Fundamentals of Image Processing

Book of Exercises
Part 2

Master 1 Computer Science - IMAge & DIGIT

Sorbonne Université

Year 2023 – 2024

Contents

6. Keypoint Detection

Exercise 1 — Moravec keypoint (corner) detection

Let I be an image, and W a window of size $W_x \times W_y$. The mean squared intensity variation $E_{u,v}(x_1, y_1)$ between the region W centered at pixel (x_1, y_1) and the region W centered at pixel $(x_1 + u, y_1 + v)$, $(u, v) \in \mathbb{Z}^2$, is defined as (see Figure 1):

$$E_{u,v}(x_1, y_1) = \sum_{k=-\frac{W_x}{2}}^{\frac{W_x}{2}} \sum_{l=-\frac{W_y}{2}}^{\frac{W_y}{2}} [I(x_1 + k + u, y_1 + l + v) - I(x_1 + k, y_1 + l)]^2 \quad (1)$$

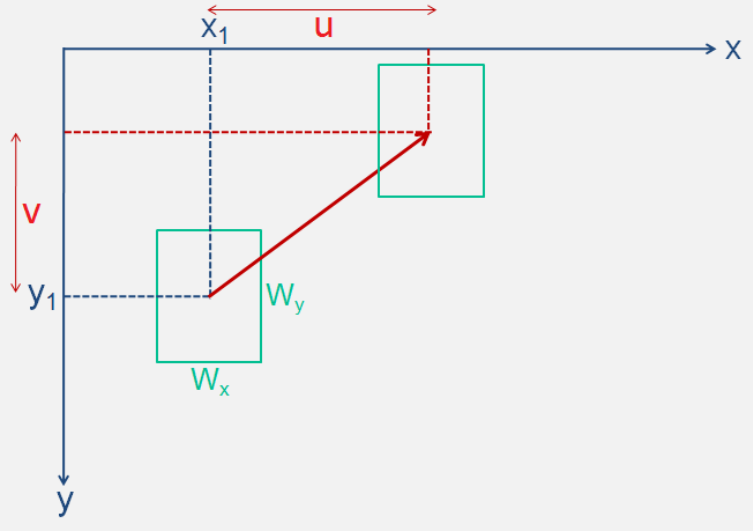


Figure 1: Illustration of the principle of Moravec detector.

The Moravec keypoint (corner) detector [4] is as follows:

- Using a 3×3 window W, compute $E_{u,v}(x, y)$ (Equation 1) for each pixel (x, y) of the image, for all possible translations in a 8-neighbors model:
  $\{(1, 0); (1, 1); (0, 1); (-1, 1); (-1, 0); (-1, -1); (0, -1); (1, -1)\}$.
- Compute a corner map as: $C(x, y) = \min_{u,v} \ E_{u,v}(x, y)$.
- Compute the local maxima, leading to the final detection.

Example: Let I be defined as:

$$I = \begin{bmatrix} 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 \\ 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 \\ 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & \boxed{0} & 255 & 255 \\ 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 \\ 255 & 255 & 255 & 255 & \boxed{255} & 255 & 255 & 255 & 255 & 255 & 255 \\ 0 & \boxed{255} & 0 & 0 & 0 & \boxed{0} & 255 & 255 & 255 & 255 & 255 \\ 0 & 0 & 0 & 0 & 0 & 0 & 255 & 255 & 255 & \boxed{255} & 255 \\ 0 & 0 & 0 & 0 & 0 & 0 & 255 & 255 & 255 & 255 & 255 \\ 0 & 0 & 0 & 0 & 0 & 0 & 255 & 255 & 255 & 255 & 255 \end{bmatrix}$$

1. Compute $E_{u,v}(x,y)$ and $C(x,y)$ at the boxed pixels of I.
2. What do these chosen points represent?
3. Discuss the detection performance of this detector.

Exercise 2 — Harris corner detector

Harris & Stephens [2] proposed to modify the Moravec detector as follows:

- Use a Gaussian window $w(x,y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2+y^2}{2\sigma^2}}$ as a weighting function in the mean squared intensity variation:

$$E^1_{u,v}(x_1,y_1) = \sum_{x,y} w(x_1-x, y_1-y) \times [I(x+u, y+v) - I(x,y)]^2 \tag{2}$$

- Compute the Taylor series expansion of $I(x+u, y+v)$ at order 1:

$$I(x+u,y+v) = I(x,y) + u\frac{\partial I}{\partial x}(x,y) + v\frac{\partial I}{\partial y}(x,y) + \mathcal{O}(u^2,v^2) \approx I(x,y) + uI_x(x,y) + vI_y(x,y) \tag{3}$$

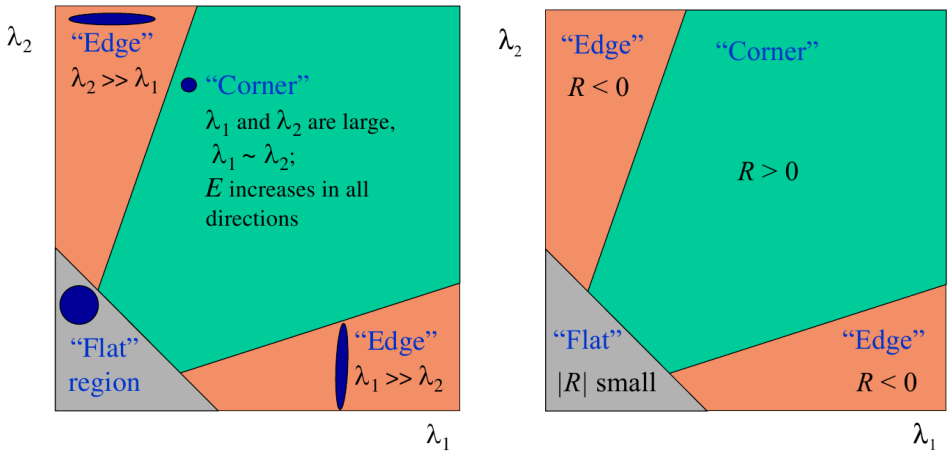
1. Prove that $E^1_{u,v}(x,y) = (u,v)M(u,v)^T$, with:

$$M(x,y) = \begin{bmatrix} w \star (I_x^2) & w \star (I_x I_y) \\ w \star (I_x I_y) & w \star (I_y^2) \end{bmatrix} = \begin{bmatrix} A & C \\ C & B \end{bmatrix}$$

2. The matrix $M(x,y)$, called auto-correlation matrix, represents the local structure of function $E^1(x,y)$ in a neighborhood of pixel (x,y). Harris & Stephens [2] proposed to compute the eigenvalues λ_1 and λ_2 of E^1 (Figure 2(a)). To reduce the computational complexity, these authors proposed the following criterion $R(x,y)$ (Figure 2(b)):

$$R(x,y) = \det(M(x,y)) - k\left[\text{trace}(M(x,y))\right]^2 \tag{4}$$

where $\det(M) = AB - C^2 = \lambda_1\lambda_2$, $\text{trace}(M) = A + B = \lambda_1 + \lambda_2$.
Prove the equivalence between Figures 2(a) and 2(b).



(a) Eigenvalues (b) Harris criterion

Figure 2: Harris detector.

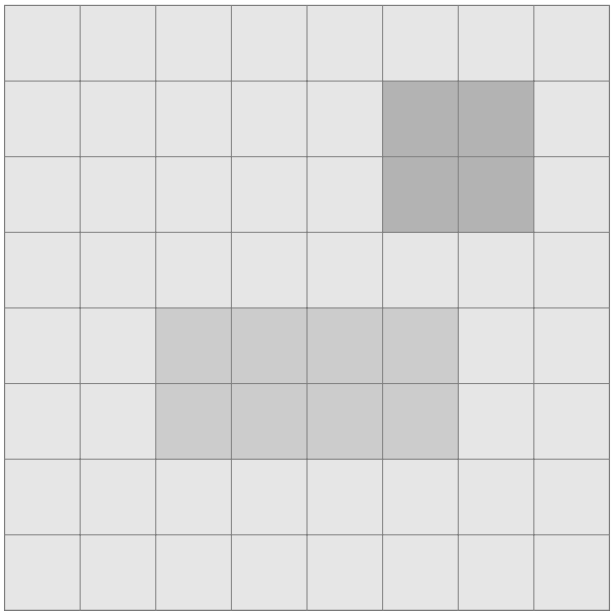
3. Prove that the Harris detector is invariant by rotation. Prove that the local maxima of $R(x, y)$ are invariant with respect to affine intensity variations, i.e. that can be written as $I'(x, y) = aI(x, y) + b$. Is the detector scale invariant?

4. Propose a simple method to detect corners at different scales, based on Harris detector.

7. Segmentation: split and merge

Exercise 3 — Split principle

1. Based on the example detailed during the lecture, propose a **recursive** segmentation algorithm using splitting and producing a quadtree.
2. Illustrate the steps of the algorithm on the following 64×64 image (each element of the grid represents a 8×8 square), and build the quadtree:



3. Which splitting predicate can be used to segment correctly this image?
4. Compute the mean and variance of each leaf in the quadtree, given that the gray levels of the background, square and rectangle are 150, 50 and 100, respectively.
5. Now a Gaussian noise with 0 mean and variance equal to 25 is added to this image. Which threshold on the variance should be applied in order to get the same quadtree as in the previous questions?

Exercise 4 — Implementation

The implementation will use Python lists. The leaves of the quadtree provide the list of regions.

1. What does a leaf in the quadtree represent? Which information should it contain? Define a formal type in Python to represent this leaf, called *block* from now on.
2. How can the quadtree obtained in the previous exercise be represented in Python? Derive a formal type in Python.
3. Propose a simple splitting criterion. Write the splitting predicate, i.e. the function:

```
def predsplit(I,reg,*args):
```

```
    """ Array*Block*... -> bool
        return True if the block should be splitted
    """
```

The argument *args is a Python iterable, which allows passing a variable number of parameters. Here we assume that the first parameter is a threshold on the standard variation.

4. Write the `split()` function which implements the splitting algorithm:

```
def split(I,reg,pred,*args):
    """ Array*Block*(Array*Block*... -> bool)*... -> quadtree
        Applies a quadtree splitting on I and region reg
        according to predicate pred and
        parameters *args used by the predicate.
    """
```

5. Write the function printblocks(L,I) which applies depth-first traversal of the quadtree L and prints the coordinates and size of the leaves, as well as the statistics (mean and standard deviation) of the associated image block in I.

Exercise 5 — Merging blocks in a quadtree

1. From a quadtree, write a merging algorithm which relies only on a homogeneity criterion of the gray levels: pixels in homogeneous regions will be set to 1 and the other to 0.
2. Consider the set of pixels with label 1. Is this set connected (specify the connectivity)? If not, what is the minimal size of each connected component? Is this good or bad?
3. Propose a merging algorithm producing a segmentation with a unique homogeneous region. It is assumed that a function `neighbors(b, L)` is available, returning the list of elements of L which are neighbors (4 closest neighbors on the Cartesian grid) of block b.
4. Apply this algorithm on the quadtree obtained in the first exercise in this session, using a similarity criterion.
5. Let R_1 and R_2 be two distinct regions with means μ_1 and μ_2, and variances σ_1^2 and σ_2^2. Derive the formulas for the mean and variance of $R_1 \cup R_2$ based on $|R_1|$, $|R_2|$, μ_1, μ_2, σ_1^2 and σ_2^2.
6. Derive a method to compute the homogeneity of a region (whether connected or not).

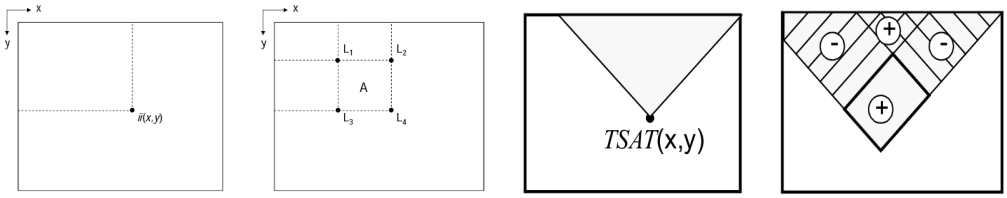
8. Image descriptors

Exercise 6 — Integral images

An ***integral image*** $I_{int}(x, y)$ is defined at pixel (x, y) as the sum of the gray levels $I(x', y')$ of all pixels to the left and above pixel (x, y):

$$I_{int}(x, y) = \sum_{x' \leq x,\ y' \leq y,} I(x', y') \tag{6}$$

Integral images have been used to compute efficiently some characteristics such as SURF [1] (a more compact and easier to compute variant of SIFT descriptor), or Haar wavelets for face detection [5].



(a) $I_{int}(x, y)$ (b) (c) $I^{\theta}_{int}(x, y)$ (d)

Figure 3: Integral image (a) and sum in a rectangular region (b). Oriented case: (c), (d).

1. Propose an algorithm using only one scan of the image and that is linear in the number of image pixels to compute $I_{int}(x, y)$.
2. Prove that it is possible to compute the sum $\sum_{x',y' \in A} I(x', y')$ of the values inside a rectangular region A (Figure 3(b)) from $I_{int}(x, y)$, in only four operations.

   A variant of the integral image consists in computing sums of intensities in oriented regions, *e.g.* rotated by $\frac{\pi}{4}$ (Figure 3(c)):

$$I^{\theta}_{int}(x, y) = \sum_{y' \leq y,\ y' \leq y - |x - x'|,} I(x', y') \tag{7}$$

3. Propose an algorithm using only one scan of the image to compute I^{θ}_{int}. What is its complexity?
4. How can the sum of values in region A of Figure 3(d) be computed efficiently?

Exercise 7 — Blob detector

A blob detector aims at detecting image regions in which the grey levels have a Gaussian distribution (with variance σ_0):

$$G_{(x_0,y_0,\sigma_0)}(x,y) = \frac{1}{2\pi\sigma_0} e^{-\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma_0}} \tag{8}$$

Reminder: the Laplacian Δ of an image I is $\Delta I = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$.

1. Prove that $\arg\min_{x,y} \ \Delta\left[G_{(x_0,y_0,\sigma_0)}(x,y)\right] = \{x_0, y_0\}$.

   Interpretation: the minima of the Laplacian of a Gaussian image correspond to the position of the center (x_0, y_0) of the Gaussian.

   Generalization:

   Let us now consider the convolution of an image I with the Gaussian kernel G: $L = I \star G_{(x_0,y_0,\sigma_0)}$. Recall that convolution and derivation commute in this case: $\Delta L = \frac{\partial^2 L}{\partial x^2} + \frac{\partial^2 L}{\partial y^2} = I \star \frac{\partial^2 G}{\partial x^2} + \frac{\partial^2 G}{\partial y^2} = I \star \Delta G$. Let us admit the following result:

   - If $I = G_{(x_0,y_0,\sigma_0)}$ and if the convolution is done by a Gaussian kernel $G_{(x_0,y_0,\sigma)}$ of variable size σ, then $\arg\min_{x,y} \ \max_{\sigma} \ \Delta\left[\sigma \cdot L(x,y,\sigma)\right] = \{x_0, y_0, \sigma_0\}$ [3].

2. Prove that $G_{(x_0,y_0,\sigma_0)}$ is a solution of the heat equation: $\frac{\partial G}{\partial \sigma} = \frac{1}{2}\Delta G$.

3. Derive that $G(k\sigma) - G(\sigma) \approx \Delta G \cdot (k-1)\sigma$. How is it then possible to build a multi-scale blob detector by convoluting the image by a difference of Gaussians (DoG) filter?

9. Principal component analysis (PCA)

Exercise 8 — Derivation of PCA

- Let $X \in \mathbb{R}^d$ be a random vector, and g the empirical expectation estimator of X: $g = \frac{1}{n}\sum_{i=1}^{n} X_i \approx \mathbb{E}[X]$ where X_i are realizations of the random variable X.
- Let Π be the projection operator on the vectorial line $\mathcal{V}$ defined by the unit vector $\overrightarrow{v}$ ($||\overrightarrow{v}|| = 1$). The coordinates of $\overrightarrow{v}$ in $\mathbb{R}^d$ are denoted by v. Then we have (see Appendix 1.1): $\Pi = vv^\top$ and $v^\top v = 1$.
- The variance σ_v^2 of X on $\mathcal{V}$ is:

$$\sigma_v^2(X) = \frac{1}{n-1}\sum_{i=1}^{n}\Big(\big(\Pi\,(X_i - g)\big)^\top \Pi(X_i - g)\Big) \qquad \left(= \frac{1}{n-1}\sum_{i=1}^{n}||\Pi(X_i - g)||^2\right) \tag{9}$$

1. Prove that $\sigma_v^2(X) = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - g)^\top vv^\top (X_i - g)$
2. Derive that $\sigma_v^2(X) = v^\top\,\Sigma\,v$, where $\Sigma = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - g)(X_i - g)^\top$ is the variance-covariance matrix of X.

   The principle of **Principal Component Analysis** (PCA) consists in determining the unit vector $\overrightarrow{v}$ that maximizes $\sigma_v^2(X)$, i.e. that maximizes $v^\top\,\Sigma\,v$, under the constraint $v^\top v = 1$:

$$\begin{cases} \max\limits_{v} & v^\top \Sigma v \\ \text{s.t.} & v^\top v = 1 \end{cases} \tag{10}$$

   This problem can be solved by the Lagrangian method (see Appendix 1.2), i.e. maximizing the following function $\mathcal{L}(v, \lambda)$ over v and λ (called Lagrange multiplier):

$$\mathcal{L}(v, \lambda) = v^\top \Sigma v + \lambda(1 - v^\top v) \tag{11}$$

3. By using the first order necessary conditions given in Appendix 1.2, prove that maximizing Equation (11) leads to: $\begin{cases} \Sigma v = \lambda v \\ v^\top v = 1 \end{cases}$
4. Derive that Equation (11) can be solved by diagonalizing Σ. Which is then the variance $\sigma_v^2(X)$ for a pair of eigenvector, eigenvalue (v_i, λ_i)?
5. Conclusion: propose a method to determine the line $\mathcal{V}$ maximizing the variance of the data defined in Equation (9).

10. Linear Discriminant Analysis (LDA)

Let $X \in \mathbb{R}^d$ be a random vector. Let us consider N realizations X_i of X, $i \in \{1...N\}$. Let Y_i be a label associated with X_i, representing the membership of X_i to one of K classes C_k: $X_i \in C_k \Leftrightarrow Y_i = k$ ($k \in \{1...K\}$).

Linear Discriminant Analysis (LDA) consists in determining **discriminant factors**, as linear combinations of the input variables, which take values as close as possible to each other for elements in a same class, and as different as possible for elements in different classes. Mathematically, LDA is formalized as the search of a data projection leading to:

- a minimal intra-class variance, and
- a maximal inter-class variance.

Exercise 9 — Decomposition of the total variance

Let g be the empirical estimation of the expectation of X: $g = \frac{1}{N}\sum_{i=1}^{N} X_i$. Let N_k be the number of elements in class k ($N_k = |C_k|$), and g_k the center (mean) of the k^{th} class: $g_k = \frac{1}{N_k}\sum_{X_i \in C_k} X_i$.

The total variance of the data writes:

$$\sigma^2(X) = \frac{1}{N}\sum_{i=1}^{N}(X_i - g)^\top\ (X_i - g) = \frac{1}{N}\sum_{k=1}^{K}\sum_{X_i \in C_k}(X_i - g)^\top\ (X_i - g) \tag{13}$$

Let $SS(k) = \frac{1}{N}\sum_{X_i \in C_k}(X_i - g)^\top\ (X_i - g)$

1. Prove that $SS(k)$ can be decomposed as:

$$SS(k) = \frac{1}{N}\left[\sum_{X_i \in C_k}(X_i - g_k)^\top (X_i - g_k)\right] + \frac{N_k}{N}(g_k - g)^\top (g_k - g) \tag{14}$$

2. Derive that the total variance $\sigma^2(X)$ can be decomposed as: $\sigma^2(X) = \sigma_b^2(X) + \sigma_w^2(X)$, where:

   - $\sigma_w^2 = \sum_{k=1}^{K}\frac{N_k}{N}\sigma_{k,w}^2$ is the weighted mean of the intra-class ("within") variances, the intra-class variance being defined as: $\sigma_{k,w}^2 = \frac{1}{N_k}\sum_{X_i \in C_k}(X_i - g_k)^\top (X_i - g_k)$.
   - $\sigma_b^2 = \sum_{k=1}^{K}\frac{N_k}{N}\sigma_{k,b}^2$ with $\sigma_{k,b}^2 = (g_k - g)^\top (g_k - g)$ is the variance of the means g_k of classes (inter-class variance, "between").

Exercise 10 — Projected variance and solution of LDA

Let Π be the projection operator on a vectorial line $\mathcal{V}$ defined by the unit vector $\overrightarrow{v}$ ($||\overrightarrow{v}|| = 1$). The coordinates of $\overrightarrow{v}$ in $\mathbb{R}^d$ are denoted by v. We have: $\Pi = vv^\top$ and $v^\top v = 1$. Remind that the projection of the total variance on $\mathcal{V}$ writes: $\sigma_v^2 = v^\top \Sigma v$, with $\Sigma = \frac{1}{N-1} \sum_{i=1}^{N} (X_i - g)(X_i - g)^\top$ (cf work on PCA).

1. Prove that the projection of the intra-class variance on $\mathcal{V}$ writes: $\sigma_{v(w)}^2 = v^\top W v$, with:

   - $W = \dfrac{1}{N} \sum_{k=1}^{K} N_k W_k$
   - $W_k = \dfrac{1}{N_k} \sum_{X_i \in C_k} (X_i - g_k)(X_i - g_k)^\top$

2. Prove that the projection of the inter-class variance on $\mathcal{V}$ writes: $\sigma_{v(b)}^2 = v^\top B v$, with:

   - $B = \dfrac{1}{N} \sum_{k=1}^{K} N_k (g_k - g)(g_k - g)^\top$

3. The projected variance can therefore be decomposed as $\sigma_v^2 = \sigma_{v(w)}^2 + \sigma_{v(b)}^2$ (projected points are still in $\mathbb{R}^d$). LDA consists in determining the line $\mathcal{V}$ maximizing $\sigma_{v(b)}^2$ and minimizing $\sigma_{v(w)}^2$. Prove that the line $\mathcal{V}$ verifies $\max_v J(v)$ with $J(v) = \left(\dfrac{v^\top B v}{v^\top \Sigma v} \right)$.

4. Note that $J(v)$ is invariant with respect to scaling transformations $v \leftarrow \alpha v$. It is then possible to choose v such that $v^\top \Sigma v = 1$. Write the Lagrangian expressing the optimization problem under constraints.
   By using the Lagrange multipliers method (see work on PCA), prove that a necessary condition to maximize $J(v) = \left(\dfrac{v^\top B v}{v^\top \Sigma v} \right)$, under the constraint $v^\top \Sigma v = 1$, is: $Bv = \lambda \Sigma v$, with $\lambda = \dfrac{v^\top B v}{v^\top \Sigma v}$.
   Indication: on can admit that $\dfrac{\partial \left(v^\top B v \right)}{\partial v} = 2Bv$ and $\dfrac{\partial \left(v^\top \Sigma v \right)}{\partial v} = 2\Sigma v$.

5. **Conclusion**: prove that LDA consists in diagonalizing $\Sigma^{-1} B$, and that the eigenvector associated with the largest eigenvalue defines the vectorial line, solution of LDA.

1 Appendix: Projection and Lagrange multipliers

1.1 Orthogonal projection on a vectorial line

In a vectorial space $\mathcal{E}$ of finite dimension, the **orthogonal projection** Π on a subspace $\mathcal{S}$ of $\mathcal{E}$ is a linear operator, which can therefore be represented by a matrix **PS**. Let **Z** be a matrix whose columns define an orthonormal basis of $\mathcal{S}$. The orthogonal projection $u = \Pi(x)$ of a vector x is given by:

$$u = \Pi(x) = ZZ^\top x = PSx \tag{15}$$

In the case of a vectorial line, we have $Z = v$ where v is a unit vector defining the vectorial line. The projection matrix then writes $PS = vv^\top$.

1.2 Optimization under constraints: Lagrange multipliers

Let $X = (x_1, ..., x_n) \in \mathbb{R}^n$, and consider the following optimization problem: $\max_X f(X)$ under the constraint $h(X) = 0$, where f is a quadratic function of X, and h is linear. The Lagrangian associated with this problem is defined as the following function $\mathcal{L}(X, \lambda)$:

$$\mathcal{L}(X, \lambda) = f(X) + \lambda h(X) \tag{16}$$

This Lagrangian allows intoducing the constraint in the objective function, with a penalty $\lambda \in \mathbb{R}$ (Lagrange multiplier). The maximization is then over both X and λ.

It can be shown that the following first order conditions are necessary in order to maximize f under the constraint h:

$$\left\{ \begin{array}{ll} \frac{\partial \mathcal{L}}{\partial x_i} & = 0 \quad \forall\, i \\ \frac{\partial \mathcal{L}}{\partial \lambda} & = 0 \end{array} \right. \quad \Leftrightarrow \quad \left\{ \begin{array}{ll} \frac{\partial f}{\partial x_i} & = \lambda \frac{\partial h}{\partial x_i} \quad \forall\, i \\ h(X) & = 0 \end{array} \right. \tag{17}$$

References

[1] H. Bay, A. Ess, T. Tuytelaars, and L. Vangool. Speeded-up robust features (surf). *Computer Vision and Image Understanding*, 110(3):346–359, June 2008.

[2] C. Harris and M. Stephens. A combined corner and edge detector. In *Proc. Fourth Alvey Vision Conference*, pages 147–151, 1988.

[3] T. Lindeberg. *Scale-Space Theory in Computer Vision.* Kluwer, December 1993.

[4] Hans Moravec. Obstacle avoidance and navigation in the real world by a seeing robot rover. In *tech. report CMU-RI-TR-80-03, Robotics Institute, Carnegie Mellon University and doctoral dissertation, Stanford University*, number CMU-RI-TR-80-03. September 1980.

[5] P.A. Viola and M.J. Jones. Robust real-time face detection. *IJCV*, 57(2):137–154, May 2004.

Practical works

Practical work materials (Jupyter notebooks and data) are to be retrieved from the course's web site: `https://frama.link/mR68kcMB`.

Practical work 6 : Harris Corner Detector

The goal of this pratical work is to implement the Harris-Stephen's corners detector (C. Harris and M. Stephens. A combined corner and edge detector. In Proc. Fourth Alvey Vision Conference, pages 147–151, 1988).

Recall the Harris detector computes a map of corners from an image I:

$$R(x,y) = \det(M) - k(\text{trace}(M))^2, (x,y)\text{pixels}$$

with $k \in [0.04, 0.06]$. M is the auto-correlation of image I:

$$M = \begin{pmatrix} \sum_{x,y\in W} w(x,y)I_x^2 & \sum_{x,y\in W} w(x,y)I_xI_y \\ \sum_{x,y\in W} w(x,y)I_xI_y & \sum_{x,y\in W} w(x,y)I_y^2 \end{pmatrix} = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

with $w(x,y) = \frac{1}{2\pi\sigma^2} e^{\frac{(x-x_c)^2+(y-y_c)^2}{2\sigma^2}}$ a Gaussian mask centered on the window W. Partial derivatives I_x and I_y are estimated by one of the following kernels : - Gradient: $G_x = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & -1 \\ 0 & 0 & 0 \end{pmatrix}$, $G_y = G_x^T$ - Prewitt: $G_x = \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix}$, $G_y = G_x^T$ - Sobel: $G_x = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}$, $G_y = G_x^T$

[ ]:
```
# Load useful libraries
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np
import scipy.signal

# Useful functions
def gaussianKernel(sigma):
    """ double -> Array
        return a gaussian kernel of standard deviation sigma
    """
    n2 = np.int(np.ceil(3*sigma))
    x,y = np.meshgrid(np.arange(-n2,n2+1),np.arange(-n2,n2+1))
    kern = np.exp(-(x**2+y**2)/(2*sigma*sigma))
    return  kern/kern.sum()
```

Exercise 1: Harris response calculation

1) Write a function `computeR(I, scale, kappa)` that returns the Harris response R from an image I and a scale *scale*. You will use 5 steps:

- Computation of the directionnal derivate I_x and I_y. Use the Sobel kernel.
- Computation of the products I_x^2, I_y^2, $I_x.I_y$.
- Computation of the convolution of I_x^2, I_y^2 and $I_x.I_y$ by a gaussian kernel of size N (use given function `gaussianKernel()`)
- Computation of $\det(M(x,y))$ and $\text{trace}(M(x,y))$ for each pixel
- Computation of $R(x,y) = \det(M(x,y)) - k.(\text{trace}(M(x,y)))^2$. You can use $k = 0.04$.

You can compute the convolutions by using the `scipy.signal.convolve2d` function.

[6]:
```
def computeR(image,scale,kappa):
    """ Array[n, m]*float*float->Array[n, m]
    """
```

2) Write a script that displays the Harris response for the image `img/house2.png` along with the original image. Use a gaussian window of size $W = 15$ pixels.

[ ]:
```

```

3) Write in a few lines an interpretation of the results, explaining how the Harris response allows to detect and discriminate homogeneous areas, edges and corners.

Your answer...

Exercise 2 : Harris corner detector

From the Harris response calculated at exercise 1, we will write all the functions needed for the Harris detector. Write the following functions:

1) A function `thresholdR(R, thres)` that calculates and returns the binary thresholding R_b of the response R according to the threshold *thres*

[ ]:
```
def thresholdR(R, thres):
    """ Array[n, m] * float -> Array[n, m]
    """
```

2) A function `Rnms(R, Rbin)` that performs a non-maximum supression from the response R and the binarized response R_{bin}. It returns the image R_{locmax} (same size as R) =1 where $R_{bin} = 1$ and the pixel has a greater value R than its 8 nearest neighbors.

Bonus: Write a faster version of the script using Numpy function `np.roll()`.

[1]:
```
def rnms(image_harris):
    """ Array[n, m] -> Array[n, m]
    """
```

3) Write a function `cornerDetector(image, scale, kappa, thresh)` that returns an array of the same size as the image. The array takes two values: 1 where a corner is detected and 0 elsewhere.

[ ]:
```
def cornerDetector(image, sigma, kappa, thres):
    """ Array[n, m]*float*float*float -> Array[n, m]
    """
```

4) Display the detected corners on the original image for the image `img/house2.png`. Each detected corner will be displayed as a small red disk. You can use the functions `np.nonzero()` and `plt.scatter()` to that purpose.

[ ]:
```

```

5) Evaluate the performances of the corner detector. Try to find good values for Sigma and Threshold.

Exercise 3 : Properties of Harris corner detector

The goal of this exercice is to study some invariance properties of Harris detector.

1) Write a script that detects the corners on the images `img/toyHorse1.png` and `img/toyHorse2.png` with a scale of 15 and appropriate threshold value. Display the detected corners on the images.

[ ]:

2) What are the dynamic ranges of these two images ?

Your answer...

3) What are the transformations beetween the two images ?

Your answer...

4) Using a fixed threshold, is the detection invariant to rotation ? To affine transformation of brightness ?

Your Answer

Practical work 7: Split and Merge

In this practical work, we implement and test the split and merge algorithm.

[1]:
```
### Usefull libraries
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt

### Data
img_test = np.full((64,64),150,dtype=np.uint8)
img_test[32:48,16:16+32] = 100
img_test[8:24,40:56] = 50
angio = np.array(Image.open('img/angiogra.png'))
cam = np.array(Image.open('img/cameraman.png'))
muscle = np.array(Image.open('img/muscle.png'))
prisme = np.array(Image.open('img/prisme.png'))
seiche = np.array(Image.open('img/seiche.png'))

### Usefull functions
def neighbors(b,K):
    """ blockStat*list[blockStat]->list[blockStat]
        returns the list of neighbors of b and elements of K
    """
    def belongsTo(x,y,a):
        """ int*int*BlockStat -> bool
            Test if pixel (x,y) belongs to block a
        """
        return x>=a[0] and y>=a[1] and x<a[0]+a[2] and y<a[1]+a[3]
    def areNeighbors(a,b):
        """ BlockStat**2 -> bool
            Test if a and b are neighbors
        """
        if a[2]>b[2] and a[3]>b[3]:
            a,b=b,a
        x,y = a[0]+a[2]//2,a[1]+a[3]//2
        return belongsTo(x+a[2],y,b) or belongsTo(x-a[2],y,b) or␣
↪belongsTo(x,y+a[3],b) or belongsTo(x,y-a[3],b)
    return [n for n in K if areNeighbors(b,n)]
```

Exercise 1

Question 1

Write the recursive function `split()` discussed in tutorial work. It takes as input the image, a region, a predicate, and a variable number of arguments. The region is a Python formal type `Block` defined by:

```
type Block = tuple[int**4]
```

The function `split()` returns a quadtree, a Python formal type, recursivelly defined by:

`type QuadTree = list[(QuadTree**4|Block)]`

The predicate is a Python function with the following signature:

`Array*Block*...->bool`

It can take a variable number of parameters which correspond to the parameters required by the predicate.

[4]:
```
# type Block = tuple[int**4]
# type QuadTree = list[(QuadTree**4|Block)]

def split(I,reg,pred,*args):
    """ Array*Block*(Array*Block*...->bool)*... -> 4-aire
        Performs a quadtree splitting of image I drived by a predicate
    """
```

Question 2

Write the function `predsplit(I,B,*args)` with signature:

`Array*Block*... -> bool`

that returns True if the standard deviation of image `I` computed in region `B` is greater than the first value of argument `*args` (it can be accessed simply by `*args[0]`).

[1]:
```
def predsplit(I,reg,*args):
    """ Array*Block*... -> bool
    """
```

Question 3

Write the function `listRegions()` which applies a depth-first search on the quadtree given as parameter, and returns the list of the leaves of the quadtree.

Some recalls about lists in Python; - Initialization: `L = []` (empty list) - Add a element `a` into a list `L`: `L.append(a)`

[ ]:
```
def listRegions(L):
    """ QuadTree -> list[Block]
    """
```

Question 4

Test your codes on the synthetic image `img_test` seen in tutorial work. Print the value returned by `split()` as well as the one returned by `listRegions()`.

[ ]:

Question 5

Write the function `drawRegions(L,I)` which takes as arguments a list of regions, an image, and returns an image where the boundaries of each region have been traced with red color. Indication: the returned image is a hypermatrix of dimension 3, the third dimension is of size 3 and encodes the red, green and blue components of a RGB colorspace. Test the function on the previous example.

[5]:
```
def drawRegions(LL,I):
    """ list[Block]*Array -> Array
        parcours de la liste dessin des régions
    """
```

Question 6

Add a Gaussian noise with standard deviation 5 to the image `img_test`. Apply the quadtree splitting on the noisy image by adjusting the threshold to obtain the same result as in the previous question. Which threshold value should be chosen? Does this make sense to you?

Hint: use the Numpy function `random.randn()` which generates random values according to a normal distribution (Gaussian distribution of null mean and variance 1). To obtain realizations of a Gaussian distribution of standard deviation σ, it is sufficient to multiply by σ the realizations of a normal distribution.

[ ]:
```
from numpy import random
```

Exercise 2

Experiment the split algorithm on the 4 natural images provided. For each image try to find the threshold that seems to you visually the best. Display the number of regions obtained after splitting.

[ ]:

Exercise 3

Question 1

Modify the function `listRegions(L)` to make it a function `listRegionsStat(L,I)` which computes the list of leaves of the quadtree `L`. Each element of this list will be enriched with three scalar values: the first being the size, the second the mean and the third the variance of pixel values of the block in the image `I`. This function then returns a list whose elements have the following formal type:

```
type BlockStat = tuple[int**4,int,float**2]
```

The first four values are those of the `Block` type, the fifth is the size of the block (in number of pixels) and the last two values are the mean and variance calculated over the region.

[3]:
```
# type BlockStat = tuple[int**4,int,float**2]

def listRegionsStat(L,I):
```

```
    """ QuadTree*Array -> list[BlockStat] """
```

Question 2

In the remainder, the formal type is considered:

`type Region = list[BlocStats]`

A region, as seen during the tutorial work, is therefore a list of blocks. Write the predicate `predmerge(b,R,*args)` as seen in tutorial work. This function returns `True` if the `b` block should merge into the `R` region. If a merge happens, then the first item of `R` will have its statistics updated to describe the statistics of the region `R` merged with `b`.

[ ]:
```
def predmerge(b,R,*args):
    """ BlocsStat*Region*... -> bool
        If merge, R[0] is modified
    """
```

Question 3

Using `predmerge()` and `neighbors()` functions, given at the beginning of the notebook, write the function `merge()` discussed in tutorial work (exercise 7.6).

Recalls on Python lists: - Remove an element `a` from a list `L`: `L.remove(a)` - Test if `a` belongs to a list `L`: `a in L` - Iterate the elements of a list `L`: `for a in L:` - Access to an element of a list: as with numpy arrays

[4]:
```
def merge(S,I,pred,*args):
    """ QuadTree*Array*(BlockStat*Region*...->bool) -> list[Region]
        Merge the leaves of S in a list of regions
    """
```

Question 4

Test the previous functions using the synthetic image `img_test`. In particular, check that merge() returns a list of 3 elements (i.e. 3 regions).

[ ]:
```
QT = split(img_test, predsplit, ?)
M = merge(QT,predmerge, ?)
assert len(M) == 3
```

Question 5

Write a function `regions(LR,shape)` that takes as arguments a list of regions (such as returned by the function `merge()`) and an image size, and returns an image of the regions. Each region will be colored with the gray level corresponding to the average of the region. The `shape` parameter gives the size of the image to be produced.

Test the function on the previous example.

[ ]:
```
def regions(LR,shape):
    """ list[Region]*tuple[int,int] -> Array """
```

Exercise 4: experiments

Question 1

Test the function `merge()` on the images `angio`, `cam`, `muscle`, `prisme` and `seiche`. Try to produce the best segmentations.

[ ]:

Question 2

The result of the merge algorithm highly depends on how you visit the regions. One can then sort the leaves of the quadtree, for example, from the smallest to the largest blocks, or the opposite (use the Python function `sorted()`). The same question arises when calculating the set of neighbors of the merged region. Should they be sorted? If yes, according to which criteria? their size? their proximity? Obviously there is no universal answer but it should be adapted to each type of problem. Do some tests to see the influence of these sortings on the result of the merger.

[ ]:

Question 3 (bonus)

Imagine and experiment alternative predicates for both the split and the merge steps. It is possible to use edges-based predicates, and also to combine with variance-based predicates.

[ ]:

Practical work 8: Color quantification and search by content

In this practical work session, we will:

- Develop a color based descriptor that can be applied to every image in a database
- Use this color descriptor to create a method that searches images by content: the goal is to find the images that are the most similar to a query.

[3]:
```
# Load useful library

from PIL import Image
import matplotlib.pyplot as plt
import numpy as np
import scipy.signal
import scipy.ndimage
from skimage.color import rgb2hsv, hsv2rgb

# Usefull functions
def setColors(nH, nS, nV):
    """ int**3 -> Array[nH*nS*nV,3]*Array[nH,nS,nV,3]
        computes an RGB palette from a sampling of HSV values
    """
    pal1 = np.zeros((nH*nS*nV, 3))
    pal2 = np.zeros((nH, nS, nV, 3))
    tH, tS, tV = 1/(2*nH), 1/(2*nS), 1/(2*nV)

    idx = 0
    for i in range(nH):
        for j in range(nS):
            for k in range(nV):
                HSVval = np.array([[[i/nH + tH, j/nS + tS, k/nV + tV]]])
                pal1[idx, :] = hsv2rgb(HSVval)*255
                pal2[i, j, k, :] = hsv2rgb(HSVval)*255
                idx += 1
    return pal1, pal2

def viewQuantizedImage(I,pal):
    """ Array*Array -> Array
        Display an indexed image with colors according to pal
    """
    Iview = np.empty(I.shape)
    n, m, c = I.shape
    for i in range(n):
        for j in range(m):
            h, s, v = I[i, j, :]
            Iview[i, j, :] = pal[ np.int(h), np.int(s), np.int(v), :]
    print( Iview.max())
    plt.imshow(Iview/255)
```

```
    plt.show()

def display5mainColors(histo, pal):
    """ Array*Array -> NoneType
        Display the 5 main colors in histo
    """
    idx = np.argsort(histo)
    idx = idx[::-1]
    K = 5
    for i in range (K):
        Ia = np.zeros((1, 1, 3), dtype=np.uint8)
        Ia[0,0,0] = pal[idx[i], 0]
        Ia[0,0,1] = pal[idx[i], 1]
        Ia[0,0,2] = pal[idx[i], 2]
        plt.subplot(1, K, i+1)
        plt.imshow(Ia)
        plt.axis('off')
    plt.show()

def display20bestMatches(S, indexQuery):
    """ Array*int -> NoneType
    """
    L = S[indexQuery, :]
    Idx = np.argsort(L)[::-1]
    cpt = 1
    plt.figure(figsize=(15, 10))
    for idx in Idx[:20]:
        plt.subplot(5, 4, cpt)
        indexQuery = idx
        imageName = (pathImage+NomImageBase[indexQuery]).strip()
        plt.imshow(np.array(Image.open(imageName))/255.)
        plt.title(NomImageBase[indexQuery])
        plt.axis('off')
        cpt += 1
    plt.show()
```

Exercise 1: HSV histogram computation

Each image of the base will be represented by its color histogram in the HSV representation. We use the HSV representation rather that the RGB representation because it is a perceptual color space: two colors that look similar will have close HSV vectors.

1) Write a function `iv = quantize(v,K)` that returns the quantize interval of `v` considering a uniform quantization of values over the range $[0, 1]$ with `K` evenly spaced intervals. For a image value v=1, the function will return K-1.

```
[ ]: def quantize(I, k):
         """ float*int -> int
         """


     # You can test your function with the following lines:
     h = np.zeros((8))
     for i in range(256):
         h[quantize(i/255.,8)] += 1
     assert (h == 32*np.ones((8))).all()
```

2) Write a function `[Iq, histo] = quantizeImage(I,Nh,Ns,Nv)` that takes as input one image I of size N x M x 3 in the HSV representation and the number of quantification interval needed for H, S and V. Your function will return:

- `Iq`: the quantified image for each channel, of size N x M x 3
- `hist`: a 3D histogram of size Nh x Ns x Nv that counts the number of pixel for each quantification bin (iH, iS, iV)

```
[ ]: def quantizeImage(I, nH, nS, nV):
         """ Array*int -> Array*Array
         """
```

3) Write a function `normalized_histo = NormalizeHistL2(histo)` that applies a normalization on the histogram `histo` according to the L2 norm. The L2 norm of x can be computed using the function `numpy.linalg.norm(x,2)`

```
[ ]: def normalize(H):
         """ Array -> Array"""
```

4) Test of the HSV histogram on an image: Complete the following code with your functions in order to apply it on one of the images of the base. The code will follow the following steps:
   1. Open the image and convert it into HSV representation.
   2. Compute the color palette for the display using the given `setColors(nH, nS, nV)` function.
   3. Compute the quantization of the image then visualize the quantized image using `viewQuantizedImage(I,pal)`.
   4. Transform the 3D histogram into a 1D histogram, normalize it according to L2 norm then visualize it.
   5. Display the 5 most prevalent colors on the image using `display5mainColors(histo, pal)`.

   You can try this on the image `Paysages67.png`with `nH = 12`, `nS = 3` and `nV = 8` and find a result similar to Figures 1, 2, 3, and 4.

Figure 1: Paysage67.png

Figure 2: Paysage67.png quantized

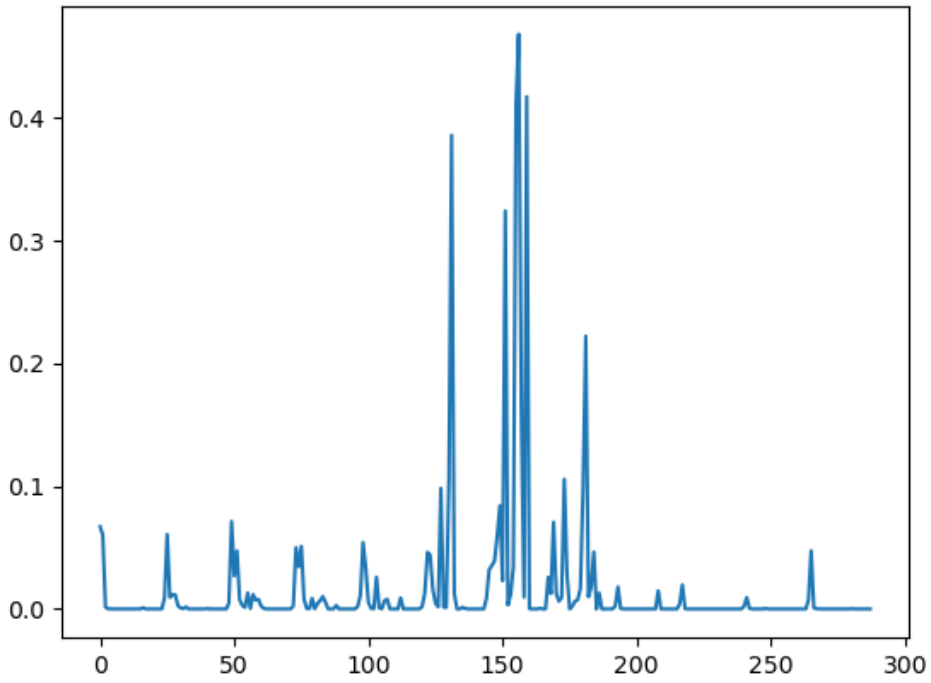


Figure 3: Histogram of HSV image (288 bins)

Figure 4: 5 main colors

[ ]:
```
PathImage = './Base/'  # to be completed
nom = ''  # to be completed

# quantization parameters
nH = # to be completed
nS = # to be completed
nV = # to be completed


filename= nom;
I = np.array(Image.open(PathImage + filename))
I = I / 255.  # I values range in [0,1]
plt.figure();
plt.imshow(I);
plt.show()

# conversion RGB->HSV
J = rgb2hsv(I)

# color palette computation
palette, palette2 = setColors( nH, nS , nV );
```

```
# Image quantization (your function)
Iq, histo = quantizeImage(J, nH, nS, nV)

# Visualisation of quantized image
viewQuantizedImage(Iq , palette2)

# flat a 3D histogram to 1D
histo = histo.flat

# Histogram normalization (your function)
histo = normalize(histo)

plt.figure()
plt.plot(histo)
plt.show()

## Determine 5 more frequent colors
idx_most_prevalent = (-histo).argsort()[:5]
hsv_most_prevalent = [np.unravel_index(idx,( nH, nS , nV )) for idx in␣
 ↪idx_most_prevalent]

display5mainColors(histo, palette)

print(hsv_most_prevalent)
```

5) Change the values of `nH`, `nS` and `nV` and analyze the results. You can try with other images in the base.

[ ]:

6) What can you say about the results?

Your answer: .....

Exercise 2: Similarity between images:

In this exercice, we will compute a measure of similarity between two images from the normalized histogram. This measure of similarity will be used in order to find images that are the most similar to a given image.

Question 2.1: Computation of the histograms for the whole base

Complete the following script to compute the histograms for every image in the base. As the computation can take a lot of time, we will do it only one time and store the result in `ListHisto.mat`. The results will be stored as a `N x M` array `listHisto` with `N = 1040` and `M = nH x nS x nV`. We will also save the names of the images as `listImage`

Set `bcomputed = False` for the first run to compute the database histograms and then set it to 1.

```
[ ]: import os
from scipy.io.matlab.mio import loadmat, savemat

#####

pathImage = './Base/'
pathDescriptors = './'

# Quantization HSV
nH =
nS =
nV =

bcomputed = True # Set to False to compute the histogramm database

if not bcomputed:
    listImage = os.listdir(pathImage)
    listHisto = []
    print('Histogram database computation ... it may take a while ...')
    for imageName in listImage:
        if os.path.isfile(pathImage+imageName) and imageName[-4:] == '.png':
            print( imageName)
            # read image
            I = np.array(Image.open(pathImage+imageName)) / 255.

            # conversion RGB->HSV
            J = rgb2hsv(I);
            # Image quantization (your function tested in Exo 1)
            _,histo = quantizeImage(J, nH, nS, nV)

            # flat a 3D histogram in 1D
            histo = histo.flatten()

            # Normalize histogramme (your function tested in Exo 1)
            listHisto.append(normalize(histo))

    print(len(listHisto), "histograms computed")
    nomList = pathDescriptors+'ListHisto.mat'
    savemat(nomList, {'listHisto': np.array(listHisto),
                      'listImage': np.array(listImage)})
else:
    print("Histogram database computation already done.")
```

Question 2.2: Computation of the similarity between every images in the base.

1. Write a function `similarityMatrix()` or a script that performs the similarity computation for every pair of images in the base from the histograms stored in `listHisto` and store the

result in a `1024 x 1024` matrix `S`. It is possible to make the operation much faster by using only one matrix operation.

[ ]:
```
mat = loadmat(pathDescriptors+'ListHisto.mat')
listHisto = mat['listHisto']
listImage = mat['listImage']

### you code below
```

2. Display the matrix `S` as an image. What can we say about this it ?

Your answer: .....

[ ]:
```

```

3. Assuming `S` is already computed and using function `display20bestMatches()`, test on the image `Liontigre1.png` (indexQuery = 349). You should obtain something similar to Figure 5.

Figure 5: 20 best matches of image 'Liontigre1.png'

[ ]:
```
# a supprimer je pense !!!!
indexQuery = 349
display20bestMatches(S, indexQuery)
imageName = (pathImage+NomImageBase[indexQuery]).strip()
```

4. Assuming `S` is already computed, generate a random query (an integer in range $[0, 1030]$), and display the 20 best matches.

[ ]:
```
import random
```

5. What can you say about the results ? What are the limitations and the advantages of this method ?

Your answer: .....

Practical works 9 & 10 : Face recognition by Eigenfaces method

The objective of this practical work is to study the properties of the Eigenfaces face recognition method.

We propose to develop a system capable of: - identify a face from a database of faces - determine whether an image contains a face present in the database - to decide whether an image represents a face or not

Tools developed in this practical work will be applied on the Yale Faces Database.

General principle

The problem of face recognition is defined as follows: given a face image, one wishes to determine the identity of the corresponding person. To this end, it is necessary to have reference images, in the form of a database of faces of all persons known by the system. Each face is associated to a vector of characteristics. These characteristics are supposed to be invariant for the same person, and different from one person to another one. Face recognition then consists of comparing the vector of characteristics of the face to be recognized with that of each of the faces of the database. This makes it possible to find the person in the database having the most similar face.

There are several types of methods, distinguished by the type of characteristics used, see *S.A. Sirohey, C.L. Wilson, and R. Chellappa. Human and machine recognition of faces: A survey. Proceedings of the IEEE, 83(5), 1995* for a state of the art:

- The approaches by face models proceed to a biometric analysis of faces. Pertinent biometrics are the distance between the eyes, length of the nose, shape of the chin...
- Image approaches, on the contrary, directly compare faces, considering them as images, for which measures of pre-attentive similarities (without a priori model) are defined.
- Hybrid approaches use the notions of similarity between images, but add a priori knowledge about the structure of a face.

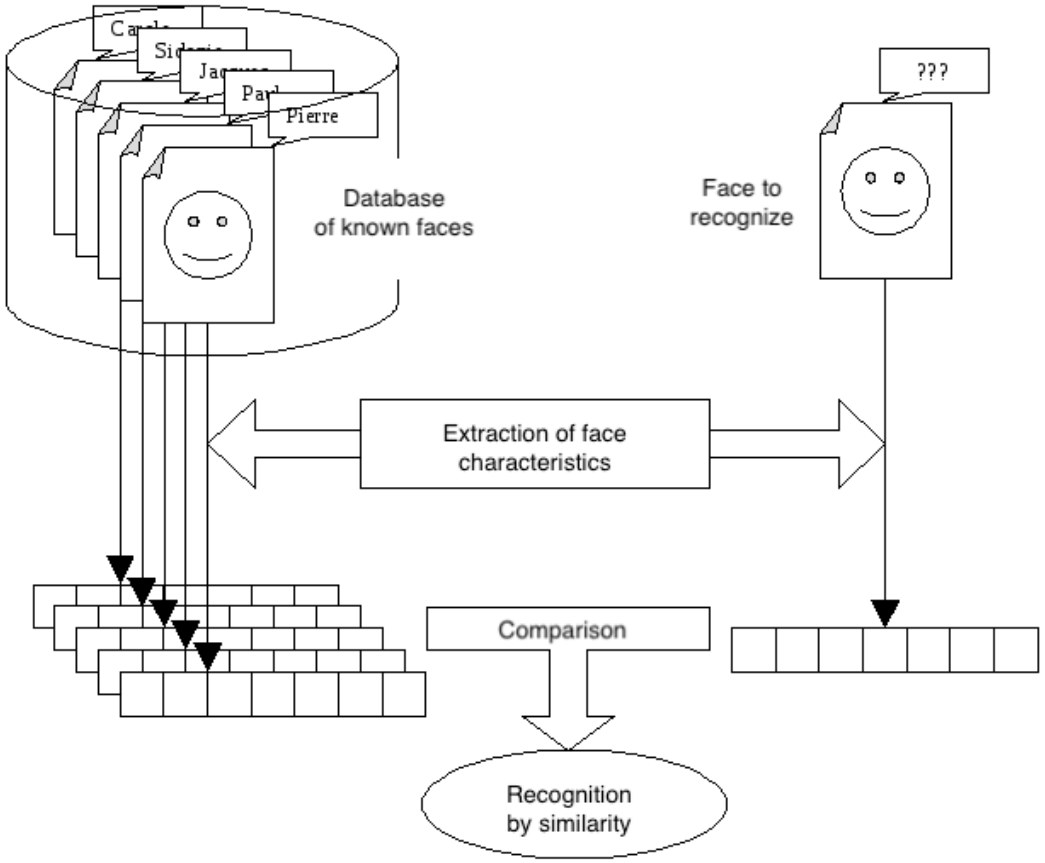


Figure 1: General Principle of a Face Recognition System

Analysis by Eigenfaces

Face recognition by Eigenfaces is an image-based approach. Each face image is considered as a vector in a space having as many dimensions as the number of pixels in the image. The image characteristics are extracted by a method of dimensionality reduction based on principal component analysis (PCA). This approach was originally proposed in 1991, see Mr. Turk and A. Pentland. Eigenfaces for recognition. J. Cognitive Neuroscience, 3(1) :71-86, 1991_.

In the following, we will use the italic notation to designate the scalars ($m, K, \ldots$) and vectors (x, u), as well as boldface for the matrices ($\mathbf{X}, \mathbf{X}_c, \mathbf{W}, \ldots$).

A face image is noted x and presented as a vector of d components. $x[i](i = 0, \cdots, d-1)$ is the pixel number i of that image. A set of faces form a cloud of points in the space $\mathbb{R}^d$. The database is divided in two parts: the *trainning* or *reference* part, used to learn the faces, and the *test* part, used to test the method. Faces of the training part are noted x_k^{train} $(k = 0, \cdots, N_{train} - 1)$ and faces of the test part are noted x_k^{test} $(k = 0, \cdots, N_{test} - 1)(k = 0, \cdots, N_{test} - 1)$.

We note $x_{average}$ the average of the reference faces, or average face. The principle of the Eigenfaces method is to model the difference of any face in relation to this average face by a set limited number of images u_h, called Eigenfaces. One image of face $x \in \mathbb{R}^d$ is thus expressed as the average face to which is added a linear combination of eigenfaces :

$$x = x_{average} + \sum_h a_h u_h + \varepsilon$$

where a_h represents the weight of *the eigenface* of index h in the face x, and ε represents the error between x and its approximation by eigenfaces (error is due to the truncation of the basis of eigenvectors). Coefficients a_h play a very important role for face recognition, because they correspond to the face coordinates x in the face subspace.

The Eigenfaces method is based on the fact that the number of eigenfaces is much smaller than the total size of the space, which is what the this is called dimensional reduction. In other words, the basis of eigenfaces is truncated, keeping only the vectors coding for the most significant information. The images are therefore analysed in a sub-space of reduced dimensions, which represents more specifically faces, among all possible types of images.

The average face is always the same for a fixed reference database, each face is examined after subtraction by the average face.

Face database

We use the Yale Faces image database, http://cvc.cs.yale.edu/cvc/projects/yalefaces/yalefaces.html. In this database, all the faces have been preprocessed, in order to resize and crop them to the size 64×64 pixels, so the images can be compared pixel per pixel.

This database contains 120 greyscale images, representing the faces of 15 people. There are 8 images per person, each corresponding to a category of images varying according to the following criteria (see figure 2): - variation of facial expression: normal, sad, sleepy, surprised, wink, happy - variation of accessories: glasses, noglasses,

Figure 2: Illustration of the shooting categories

The database is divided into two groups: the reference group will be used as a training set, the other group as a test set: - the reference base contains n images, each with a number of pixels $d = n_l \times n_c$. There are 6 images per person in the training database, so $n = 6 \times 15 = 90$. Each image is 64×64, hence $d = 4096$, - the test base contains 2 images per person so a total of 30 images. Each image is again 64×64.

In the following, we always manipulate face images in the form of vectors, and a set of faces in the form of a matrix where each column is a face. As we use Numpy, the images are stored in a multidimensionnal array of real (double). This array is viewed as a matrix $\mathbf{X}$ of size $d \times n$:

$$\mathbf{X} = [x_0, ..., x_{n-1}]$$

.

The matrix $\mathbf{X}$ is split in $\mathbf{X}^{train}$ and $\mathbf{X}^{test}$ respectively of size $d \times N_{train}$ and $d \times N_{test}$.

[ ]:

Exercise 1: loading the database, display and centring of faces

Vectors *id* and *cat* give informations about the images: $id[k]$ and $cat[k]$ are respectively the identification (an index) and the cagetory of face k. Theses vectors are available for the reference and the test bases and will be useful in the following.

To load the database, we simply have to read the Matlab file `YaleFaces.mat` provided with this notebook: it provides the matrices and vectors $\mathbf{X}^{train}, \mathbf{X}^{test}, id^{train}, id^{test}, cat^{train}, cat^{test}$.

The following code loads the database and creates the various matrices and vectors:

[12]:
```
##### Useful libraries
import numpy.linalg
import numpy as np
import matplotlib.pyplot as plt

## Loading YaleFaces database
import scipy.io

yaleFaces = scipy.io.loadmat('./YaleFaces.mat')

# The training set (90 faces)
X_train = yaleFaces['X_train']
cat_train = yaleFaces['cat_train'][0]
id_train = yaleFaces['id_train'][0]-1

# The test set (30 faces)
X_test = yaleFaces['X_test']
cat_test = yaleFaces['cat_test'][0]
id_test = yaleFaces['id_test'][0]-1

# Additional images that don't contain faces
X_noface = yaleFaces['X_noface']
```

1. Write a function that computes the average face x_{moy}.
   Tip: use `mean` function from Numpy.

[ ]:
```
def meanFaces(X):
    """ Array[d,n] -> Vector[d] """
```

2. Write a function that centers the faces.
   Recall: center means subtract the average face.

[ ]:
```
def centeredFaces(X):
    """ Array[d,n]*Vector[d] -> Array[d,n] """
```

3. Write a function `deflat()` that takes as argument a face, represented as a vector of 4096 elements, and returns an image of size 64×64.
   Important: the Yale Faces database has been created in Matlab, for which the matrices are organized column by column. It may be useful to transpose the matrix.

[ ]:
```
def deflat(V):
    """ Vector[4096] -> Array[64,64] """
```

4. Display the average face, as well as a few faces with the associated centered faces. Here is an example of the expected result:

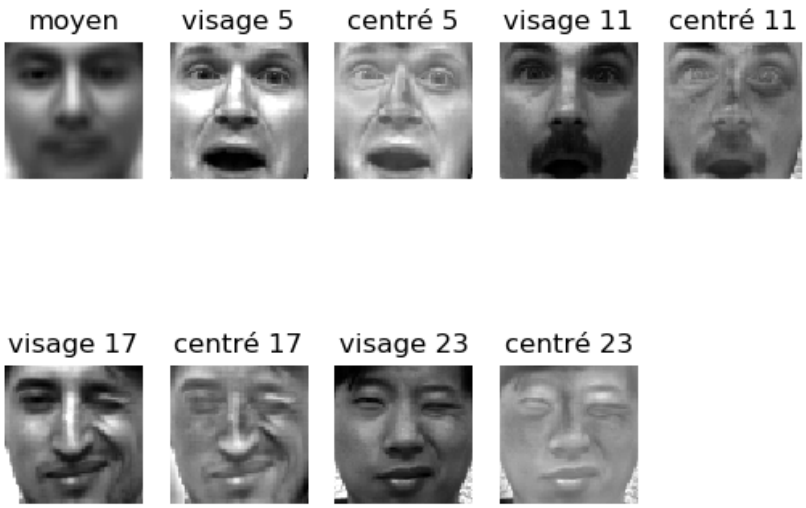


Figure 3: average face and centering of the database

[ ]:

Exercise 2: Computation of Eigenfaces (PCA)

The method developed by Turk and Pentland defines the eigenfaces as the main axes obtained by carrying out a principal component analysis (PCA) of the vectors associated with the reference faces. **The eigenfaces are thus the eigenvectors of the covariance matrix $\mathbf{X}_c\mathbf{X}_c^\top$**, of size $d \times d$, where the matrix $\mathbf{X}_c$ of the same size as $\mathbf{X}$ represents all the centered faces:

$$\mathbf{X}_c = \left[x_0 - x_{moy}, \cdots x_{n-1} - x_{moy}\right]$$

Each line of $\mathbf{X}_c$ corresponds to a pixel p, each column of $\mathbf{X}_c$ corresponds to a reference face of index $k^.

Rather than using eigenvalue decomposition, we will use singular value decomposition (SVD). The SVD decomposes the matrix $\mathbf{X}_c$ of size $d \times d$ into 3 matrices $\mathbf{U}$, $\mathbf{S}$, $\mathbf{V}$ such as :

$$\mathbf{X}_c = \mathbf{U}\mathbf{S}\mathbf{V}^\top$$

where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices ($\mathbf{U}\mathbf{U}^\top = \mathbf{U}^\top\mathbf{U} = \mathbf{I}_d^d$ and $\mathbf{V}\mathbf{V}^\top = \mathbf{V}^\top\mathbf{V} = \mathbf{I}_d^n$) of respective sizes $d \times d$ and $n \times n$, and $\mathbf{S}$ is a matrix of size $d \times n$ with null elements everywhere except on the main diagonal.

This decomposition has the following properties: - the columns of $\mathbf{V}$ are the eigenvectors of $\mathbf{X}_c^\top\mathbf{X}_c$, - the columns of $\mathbf{U}$ are the eigenvectors of $\mathbf{X}_c\mathbf{X}_c^\top$, - the matrix $\mathbf{S}$ is diagonal. The diagonal represents the singular values of $\mathbf{X}_c$, equal to the square roots of the eigenvalues λ_k of $\mathbf{X}_c^\top\mathbf{X}_c$ and $\mathbf{X}_c\mathbf{X}_c^\top$.

With Numpy, the SVD can be calculated by this way:

```
U, S, V = numpy.linalg.svd(Xc)
```

In our case, $n < d$, and the eigenvalues λ_k of $\mathbf{X}_c\mathbf{X}_c^\top$ are therefore all null for $k > n$. We will not need the associated eigenvectors $k > n$. The `svd` function has a fast mode, which calculates only the eigenvectors corresponding to the columns of the matrix passed as argument:

```
U, S, V = svd(Xc, full_matrices=False)
```

This command returns the matrices $\mathbf{U}$ and $\mathbf{V}$, of size $d \times n$ and $n \times n$, and the matrix $\mathbf{U}$ matrix has been truncated, only the first n columns are retained:

$$\mathbf{U} = [u_1, \cdots, u_n]$$

Finally `S` is a vector of size n and represents the diagonal matrix $\mathbf{S}$.

1. Write a function `eigenfaces(Xc)` which returns a t-uple consisting of the $\mathbf{U}$ matrix of eigenfaces, computed from a centered database $\mathbf{X}_c$, and the table of associated eigenvalues.

[ ]:
```
def eigenfaces(Xc):
    """ Array[d,n] -> Array[d,n]*Vector[n] """
```

2. Use this function to calculate $\mathbf{U}$ and $\mathbf{S}$. Normalize then the eigenvalues so that their sum is equal to 1.

[ ]:
```

```

3. Display the average face and the first 15 eigenfaces (see figure 4, use the `plt.subplot()` function). and their associated own values. Give your interpretation of the eigenfaces images?

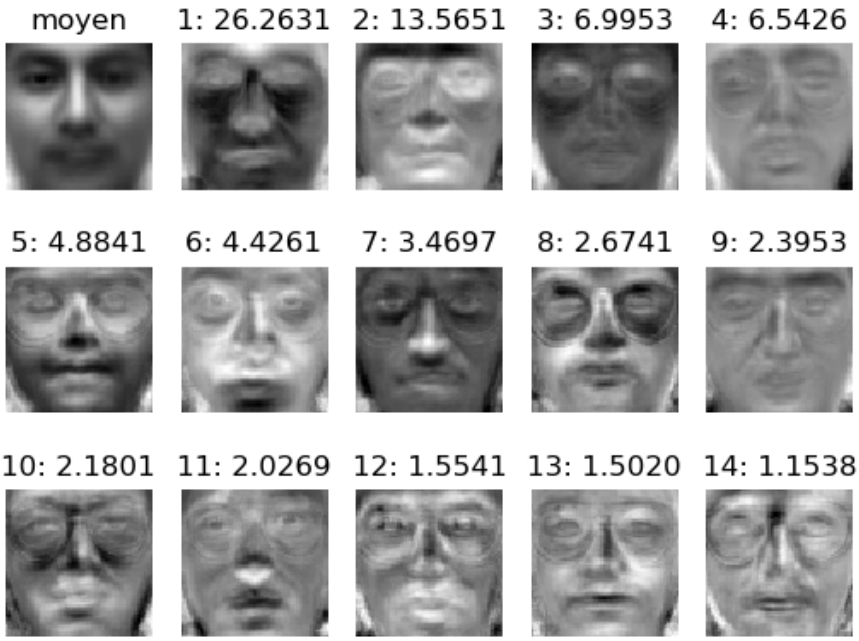


Figure 4: the 15 first eigenfaces

[ ]:

4. Plot the curve of the cumulative sum of the normalized eigenvalues (see figure 5 for the expected result), to see how much variation is captured by the first K eigenfaces. How many eigenfaces are needed to obtain a good reconstruction?

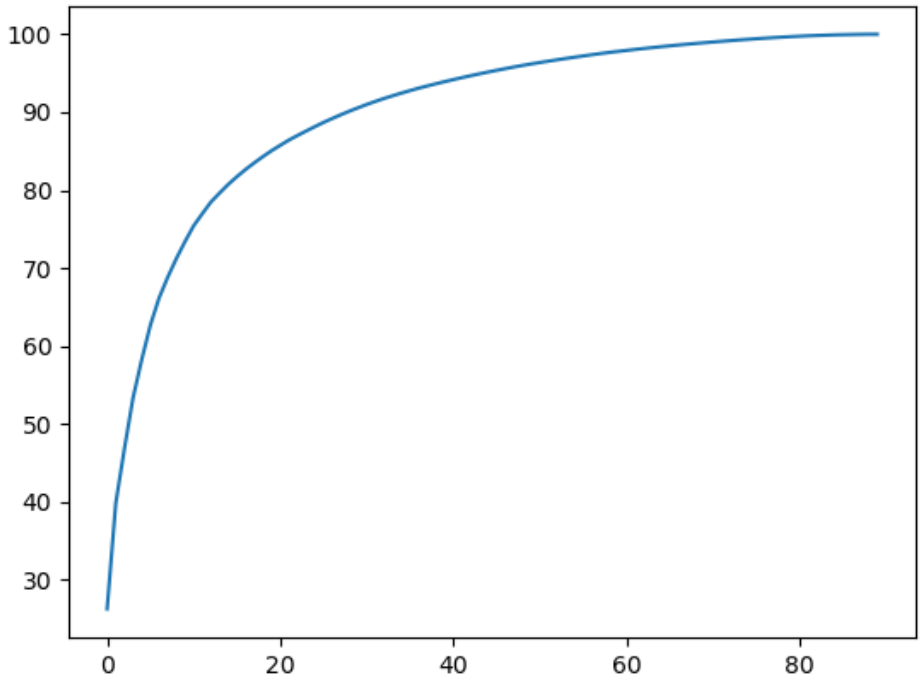


Figure 5: cumulative sum of eigenvalues

[ ]:

Exercise 3: projection in the subspace of faces

In the following, we use a reduced number of eigenfaces/eigenvectors. The vectorial space of faces, $\mathbf{W}_K$, is spanned by the basis formed with the K first eigenvectors:

$$\mathbf{W}_K = [u_1, ..., u_K]$$

Note that the set of columns of W_K is an orthonormal basis, so $\mathbf{W}_K^\top \times \mathbf{W}_K = \mathbf{I}_d^K$.

The **projection of a face image x in the face subspace** is simply done by subtracting from x the average face and applying the scalar product with each eigenvector. This gives the coordinates of the image x in the subspace of faces, which is of dimension K.

Each face therefore has several representations: - the original image, a vector $x \in \mathbb{R}^n$ - the coordinates of the projected image z in the basis of eigenfaces, $\{a_h\}$, $h \in \{1; K\}$ (subspace of faces):

$$z = \mathbf{W}_K^\top (x - x_{average})$$

- its reconstruction in the original space $\mathbb{R}^n$, $\tilde{x}$:

$$\tilde{x} = x_{average} + \sum_h a_h u_h = x_{moy} + \mathbf{W}_K z$$

The reconstruction error is defined as the distance between a face x and the associated reconstruction $\tilde{x}$:

$$E^{recons}(x) = \|x - \tilde{x}\|_2 = \sqrt{\sum_{p=1}^{n} (x(p) - \tilde{x}(p))^2}$$

1. Write a function `projEigenfaces()` which takes as arguments a face, x, the average face, $x_{average}$, the subspace of faces $\mathbf{W}$, the number of eigenfaces K, and computes the coordinates of projected face z in the subspace $\mathbf{W}_K$ of faces.

[ ]:
```
def projEigenface(x, x_mean, W, K):
    """ Vector[d]*Vector[d]*Array[d,n]*int -> Vector[K] """
```

2. Write a function `reconstruct()` which takes as arguments a projected face, z, the average face, $x_{average}$, and the truncated subspace of face, W and K, and computes the coordinate of x in the original space ($\mathbb{R}^n$).

[ ]:
```
def reconstruct(z, x_mean, W, K):
    """Vector[K]*Vector[d]*Array[d,n]*int -> Vector[d] """
```

3. Write a function `errorReconstruct()` which computes the recontruction error between $\tilde{x}$ and x.

[ ]:
```
def errorReconstruct(x_r, x):
    """Vector[d]*Vector[d] -> double """
```

4. Write a function `affiche_Reconstruction()` which displays:
   - the original face x,
   - the reconstructed faces x_r for varioous values of K (for instance, $K = 5, 10, 25, 50, 90$).

[1]:
```
def affiche_reconstruction(x, x_moy, W, listK):
    """ Vector[d]*Vector[d]*Array[d,n]*list[int] -> NoneType """
```

5. Test the previous functions by displaying the projection/reconstruction result for several images (from the training and test bases). Figure 6 shows the result of the reconstruction for image 50 of the training base. For image 55 of the training base, what is the reconstruction error for $K = n = 90$? Is the image identical to its reconstruction? Same question for image 17 of the test base.

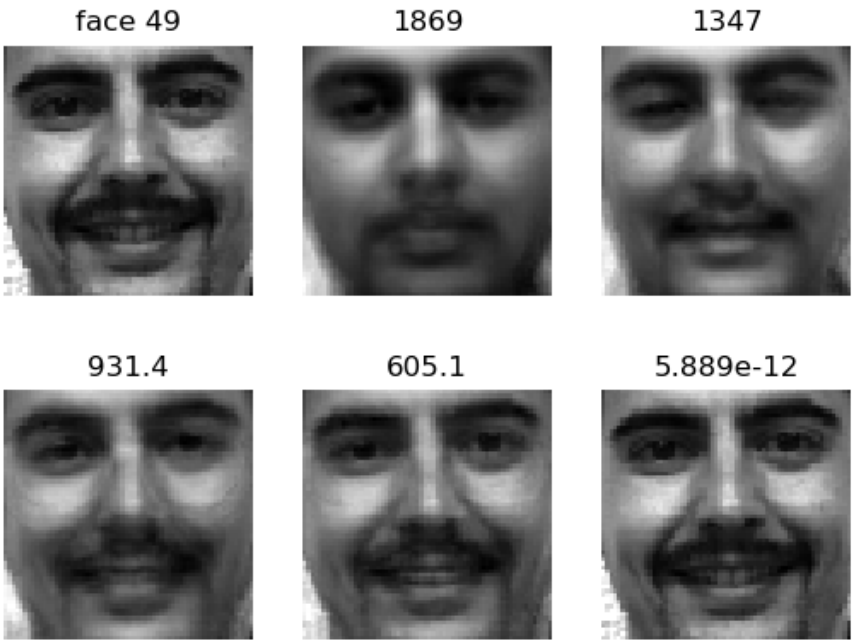


Figure 6: example of reconstruction for image number 50

[ ]:

6. Is there a difference between the reconstructions of the faces from the training base and those from the test base? Why?

[ ]:

7. Bonus question__: Plot the evolution of the average error of reconstruction of test faces when K varies from 1 to N. Is this evolution consistent with the cumulative sum previously calculated (exercise 2, question 4)?

[ ]:

Exercise 4: Face recognition and identification

Each reference face x_k^{train} has an identity associated with it, in the form of a $id^{train}(k)$ number. In this section we try to identify a face x^{test} from the reference faces.

The simplest method is to compare the projection z^{test} of the test face x^{test} with the projection z_k^{train} of each reference image x_k^{train} (see figure 7). The dissimilarity between the two projected vector is quantified by the distance in subspace $E_k(x^{test})$:

$$E_k(x^{test}) = ||z^{test} - z_k^{train}||_2$$

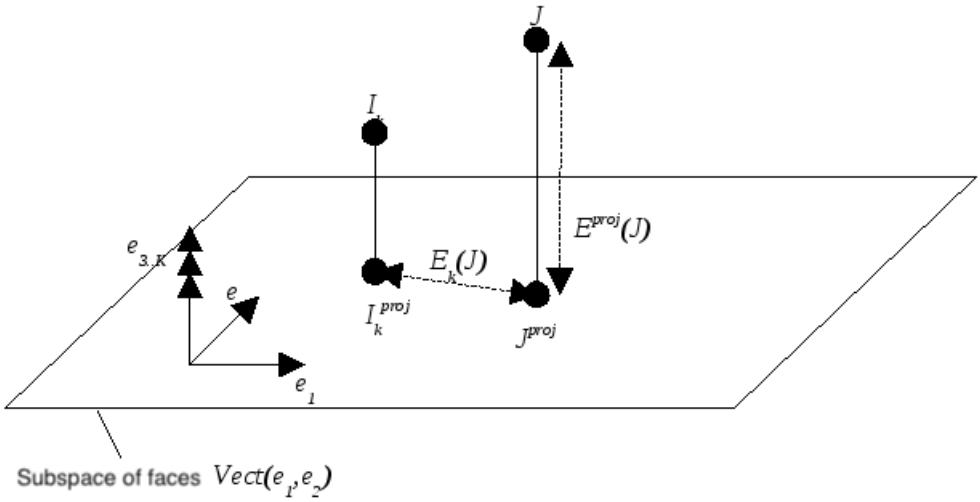


Figure 7: Projection of an image J in the subspace of faces and comparison with a reference face I_k, in case $K = 2$.

This distance is evaluated for each reference face, we can determine the reference face x_k^{train} closest to the test face x^{test}. **Its identifier $id^{train}(k)$ then allows the recognition of the tested face.**

1. What is the advantage of calculating the distance $E_k(x^{test})$ in the subspace of faces rather than in the starting space?

Your answer:

2. Write a function `computeMatDist()` which takies as arguments the training set of centered faces, $\mathbf{X}c_{train}$ of size $d \times N_{train}$, the test set of centered faces, $\mathbf{X}c_{test}$ of size $d \times N_{test}$, the subspace of eigenfaces, $\mathbf{W}$ and K, and computes the matrix D of distance between a face of the test set and a face of the training set. D is of size $N_{test} \times N_{train}$.

[3]:
```
def calculMatDist(X_train, X_test, W, K):
    """ Array[d,n]*Array[d,m]*Array[d,n]*int -> Array[m,n]"""
```

3. Write a function `identification()` which takes as argument the matrix of distances D (computed by the previous function), the vector of identification of the training set id^{train}, and returns the identificaton vector $\hat{id}^{test}$ of size N_{test} of the elements of the test set.

[2]:
```
def identification(D, id_train):
    """ Array[m,n]*Array[n] -> Array[m]"""
```

4. Compute for $K = 30$ the identification rate by comparing $\hat{id}^{test}$ to id^{test} labels. Then vary K, and plot the curve of the number of recognized faces as a function of K. Explain the shape of the curve obtained. What value of K can be taken to have a good recognition and a low calculation time?

[ ]:
```

```

5. **Bonus question**: for $K = 30$, calculate for each face of the training set its distance in the subspace $\mathbf{W}_K$ from each element of the training set. Display the result as the image of a matrix. Comment the result.

[ ]:

6. **Bonus question**: What are the minimal and maximal distances between two faces of the same category (i.e. same person)? Between two faces of different categories? If we want to choose a threshold θ to detect the presence of an unknown face, what indications do the previous min/max values give us?

[ ]:

Exercise 5: face/non-face classification

Until now, we focused on comparing facial images with each other. But the method provides information that we have not yet used. In particular, the reconstruction error can be used to verify that an image is indeed an image of a face. When an image contains something other than a face (image of a flower, a person seen in its entirety, a random image...), we can say that it is a non-face (database *noface*).

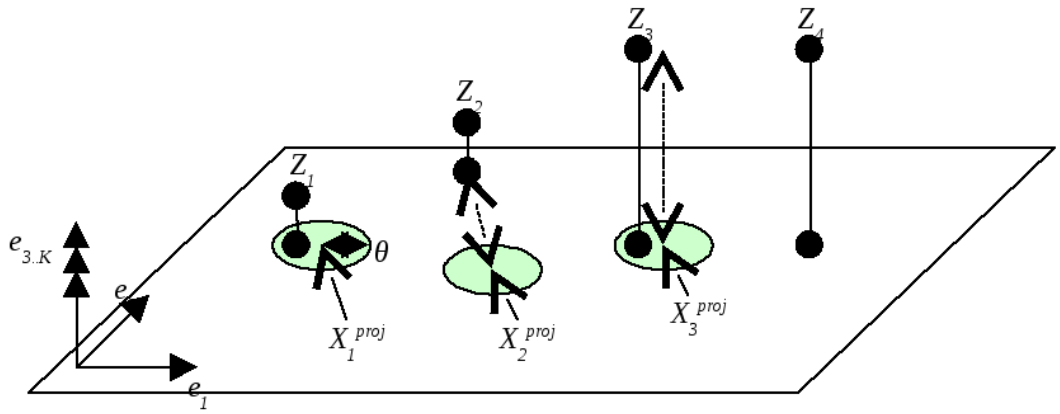


Figure 8: illustration of possible cases for classifying an image: case 1,2) Z close to subspace: it is a face 3,4) Z far from subspace: it is not a face, 1) Z is an identified face, case 2) z is an unknown face, case 3) risk of identifying Z as a face when it is not.

[ ]:

1. For each set: training set, test set, and *noface* set, plot the reconstruction error of all the images of each set (this provides 3 plots). Compute the minimal, average, maximal errors for the three sets. What conclusion can be drawn?

[ ]:

2. Visualize the reconstruction error by displaying the original image and the reconstructed image for 10 images of the face database, and for the 10 images of the noface database. Comment the results.

[ ]: