)
                estByte = 0;
        end
end
```

A.2.3.3 trouverLongueur

```
function lon = trouverLongueur(block)
% renvoie le nombre de caractères dans le qr code.
     lon = block(1,1) +...
          block(2,1)*4 +...
          block(3,1)*16 +...
          block(4,1)*64 +...
          block(1,2)*2 +...
          block(2,2)*8 +...
          block(3,2)*32 +...
          block(4,2)*128;
end
```

A.2.3.4 trouverSens

```
function sens = trouverSens(block, blockPrecedent,sens)
% on compare la taille de l'ancien block et du block actuel. Si ces tailles
% sont différentes, on sait que l'on doit changer le sens de lecture.
% 1 : de bas à haut
% 2 : de droite à gauche en haut
% 3 : de haut à bas
% 4 : de droite à gauche en bas
        if size(block) ~= size(blockPrecedent)
        sens = sens + 1;
        if sens > 4
            sens = 1;
        end
    end
end
```

A.2.3.5 blockToLettre

```
function lettre = blockToLettre(block,blockType)
% Transforme un block en une lettre de la table ASCII
% Il y trois types de block sur un Qr-Code et chaque type se lit de manière
% différente
    array = zeros(1,8);
    ascii = 0;
         switch blockType
                     case 1        % de bas à haut
                        array(1) = block(4,2)*128;
                        array(2) = block(4,1)*64;
                        array(3) = block(3,2)*32;
                        array(4) = block(3,1)*16;
                        array(5) = block(2,2)*8;
                        array(6) = block(2,1)*4;
                        array(7) = block(1,2)*2;
                        array(8) = block(1,1);
                case 2        % droit hauit gauche
                        array(1) = block(2,4)*128;
                        array(3) = block(1,4)*32;
                        array(2) = block(2,3)*64;
                        array(4) = block(1,3)*16;
                        array(7) = block(2,2)*2;
                        array(5) = block(1,2)*8;
                        array(8) = block(2,1);
                        array(6) = block(1,1)*4;
                case 3        % de haut en bas
```

```
                    array(7) = block(4,2)*2;
                    array(8) = block(4,1);
                    array(5) = block(3,2)*8;
                    array(6) = block(3,1)*4;
                    array(3) = block(2,2)*32;
                    array(4) = block(2,1)*16;
                    array(1) = block(1,2)*128;
                    array(2) = block(1,1)*64;
                case 4          % doit bas gauche
                    array(3) = block(2,4)*32;
                    array(1) = block(1,4)*128;
                    array(4) = block(2,3)*16;
                    array(2) = block(1,3)*64;
                    array(5) = block(2,2)*8;
                    array(7) = block(1,2)*2;
                    array(6) = block(2,1)*4;
                    array(8) = block(1,1);
    end
    % On obtient un entier qui correspond à une lettre dans la table ASCII
    for i = 1 : 8
        ascii = ascii + array(i);
    end
   % On transfome cet entier en une lettre
    lettre = char(ascii);
end
```

Annexe B

Autre

B.1 Table ASCII

La norme ASCII est l'une des première norme utilisé afin d'associer à un entier un caractère. Elle est aujourd'hui remplacée par la norme Unicode. Cette table ASCII est cependant toujours utilisée quand on veut encoder un message très simple, comme dans un code QR par exemple

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	*[NULL]*	48	30	110000	60	**0**	96	60	1100000	140	`
1	1	1	1	*[START OF HEADING]*	49	31	110001	61	**1**	97	61	1100001	141	**a**
2	2	10	2	*[START OF TEXT]*	50	32	110010	62	**2**	98	62	1100010	142	**b**
3	3	11	3	*[END OF TEXT]*	51	33	110011	63	**3**	99	63	1100011	143	**c**
4	4	100	4	*[END OF TRANSMISSION]*	52	34	110100	64	**4**	100	64	1100100	144	**d**
5	5	101	5	*[ENQUIRY]*	53	35	110101	65	**5**	101	65	1100101	145	**e**
6	6	110	6	*[ACKNOWLEDGE]*	54	36	110110	66	**6**	102	66	1100110	146	**f**
7	7	111	7	*[BELL]*	55	37	110111	67	**7**	103	67	1100111	147	**g**
8	8	1000	10	*[BACKSPACE]*	56	38	111000	70	**8**	104	68	1101000	150	**h**
9	9	1001	11	*[HORIZONTAL TAB]*	57	39	111001	71	**9**	105	69	1101001	151	**i**
10	A	1010	12	*[LINE FEED]*	58	3A	111010	72	**:**	106	6A	1101010	152	**j**
11	B	1011	13	*[VERTICAL TAB]*	59	3B	111011	73	**;**	107	6B	1101011	153	**k**
12	C	1100	14	*[FORM FEED]*	60	3C	111100	74	**<**	108	6C	1101100	154	**l**
13	D	1101	15	*[CARRIAGE RETURN]*	61	3D	111101	75	**=**	109	6D	1101101	155	**m**
14	E	1110	16	*[SHIFT OUT]*	62	3E	111110	76	**>**	110	6E	1101110	156	**n**
15	F	1111	17	*[SHIFT IN]*	63	3F	111111	77	**?**	111	6F	1101111	157	**o**
16	10	10000	20	*[DATA LINK ESCAPE]*	64	40	1000000	100	**@**	112	70	1110000	160	**p**
17	11	10001	21	*[DEVICE CONTROL 1]*	65	41	1000001	101	**A**	113	71	1110001	161	**q**
18	12	10010	22	*[DEVICE CONTROL 2]*	66	42	1000010	102	**B**	114	72	1110010	162	**r**
19	13	10011	23	*[DEVICE CONTROL 3]*	67	43	1000011	103	**C**	115	73	1110011	163	**s**
20	14	10100	24	*[DEVICE CONTROL 4]*	68	44	1000100	104	**D**	116	74	1110100	164	**t**
21	15	10101	25	*[NEGATIVE ACKNOWLEDGE]*	69	45	1000101	105	**E**	117	75	1110101	165	**u**
22	16	10110	26	*[SYNCHRONOUS IDLE]*	70	46	1000110	106	**F**	118	76	1110110	166	**v**
23	17	10111	27	*[ENG OF TRANS. BLOCK]*	71	47	1000111	107	**G**	119	77	1110111	167	**w**
24	18	11000	30	*[CANCEL]*	72	48	1001000	110	**H**	120	78	1111000	170	**x**
25	19	11001	31	*[END OF MEDIUM]*	73	49	1001001	111	**I**	121	79	1111001	171	**y**
26	1A	11010	32	*[SUBSTITUTE]*	74	4A	1001010	112	**J**	122	7A	1111010	172	**z**
27	1B	11011	33	*[ESCAPE]*	75	4B	1001011	113	**K**	123	7B	1111011	173	**{**
28	1C	11100	34	*[FILE SEPARATOR]*	76	4C	1001100	114	**L**	124	7C	1111100	174	**|**
29	1D	11101	35	*[GROUP SEPARATOR]*	77	4D	1001101	115	**M**	125	7D	1111101	175	**}**
30	1E	11110	36	*[RECORD SEPARATOR]*	78	4E	1001110	116	**N**	126	7E	1111110	176	**~**
31	1F	11111	37	*[UNIT SEPARATOR]*	79	4F	1001111	117	**O**	127	7F	1111111	177	*[DEL]*
32	20	100000	40	*[SPACE]*	80	50	1010000	120	**P**					
33	21	100001	41	**!**	81	51	1010001	121	**Q**					
34	22	100010	42	**"**	82	52	1010010	122	**R**					
35	23	100011	43	**#**	83	53	1010011	123	**S**					
36	24	100100	44	**$**	84	54	1010100	124	**T**					
37	25	100101	45	**%**	85	55	1010101	125	**U**					
38	26	100110	46	**&**	86	56	1010110	126	**V**					
39	27	100111	47	**'**	87	57	1010111	127	**W**					
40	28	101000	50	**(**	88	58	1011000	130	**X**					
41	29	101001	51	**)**	89	59	1011001	131	**Y**					
42	2A	101010	52	*****	90	5A	1011010	132	**Z**					
43	2B	101011	53	**+**	91	5B	1011011	133	**[**					
44	2C	101100	54	**,**	92	5C	1011100	134	****					
45	2D	101101	55	**-**	93	5D	1011101	135	**]**					
46	2E	101110	56	**.**	94	5E	1011110	136	**^**					
47	2F	101111	57	**/**	95	5F	1011111	137	**_**					

FIGURE B.1 – Table ASCII