

UNIVERSITÉ PARIS 8

COURS DE TRAITEMENT DE SIGNAL ET D'IMAGE

FaceWorker

*Panchalingamoorthy Gajenthran
Hu Sacha*

16 avril 2019

Table des matières

1	Introduction	1
2	Filtres	2
2.1	Nuance de gris	2
2.2	Egalisation d'histogramme	3
2.3	Sobel	3
2.4	Prewitt	4
2.5	Laplacien	5
2.6	Sharpen	5
2.7	Moyenneur	5
2.8	Médian	6
2.9	Inversion des couleurs	6
2.10	Sepia	7
2.11	Luminosité	7
2.12	Tourbillon	8
2.13	Bilateral	8
2.14	Interpolation bilinéaire	9
2.15	Rotation	9
2.16	Miroir	10
2.17	Binaire	10
2.18	Complément binaire	11
3	IRecognition : Eigenfaces	11
3.1	Dataset : base de données d'apprentisage	12
3.2	Testset : les images à tester	13
3.3	Reconnaissance et identification	13
4	VRecognition : SOM	14

1 Introduction

Notre projet s'appelle FaceWorker. Une application qui est capable d'appliquer des filtres à une photo, de reconnaître un visage sur une photo, ou même de reconnaître une personne en video ou en live.

L'application se présente ainsi : 3 boutons pour accéder aux trois fonctionnalités principales que propose FaceWorker : FaceFilter, IRecognition (Eigenfaces) ou VRecognition (reconnaissance faciale en temps réel).

La première permet de charger une image, de sélectionner un ou plusieurs filtres via une interface très intuitive qui propose de nombreux filtres dans

un menu déroulant et accessibles via des boutons sur le côté droit de l'image. Lorsque l'image est chargée, on peut directement visualiser les modifications apportées sur l'image. Lorsque les modifications sont faites, on peut sauvegarder l'image.

La deuxième, permet de charger une image et de détecter le visage puis en parcourant une base de données de visages pré-enregistrées, reconnaître à qui appartient cette photo.

La dernière option, est malheureusement encore en cours d'implémentation car elle s'est avérée plus complexe que prevue. Nous allons tout de même expliquer son fonctionnement. Elle permet de reconnaître en temps réel le visage d'une personne et de lui associer un nom.

Toute cette interface a été réalisée à l'aide de la toolbox Guide qui permet de créer de très bonnes interfaces graphiques.

2 Filtres

Cette partie de notre application permet d'appliquer toutes sortes de filtres. Nous avons implémenté à la main, 17 filtres différents dont les filtres passe haut, passe bas ou des filtres utilisant les histogrammes mais pas seulement (une liste exhaustive est donnée plus bas). Il est également possible de cumuler différents filtres et ainsi de rajouter un second filtre par-dessus un premier filtre. Enfin, lorsque tous les filtres souhaités ont été rajoutés, il est possible d'enregistrer l'image, ce qui crée un nouveau fichier utilisable par la suite.

L'ensemble des filtres se trouvent dans le fichier *fwFilter.m*. A noter que tous les fichiers métiers comporteront le préfixe **fw** pour FaceWorker.

Il existe 17 filtres différents dans notre application : nuance de gris, binaire, complément binaire, égalisation d'histogramme, Sobel, Sepia, Prewitt, luminosité, Laplacien, Sharpen, moyenneur, médian, inversion de couleurs, tourbillon, bilatéral, interpolation bilinéaire et miroir.

2.1 Nuance de gris

Ce filtre (`applyGrayscale`) permet de faire passer une image en couleur en une image composée de nuance de gris. D'abord, nous vérifierons si l'image sélectionnée est bien une image en couleur puis appliquer notre algorithme. Matlab possède déjà la fonction `rgb2gray` mais nous avons décidé de créer la notre qui ajoute la vérification précédemment expliquée.

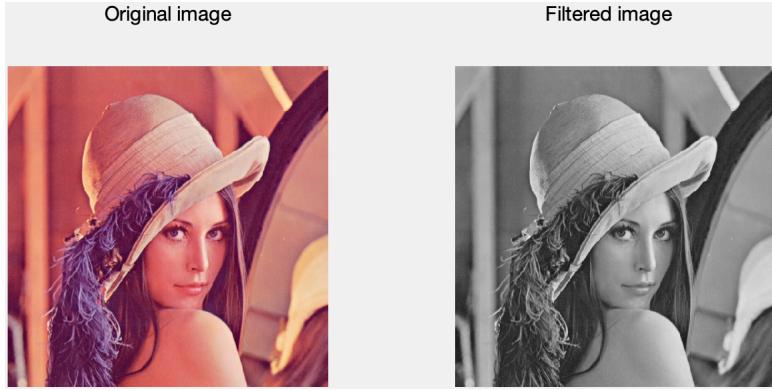


FIGURE 1 – Nuance de gris

2.2 Egalisation d'histogramme

L'égalisation d'histogramme (`applyHistEq`) va nous permettre d'ajuster le contraste d'une image en s'aidant de l'histogramme de celle-ci. Pour réaliser ce filtre, trois étapes sont à noter : la création de l'histogramme de l'image, le calcul de l'histogramme cumulée grâce à l'histogramme créé précédemment et enfin l'application de l'égalisation de l'histogramme pour chaque pixel de l'image.

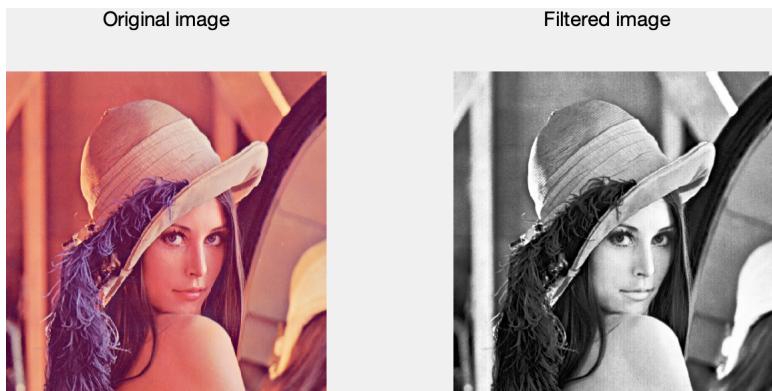


FIGURE 2 – Egalisation d'histogramme

2.3 Sobel

Le filtre de Sobel (`applySobel`) est un filtre de passe-haut car il permet de détecter les contours de l'image en utilisant des matrices de convolution. Ainsi on obtiendra le gradient X pour une image I et le gradient Y pour une image I. La combinaison des deux gradients permet d'obtenir une approximation de la

norme du gradient à l'aide de la formule suivante : $G = \sqrt{G_x^2 + G_y^2}$; Notre algorithme est plutôt lent vu qu'il s'agit de faire des boucles imbriquées les unes sur les autres et également plusieurs calculs (c'est pour cela que nous avons remplacé $G = \sqrt{G_x^2 + G_y^2}$; par $G = \text{abs}(G_x) + \text{abs}(G_y)$; afin de réduire le temps d'exécution). Une alternative aurait été d'utiliser la méthode `conv2` offerte par Matlab.

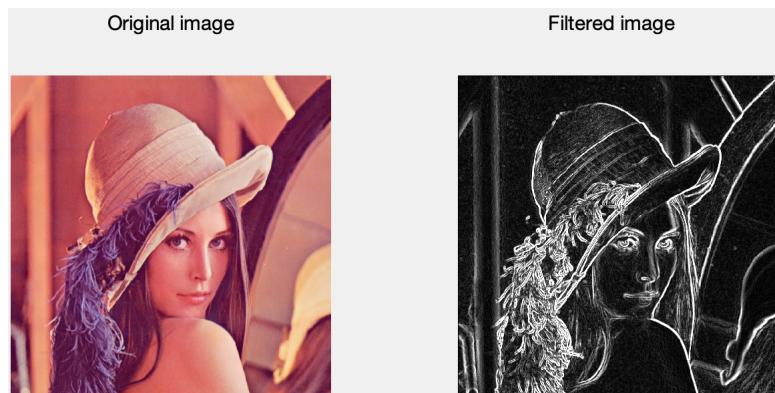


FIGURE 3 – Filtre de Sobel

2.4 Prewitt

De la même manière que le filtre de Sobel, le filtre de Prewitt (`applyPrewitt`) va nous permettre de détecter les contours de l'image. Seulement, les matrices qui vont convoluer vont être différentes des matrices du filtre de Sobel donc le rendu concernant les contours seront différents comme nous pouvons le voir dans l'image ci-dessous.

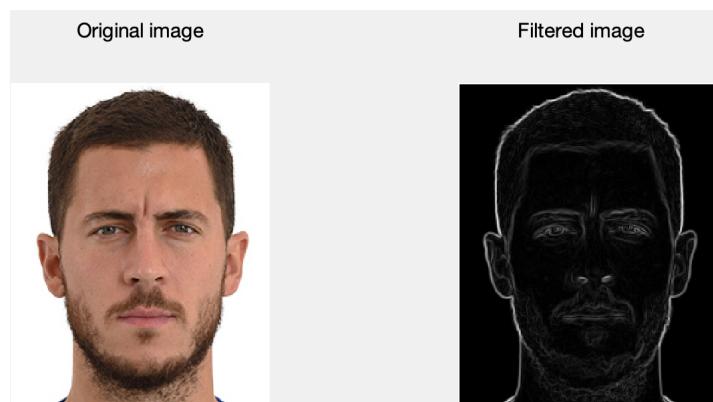


FIGURE 4 – Filtre de Prewitt

2.5 Laplacien

Exactement comme les deux autres cités auparavant (Sobel et Prewitt), ce filtre (`applyLaplacian`) est également un filtre passe-haut.

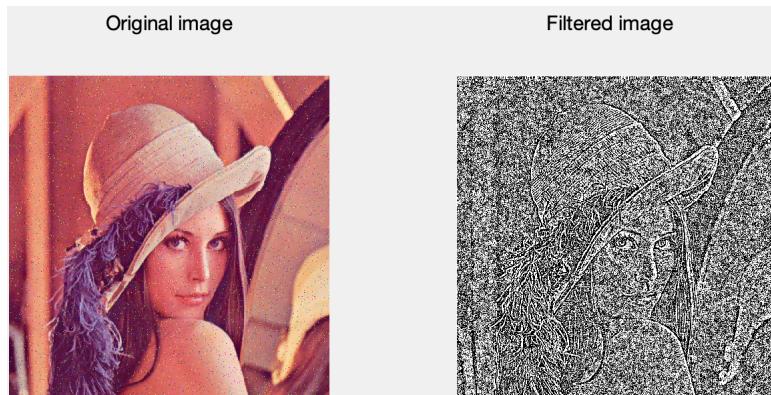


FIGURE 5 – Filtre laplacien

2.6 Sharpen

Pour finir sur les filtres passe-haut, le filtre Sharpen permet également de calculer le contours en utilisant des gradients différents.

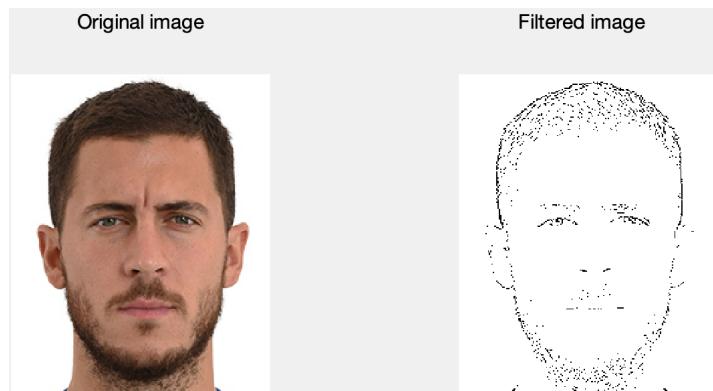


FIGURE 6 – Filtre Sharpen

2.7 Moyenneur

Le filtre moyenneur (`applyAverage`) est un filtre de passe-bas : en effet, il réduit les détails (mais aussi le bruit) et permet d'obtenir une image plus lisse. Il consiste à remplacer le pixel sélectionné par la moyenne de ses pixels

voisins. Une amélioration de ce filtre serait le filtre Gaussian car il offre un meilleur lissage et moins de bruit.

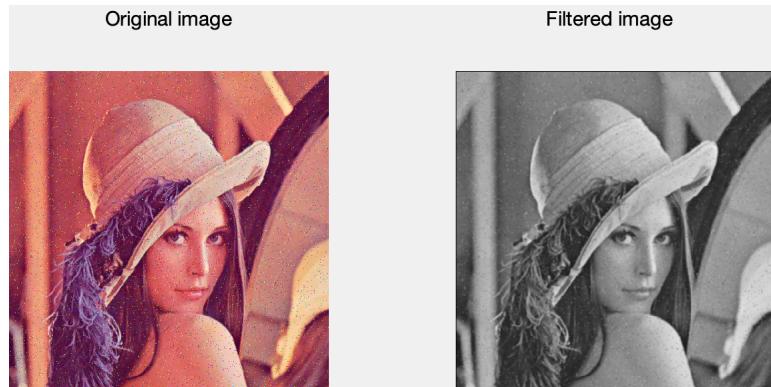


FIGURE 7 – Filtre moyenneur

2.8 Médian

Le filtre médian (`applyMedian`) est une opération non linéaire, plutôt utilisé pour réduire les bruits d'une image. Le principe est assez simple : sur un pixel sélectionné, on applique la médiane de tous les pixels voisins.

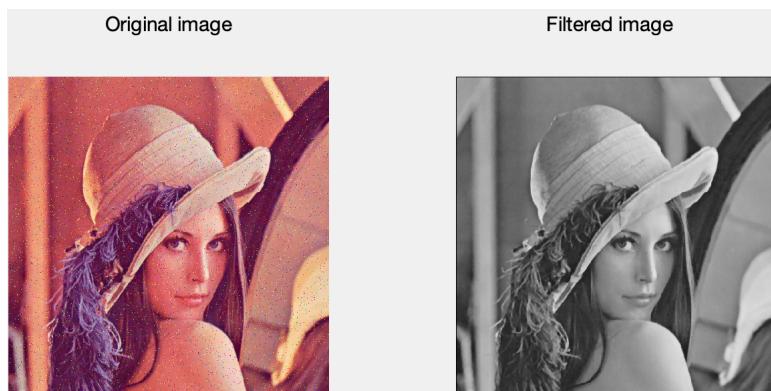


FIGURE 8 – Filtre médian

2.9 Inversion des couleurs

Comme son nom l'indique, ce filtre (`applyInvert`) va nous permettre d'inverser les couleurs d'une image. L'implémentation de l'inversion des couleurs est assez simple : il suffit de prendre la valeur maximale de chaque couleur à savoir 255 et de la soustraire par la couleur actuelle.

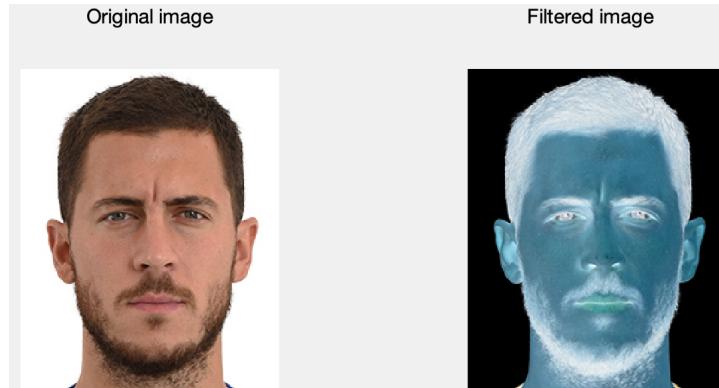


FIGURE 9 – Inversion des couleurs

2.10 Sepia

Le filtre Sépia (`applySepia`) va nous permettre d'apporter une teinte plus ancienne à notre image et offrir une photo plus vintage (avec une domination de la couleur jaune). Pour chaque couleur de l'image, on va additionner les 3 couleurs (RGB) en appliquant un coefficient différent pour chaque couleur. Par exemple pour le nouveau rouge du Sépia $NR = (R * 0.393) + (G * 0.769) + (B * 0.189)$;

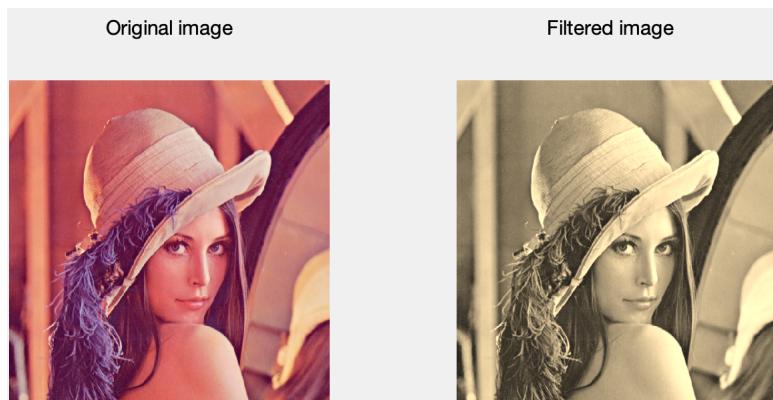


FIGURE 10 – Filtre Sépia

2.11 Luminosité

Améliorer la luminosité (`applyBrightness`) se réalise de manière assez simple : il suffit de décaler l'histogramme $I'(i, j) = I(i, j) + b$ où b est le facteur de décalage. Dans notre application, pour gérer le mieux possible

la luminosité de l'image, un slide-bar est mis à disposition représentant le facteur de décalage (entre 0 et 100)

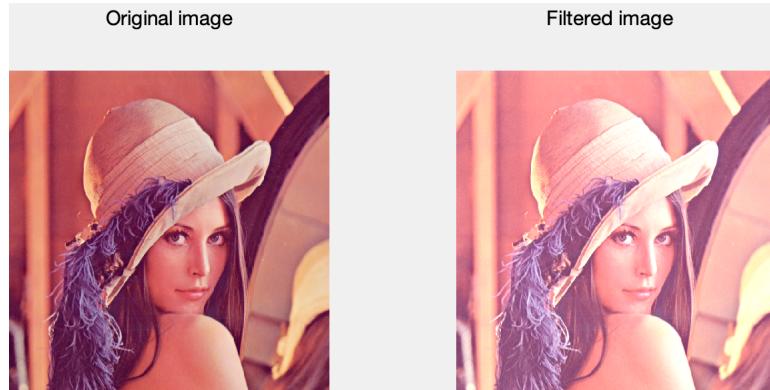


FIGURE 11 – Luminosité

2.12 Tourbillon

Le filtre Tourbillon (`applySwirl`) va créer une sorte de spirale déformant l'image en plusieurs morceaux donnant l'impression que l'image se fait aspirer par un tourbillon. On utilisera les fonctions `cos` et `sin` pour réaliser ce filtre. Comme pour la luminosité, on bénéficiera d'un slide-bar gérant le degré de transformation de l'image.

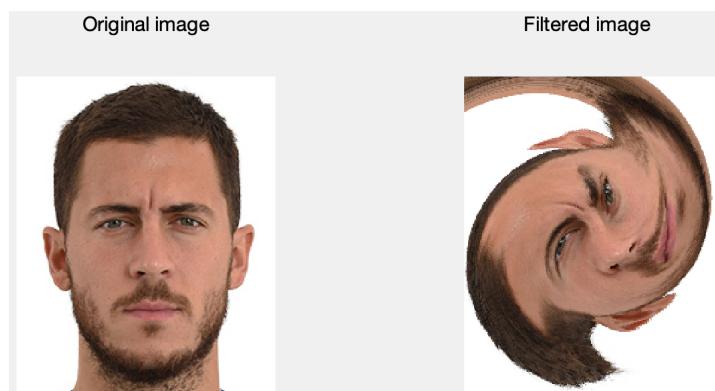


FIGURE 12 – Filtre swirl

2.13 Bilateral

Le filtre bilateral (`applyBilateralRGB`) est un filtre de passe-bas permettant de réduire les bruits tout en rendant l'image plus lisse. Il peut être utilisé

pour transformer une image réel en une image en cartoon par exemple.

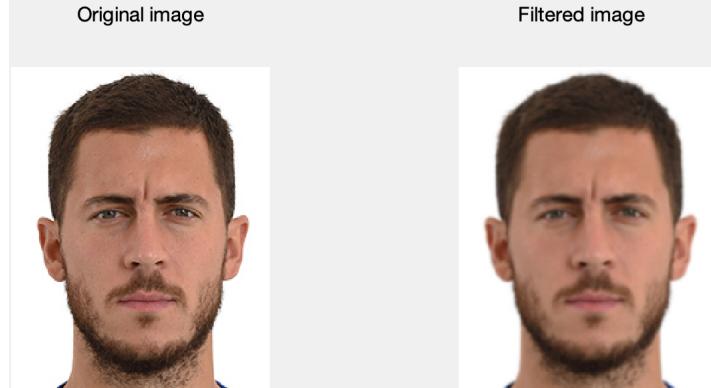


FIGURE 13 – Filtre bilatéral

2.14 Interpolation bilinéaire

L'interpolation bilinéaire `applyInterpolationB` consiste à agrandir les dimensions de l'image par copie de pixels. Le principe est assez simple, chaque pixel de la nouvelle image sera de la forme : $I'(i, j) = I(i/\text{scale}, j/\text{scale})$ où `scale` correspond au coefficient séparant l'échelle de départ et la nouvelle échelle. On peut très bien créer deux `scale` différents (un pour la colonne et un autre pour la ligne).

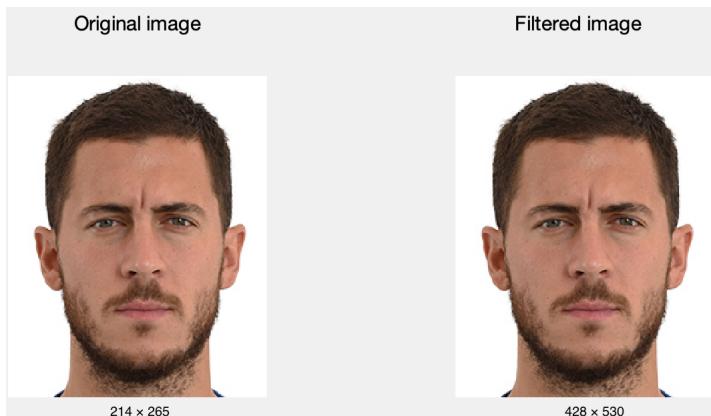


FIGURE 14 – Interpolation bilinéaire

2.15 Rotation

Le filtre de rotation `applyRotation` permet de réaliser une rotation de l'image selon un angle de rotation `a` autour de son centre en appliquant pour

chaque pixel la formule suivante :

$$x = W/2 + (x_0 - W/2) * \cos(a) - (y_0 - H/2) * \sin(a)$$

$$y = H/2 + (y_0 - H/2) * \cos(a) + (x_0 - W/2) * \sin(a)$$

Notre formule est légèrement différente de celle-ci mais réalise la même rotation.

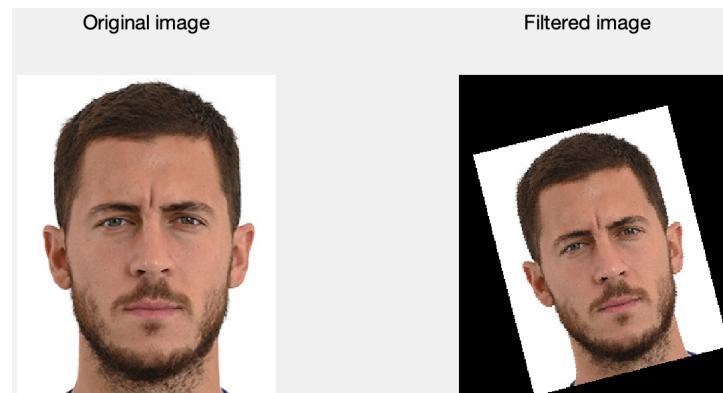


FIGURE 15 – Rotation

2.16 Miroir

Le miroir `applyMirror` permet de retourner l'image de la gauche vers la droite. Concernant l'implémentation, soient $w(j) = 1 + j$ et $w'(j) = width - j$, on a alors $I'(i, w(j)) = I(i, w'(j))$

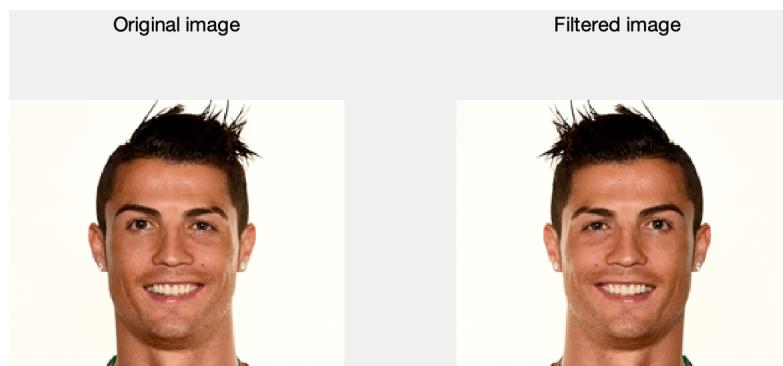


FIGURE 16 – Filtre Miroir

2.17 Binaire

Le filtre binaire (`applyBinary`) consiste à représenter l'image avec seulement des valeurs composées de 0 et de 1 et donc une image composée de noir

et blanc. Pour cela, on transforme l'image en gris si ce n'est pas le cas, puis on calcule moyenne de la valeur de gris (qui sera notre seuil) et enfin toutes les valeurs au-dessus de ce seuil seront représentées par des 1 et le reste par des 0.



FIGURE 17 – Filtre binaire

2.18 Complément binaire

En admettant que notre image est en binaire (sinon on applique le filtre binaire vu ci-dessus), on peut ainsi appliquer son complément (`applyComplementBinary`), c'est-à-dire transformer tout ce qui est en noir, en blanc et tout ce qui est en blanc, en noir, ce qui revient donc à inverser les 0 en 1 et vice-versa.



FIGURE 18 – Filtre complément binaire

3 IRecognition : Eigenfaces

Cette fonctionnalité permet de reconnaître une image parmi une base de données à l'aide de la méthode Eigenfaces. Les eigenfaces sont un ensemble

de vecteurs propres utilisés dans le domaine de la vision artificielle afin de résoudre le problème de la reconnaissance du visage humain. Le recours à des eigenfaces pour la reconnaissance a été développé par Sirovich et Kirby (1987) et utilisé par Matthew Turk et Alex Pentland pour la classification de visages. Cette méthode est considérée comme le premier exemple réussi de technologie de reconnaissance faciale. Ces vecteurs propres sont dérivés de la matrice de covariance de la distribution de probabilité de l'espace vectoriel de grande dimension des possibles visages d'êtres humains (source : Wikipedia). Cette méthode va permettre d'identifier le visage en analysant l'ensemble du visage, on parle alors d'analyse en composant principale (PCA) se différenciant ainsi des méthodes géométriques ou locales qui eux se concentrent sur les signes/marques du visage (et qui peuvent parfois paraître imprécises). La méthode Eigenfaces est un algorithme plutôt simple à implémenter et efficace : elle présente un bon taux de réussite (évidemment plus la base de données est grande, plus le taux de réussite est significatif).

3.1 Dataset : base de données d'apprentissage

Pour les données d'apprentissage, nous avons collecté les images du laboratoires AT&T pour avoir une base de données assez conséquente, puis par dessus nous avons rajouter d'autres images sous différents formats afin d'être capable de traiter tous les cas. Ainsi, la BDD contient 43 dossiers où chaque dossier comporte 10 images.

Les fonctions `loadFaceDatabase` et `loadFaceDatabaseCD` permettent de charger la base de données et d'exploiter un ensemble d'image sous forme de matrice. Vu que `loadFaceDatabaseCD` présente quelques inconvénients du fait qu'elle sollicite l'utilisation de `cd` (change directory), nous allons plutôt privilégier `loadFaceDatabase`.

Pour chaque image dans la base de données, on se chargera de vérifier quelques critères importants. Ce programme ne prend que des images au format `.pgm`, donc il faudra faire gaffe au format des fichiers. L'image doit être d'une certaine taille : 92×112 . Cependant, si elle n'est pas conforme à ces dimensions, elle sera juste redimensionnée (à l'aide de la fonction `imresize` en lui précisant la taille souhaitée) afin que l'on puisse continuer. Afin d'utiliser la méthode Eigenfaces, il faut également que l'image soit en nuances de gris (à l'aide de la fonction `rgb2gray`, cette fois-ci pas celle que nous avons créé afin de ne pas avoir de soucis par rapport au temps). Evidemment, si ce n'est pas déjà le cas, nous transformons l'image en nuances de gris. Dans un premier temps, nous voulions utiliser la fonctionnalité de Matlab permettant de rogner l'image (avec `imcrop` et `vision.CascadeObjectDetector` en ne conservant que le visage, partie qui nous intéresse, mais celle-ci découpaît un

visage trop réduit et retirait certains traits de visage comme le menton ou les cheveux. Nous avons donc testé sans rogner et cela fonctionnait bien mieux. Donc nous devons faire en sorte que le visage soit correctement cadré. De plus, il faut également faire attention aux fichiers cachés (comme les fichiers *.DS_Store* pour les utilisateurs de MacOS)

3.2 Testset : les images à tester

Une fois, la base de données chargée, il faut maintenant choisir une image en vérifiant également certains critères. Si un format d'image autre que *.pgm* est chargée, une copie de cette image sera créée, cette fois au format *.pgm*, et on utilise par la suite cette image (on s'occupera également de la suppression de ce fichier créé une fois l'application fermée). On se contente aussi de vérifier si l'image en question en nuance de gris, avec des dimensions correctes et se chargera de la mettre en *uint8* pour pouvoir l'utiliser lors de la reconnaissance. Toutes ces vérifications et transformations se trouve dans la fonction *transformImage*.

3.3 Reconnaissance et identification

Apres cela, nous appliquons l'algorithme de Eigenfaces pour trouver les Eigen vectors de l'image chargée à l'aide de la function *eig* (qui est l'équivalent de *spec* en SciLab), puis, on parcourt toutes les images de la base de donnée en comparant les Eigen vectors de chaque image. Pour chaque image, on obtiendra ainsi les Eigen faces où seront dessinés les particularités de chaque image pour la reconnaissance. L'image dont la différence de Eigen vector est la plus faible est l'image de la personne en question. Autrement dit, on vient de retrouver le visage de la personne sur une autre photo dans la base de données. Cette photo est ensuite affichée. Par la suite nous aimerions que chaque image de la base de donnée soit liée a un prénom, de manière a ce que lorsqu'on reconnaît une photo, on puisse directement afficher le nom de la personne.

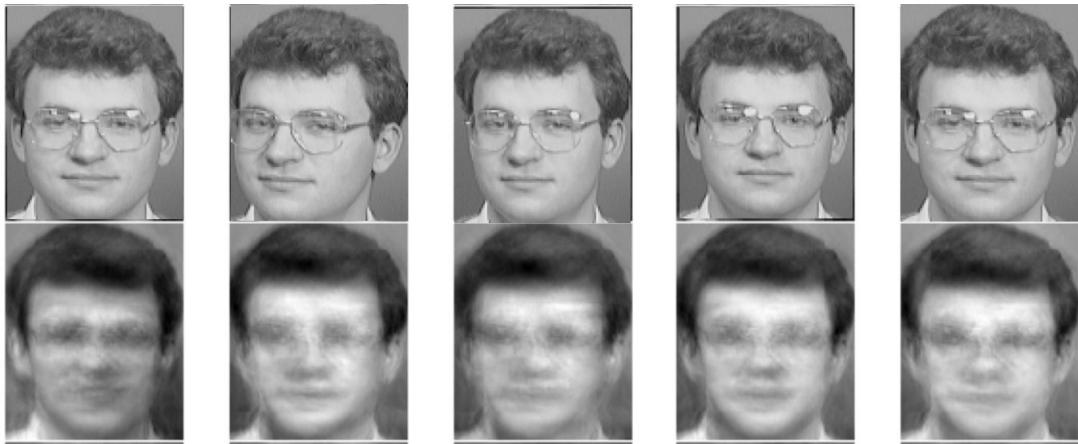


FIGURE 19 – Visages référents et les Eigenfaces correspondants

4 VRecognition : SOM

Dans cette partie encore en cours de développement, nous voulions utiliser le principe de SOM aussi connu comme « carte auto adaptative » pour reconnaître un visage au sein d'une photo. Voici le fonctionnement. Tout d'abord, nous avons réalisé un traqueur de visage qui place un carré au-dessus du visage de la personne. Pour cela nous avons utilisé une toolbox Matlab pour détecter le visage plus facilement. Ensuite nous plaçons sur le visage 100 points. Si le visage bouge relativement lentement, les points suivent le visage. Cependant, si le visage bouge trop vite ou que quelque chose vient à cacher le visage, les points sont perdus. Pour éviter de perdre la trace d'un visage ainsi, nous recréons des points toutes les 10 frames afin de s'assurer qu'il y ait constamment 100 points sur le visage. Cela permet de suivre le visage avec une efficacité quasi-parfaite. La suite était nettement plus compliquée et nous a énormément retardé. Nous voulions utiliser l'algorithme de SOM ainsi : pour chaque nouveau visage, on prend 8 photos à intervalle de temps régulier. Puis on demande à l'utilisateur d'entrer le nom de la personne. On rogne la photo pour m'avoir que le visage et on transmet cette photo à notre carte auto adaptative. À partir de là, on utilise les 8 photos pour créer une carte auto adaptative unique, représentant le visage de la personne. Lorsqu'il faut reconnaître le visage d'une personne, il n'y a plus qu'à mesurer la distance euclidienne entre chaque pixel de l'image (du visage à reconnaître) avec chaque carte auto adaptative de la base de données. Plus l'image se rapproche d'une certaine carte auto adaptative, plus on peut penser qu'il s'agit d'une seule et même personne. Pour pouvoir signaler qu'une personne n'existe pas dans la base de données, on peut également imposer un seuil de

validité. Si la carte auto adaptative la plus proche et la photo n'ont qu'un pourcentage de ressemblance très faible, ou inférieur à 70%, on peut penser qu'il ne s'agit pas de la bonne personne et donc que la personne n'existe pas dans la base de données. Malheureusement, nous avons commencé à implémenter SOM en C pour ensuite appeler la fonction depuis Matlab, mais pris par le temps nous avons délaissé cette idée.