

# IDIAP

Martigny - Valais - Suisse



## HANDWRITTEN DIGITS RECOGNITION

Eric Grand

IDIAP-RR 00-07

May 2000

Dalle Molle Institute  
for Perceptual Artificial  
Intelligence • P.O.Box 592 •  
Martigny • Valais • Switzerland

phone +41 - 27 - 721 77 11  
fax +41 - 27 - 721 77 12  
email [secretariat@idiap.ch](mailto:secretariat@idiap.ch)  
internet <http://www.idiap.ch>

---

# *Table des matières*

---

	<b>Préambule</b>	<b>v</b>
	Objectif	v
	Informations sur le rapport	vi
	Remerciements	vi
<b>SECTION I</b>	<b>TRAITEMENT DE L'IMAGE</b>	
	<b>Introduction au traitement de l'image</b>	<b>2</b>
	Utilité	2
	Le filtrage	4
	Introduction	4
	Le filtrage linéaire	4
	Optimisation pour la rapidité de calcul	6
	<b>Segmentation</b>	<b>7</b>
	Méthode	7
	Implémentation	8
	Histogramme	9
	Fonctions C++	10
	<b>Prétraitements</b>	<b>11</b>
	Flou	11
	Filtre de Gauss	11
	Suppression des taches	13
	Algorithme	13
	Correction de l'inclinaison des caractères	14
	Définitions	14
	Principe de l'algorithme	15

Centrage	16
Algorithme	17
Changer la taille de l'image	17
Normalisation de la taille des images	19
Méthode	19
Manipulation de l'image	19
copier	20
décaler	20
découper	20
Découper au minimum	21
Fonctions C++	21

**SECTION II****RECONNAISSANCE**

<b>Reconnaisseur SVM</b>	<b>23</b>
Introduction	23
Entraînement	24
Principe général	24
Classificateur	25
Algorithme	26

**SECTION III****DÉMONSTRATEUR**

<b>Modélisation UML</b>	<b>28</b>
Introduction	28
Diagramme des cas d'utilisation	28
Définition des acteurs	29
Définition des cas d'utilisation	29
Représentation graphique	29
Diagrammes de séquences	30
Représentation graphique	31
Diagramme de classes	35
<b>Démonstrateur</b>	<b>37</b>
Environnement	37
Installation	37
Lancement de l'application	37
Mode d'emploi	38
Mode en ligne	38
Mode image par image	39

## SECTION IV

## EXPÉRIENCES

**Tests** 42

Constitution d'une base de données 42

Tests 43

Résultats 44

**Formats de fichiers** 46

"Big-endian" et "little-endian" 46

Avantages et inconvénients 46

Fichiers Images 47

MNIST 47

PBM 48

SVM 49

Fichiers labels 50

MNIST 50

**Conclusions** 51

Améliorations possibles 51

Bug MFC 51

## ANNEXES

**A. Codes sources** 55

*Ce travail de diplôme s'est effectué au sein du groupe VISION de l'IDIAP<sup>1</sup> en collaboration avec un autre diplômé de l'Ecole d'Ingénieurs du Valais: M. Philippe Gillioz.*

## Objectif

L'objectif est de développer un logiciel de démonstration capable de reconnaître l'écriture manuscrite.

Ce logiciel sera écrit en C++ et fonctionnera sur un système d'exploitation Windows 95 ou Windows NT. Il sera capable de charger des images de provenances diverses, telles qu'un fichier ou d'un stylo scanner via le port série du PC, de les traiter et d'en effectuer la reconnaissance de caractères. Nous allons travailler uniquement avec des codes postaux, ce qui limitera la reconnaissance à des chiffres.

Ce travail fait partie d'un projet global, comme l'illustre la figure 1-1.

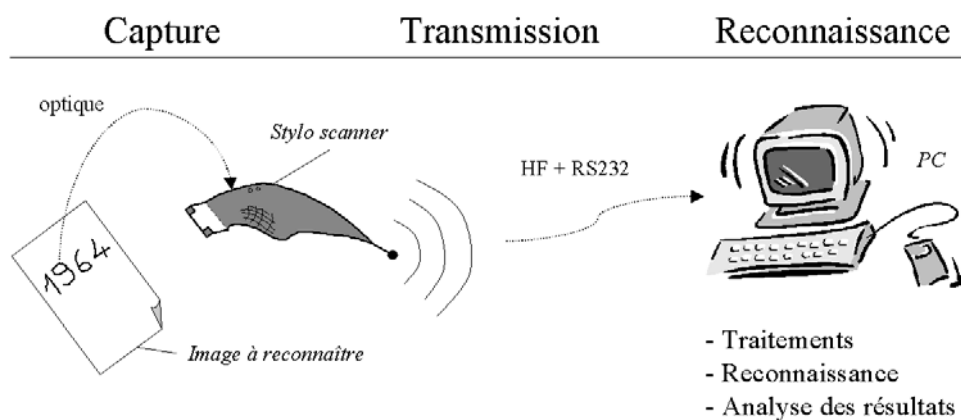
D'une part, M. Philippe Gillioz effectuera l'acquisition de données, en l'occurrence des images, par la conception et la réalisation d'un stylo scanner. Il s'occupera aussi de l'intégralité de la transmission, du stylo scanner vers un PC, qui s'effectuera sans fil, par voie herzienne et de manière asynchrone. Une petite interface convertira le signal HF en un signal électrique RS232. Il devra aussi écrire le code C++ recevant les données sur le port série du PC.

D'autre part interviendra le démonstrateur qui fait partie intégrante de mon travail de diplôme.

---

1. Institut Dalle Molle d'Intelligence Artificielle Perceptive <http://www.idiap.ch>

**FIGURE 1-1**  
Globalité du projet



## Informations sur le rapport

Ce rapport est divisé en quatre grandes parties. La première partie se consacre au **traitement de l'image**, la seconde expose les principes généraux de la **reconnaissance** et plus précisément ceux du reconnaisseur SVM. La troisième partie présente la modélisation du **logiciel** et son mode d'emploi tandis que la dernière est réservée aux **tests** de reconnaissance.

Vous trouverez à la fin de chaque chapitre les principales fonctions C++ s'y rapportant.

## Remerciements

Merci à M. Gianni Pante, GPIL à Martigny, pour son dynamisme, sa maîtrise de FrameMaker et pour son amitié.

Merci à M. Juergen Luetin, responsable du groupe Vision de l'IDIAP à Martigny, pour ses encouragements et sa disponibilité.

Merci à M. Gilbert Maître, professeur de traitement du signal à l'EIV, pour son goût des mathématiques.

Merci à Mme Marylène Michelloud, pour m'avoir transmis le virus de l'informatique.

Merci encore à toutes les personnes travaillant à l'IDIAP pour leur convivialité et leur sympathie.

---

*SECTION I*

*Traitement  
de l'image*

---

# Introduction au traitement de l'image

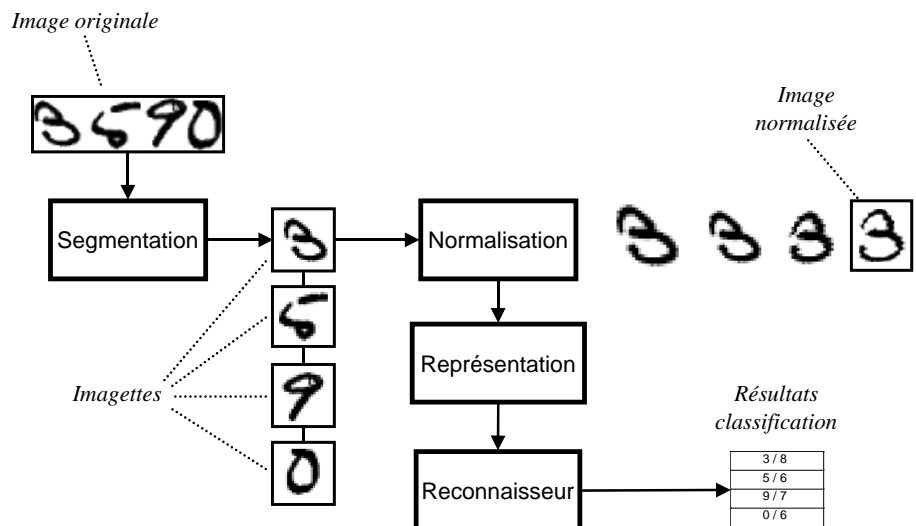
*Vous découvrirez dans ce chapitre la place essentielle que possède le traitement de l'image en vue de la reconnaissance.*

## Utilité

Qui donc peut me reconnaître physiquement s'il ne m'a jamais vu ou s'il n'a reçu aucune description détaillée de ma personne auparavant? Pour cette même raison, un reconnaiseur a besoin de passer préalablement par une phase d'apprentissage avant de pouvoir accomplir pleinement sa tâche. Afin d'obtenir de bons résultats, il est primordial que les objets à reconnaître ressemblent le plus possible aux objets appris.

C'est pour cela, que nous allons procéder, comme le montre la figure 2-1, à toute une série de traitements pour que notre image se présente sous une forme connue du reconnaiseur.

**FIGURE 2-1**  
Différentes phases de traitements de l'image





L'image d'origine va donc passer par quatre grandes étapes qui sont la **segmentation**, la **normalisation**, la **représentation** et enfin le **reconnaisseur**.

Nous pouvons visualiser ses étapes comme étant des blocs avec chaque fois une entrée et une ou plusieurs sortie.

Voici donc une description générale de chacun des blocs:

**Segmentation:** Ce bloc reçoit en entrée une image contenant la séquence de caractères manuscrits. Il en ressort une liste d'images contenant chacune un caractère.

**Normalisation:** Ce bloc reçoit une image et lui affecte plusieurs traitements tels que la suppression de taches, la correction de l'inclinaison, la normalisation de l'épaisseur du trait, la normalisation de la taille ainsi que le centrage. Cette phase est aussi appelée prétraitement.

**Représentation:** Ce bloc reçoit en entrée une image normalisée. Plusieurs transformations différentes peuvent lui être appliquées, cela dépend du reconnaiseur. Par exemple, une image peut être représentée sous une forme de vecteur à  $n$  dimensions,  $n$  étant le nombre de pixels de l'image. Elle peut aussi être représentée sous forme d'histogrammes contenant ses projections horizontales, verticales et diagonales.

**Reconnaisseur:** Ce bloc contient le reconnaiseur qui a suivi préalablement un entraînement. Il est capable, d'après la représentation qu'on lui donne de l'image, de créer une liste de candidats avec un ordre de vraisemblance. Pour un reconnaiseur entraîné avec des lettres, il est envisageable de chercher dans un dictionnaire électronique les mots correspondant à la séquence de lettres reconnues. Si ce mot n'existe pas, il est toutefois possible de changer l'ordre de vraisemblance d'un ou plusieurs caractères. Si cela ne suffit pas, il faut modifier la segmentation de l'image, soit par une fusion ou soit par une scission.

Avant de passer à une description plus détaillée de ces étapes dans les chapitres suivants, voici une notion incontournable dans le traitement de l'image: le filtrage.

## Le filtrage

### Introduction

En générale, une image possède une certaine redondance spatiale, c'est-à-dire qu'entre deux pixels voisins, les caractéristiques sont très semblables. Le bruit dans une image peut alors être défini de la manière suivante.

C'est un phénomène brusque de variation d'un pixel isolé par rapport à ses voisins.

Ainsi pour lutter contre les effets du bruits, il est nécessaire d'opérer des transformations qui pour chaque pixel tiennent compte de son voisinage. Les méthodes parmi les plus utilisées sont les techniques dites de *filtrage* de l'image. Dans le but d'atténuer les effets du bruits, elles portent le nom de filtrage *passé-bas*. Il existe deux types de filtrages passé-bas:

- le filtrage linéaire où la transformation d'un pixel est le fruit d'une combinaison linéaire des pixels voisins
- le filtrage non linéaire où les pixels voisins interviennent suivant une loi non linéaire.

Dans notre cas nous nous concentrerons sur le filtrage linéaire, car sa mise en oeuvre est très simple et son efficacité remarquable.

### Notation:

Voici la notation que nous utiliserons par la suite pour représenter une image de largeur  $n$  et de hauteur  $m$  :

$$I = [i_{xy}] = \begin{bmatrix} i_{11} & i_{12} & \dots & i_{1n} \\ i_{21} & i_{22} & \dots & i_{2n} \\ \cdot & \cdot & \dots & \cdot \\ i_{m1} & i_{m2} & \dots & i_{mn} \end{bmatrix} \quad (2.1)$$

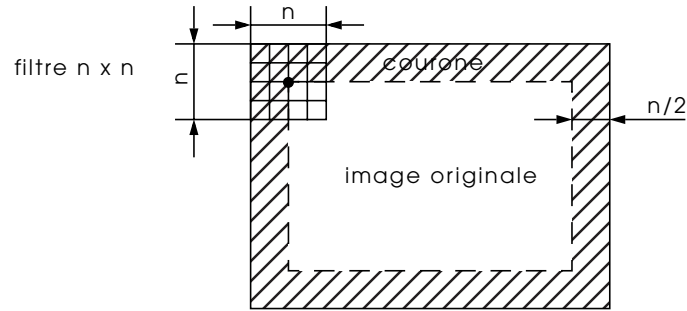
Voici la notation pour la représentation d'un pixel de l'image:

$$p_{xy} = I[x, y] \quad (2.2)$$

### Le filtrage linéaire

Afin d'éliminer les effets de bords, nous allons ajouter pour un filtre  $G$  de dimension  $n \cdot n$  une couronne de largeur  $n/2$ , comme vous pouvez le voir à la figure 2-2.

**FIGURE 2-2**  
Suppression de l'effet  
de bord



Le filtre  $G$  est en fait une matrice de nombres. Il peut être de dimension rectangulaire, ce qui changera évidemment les dimensions de la couronne.

L'équation 2.4 représente un filtre  $3 \times 3$ .

$$G_{(3 \times 3)} = \begin{bmatrix} G_{11} & G_{12} & G_{13} \\ G_{21} & G_{22} & G_{23} \\ G_{31} & G_{32} & G_{33} \end{bmatrix} \quad (2.3)$$

L'opération mathématique appliquée à chaque pixel de l'image s'appelle le *produit de convolution* ( $\otimes$ ) si la combinaison d'un pixel avec ses voisins est faite de la même manière pour tous les endroits de l'image.

Voici un exemple de produit de convolution d'un filtre  $G$  avec une image

$I_{in}$ :

$$\begin{aligned} I_{out} &= I_{in} \otimes G \\ &= I_{in} \otimes G_x \otimes G_y \end{aligned} \quad (2.4)$$

Ici,  $G_x$  représente la matrice horizontale et  $G_y$  la matrice verticale. Voici un exemple toujours pour une matrice  $3 \times 3$ :

$$\begin{aligned} G &= G_x \otimes G_y \\ &\text{avec} \\ G_x &= \begin{bmatrix} G_{x1} & G_{x2} & G_{x3} \end{bmatrix} \\ G_y &= \begin{bmatrix} G_{y1} \\ G_{y2} \\ G_{y3} \end{bmatrix} \end{aligned} \quad (2.5)$$

Ceci signifie que nous pouvons filtrer l'image d'abord avec  $G_x$  puis avec  $G_y$ .

Il est donc possible d'appliquer des filtres  $H(f)$  utilisé en électricité sur des images. Les fréquences représentent, pour une image, les variations de couleurs entre deux pixels voisins. Pour transposer ce type de filtres, il suffit de calculer la transformée de Fourier inverse du filtre et d'introduire les valeurs dans les matrices  $G_x$  et  $G_y$ .

$$G_x = F^{-1}[H(f)] \quad (2.6)$$

Un exemple concret sera donné au chapitre "4. Prétraitements", concernant un filtre de Gauss passe-bas.

### Optimisation pour la rapidité de calcul

Pour un filtre de dimensions 10x10, le calcul de chaque pixel demandera 100 multiplications si nous convoluons l'image directement avec le filtre, tandis qu'il faudra seulement 20 multiplications si nous convoluons l'image d'abord avec le filtre horizontal  $G_x$  puis ensuite avec le filtre vertical  $G_y$ .

La convolution est une opération qui nécessite beaucoup de temps de calcul du fait qu'elle comporte un grand nombre de multiplications. Il est toutefois possible d'y remédier en passant dans le domaine des fréquences. En effet, un convolution dans le domaine du temps correspond à une multiplication dans le domaine des fréquences. Cela diminue fortement le temps de calcul pour un système informatique.

$$\begin{aligned} I_{out}(t) &= I_{in}(t) \otimes G(t) \\ F(I_{out}(t)) &= F(I_{in}(t)) \cdot F(G(t)) \\ I_{out}(t) &= F^{-1}(F(I_{in}(t)) \cdot F(G(t))) \end{aligned} \quad (2.7)$$

Il faudra cependant calculer la fonction de Fourier de l'image d'entrée et du filtre, puis après multiplication, calculer la fonction inverse de Fourier.

### Remarque

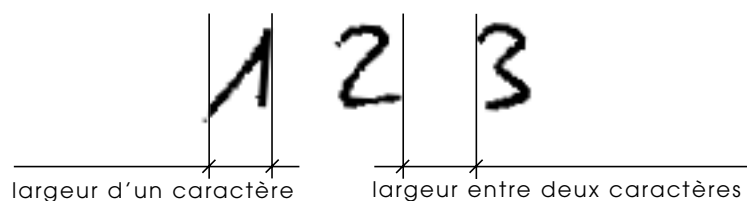
Dans mon code C++, j'ai filtré les images en utilisant l'opération de convolution car la rapidité n'était pas exigée. Il serait néanmoins possible d'implémenter la fonction de Fourier par une FFT (Fast Fourier Transform).

*Segmenter signifie diviser en segments ou diviser (un tout) en plusieurs parties. En effet, dans ce chapitre, nous allons découper l'image contenant la séquence de caractères manuscrits en plusieurs imquettes contenant chacune un caractère.*

## Méthode

Pour procéder à cette opération, nous allons utiliser un algorithme des plus simples qui consiste à mesurer l'espace blanc entre les projections verticales des caractères. Pour qu'une segmentation s'effectue correctement, il faudra donc que les caractères se détachent distinctement les uns des autres.

**FIGURE 3-1**  
Mesures d'espaces



Afin de faciliter cette tâche, il a été nécessaire de projeter tous les pixels de l'image sur un axe horizontal.

$$V[x] = \sum_i p(x, y_i) \quad (3.1)$$

Ci-dessus,  $V[x]$  représente la valeur de la projection vertical de l'image au point  $x$  de l'abscisse et  $p(x, y)$  la valeur d'un pixel aux coordonnées  $x, y$ .

Il est tout à fait possible de faire de même pour l'axe vertical:

$$H[y] = \sum_i p(x_i, y) \quad (3.2)$$

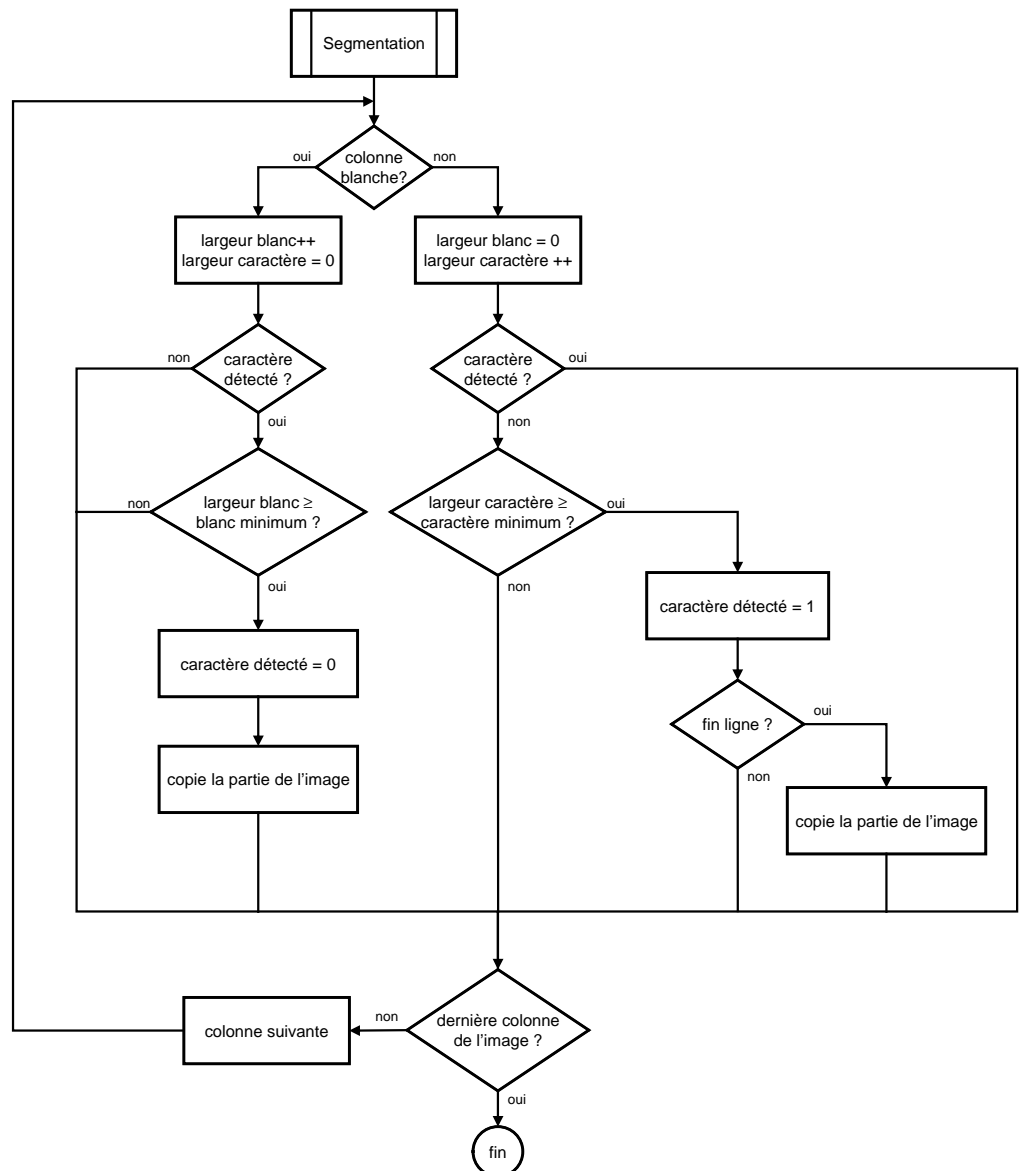
Cette projection horizontale peut être utile pour segmenter plusieurs lignes.

### Implémentation

Voici, comme l'illustre la figure 3-2, le structogramme de la fonction de segmentation. Il faudra simplement passer trois paramètres qui sont:

- un pointeur sur l'image à segmenter
- l'espace minimum requis entre les caractères (blanc minimum)
- la largeur minimale d'un caractère (caractère minimum)

**FIGURE 3-2**  
Structogramme de la  
fonction principale de  
segmentation



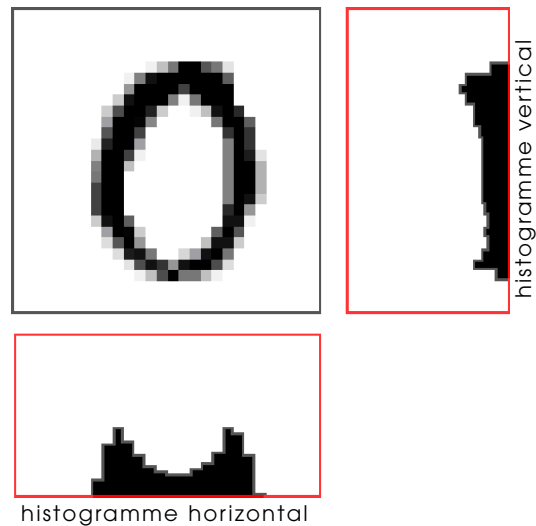
Cette fonction nous retourne une liste de pointeurs sur chaque imagette.

Les termes **largeur caractère** et **largeur blanc** représentent simplement la largeur en pixels d'un caractère et l'espace séparant deux caractères comme le représente la figure 3-1.

## Histogramme

Par la suite, nous appellerons la projection verticale: histogramme horizontal de par sa forme horizontale et la projection horizontale: histogramme vertical. Voici à la figure 3-3 un exemple de projections verticale et horizontale d'une imagette.

**FIGURE 3-3**  
Histogrammes horizontal et vertical



## Fonctions C++

**TABLE 3-1** : Fonctions relatives à la segmentation

classe	fonction	description
CSegmentation	DoSegmentation()	Segmente une image en retournant une liste d'images contenant les caractères
CSegmentation	GetNumberImages()	Retourne le nombre d'images après une segmentation
CHisto	DoHisto()	Calcul les projections horizontale et verticale d'une image
CDisplay	DisplayHistoH() DisplayHistoV()	Affiche l'histogramme horizontal Affiche l'histogramme vertical



---

*Ce chapitre décrit les différents prétraitements qui ont été utilisés pour le démonstrateur.*

## ***Flou***

---

Le flou va nous permettre de transformer une image noir-blanc en une image à niveaux de gris. Pour cela, nous allons adoucir les flancs par filtrage. Il existe une multitude de filtres plus ou moins utilisés et il serait fastidieux de vouloir tous les tester. Notre choix s'est porté sur le filtrage dit *gaussien* car il représente un filtre linéaire parmi les plus courants et son efficacité n'est plus à prouver.

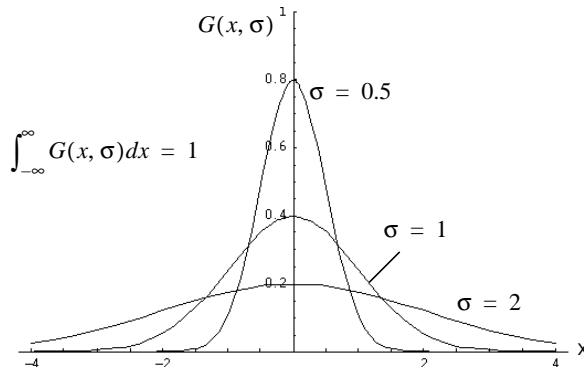
### **Filtre de Gauss**

Ce filtre tire son nom de la valeur de ses coefficients qui sont ceux d'une courbe de Gauss. Pour commencer, prenons l'équation qui décrit une courbe de gauss à une dimension.

$$G(x, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{x^2}{2\sigma^2}} \quad (4.1)$$

La figure 4-1 représente cette fonction. Nous pouvons remarquer que le paramètre  $\sigma$  permet de modifier la sélectivité du filtre. Nous utiliserons la plupart du temps  $\sigma = 1$ .

**FIGURE 4-1**  
Courbe de Gauss à une dimension



Comme nous l’avons vu au chapitre “ 2. Introduction au traitement de l’image ”, nous pouvons dimensionner notre filtre  $G$ . Prenons pour exemple un filtre de dimensions  $3 \times 3$  et un  $\sigma = 1$ .

Il suffit seulement de calculer les filtres  $G_x$  et  $G_y$  et de les appliquer l’un après l’autre:

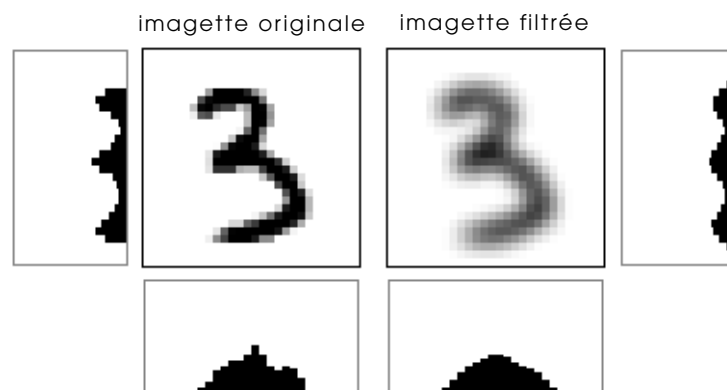
$$G_x = \frac{1}{0.88} \begin{bmatrix} 0.24 & 0.40 & 0.24 \end{bmatrix}$$

$$G_y = \frac{1}{0.88} \begin{bmatrix} 0.24 \\ 0.40 \\ 0.24 \end{bmatrix} \tag{4.2}$$

Le facteur  $\frac{1}{0.88}$  sert à garder le niveau de couleur normal, c’est-à-dire que si tous les pixels de l’image sont à 1, les filtres ci-dessus ne doivent avoir aucune influence.

Voici à la figure 4-2 une application d’un filtre de Gauss sur une imagerie.

**FIGURE 4-2**  
Application d’un filtre gaussien de  $5 \times 5$  pixels



Nous voyons nettement l’effet du filtre passe-bas à l’aide des histogrammes.

## Suppression des taches

Lors d'une capture d'image, quelle qu'en soit la source, la présence de bruit est inévitable. Ce bruit doit être supprimé car il vient perturber l'information utile. Il provient généralement de défaut présent dans la structure du papier, de taches quelconques, d'un quadrillage en arrière plan ou d'une mauvaise opacité du papier, faisant transparaître le recto sur le verso.

### Algorithme

Seul un algorithme de suppression de taches a été implémenté dans le démonstrateur. Mais avant de le présenter, je vais tout de même expliquer brièvement comment il est possible de supprimer un fond d'image.

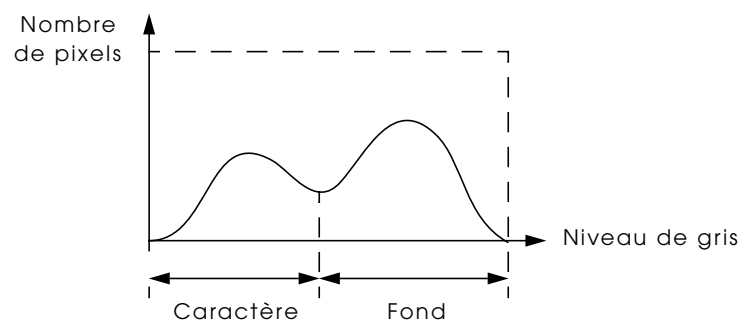
### Suppression d'un fond d'image

D'abord nous devons établir l'histogramme des niveaux de gris ou des niveaux de couleurs de l'image. Avec l'imagette de la figure 4-3, nous obtenons un histogramme qui ressemble à la figure 4-4.

**FIGURE 4-3**  
Imagette avec un bruit de fond



**FIGURE 4-4**  
Histogramme d'une image avec un arrière plan



Ensuite il devient assez facile de trouver la limite entre l'information utile, ici le caractère, et le bruit de fond car ils se distinguent généralement assez bien.

Revenons à nos moutons en ce qui concerne la suppression des taches.

### Suppression des taches

L'algorithme utilisé ici est un peu particulier. J'ai en premier lieu effectué un adoucissement de l'image avec l'algorithme de flou que nous avons vu précédemment, ce qui dissout les points noirs isolés. Puis un seuillage est effectué, ce qui supprime tous les pixels en dessous d'une certaine valeur. Ce seuil est calculé en fonction du poids moyen de l'image et d'un coefficient  $a$  fixé arbitrairement.

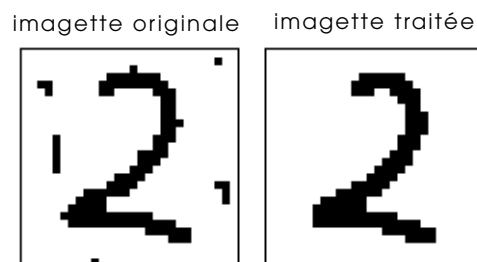
$$\text{seuil} = \frac{a}{x \cdot y} \sum_I p(x, y) \quad (4.3)$$

Ci-dessus,  $\sum_I$  représente le parcours de tous les pixels de l'image et

$p(x, y)$  la valeur ou le poids d'un pixel aux coordonnées  $(x, y)$ .

Comme nous pouvons le constater à la figure 4-5, cet algorithme très simple nous donne de bons résultats.

**FIGURE 4-5**  
Exemple de suppression des taches



## Correction de l'inclinaison des caractères

### Définitions

L'algorithme repose sur une mesure des moments d'ordre 1 et 2 de l'objet 2D constitué par les pixels noirs de l'imagette.

Ci-après  $N$  représente le nombre de pixels noirs de l'imagette et  $\sum_N$  la somme sur les pixels noirs de l'image.

Les **moments d'ordre 1** (ou moments linéaires) sont les moyennes des coordonnées  $x$  et  $y$  des pixels noirs:

$$\bar{x} = \frac{1}{N} \sum_N x \quad \bar{y} = \frac{1}{N} \sum_N y \quad (4.4)$$

La paire  $(\bar{x}, \bar{y})$  constitue les coordonnées du “centre de masse” de l’objet.

Les **moments d’ordre 2** (ou moments quadratiques) relativement à l’origine du système de coordonnées sont définis par les formules suivantes:

$$\overline{x^2} = \frac{1}{N} \sum_N x^2 \quad \overline{y^2} = \frac{1}{N} \sum_N y^2 \quad \overline{xy} = \frac{1}{N} \sum_N xy \quad (4.5)$$

Les moments d’ordre 2 relativement au centre de masse sont:

$$\begin{aligned} \overline{(x - \bar{x})^2} &= \frac{1}{N} \sum_N (x - \bar{x})^2 \\ \overline{(y - \bar{y})^2} &= \frac{1}{N} \sum_N (y - \bar{y})^2 \\ \overline{(x - \bar{x})(y - \bar{y})} &= \frac{1}{N} \sum_N (x - \bar{x})(y - \bar{y}) \end{aligned} \quad (4.6)$$

ils vérifient les égalités suivantes:

$$\begin{aligned} \overline{(x - \bar{x})^2} &= \overline{x^2} - \bar{x}^2 \\ \overline{(y - \bar{y})^2} &= \overline{y^2} - \bar{y}^2 \\ \overline{(x - \bar{x})(y - \bar{y})} &= \overline{xy} - \bar{x}\bar{y} \end{aligned} \quad (4.7)$$

Pour chacune de ces égalités, les termes de droite seront utilisés pour l’implémentation de l’algorithme car ils consomment beaucoup moins de ressources processeur pour le calcul.

### Principe de l’algorithme

L’algorithme applique la transformation

$$(x' = x - a(y - \bar{y})) \quad (4.8)$$

sur l'objet avec

$$a = \frac{\overline{(x - \bar{x})(y - \bar{y})}}{\overline{(y - \bar{y})^2}} \quad (4.9)$$

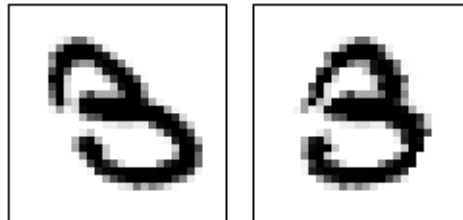
ce qui a pour effet d'annuler le moment mixte d'ordre 2  $\overline{(x' - \bar{x})(y - \bar{y})}$  (cf. preuve ci-après)

Moment mixte d'ordre 2 de l'objet transformé

$$\begin{aligned} \overline{(x' - \bar{x})(y - \bar{y})} &= \frac{1}{N} \sum_N ((x - \bar{x})(y - \bar{y}) - a(y - \bar{y})^2) \\ &= \overline{(x - \bar{x})(y - \bar{y})} - a \frac{1}{N} \sum_N (y - \bar{y})^2 \\ &= \overline{(x - \bar{x})(y - \bar{y})} - \overline{a(y - \bar{y})^2} \\ &= 0 \end{aligned} \quad (4.10)$$

La figure 4-6 illustre l'application de cet algorithme.

**FIGURE 4-6**  
Exemple de correction  
de l'inclinaison



## Centrage

Ce traitement a pour but de normaliser la position des caractères à l'intérieur de l'imagerie. Le centre physique de l'imagerie correspondra toujours au centre de gravité du caractère.

## Algorithme

Commençons par calculer le centre de gravité de l'image  $I$ . Voici le poids total de l'image:

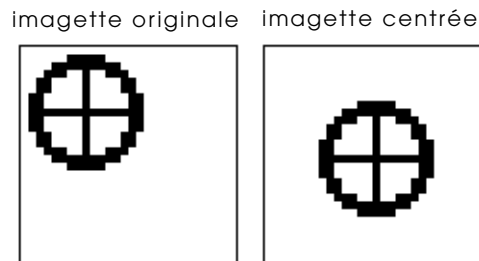
$$A = \iint_I p(x, y) dx dy \quad (4.11)$$

Ensuite vient le point d'annulation des moments du premier ordre sur les axes  $x$  et  $y$  qui représentent le centre de gravité:

$$\begin{aligned} \bar{x} \iint_I p(x, y) dx dy &= \iint_I xp(x, y) dx dy \\ \bar{y} \iint_I p(x, y) dx dy &= \iint_I yp(x, y) dx dy \end{aligned} \quad (4.12)$$

**FIGURE 4-7**

Exemple de centrage d'une image sur son centre de gravité. Ici le dessin d'une cible montre bien l'efficacité de l'algorithme.



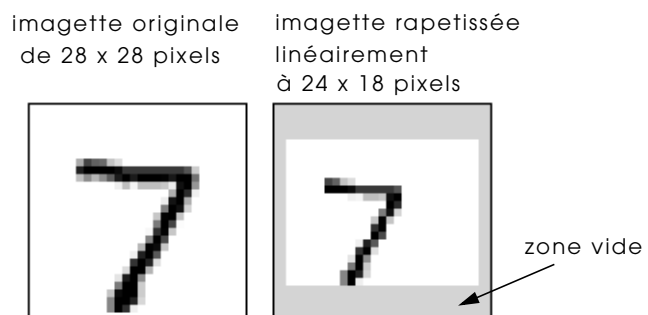
Nous remarquons tout de même que l'imagette est légèrement décallée en bas à droite. Ceci est dû au fait qu'elle contient un nombre pair de pixels. Il a fallu donc faire un choix pour déterminer de quel côté l'image devait basculer.

## Changer la taille de l'image

Cette fonction nous permet d'agrandir ou de rapetisser une image sans l'étirer, c'est-à-dire en gardant les proportions de l'image d'origine.

**FIGURE 4-8**

Changement de la taille d'une image de manière linéaire



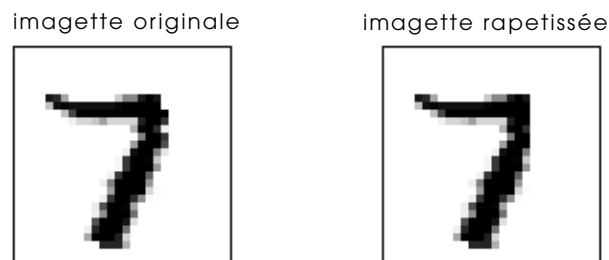
Une fonction très simple consiste à calculer les rapport des hauteurs et des largeurs entre l'image finale et de l'image originale. Ces rapports représentent les pas ou la fréquence à laquelle il faudra rééchantillonner l'image.

$$\text{pas}_{horizontal} = \frac{\text{largeur finale}}{\text{largeur originale}} \quad (4.13)$$

Pour conserver les proportions, il faut prendre comme pas la plus grande valeur entre le pas horizontal et le pas vertical.

Cependant, un problème persiste lorsque nous voulons rapetisser une image d'un pixel seulement. Nous voyant en effet, sur la figure 4-9, qu'avec notre premier algorithme, la dernière colonne de pixels est supprimée. Cela est dû au pas qui prend seulement une valeur suffisante en fin d'image pour sauter un pixel.

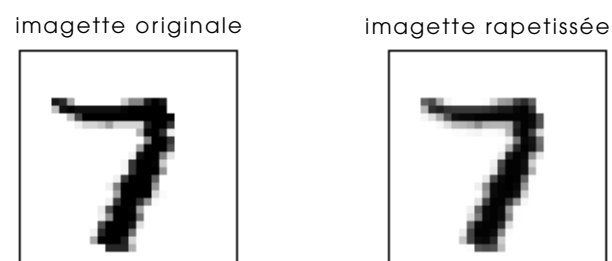
**FIGURE 4-9**  
Réduction de la taille d'une imagette d'un pixel



La solution à ce problème consiste à rééchantillonner l'image avec une meilleure résolution.

Pour ce faire, l'image sera agrandie d'environ un facteur dix à l'aide du premier algorithme de changement de taille. Ensuite elle passera à travers un filtre passe-bas de largeur de la moitié de ce même facteur. Et pour terminer, elle sera rapetisser à la taille désirée, toujours avec notre premier algorithme. Voici à la figure 4-10 le résultat de ce nouvel algorithme.

**FIGURE 4-10**  
Réduction de la taille d'une imagette d'un pixel après rééchantillonnage



Dans le chapitre “ 8. Tests ”, vous verrez dans les tableaux de tests les deux traitements suivants: normalisation de la taille<sup>(1)</sup> et normalisation de la taille<sup>(2)</sup>. Cela signifie que pour la normalisation de la taille<sup>(1)</sup>, nous avons

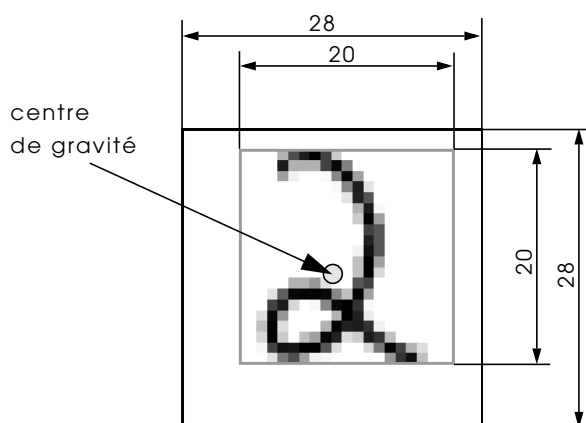


utilisé le premier algorithme de changement de taille et pour la normalisation de la taille<sup>(2)</sup> le deuxième.

## Normalisation de la taille des images

Toutes les images des bases de données MNIST<sup>1</sup> qui servent à l'entraînement et aux tests sont normalisées de la manière suivante:

**FIGURE 4-11**  
Exemple d'une image normalisée



Elles ont une taille de 28 x 28 pixels et le caractère, ici le chiffre 2, est contenu dans une zone de 20 x 20 pixels et centré sur son centre de gravité.

### Méthode

Voici comment nous allons procéder pour obtenir cette normalisation de la taille pour une image:

- découper le caractère
- l'agrandir ou le rapetisser linéairement à la taille désirée
- agrandir le canevas pour obtenir la bonne grandeur de l'image finale
- centrer l'image sur son centre de gravité

La plupart de ces fonctions ont été décrites plus haut, il suffit simplement de les appliquer une à une.

## Manipulation de l'image

Voici encore quelques fonctions complémentaires permettant de manipuler une image. Les algorithmes ne sont pas décrits en raison de leur extrême simplicité. Voici simplement des exemples. Dans chaque cas,

1. Voici l'adresse décrivant les bases de données MNIST:  
<http://www.research.att.com/~yann/ocr/mnist/index.html>

l'unité utilisé est le pixel et chaque imagerie mesure 28 x 28 pixels. L'origine des coordonnées (0, 0) se trouve toujours sur le premier pixel en haut à gauche de l'image.

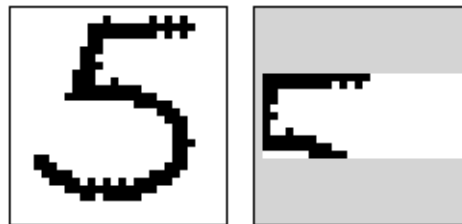
### copier

Cette fonction a pour but de copier entièrement ou partiellement une image. Si la zone à copier n'est pas totalement incluse dans la surface de l'image actuelle, alors ce qui se trouve en dehors de celle-ci sera rempli de pixels blancs.

**FIGURE 4-12**

Exemple de copie d'une partie de l'image du point (10, 2) au point (35, 12)

imagerie originale      copie d'une partie de l'imagerie



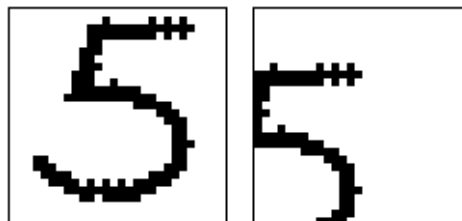
### décaler

Cette fonction permet de décaler l'image sur les deux axes x et y. Les nouveaux pixels remplaçant le vide après le décalage seront blancs.

**FIGURE 4-13**

Décalage de -10 sur l'axe des x et +6 sur l'axe y

imagerie originale      imagerie décalée



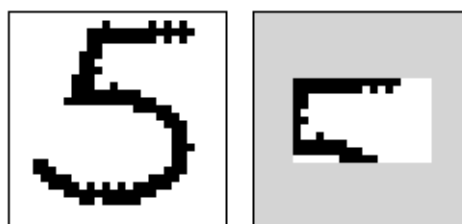
### découper

Cette fonction découpe une partie de l'image. Si la zone de découpage sort de l'image, alors cette partie extérieure ne sera pas prise en compte comme vous pouvez le constater sur la figure 4-14.

**FIGURE 4-14**

Découpage de l'image du point (10, 2) au point (35, 12)

imagerie originale      imagerie découpée



## Découper au minimum

Ce qui différencie cette fonction de la précédente réside dans le fait qu'elle calcule elle-même la zone à découper afin que cette zone ne contienne uniquement que l'information essentielle

**FIGURE 4-15**  
Découpage minimum  
d'une image



## Fonctions C++

**TABLE 4-1** : Diverses fonctions relatives aux prétraitements de l'image

classe	fonction	description
CPreprocessing	DoBlur()	Effectue un adoucissement de l'image (flou gaussien)
CPreprocessing	RemoveSmallBlobs()	Enlève les taches sur l'image
CPreprocessing	SlantCorrectChar()	Corrige l'inclinaison du caractère contenu dans l'image
CPreprocessing	CenterMass()	Centre l'image sur son centre de gravité
CPreprocessing	SizeNormalize()	Normalise la taille des caractères ainsi que de l'image globale
CPreprocessing	Shift()	Décale l'image dans les deux dimensions
CImage	ResizeLinear()	Agrandit ou rapetisse une image
CImage	Copy()	Copy une zone de l'image
CImage	Crop()	Découpe une partie de l'image
CImage	CropMin()	Découpe l'image au minimum

---

*SECTION II*

*Reconnaissance*

---

---

*Ce chapitre est consacré exclusivement au reconnaisseur de type SVM, car il serait trop long et fastidieux de vouloir décrire d'autres types de reconnaisseurs et cela ne fait pas l'objet de ce rapport.*

## Introduction

---

Pourquoi avons-nous choisi un reconnaisseur de type SVM (Support Vector Machine, Machine à Vecteurs de Support)? Celui-ci présente un certain nombre d'avantages par rapport aux autres types de reconnaisseurs tels que les réseaux de neurones, les arbres de décision ou les classificateurs bayésiens.

En effet, sa mise en oeuvre est très simple car il n'y a qu'un paramètre à définir: son type de noyau que nous verrons plus bas. De plus, ses performances sont excellentes, comme vous pourrez le constater au chapitre " 8. Tests ". Le numéro de vecteur support est choisi automatiquement.

Afin de pouvoir être fonctionnel, le reconnaisseur doit passer par deux étapes qui sont *l'entraînement* ou apprentissage et *la classification* ou reconnaissance.

La seule ombre au tableau réside dans son entraînement. Etant donné qu'il ne peut classifier que deux classes à la fois, il faudra faire pour chacune des classes un entraînement avec toutes les autres. Comme il est pourvu d'un algorithme itératif pour choisir les bons vecteurs support, cela implique un temps de calcul considérable. Ne dit-on pas: "*La patience est la mère des vertus!*"

## Entraînement

### Principe général

L'entraînement permet de créer des modèles à partir d'objets, comme l'illustre la figure .

**FIGURE 5-1**  
Phase d'entraînement



Ces objets peuvent être de nature quelconque. Dans notre cas, les objets correspondent à la représentation d'une image normalisée sous forme de vecteur à  $n$  dimensions,  $n$  étant le nombre de pixels de l'image.

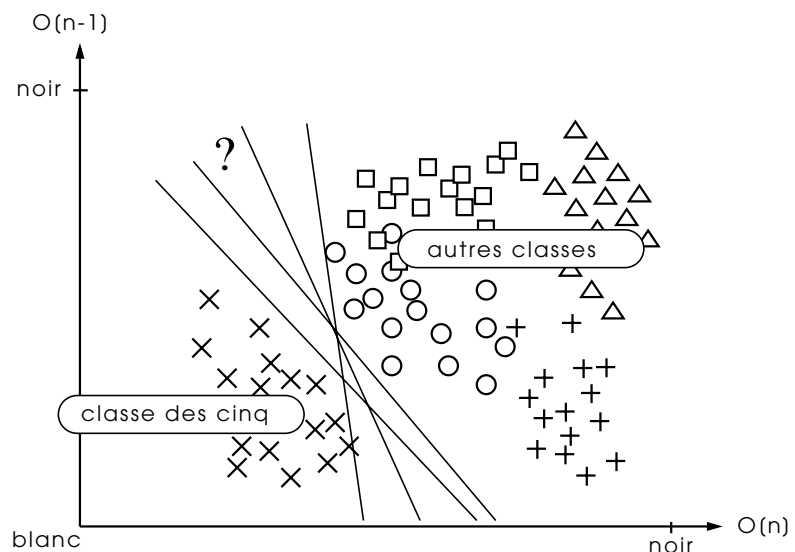
$$\text{objet} = \overrightarrow{\text{image}[n]} \quad n = \text{largeur} \cdot \text{hauteur}$$

Les modèles, quant à eux, contiennent une liste de vecteurs support qui seront décrits un peu plus bas.

Pour la suite de l'explication, nous n'allons prendre que deux dimensions, car au delà de la troisième dimension, il devient assez difficile de se faire une représentation. Dans notre cas, chaque axe représente le niveau de couleur d'un pixel.

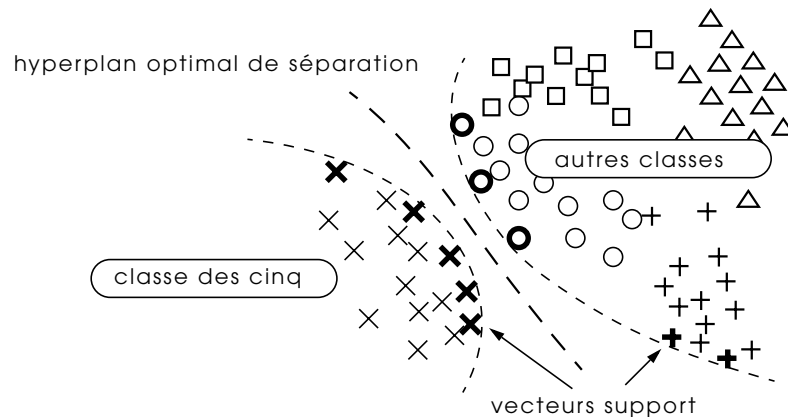
Prenons l'exemple d'un reconnaissseur qui doit être entraîné avec des images de deux classes différentes, par exemple des cinq et toutes les autres classes. Voici donc, à la figure 5-2, la représentation de ces deux classes d'images sur deux dimensions seulement.

**FIGURE 5-2**  
Représentation de deux dimensions



Il s'agit maintenant de distinguer ces deux classes. La méthode utilisée par le reconnaisseur SVM consiste à chercher la plus grande surface (hyperplan) possible séparant les deux classes afin de minimiser l'erreur de classification. Cette surface peut être délimitée de manière linéaire, mais aussi par des polynômes d'ordre  $n$  ou des courbes de Gauss. Ce choix représente le type de noyau du reconnaisseur. Les vecteurs tangents à cette surface sont appelés vecteurs support.

**FIGURE 5-3**  
vecteurs support

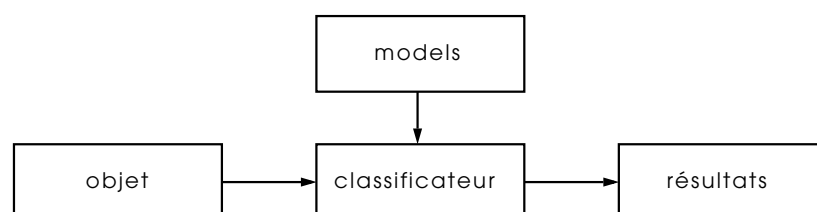


Ainsi est créé, pour chaque classe, un nouveau modèle contenant tous les vecteurs support séparant cette classe des autres. Ils seront ensuite utilisés par le classificateur. Il est à noter que tous les vecteurs support de la classe d'apprentissage sont de signes positifs contrairement aux vecteurs support des autres classes qui sont eux de signes négatifs.

## Classificateur

La reconnaissance de chiffres va pouvoir s'effectuer à l'aide du classificateur. Comme son nom l'indique, il va classifier les objets qu'il recevra en entrée. Pour cela il a besoin des modèles préalablement calculés par l'entraînement.

**FIGURE 5-4**  
Phase de classification



## Algorithme

L'équation ci-dessous, est l'algorithme utilisé pour le reconnaisseur SVM.

$$f(\vec{x}) = \sum_i S_i w_i K_{rbf}(\vec{x}, \vec{SV}_i) \quad (5.1)$$

La classification se fait classe par classe. Le principe est qu'il calcule la somme des distances de tous les vecteurs support d'une classe à l'image à reconnaître, sachant que les distances à la classe en question auront un signe  $S_i$  positif et les autres distances un signe négatif.

Nous obtenons donc une liste de vraisemblance pour chacune des classes.

$w_i$  représente une pondération qui a été calculée à l'entraînement.

Voici ci-dessous l'équation du type de noyau que nous avons utilisé.

$$K_{rbf}(\vec{x}, \vec{SV}) = e^{-\gamma \|\vec{x} - \vec{SV}\|^2} \quad (5.2)$$

Il a été choisi en fonction des résultats des tests faits à l'IDIAP. L'explication de cet algorithme ne fait pas ici l'objet de ce travail.



---

*SECTION III*

*Démonstrateur*

---

---

*Ce chapitre présente les grandes lignes de modélisation du logiciel nécessaires à la compréhension de sa structure général. Pour cette partie, il est recommandé que le lecteur connaisse les grandes lignes de la programmation orientée objets ainsi que la notation UML. Pour plus de détails concernant la programmation, il faut se reporter au code C++ en annexe.*

### ***Introduction***

---

La modélisation est une phase primordiale dans l'élaboration d'un projet informatique. Elle représente le squelette sur lequel la programmation va se greffer. Le concept de réutilisabilité doit y être intégré au maximum.

Pour cela, j'utiliserai principalement la notation unifiée UML<sup>1</sup> qui est née de la fusion des trois principales méthodes de modélisation: Booch, Rumbaugh et Jacobson. Elle est très bien adaptée pour la programmation orientée objets.

### ***Diagramme des cas d'utilisation***

---

Le diagramme des cas d'utilisation permet d'obtenir un aperçu général de la fonctionnalité du logiciel. Son élaboration se fait de manière itérative pour aboutir enfin à la meilleure représentation du projet. Pour cela il est très important de bien connaître le cahier des charges. Il doit rester à un niveau d'abstraction élevé, sans entrer dans les détails. Il se décompose en trois phases principales:

- 
1. Unified Modeling Language

## Définition des acteurs

Cette première phase sert à identifier les différentes entités qui entrent en interaction avec le démonstrateur.

Dans notre cas, deux acteurs se présentent:

- **Utilisateur**: Personne qui utilise le logiciel et désire effectuer la reconnaissance de caractères.
- **Stylo scanner**: Unité de capture et de transfert d'images.

## Définition des cas d'utilisation

Les cas d'utilisations représentent les différentes actions pouvant être activées par les acteurs. Un cas d'utilisation peut en activer un autre.

### Utilisateur:

- Paramètre la transmission de données (données en provenance du stylo scanner)
- Choisit les prétraitements à effectuer
- Choisit le mode de reconnaissance
- Commande le stylo scanner

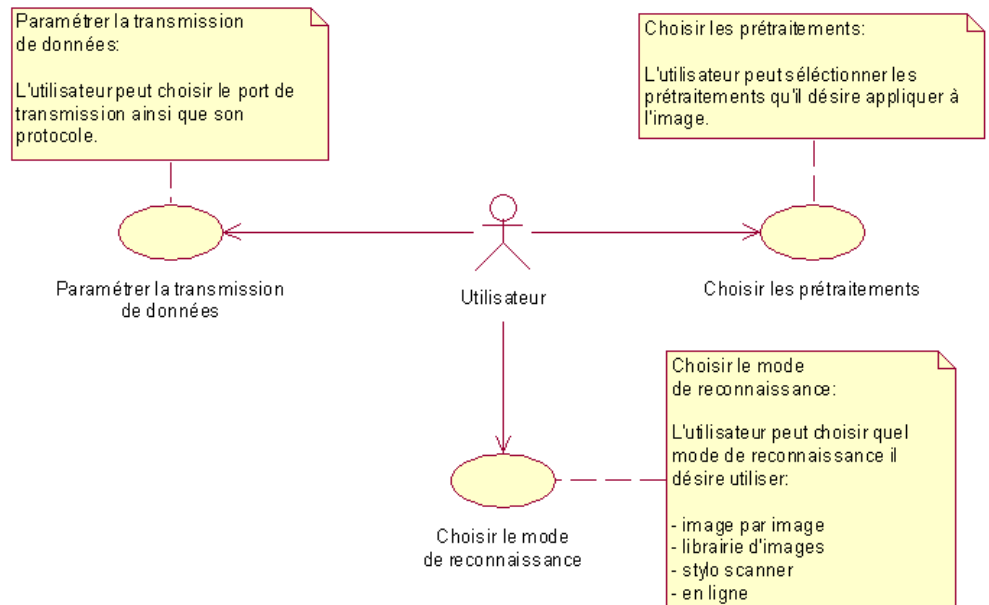
### Stylo scanner:

- Envoie une image

## Représentation graphique

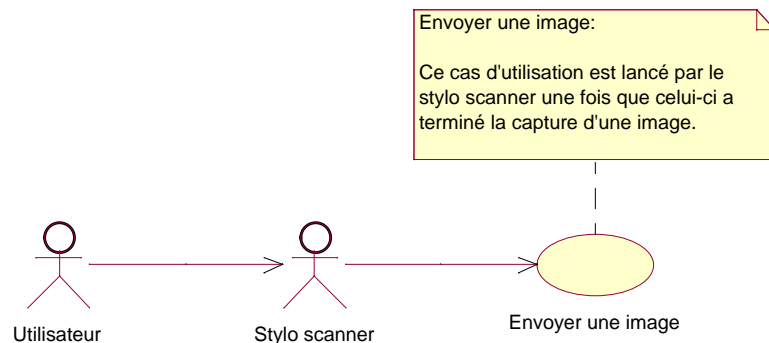
J'ai utilisé le logiciel *Rational Rose 98* comme outil de modélisation, car il permet de réaliser l'ensemble des diagrammes nécessaires pour la description d'un logiciel, le tout en notation UML.

**FIGURE 6-1**  
Diagramme des cas d'utilisation autour de l'utilisateur



Les figures 6-1 et 6-2 nous montrent bien les relations qu'il y a entre les acteurs et les différents cas d'utilisation. Par ailleurs, nous pouvons constater l'extrême simplicité de ces diagrammes, ce qui fait toute leur force.

**FIGURE 6-2**  
Diagramme des cas d'utilisation autour du stylo scanner



Nous voyons à la figure 6-2 que l'utilisateur intervient sur le stylo scanner. En effet, c'est lui qui donnera l'ordre au stylo scanner de commencer la capture de l'image.

## Diagrammes de séquences

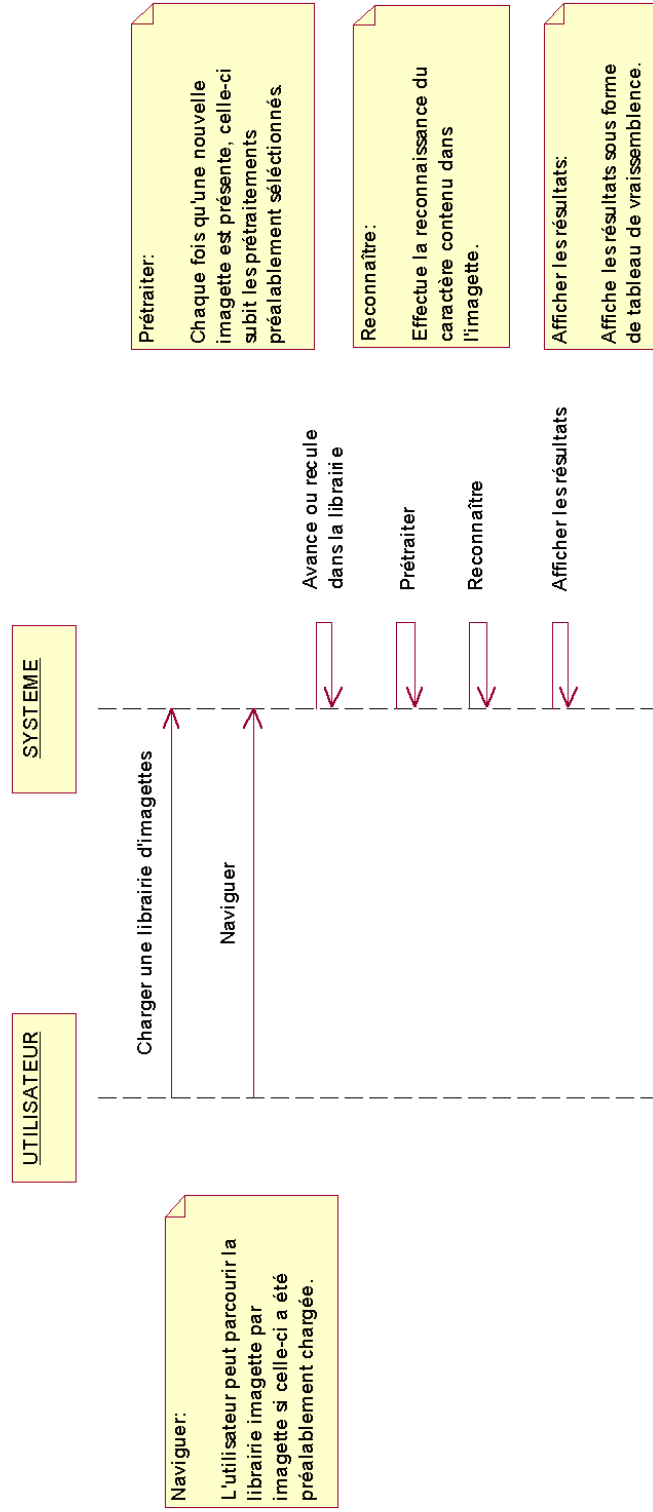
Il s'agit maintenant d'établir les principales séquences de chaque cas d'utilisation. Les lignes verticales représentent l'écoulement du temps. Ici, ces diagrammes restent très généraux, car ils font partie des premières itérations. Ils nous permettent déjà de déterminer les classes principales du projet.

Nous allons rester à ce niveau d'abstraction car il est inutile de décrire avec la représentation UML ce qui peut être décrit avec du code.

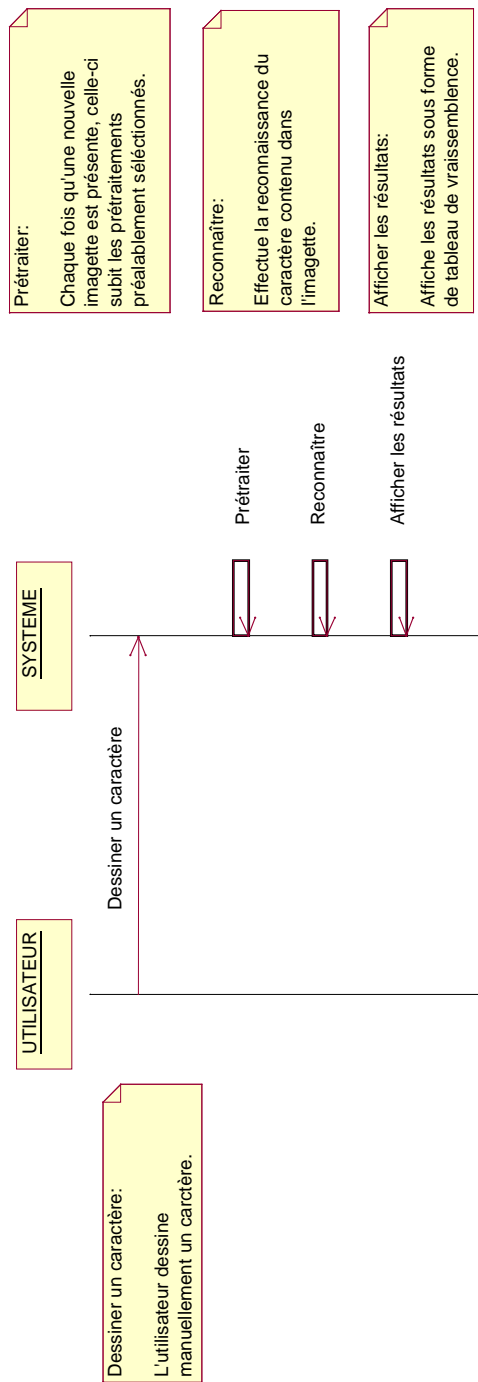
### Représentation graphique

Voici ci-dessous les quatre diagrammes de séquences du cas d'utilisation "choisir le mode de reconnaissance".

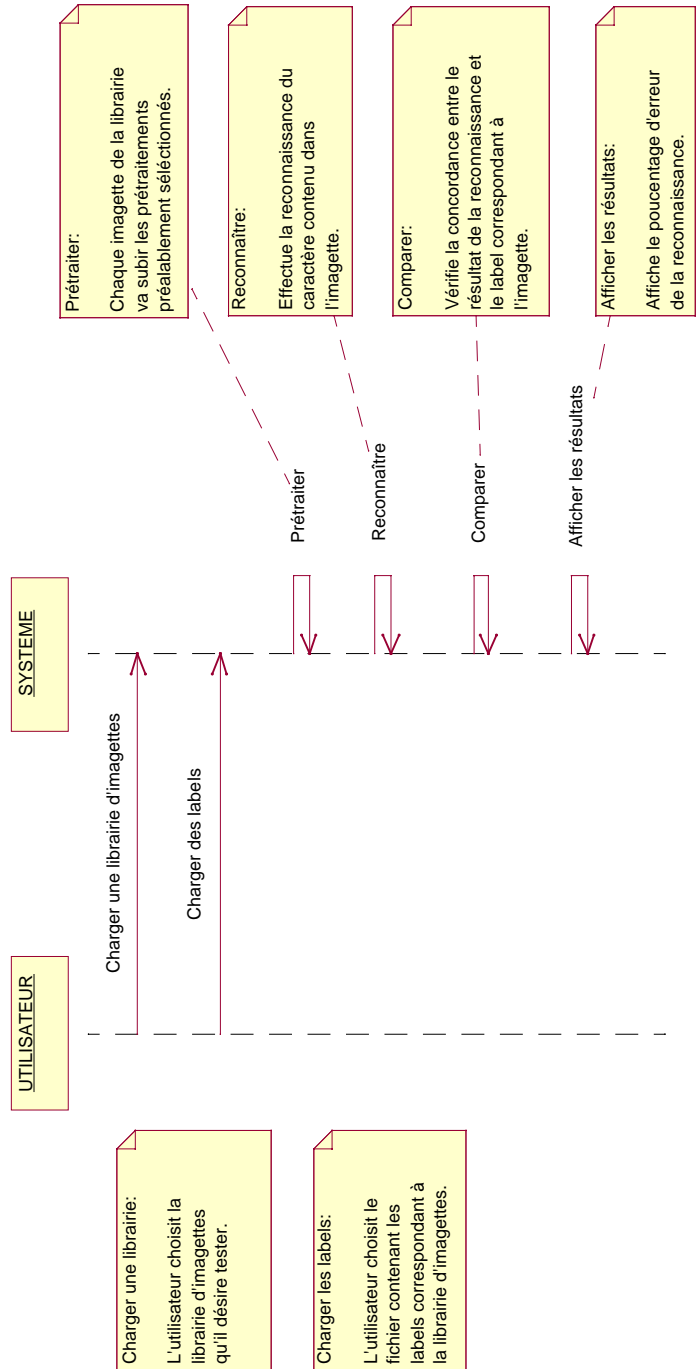
**FIGURE 6-3**  
Mode de reconnaissance: image par image



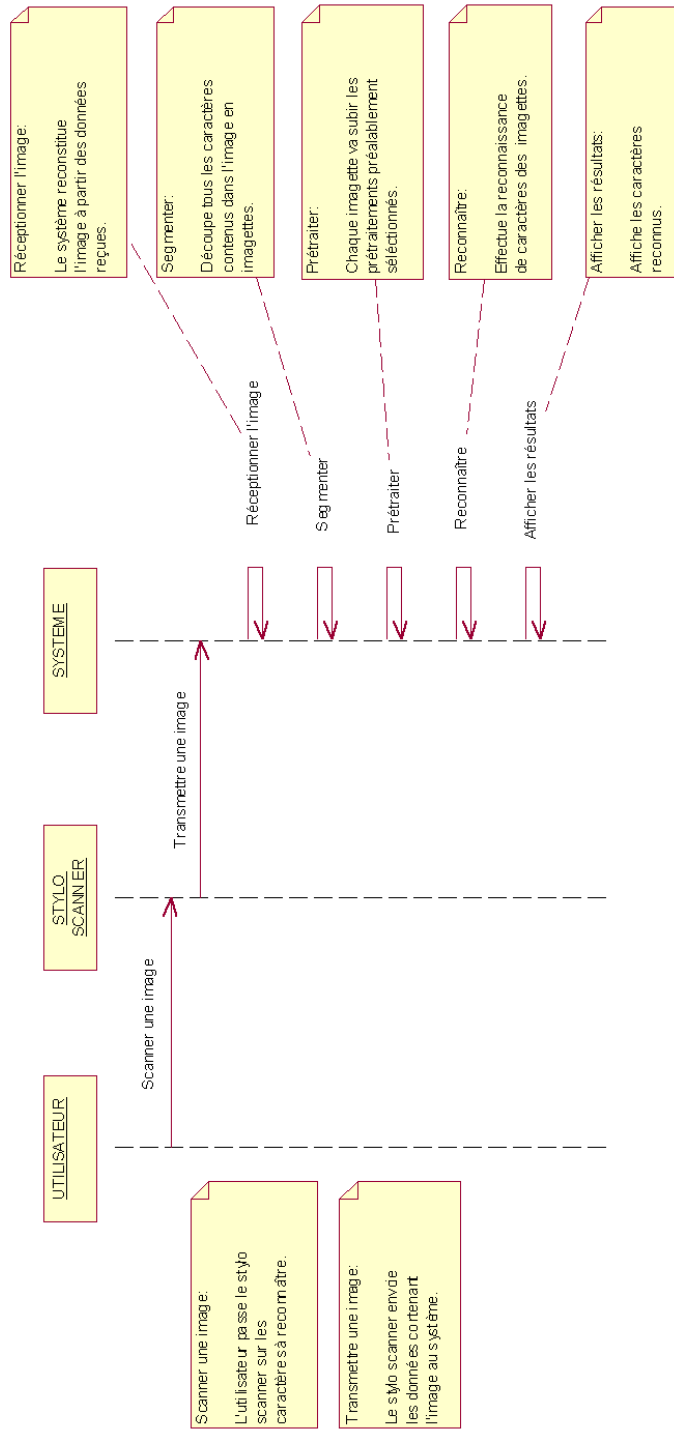
**FIGURE 6-4**  
Mode de reconnaissance:  
en ligne



**FIGURE 6-5**  
Mode de reconnaissance:  
librairie

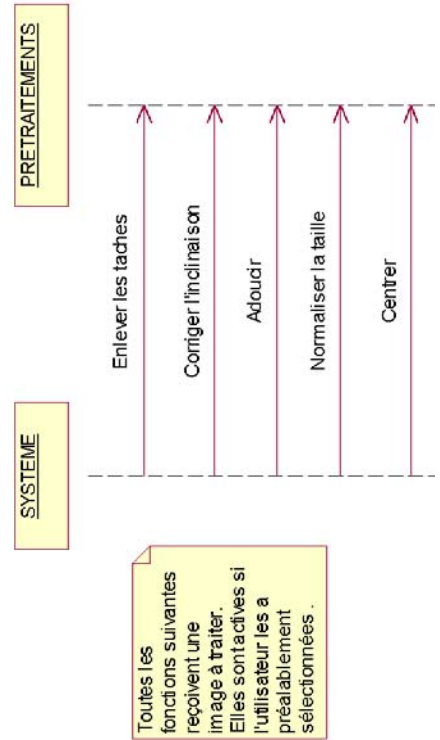


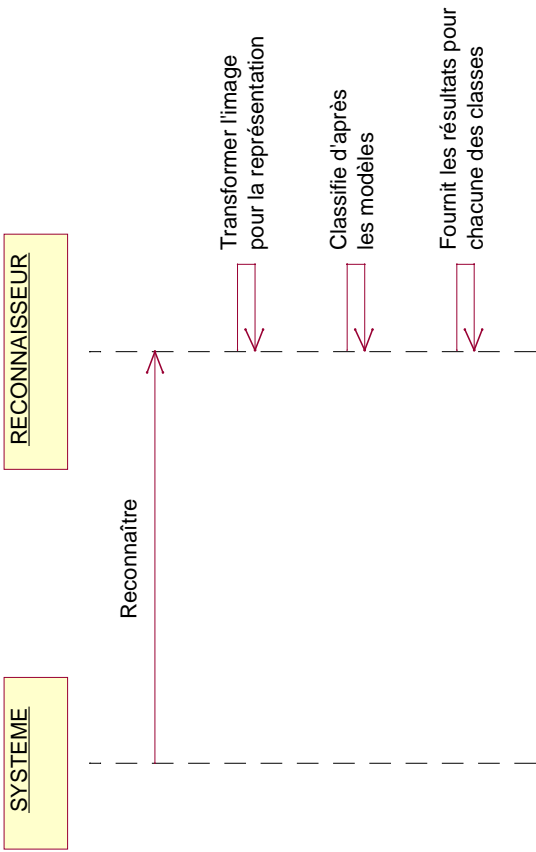
**FIGURE 6-6**  
Mode de reconnaissance: stylo scanner



Voici maintenant aux figures 6-7 et 6-8 les sous-diagrammes de séquence correspondant aux fonctions "prétraiter" et "reconnaître".

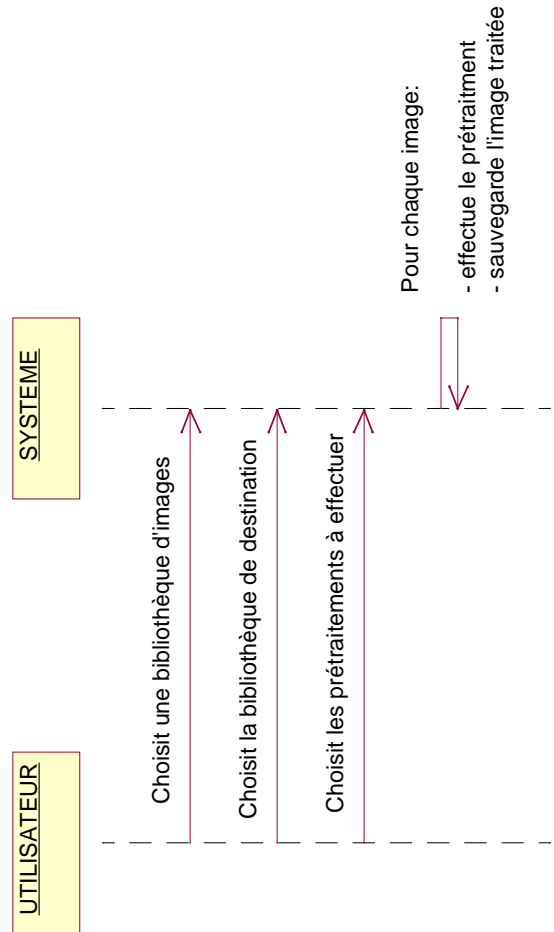
**FIGURE 6-7**  
Diagramme de séquence: Prétraiter





**FIGURE 6-8**  
Diagramme de séquence:  
Reconnaître

La figure 6-9 représente le diagramme de séquence d'une fonction particulière permettant le prétraitement d'une librairie entière d'images.



**FIGURE 6-9**  
Diagramme de séquence:  
Automatisation des prétraitements





Nous pouvons remarquer, dans le diagramme de classes à la figure 6-10, l'apparition de classes particulières dérivées de `CFileImage`. Nous appelons ce genre de classes des *class adapter* ou en français des *adapteurs de classes*. Ce nom vient du fait qu'elles contiennent des déclarations de méthodes communes dont leur contenu diffère.

Voici un petit exemple concret qui vaudra mieux qu'une grande théorie.

Supposons que je désire ouvrir un fichier d'image quelconque, puis lire son contenu avec une méthode `OpenFile()`. Chaque type de fichier contiendra évidemment sa propre structure de données. Il faut donc qu'à l'ouverture du fichier d'image avec la méthode `OpenFile()`, celui-ci lise la bonne structure de donnée correspondant au type de fichier.

extrait de code:

```
// déclaration des variables

CFileImage* m_pFileImage;           // pointeur sur la classe
                                   // de base
TypeFile m_TypeFile;                // type de fichier

...
// l'utilisateur choisit le fichier à ouvrir
...
// contrôle le type de fichier d'après son extension
...

switch (m_TypeFile)                 // choix multiple de types
{
    case MNIST:    m_pFileImage = new CFileMNIST; // crée une nouvelle instance
                                                           // d'une classe adaptateur
                   break;
    case PBM:     m_pFileImage = new CFilePBM;
                   break;
    ...
}

m_pFileImage ->OpenFile();           // appelle la fonction
                                   // correspondant au type
                                   // de fichier
```

Ces classes jouent donc le rôle d'interfaces entre la classe de base et les classes sous-jacentes.

La classe `CRecoSVM` est aussi une *class adapter*. Cela deviendrait très utile si nous voudrions par la suite intégrer un autre type de reconnaiseur.

Pour le reste des classes, il faut se référer au code C++ en annexe.

---

*Vous trouverez dans ce chapitre un petit mode d'emploi du logiciel de démonstration, permettant d'en découvrir les principales fonctionnalités.*

### **Environnement**

Le logiciel de démonstration fonctionne sur les systèmes d'exploitation Windows 95 et NT. Il est recommandé de posséder 64 Mo de RAM, car en fonctionnement normal le logiciel utilise 32 Mo de RAM. L'espace requis sur le disque dur est de 50 Mo. Un processeur de type Pentium au minimum et tournant à 100 MHz est conseillé pour obtenir un temps de reconnaissance correct.

### **Installation**

Pour l'instant, il n'existe pas d'installation automatique. Il faut donc installer le logiciel manuellement. Inutile de faire une crise d'angoisse, seule une manipulation est nécessaire:

Copiez simplement le répertoire "models" avec tout son contenu sur le premier niveau du disque c:\.

### **Lancement de l'application**

Exécutez le fichier "RecoSVM.exe". Ne vous inquiétez pas si la fenêtre d'application n'apparaît pas tout de suite, cela est dû au chargement en mémoire vive de tous les models indispensables à la reconnaissance.

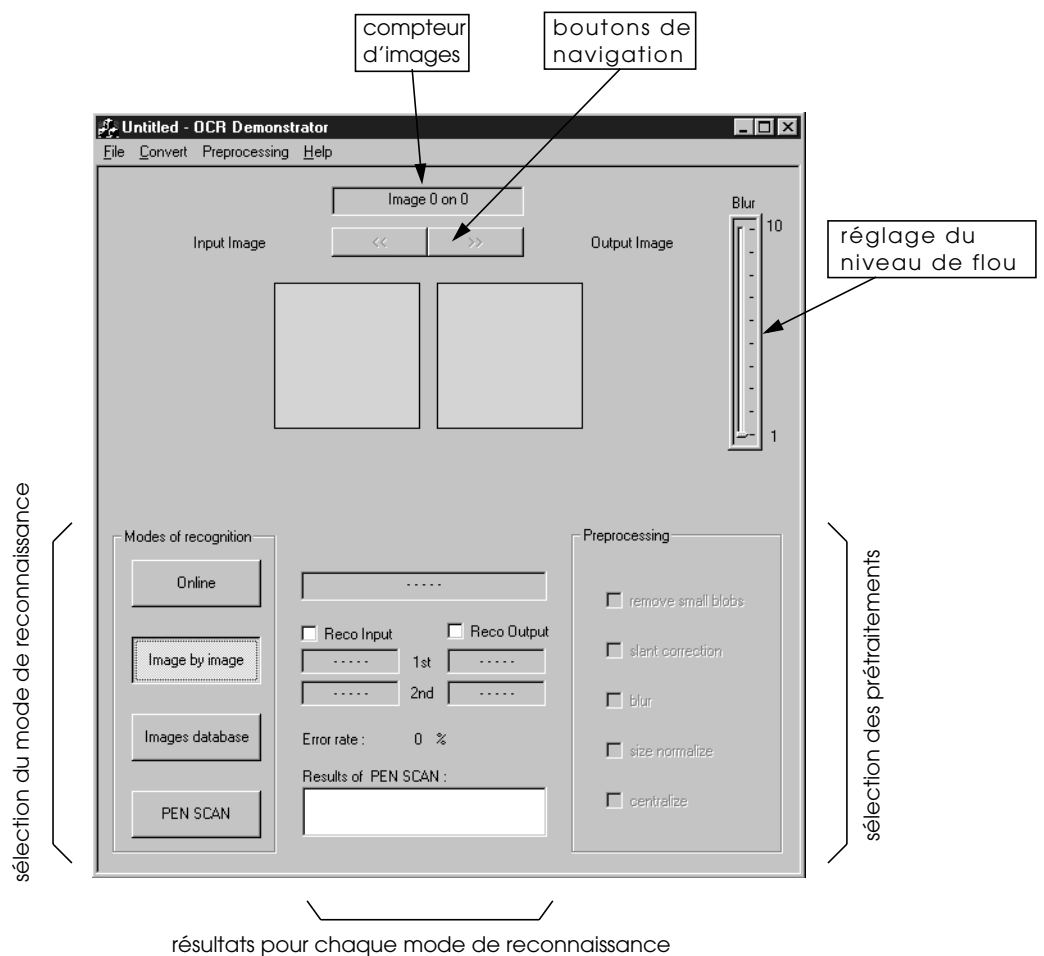
## Mode d'emploi

Comme nous l'avons vu au chapitre " 6. Modélisation UML ", il existe quatre modes de reconnaissance. Seuls deux modes sont actuellement fonctionnels:

- mode en ligne
- mode image par image

Avant de passer à une explication plus détaillée sur l'utilisation de ces deux modes de reconnaissance, voici une description du contenu de la fenêtre principale comme vous pouvez le voir à la figure 7-1.

**FIGURE 7-1**  
Fenêtre principale de l'application

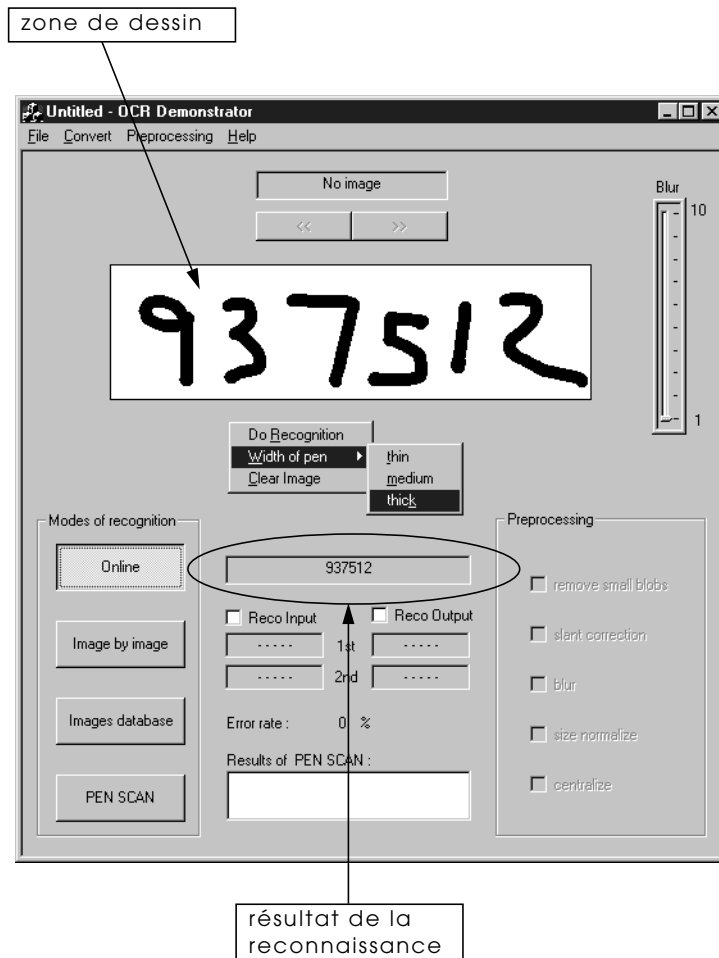


Pour commencer, vous devez sélectionner un mode de reconnaissance.

### Mode en ligne

Ce mode de reconnaissance est le plus démonstratif, car vous pouvez directement dessiner dans la zone prévue à cet effet comme l'illustre la figure 7-2.

**FIGURE 7-2**  
Mode de reconnaissance  
"en ligne"



A l'aide de la touche droite de la souris, vous avez accès aux fonctionnalités suivantes:

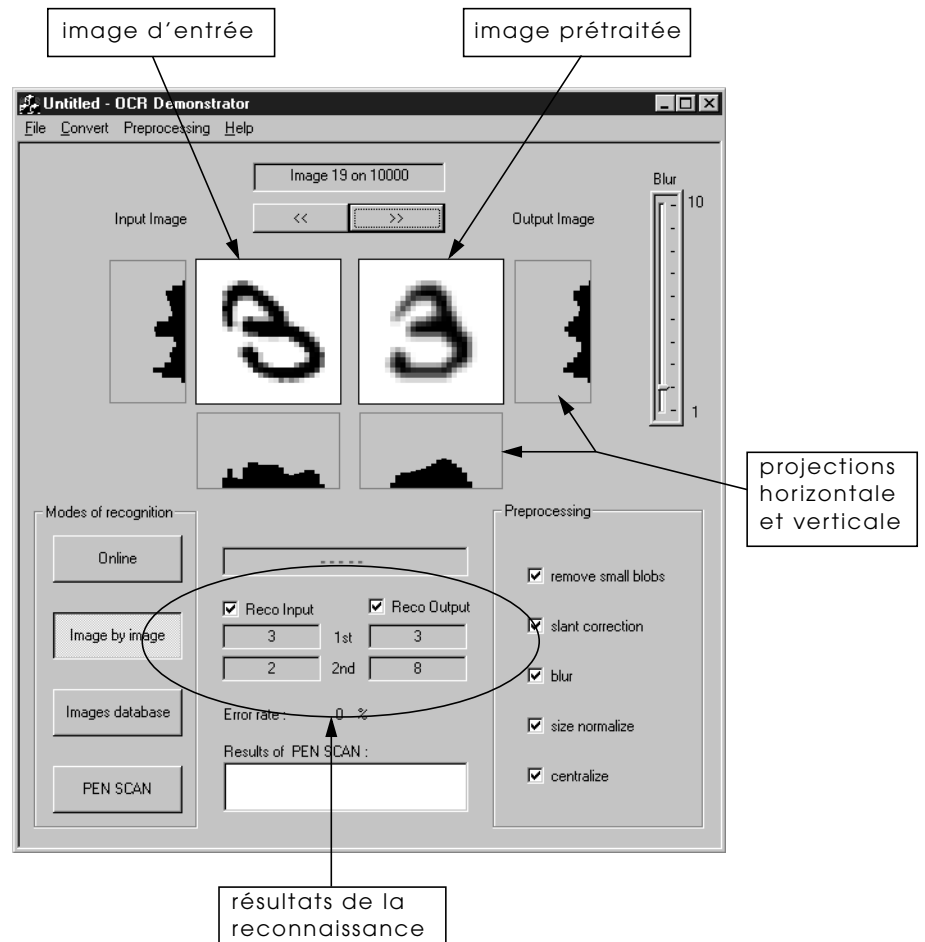
- effectuer la reconnaissance des chiffres dessinés
- changer l'épaisseur du stylo
- effacer l'image

Ce mode a été créé pour simuler les images en provenance du stylo-scanner. La hauteur de la zone de saisie correspond exactement à la résolution du stylo-scanner qui est de 102 pixels.

### Mode image par image

Vous devez charger une bibliothèque d'images de test MNIST en sélectionnant "Open" dans le menu "File". Vous pouvez maintenant visualiser toutes les images de la bibliothèque à l'aide des boutons de navigation. Vous avez la possibilité d'effectuer sur chaque image d'entrée différents prétraitements qu'il suffit de cocher. L'image de droite correspond au résultat après le prétraitement.

**FIGURE 7-3**  
Mode de reconnaissance "image par image"



La partie la plus intéressante réside dans la reconnaissance du chiffre contenu dans l'imagette. Cochez simplement la case adéquate. Vous verrez alors apparaître deux chiffres. Le premier est celui qui a obtenu le meilleur score de reconnaissance et le deuxième le second score.

Remarquez que les résultats diffèrent pour une imagette avant et après son prétraitement.

---

*SECTION IV*

*Expériences*

---

*Ce chapitre prouve par des tests l'efficacité des prétraitements d'images sur la reconnaissance.*

Pour effectuer nos tests, nous avons utilisé deux bases de données d'imagettes. L'une, déjà existante, contient 10'000 chiffres et l'autre à dû être créée.

## Constitution d'une base de données

Afin de construire ma propre base de données, j'ai distribué des feuilles qui ont été remplies par onze personnes différentes. Ces feuilles comportent cinq numéros postaux qui possèdent eux-mêmes huit styles d'écritures différents, comme le montre la figure 8-1.

**FIGURE 8-1**  
Différents styles d'écriture

	normal	italique	espacé	serré
grand	2954	2954	2 9 5 4	2954
petit	2954	2954	2 9 5 4	2954

Pour les tests, seuls les styles "grand" et "espacé" ont été pris en considération étant donné qu'il nous est possible de segmenter uniquement des caractères qui ne se chevauchent pas comme nous l'avons vu au chapitre "3. Segmentation".

Voici la procédure que nous avons suivie pour constituer notre base de données d'imagettes à partir des feuilles brutes.



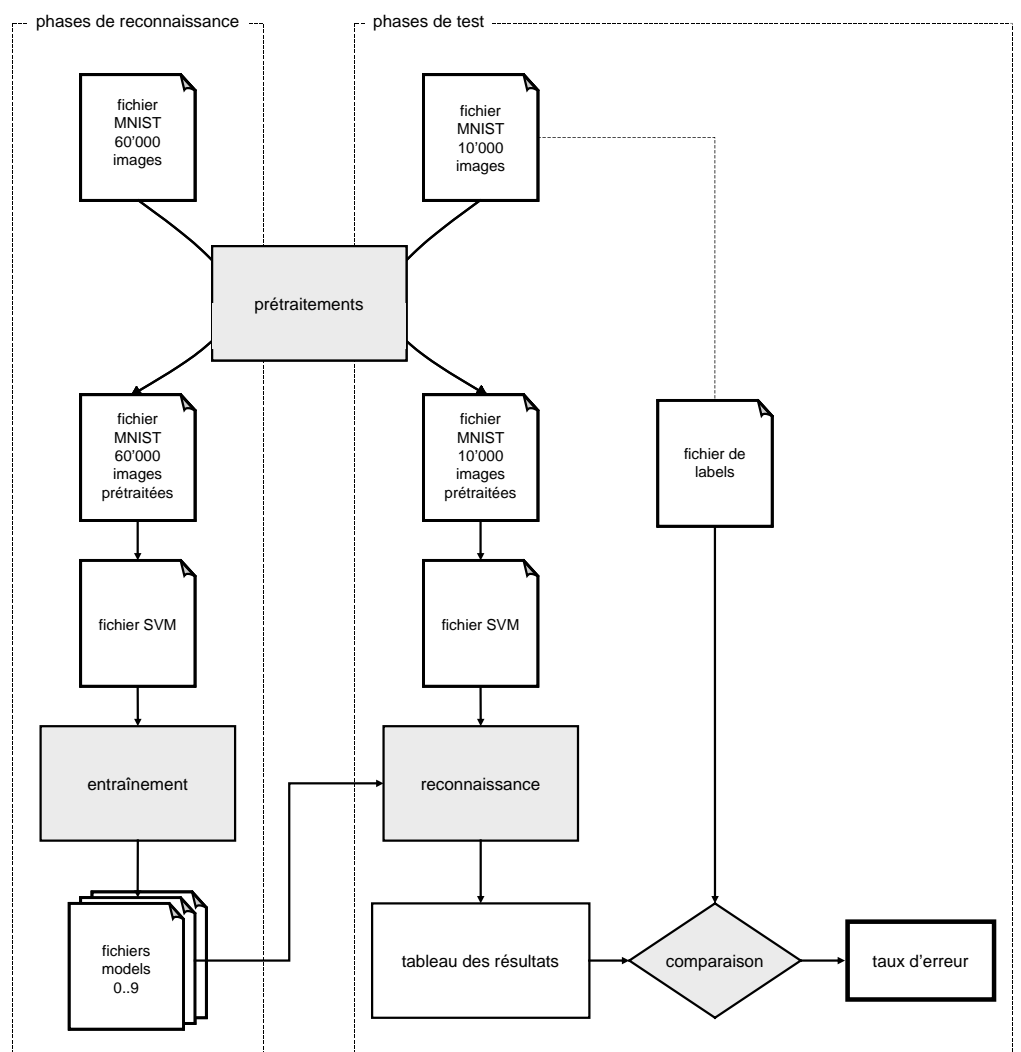
Il a fallu, dans un premier temps, découper chacun des codes postaux de la page, ensuite les trier par styles, segmenter chaque chiffre, les normaliser et enfin les mémoriser dans une librairie d'images au format MNIST (voir au chapitre 9).

C'est le seul endroit où l'histogramme vertical (projection horizontale de l'image) est intervenu. Il nous a permis de segmenter les lignes.

## Tests

La figure 8-2 illustre très bien le mécanisme de test.

**FIGURE 8-2**  
Test de l'efficacité des  
prétraitements



Il y a peut être une chose à souligner dans ce diagramme, c'est que les images d'entraînement doivent passer par les mêmes prétraitements que les images de tests.

## Résultats

TABLE 8-1 : Résultats des tests pour 220 chiffres

traitement	taux d'erreur
aucun	11.36 %
correction de l'inclinaison + centrage	11.36 %
correction de l'inclinaison + normalisation de la taille <sup>(2)</sup> + centrage	12.73 %

Les taux d'erreur de notre base de données "artisanale" sont très élevés. La raison principale repose sur le fait que le reconnaisseur a subi un entraînement avec des chiffres qui ont été écrits par des américains alors que la base de données contient des chiffres écrits par des européens. Il y a des différences notables pour les chiffres: 1, 7 et 9.

TABLE 8-2 : Résultats des tests pour 10'000 chiffres (MNIST)

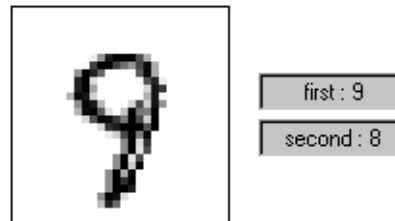
entraînement	test	taux d'erreur
aucun prétraitement	aucun prétraitement	1.36 %
<b>correction de l'inclinaison + centrage</b>	<b>correction de l'inclinaison + centrage</b>	<b>1.21 %</b>
	aucun prétraitement	1.48 %
	correction de l'inclinaison + centrage + flou gaussien 3*3	4.53 %
correction de l'inclinaison + normalisation de la taille <sup>(1)</sup> + centrage	correction de l'inclinaison + normalisation de la taille <sup>(1)</sup> + centrage	1.31 %
	aucun prétraitement	9.12 %
correction de l'inclinaison + normalisation de la taille <sup>(2)</sup> + centrage	correction de l'inclinaison + normalisation de la taille <sup>(2)</sup> + centrage	1.25 %

TABLE 8-2 : Résultats des tests pour 10'000 chiffres (MNIST)

entraînement	test	taux d'erreur
correction de l'inclinaison + flou gaussien 3*3 + normalisation de la taille <sup>1</sup> + centrage	correction de l'inclinaison + flou gaussien 3*3 + normalisation de la taille <sup>1</sup> + centrage	1.31 %
suppression des taches + correction de l'inclinaison + flou gaussien 2*2 + centrage	suppression des taches + correction de l'inclinaison + flou gaussien 2*2 + centrage	1.94 %

Les résultats des tests avec la librairie d'images MNIST sont excellents. Le plus faible taux d'erreur descend à 1.21% ! Il ne peut être abaissé à zéro en raison de certains chiffres dont nous n'arrivons pas nous-mêmes à reconnaître.

**FIGURE 8-3**  
Confusion entre un 8 et un 9



*Vous découvrirez dans ce chapitre la structure des formats de fichiers utilisés par le logiciel. Pour des raisons de temps, nous n'avons pas spécialement choisi les formats d'image les plus courants, mais les plus rapides à mettre en oeuvre.*

## **“Big-endian” et “little-endian”**

---

Avant de décrire les différents formats de fichier, il est important de connaître leur structure fondamentale. Dans un fichier, comme dans une mémoire, les données peuvent être organisées de différente manière. En fait, il existe deux principales architectures: *big-endian* et *little-endian*. Seul la direction des octets dans un mot les différencie.

L'architecture *big-endian* contient l'octet de poids fort à la plus basse adresse, tandis que l'architecture *little-endian* contient l'octet de poids faible à la plus basse adresse. Voici un exemple avec une valeur de 32 bits 0x12345678 en hexadécimal qui serait enregistrée dans la mémoire:

<b>Adresse</b>	<b>00</b>	<b>01</b>	<b>02</b>	<b>03</b>
big-endian	12	34	56	78
little- endian	78	56	34	12

### **Avantages et inconvénients**

Il y a des avantages et des inconvénients pour chaque méthode. Les principaux avantages du *big-endian* sont:

- facilité accrue pour la lecture d'une suite d'octets dans une zone mémoire.
- manipulations plus aisées pour la plupart des fichiers graphiques, car ceux-ci utilisent généralement des mots supérieurs au byte.

Et voici quelques inconvénients de l'architecture *big-endian*:

- si nous lisons une valeur de mauvaise taille, le résultat sera toujours faux contrairement au *little-endian* qui lui, peut parfois, donner un résultat correct pour autant que cette valeur soit ne soit pas trop grande.
- la plupart des architectures *big-endian* ne permettent pas à des mots d'être écrit dans une adresse impaire.

Les avantages et inconvénients du *little-endian* sont fondamentalement opposés à ceux cités ci-dessus.

## Fichiers Images

### MNIST

**extension:** \*.idx3-ubyte

Un fichier MNIST peut contenir une ou plusieurs images de tailles identiques. Il est de type binaire dont la structure est représentée au tableau 9-1. Les pixels des images sont organisés en lignes. Les valeurs des pixels sont comprises entre 0 et 255. 0 signifie le fond (blanc) et 255 le premier plan (noir).

TABLE 9-1 : Structure d'un fichier image MNIST

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000803 (2051)	numéro magic
0004	32 bit integer	10000	number of images
0008	32 bit integer	28	number of rows
0012	32 bit integer	28	number of columns
0016	unsigned byte	??	pixel
0017	unsigned byte	??	pixel
.....	unsigned byte	??	pixel
xxxx	unsigned byte	??	pixel

Ce type de fichier est très pratique pour sauvegarder des bibliothèques entières d'images. Dans notre cas, les bases de données images d'entraînement et de test sont sauvegardées dans ce format comme vous pouvez le voir ci-dessous.

fonction	nom du fichier	contenu
entraînement	train-images.idx3-ubyte	60'000 images normalisées
test	t10k-images.idx3-ubyte	10'000 images normalisées

En ce qui concerne la normalisation des images, je vous renvoie au chapitre " 4. Prétraitements ".

## PBM

**extension:** \*.pbm

Ce type de fichier est binaire et vous trouverez la structure de son en-tête au tableau 9-2. Il présente l'avantage d'être extrêmement facile à lire et à écrire, de plus beaucoup de logiciels de conversion d'images le supportent.

**TABLE 9-2 :** Structure de l'en-tête d'un fichier image PBM (binaire)

[valeur]	[description]
P4	numéro magic
–	espace blanc
largeur	largeur de l'image en pixels
–	espace blanc
hauteur	hauteur de l'image en pixels
–	espace blanc

L'espace blanc après la taille de l'image peut être seulement un simple caractère , typiquement une nouvelle ligne. Après cela viennent les bits, ligne par ligne de haut en bas. Dans chaque ligne, les bits sont enregistrés de gauche à droite en octets, le bit de poids fort se situe à gauche. Chaque ligne commence dans un nouvel octet, même si la largeur de l'image n'est pas un multiple de huit.

## SVM

**extension:** aucune

Ce type de fichier peut comporter un nombre indéterminé d'images au format ASCII<sup>1</sup>. Il a été inventé pour l'utilisation du reconnaisseur SVM (voir au chapitre 5). A mon avis, le choix du format ASCII est une tare, car la taille d'un fichier peut rapidement atteindre des proportions énormes si celui-ci comporte plusieurs centaines d'images.

Voici un extrait d'un fichier SVM utilisé pour un test:

```
# SVM file created by Eric Grand date : Friday, December 17, 1999
0 206:0.301961 207:0.549020 208:0.549020 233:0.796078 234:0.996078
0 233:0.301961 234:0.549020 235:0.549020
```

Le fichier commence par des lignes de commentaires précédées du caractère "#". Ensuite, chaque ligne représente une image dont les pixels sont organisés par colonnes de haut en bas et de gauche à droite.

Au début de chaque ligne, se trouve un entier qui peut varier suivant l'utilisation du fichier. En effet, si ce fichier est utilisé pour un test, cet entier prendra la valeur 0 comme dans l'exemple ci-dessus et s'il est utilisé pour l'entraînement il vaudra:

-1	: si cette image n'appartient pas à la classe à entraîner
+1	: si elle appartient à la classe à entraîner

Après cela, nous trouvons pour chaque pixel de l'image sa position dont la numérotation commence à 1 suivie du caractère ":". Ensuite vient sa valeur qui représente, dans notre cas, son niveau de couleur. Ici la valeur d'un pixel est comprise entre 0 (fond) et 1 (premier plan).

Il est important de noter que **seuls les pixels contenant l'information sont sauvegardés**. Par contre, aucune information n'est donnée quant à la taille des images car le reconnaisseur SVM peut très bien classifier des éléments autres que des images.

---

1. American Standard Code for Information Interchange

## Fichiers labels

### MNIST

**extension:** \*.idx1-ubyte

Ce fichier est de type binaire et comporte tous les labels nécessaire pour l'entraînement et les tests du reconnaisseur SVM. En effet, pour chaque objet entraîné devra correspondre la classe à laquelle il correspond. Pour notre application, ce fichier contiendra des entier compris entre 0 et 9. Voici au tableau 9-3 la structure d'un fichier.

TABLE 9-3 : Structure d'un fichier label MNIST

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000803 (2051)	numéro magic
0004	32 bit integer	10000	number of images
0008	unsigned byte	??	label
0009	unsigned byte	??	label
.....	unsigned byte	??	label
xxxx	unsigned byte	??	label



---

L'objectif de ce travail de diplôme a été atteint. Les attentes et exigences de l'institut IDIAP ont été satisfaites. Le démonstrateur fonctionne merveilleusement bien, intégrant comme prévu le reconnaiseur SVM.

## *Améliorations possibles*

---

Comme nulle chose n'est parfaite, voici quelques améliorations envisageables pour notre démonstrateur:

- Créer une aide intégrée au logiciel.
- Conversion des fichiers modèles. Actuellement les vecteurs support sont sauvegardés sous forme ASCII et prennent une très grande place mémoire, ce qui réduit la portabilité du logiciel. Il conviendrait de créer une petite interface qui s'occuperait de transformer ce fichier sous forme binaire.

### **Bug MFC**

Lorsque l'on sélectionne dans un sélecteur de fichiers un grand nombre de fichiers, ceux-ci apparaissent correctement dans la zone d'édition mais seulement les 255 premiers caractères sont réellement stockés dans le buffer `CFileDialog::m_ofn.lpstrFile`. Essayez par exemple de charger une vingtaine d'images à la fois dans un programme de retouche d'images!

Sion, le 18 janvier 2000

---

## Bibliographie

---

- [1] Chapman Davis, "*Visual C++ 6*", Simon & Schuster Macmillan, ISBN 2-7440-0552-5
- [2] Conger Jim, "*Microsoft Foundation Class Primer*", Waite Group Press, 1993, ISBN 1-878739-31-X.
- [3] Davies E. R. , "*Machine Vision*", Academic Press, 1997, ISBN 0-12-206092-X
- [4] Desmadril Michel, "*Programmer en C++*", Eyrolles, 1990.
- [5] Fowler Martin, "*UML distilled*", Addison Wesley, 1997, ISBN 0-201-32563-2.
- [6] Kay David C. , Levine John R., "*Graphics File Formats*", TAB Books, 1992, ISBN 0-8306-3060-0
- [7] Klaus Berthold, Horn Paul , "*Robot Vision*", The MIT Press, 1986, ISBN 0-262-08159-8
- [8] Kreyszig Erwin, "*Advanced Engineering Mathematics*", John Wiley, 1993, ISBN 0-471-59989-1
- [9] Kruglinski David J. , "*Visual C++ 6.0*", Microsoft Press, ISBN 2-84082-374-8
- [10] Marr David, "*Vision*", Freeman and Compny, 1982, ISBN 0-7167-1567-8
- [11] Micheloud Marylène, Rieder Medard, "*Programmation orientée objets en C++*", Presses polytechniques et universitaires romandes, ISBN 2-88074-334-6.

- [12] Powel Douglass Bruce, "*Real-Time UML*", Addison Wesley, 1998, ISBN 0-201-32579-9.
- [13] Quatrani Terry, "*Visual Modeling with Rational Rose and UML*", Addison Wesley, 1998, ISBN 0-201-31016-3.
- [14] Toumatzet Jean-Jacques, "*Traitement de l'image sur micro-ordinateur*", Sybex, 1987, ISBN 2-7361-0291-6
- [15] Ussain Zahid, "*Digital Image Processing*", Ellis Horwood, 1991, ISBN 0-13-213281-8
- [16] W. Kernighan Brian, Ritchie Dennis M. , "*The C Programming Language*", Prentice Hall, 1988, ISBN 0-13-110362-8.

---

## *ANNEXE A*

# *Codes sources*

---

*Voici le CD-ROM contenant l'intégralité des codes sources ainsi qu'une version exécutable du démonstrateur.*