# Tracking de visage avec filtre de Kalman

IMBERT Gauthier

June 2020

## Table des matières

# 1 Étude bibliographique sur le modèle

## 1.1 Propos introductif

Les filtres de Kalman font partie de la famille des filtres à réponse impulsionnelle infinie RII(ou encore filtre récursif). Ces filtres on produisent une réponse en se basant sur les valeurs du signal d'entrée ainsi que sur les valeurs des réponses antérieures (d'où le nom de filtre récursif). Les filtres de Kalman permettent d'estimer et de prédire les différents états d'un système dynamique à partir d'une série de mesures bruitées ou incomplètes.

Ces filtres sont donc principalement utilisés pour effectuer du filtrage (suppression de bruit) ou de la prédiction sur des données manquantes.

## 1.2 Historique

L'origine du filtre est assez controversée. En effet, bien que son nom vienne du mathématicien et automaticien américain Rudolf Emil Kalman, l'astronome Thorvald Nicolai Thiele (19ème siècle) et le théoriciens du radar Peter Swerling (20ème siècle) avaient développé des algorithmes très similaires avant lui.

La première application connue de ce filtre a été réalisée par Stanley F. Schmidt, un ingénieur en aérospatial qui travaillait pour la NASA. Le filtre de Kalman a été utilisé pour estimer la trajectoire de la fusée de la mission Apollo.

## 1.3 Principe général

Le fonctionnement des filtres de Kalman ayant déjà été expliqués en cours, cette partie sera assez brève.

Néanmoins, il est indispensable de définir son principe général. L'état du filtre est quantifié au cours du temps par deux paramètres :

- $x_k$ la position à l'instant $k$
- $P_k$ la matrice de covariance à l'instant $k$

Les filtres de Kalman distinguent deux phases : la phase de prédiction, et la phase de mise à jour. La phase de prédiction, comme son nom l'indique, va prédire l'état $k$ à partir le l'état $k-1$. La phase de mise à jour, quand à elle permet de corriger l'état prédit à partir de l'observation à l'instant $k$.

Pour la prédiction, on effectue les calculs suivants :

$$x_{k_{pred}} = Ax_{k-1}$$

$$P_{k_{pred}} = AP_{k-1}A^T + Q$$

Et pour la mise à jour :

$$x_k = x_{k_{pred}} + K_k v_k$$

$$P_k = P_{k_{pred}} - K_k S_k K_k^T$$

Avec :

$$v_k = x_{k_{observe}} - Hx_{k_{pred}}$$

$$S_k = HP_{k_{pred}}H^T + R$$

$$K_k = P_{k_{pred}}H^T S_k^- 1$$

Et A la matrice de transition, H la matrice d'observation, Q le bruit de transition et R le bruit d'observation du système.

Pour notre étude, on utilisera un modèle à accélération constante ; les différentes matrices évoquées précédemment seront donc de la forme :

$$A = \begin{pmatrix} 1 & 0 & dt & 0 & 0 & 0 \\ 0 & 1 & 0 & dt & 0 & 0 \\ 0 & 0 & 1 & 0 & dt & 0 \\ 0 & 0 & 0 & 1 & 0 & dt \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$H = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$x_k = \begin{pmatrix} x \\ y \\ \dot{x} \\ \dot{y} \\ \ddot{x} \\ \ddot{y} \end{pmatrix}$$

$$Q = \begin{pmatrix} a & 0 & 0 & 0 & 0 & 0 \\ 0 & b & 0 & 0 & 0 & 0 \\ 0 & 0 & c & 0 & 0 & 0 \\ 0 & 0 & 0 & d & 0 & 0 \\ 0 & 0 & 0 & 0 & e & 0 \\ 0 & 0 & 0 & 0 & 0 & f \end{pmatrix} \quad \forall a, b, c, d, e, f \in \mathbb{R}$$

$$R = \begin{pmatrix} g & 0 \\ 0 & h \end{pmatrix} \quad \forall g, h \in \mathbb{R}$$

$$P_0 = \begin{pmatrix} i & 0 & 0 & 0 & 0 & 0 \\ 0 & j & 0 & 0 & 0 & 0 \\ 0 & 0 & k & 0 & 0 & 0 \\ 0 & 0 & 0 & l & 0 & 0 \\ 0 & 0 & 0 & 0 & m & 0 \\ 0 & 0 & 0 & 0 & 0 & n \end{pmatrix} \quad \forall i, j, k, l, m, n \in \mathbb{R}$$

# 2 Étude bibliographique sur l'application

Les domaines d'applications du filtre de Kalman sont nombreux. Comme énoncé précédemment, il est utilisé dans l'aérospatial pour prédire des trajectoires. Mais il est aussi beaucoup utilisé en cartographie et localisation simultanées (ceci consiste pour un véhicule autonome à cartographier son environnement tout en évoluant dans ce dernier).

Il n'existe à vrai dire que très peu d'exemples de tracking de visage avec un filtre de Kalman. En réalité, un filtre de Kalman ne permet pas de faire du tracking, il permet de faire de la prédiction. Il existe en revanche de nombreux exemples de tracking d'objets utilisant un filtre de Kalman pour effectuer une prédiction lorsque l'objet disparaît du champs de vision.

Nous allons donc utiliser un filtre de Kalman pour effectuer de la prévision, plutôt que du tracking.

# 3 Présentation du jeu de données choisi / construit

J'ai choisi de construire mes jeux de données moi même. Pour ce faire, je me suis filmé avec ma webcam. j'ai fait en sorte qu'à un moment de la vidéo, mon visage soir caché par un objet avec de mettre en exergue l'aspect de prédiction. La vidéo résultante est nommée (de manière très explicite) *"video.mp4"*. Voici quelques propriétés de la vidéo :

- Durée : 18 secondes
- Dimensions : 640×480
- Taux de rafraîchissement : 30 images par seconde

De plus, tout au long de mon projet j'ai directement utilisé ma webcam, afin d'avoir des résultats plus ou moins en direct (avec une petite latence d'environ 1 seconde). Ainsi, mon projet est tout à fait fonctionnel avec la webcam, ce qui est très pratique pour faire des tests.

# 4 Explication de l'implémentation

Le code se décompose en deux modules, l'un dédié à la détection des visages, et l'autre à la prédiction grâce au filtre de Kalman

## 4.1 Le module FiltreKalman

Ce module est le poumon du projet, c'est dans celui ci qu'on définit toutes les opérations liées au filtre de Kalman.

Pour ce module, on aura besoin du package *numpy*, afin notamment de faciliter les calculs matriciels.

On commence par initialiser tous les paramètres du filtre, à savoir : la matrice de l'état initial, les matrices de transition, d'observation, de covariance et les matrices de bruit de transition et d'observation.

```
    class FiltreKalman(object):
    def __init__(self, dt, std_acc, x_std_meas, y_std_meas):
        """
        Classe pour l'utilisation du filtre de Kalman
        :param dt: temps d'echantillonnage
        :param std_acc: bruit de transition standard
        :param x_std_meas: bruit de deviation standard en x
        :param y_std_meas: bruit de deviation standard en y
        """

        # On initialise dt
        self.dt = dt

        # L'etat initial (on  l'initialise a 0)
        self.x = np.matrix([[0], [0], [0], [0], [0], [0]])

        # La matrice de transition A
        self.A = np.matrix([[1, 0, self.dt, 0, 0, 0],
                            [0, 1, 0, self.dt, 0, 0],
                            [0, 0, 1, 0, self.dt, 0],
                            [0, 0, 0, 1, 0, self.dt],
                            [0, 0, 0, 0, 1, 0],
                            [0, 0, 0, 0, 0, 1]])


        # La matrice d'observation H
        self.H = np.matrix([[1, 0, 0, 0, 0, 0],
                            [0, 1, 0, 0, 0, 0]])

        # Le bruit de transiton
        self.Q = np.matrix([[(self.dt**4)/4, 0, (self.dt**3)/2, 0,
    self.dt**2, 0],
                            [0, (self.dt**4)/4, 0, (self.dt**3)/2,
    0, self.dt**2],
                            [(self.dt**3)/2, 0, self.dt**2, 0, self
    .dt, 0],
                            [0, (self.dt**3)/2, 0, self.dt**2, 0,
    self.dt],
                            [self.dt**2, 0, self.dt, 0, 1, 0],
                            [0, self.dt**2, 0, self.dt, 0, 1]]) *
    std_acc**2

        # Bruit d'observation
        self.R = np.matrix([[x_std_meas**2,0],
                            [0, y_std_meas**2]])

        # La matrice de covariance
        self.P = np.eye(self.A.shape[1])

```

On définit ensuite une première fonction : *prediction* qui, comme son nom l'indique va prédire les positions. Pour cela, on applique simplement les formules vues en cours, à savoir : $x_k = Ax_{k-1}$ pour la prédiction de la position, et

$P_k = AP_{k-1}A^T + Q$ pour la mise à jour de la matrice de covariance.

```
    def prediction(self):
        """Retourne la position predite en applicant un filtre de
    Kalman"""
        # Calcul des positions : x_k = Ax_(k-1)
        self.x = np.dot(self.A, self.x)

        # Mise a jour de P : P_k = A*P_(k-1)*A' + Q
        self.P = np.dot(np.dot(self.A, self.P), self.A.T) + self.Q

        return self.x[0:2]

```

La seconde fonction (*maj*) fait la mise à jour du filtre. Les calculs se basent encore une fois sur les calculs vus en cours : $S_k = HP_kH^T + R$ ; le gain du filtre : $K_k = P_kH^TS_k^-1$ ; enfin la mise à jour de $P$ :

$$P_k = P_{k-1} - K_kS_kK_k^T \Leftrightarrow P_k = (I - K_kH)P_{k-1}$$

```
    def maj(self, z):
        """Correction de la prediction grace a l'observation k"""
        # S_k = H*P_k*H'+R
        S = np.dot(self.H, np.dot(self.P, self.H.T)) + self.R

        # Calcul du gain du filtre : K_k = P_k * H'* S_k^-1
        K = np.dot(np.dot(self.P, self.H.T), np.linalg.inv(S))

        # Calcul des coordonnes maj
        self.x = np.round(self.x + np.dot(K, (z - np.dot(self.H,
    self.x))))

        I = np.eye(self.H.shape[1])
        # Maj de P
        self.P = (I - (K * self.H)) * self.P
        return self.x[0:2]

```

Les fonctions de ce module seront utilisés dans le programme principal décrit ci-après.

## 4.2 Le programme principal : détection de visage et application du filtre de Kalman

Pour la détection et le tracking des visages, nous allons utiliser le package *opencv-python* (cv2). OpenCV (Open Source Computer Vision Library) est une librairie qui comporte de nombreuses méthodes pour effectuer tu traitement d'image, de l'analyse de vidéo, ou encore de la détection (tout ce qui se rapporte à la vision en général). Nous allons donc utiliser sa version python.

Pour reconnaître les visages, nous allons utiliser un "cascade classifier" qui fonctionne grossièrement de la façon suivante : on l'entraîne avec une base de données de visages (ou autre objet) suffisamment étoffée, puis on l'utilise pour

détecter le visage dans un environnement [1]. La base de données que nous allons utiliser est un fichier XML regroupant des données brutes permettant de détecter des visages; ce fichier est nommé *"haarcascade_frontalface_default.xml"*[2].

```
    #On cree le face cascade
    cascPath = "haarcascade_frontalface_default.xml"
    faceCascade = cv2.CascadeClassifier(cascPath)

```

Une fois notre faceCascade créée, il nous faut lire la vidéo que l'on souhaite traiter. Pour cela, on utilise la méthode *VideoCapture*.

```
    # Lecture de la video (0 pour la webcam, 'video.mp4' pour la
    video)
    #video = cv2.VideoCapture(0)
    video = cv2.VideoCapture('video.mp4')

```

On initialise également notre filtre de Kalman ainsi que deux variables : hauteur et largeur qui correspondent à la hauteur et largeur du visage.

```
    largeur = 0
    hauteur = 0
    # Initialisation du filtre
    filtre_k = FiltreKalman(1/30, 1, 0.1,0.1)

```

Nous allons par la suite entrer dans une boucle infinie afin de traiter la vidéo jusqu'à la fin, les étapes décrites ci-après sont à l'intérieur de la boucle. On commence par initialiser la liste qui va accueillir les positions des visages et à lire la vidéo image par image avec la méthode *read()*.

```
    # Une liste de couples (x,y) correspondant aux positions des
    visages
    centres_visages=[]
    # Lecture de la video image par image
    retval, image = video.read()

```

Par la suite, pour faciliter le traitement, on convertit la vidéo en niveaux de gris.

```
    # Conversion de l'image en nuances de gris (pour faciliter le
    traitement)
    image_nvgris = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

```

Une fois l'image convertie en niveaux de gris, il ne nous reste plus qu'à détecter les visages grâce à la fonction *detectMultiscale*. Cette fonction prend en entrée l'image en niveaux de gris. Elle a deux paramètres :

- *minNeighbors*, qui permet de ne pas détecter 10 fois le même visage en indiquant la distance minimale possible entre deux visages

[1] Le principe est bien mieux expliqué ici, mais ce n'est pas le propos du projet : `https://realpython.com/traditional-face-detection-python/`

[2] Le fichier est accessible à l'url suivant : `https://github.com/shantnu/FaceDetect/blob/master/haarcascade_frontalface_default.xml`

- *minSize*, qui permet de ne pas considérer des tout petits éléments comme des visages en spécifiant la taille minimal des visages

```
    # Detection des visages dans l'image en nv de gris
    visages = cascade_de_visages.detectMultiScale(
        image_nvgris,
        minNeighbors=5,
        minSize=(30, 30)
    )

```

Pour bien repérer les visages, on dessine un carré vert autour de chacun des visages

```
    # Dessine un rectangle autour de chaque visage
    for (x, y, w, h) in visages:
        cv2.rectangle(image, (x, y), (x+w, y+h), (0, 255, 0), 2)
        centres_visages.append(np.array([[x], [y]]))
        largeur = w #la largeur nous sera utile pour plus tard
        hauteur = h #la hauteur //      //      //      //

```

Vient ensuite la phase de prédiction, on calcule les coordonnées prédites grâce à la méthode *prediction* décrite précéement et on dessine un rectangle bleu (avec une légende également).

```
    # Prediction grace au filtre de Kalman
    (x_pred, y_pred) = filtre_k.prediction()
    # Dessine un rectangle a la position predite
    cv2.rectangle(image, (x_pred, y_pred), (x_pred + largeur,
    y_pred + hauteur), (255, 0, 0), 2)
    # Ajout d'une legende
    cv2.putText(image, "Position predite", (x_pred + 15, y_pred),
    0, 0.5, (255, 0, 0), 2)

```

Il ne nous reste plus qu'à mettre à jour je filtre et à éventuellement corriger la prédiction (si et seulement si on détecte un visage) en utilisant la fonction *maj*.

```
    # Fase de maj du filtre Kalman
    if centres_visages != []:
        (x_estim, y_estim) = filtre_k.maj(centres_visages[0])
        # Dessine un rectangle a la position estimee grace a Kalman
        cv2.rectangle(image, (x_estim, y_estim), (x_estim + largeur
    , y_estim + hauteur), (0, 0, 255), 2)
        # Ajout d'une legende
        cv2.putText(image, "Position estimee", (x_estim + 15,
    y_estim + 10), 0, 0.5, (0, 0, 255), 2)

```

Le code suivant est optionnel mais très pratique puisqu'il permet de sortir de la boucle infinie sans avoir à interrompre le programme manuellement. *waitKey* attends un évènement clavier (passe à True quand une touche est pressée).

```
    # Pour sortir de la boucle en appuyant sur 'q'
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

```

Il faut bien évidement ne pas oublier d'afficher les images :

```
    # Affichage du resultat
    cv2.imshow('Traitement de la video', image)

```

Et enfin, dès qu'on sort de la boucle, on arrête la lecture de la vidéo et on détruit la fenêtre affichant les résultats.

```
    # Quand on sort de la boucle, on ferme tout
    video.release()
    cv2.destroyAllWindows()

```

# 5 Choix des hyper-paramètres

Le choix des hyper-paramètres pour un filtre de Kalman est très compliqué. En effet, il est très difficile de quantifier concrètement les différents bruits (covariance, bruit de transition et bruit d'observation). Après de nombreux essais à tâtons et non concluants, j'ai commencé à effectuer des recherches. J'ai alors trouvé un exemple de filtre de Kalman, j'ai essayé avec les valeurs utilisées et j'ai obtenu des résultats bien meilleurs. Voici donc les valeurs prises pour les bruits :

$$Q = \begin{pmatrix} \frac{dt^4}{4} & 0 & \frac{dt^3}{2} & 0 & dt^2 & 0 \\ 0 & \frac{dt^4}{4} & 0 & \frac{dt^3}{2} & 0 & dt^2 \\ \frac{dt^3}{2} & 0 & dt^2 & 0 & dt & 0 \\ 0 & \frac{dt^3}{2} & 0 & dt^2 & 0 & dt \\ dt^2 & 0 & dt & 0 & 1 & 0 \\ 0 & dt^2 & 0 & dt & 0 & 1 \end{pmatrix}$$

$$R = \begin{pmatrix} 0.1^2 & 0 \\ 0 & 0.1^2 \end{pmatrix}$$

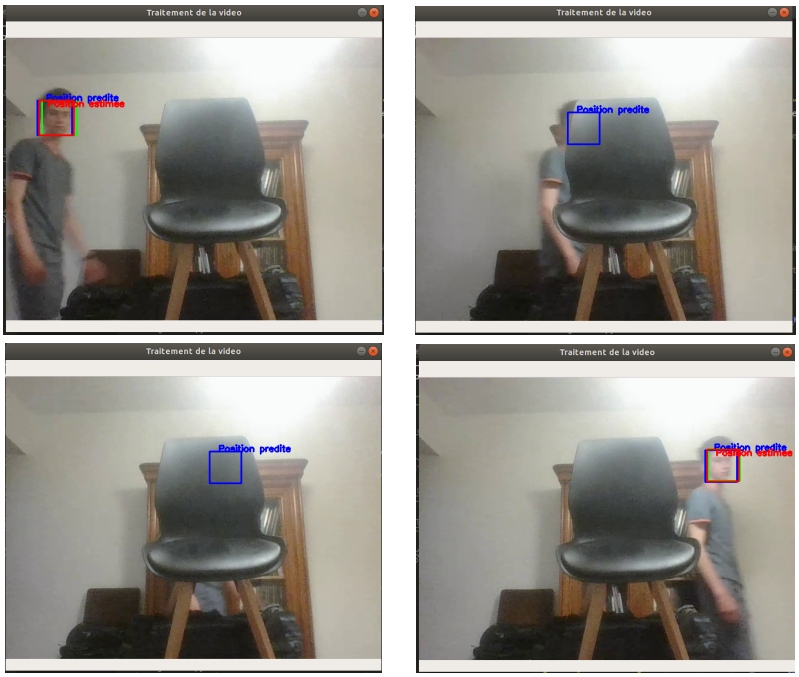
$$P_0 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Il reste un autre paramètre, celui du $dt$. Le choix de ce paramtre, contrairement aux autres, a été relativement simple. En effet, vu que la vidéo a une capacité de 30 images par secondes et que la webcam possède la même capacité, on prendra donc $dt = 1/30$.

# 6 Résultats et interprétations

Les résultats sont assez concluants, le tracking de visage fonctionne très bien et la prédiction est assez fiable malgré quelques buggs défauts. En effet, si on change totalement de direction au moment de la prédiction, cette dernière sera erronée. Aussi, le modèle choisi, est celui de l'accélération constante ; or ce n'est pas toujours le cas lors d'un déplacement de personnes, et ce n'est donc peut être pas le modèle le plus adapté pour prédire des déplacements humains, qui à l'instar de certains déplacements mécaniques ou physiques, ne sont pas totalement prévisibles.

Malgré tout, le programme semble fonctionner dans la majorité des cas, et les captures d'écran ci-dessous en témoignent.




Résultat du traitement avec la vidéo *"video.mp4"*

On constate bien sur ces captures que la prédiction de trajectoire est cohérente.

## Bibliographie

Documentation numpy : https://numpy.org/doc/

Documentation OpenCV : https://opencv-python-tutroals.readthedocs.io/en/latest/#

Filtre Kalman : https://fr.wikipedia.org/wiki/Filtre_de_Kalman
Cours filtre Kalman : CM06-Filtre de Kalman, Romain Hérault

Reconnaissance de visages : https://realpython.com/face-recognition-with-python/
Tracking 2d : https://machinelearningspace.com/2d-object-tracking-using-kalman-filter/

Tracking : https://www.guidodiepen.nl/2017/02/detecting-and-tracking-a-face-with-python-and-opencv/

# Annexes

Code complet du module FiltreKalman :

```
"""
Module du filtre de Kalman
"""
import numpy as np

class FiltreKalman(object):
    def __init__(self, dt, std_acc, x_std_meas, y_std_meas):
        """
        Classe pour l'utilisation du filtre de Kalman
        :param dt: temps d'echantillonnage
        :param std_acc: bruit de transition standard
        :param x_std_meas: bruit de deviation standard en x
        :param y_std_meas: bruit de deviation standard en y
        """

        # On initialise dt
        self.dt = dt

        # L'etat initial (on  l'initialise    0)
        self.x = np.matrix([[0], [0], [0], [0], [0], [0]])

        # La matrice de transition A
        self.A = np.matrix([[1, 0, self.dt, 0, 0, 0],
                            [0, 1, 0, self.dt, 0, 0],
                            [0, 0, 1, 0, self.dt, 0],
                            [0, 0, 0, 1, 0, self.dt],
                            [0, 0, 0, 0, 1, 0],
                            [0, 0, 0, 0, 0, 1]])


        # La matrice d'observation H
        self.H = np.matrix([[1, 0, 0, 0, 0, 0],
                            [0, 1, 0, 0, 0, 0]])

        # Le bruit de transiton
        self.Q = np.matrix([[(self.dt**4)/4, 0, (self.dt**3)/2, 0,
    self.dt**2, 0],
                            [0, (self.dt**4)/4, 0, (self.dt**3)/2,
    0, self.dt**2],
                            [(self.dt**3)/2, 0, self.dt**2, 0, self
    .dt, 0],
                            [0, (self.dt**3)/2, 0, self.dt**2, 0,
    self.dt],
                            [self.dt**2, 0, self.dt, 0, 1, 0],
                            [0, self.dt**2, 0, self.dt, 0, 1]]) *
    std_acc**2

        # Bruit d'observation
        self.R = np.matrix([[x_std_meas**2,0],
                            [0, y_std_meas**2]])

        # La matrice de covariance
        self.P = np.eye(self.A.shape[1])

```

```
    def prediction(self):
        """Retourne la position pr dite en applicant un filtre de Kalman"""
        # Calcul des positions : x_k = Ax_(k-1)
        self.x = np.dot(self.A, self.x)

        # Mise a jour de P : P_k = A*P_(k-1)*A' + Q
        self.P = np.dot(np.dot(self.A, self.P), self.A.T) + self.Q

        return self.x[0:2]

    def maj(self, z):
        """Correction de la prediction grace a l'observation k"""
        # S_k = H*P_k*H'+R
        S = np.dot(self.H, np.dot(self.P, self.H.T)) + self.R

        # Calcul du gain du filtre : K_k = P_k * H'* S_k^-1
        K = np.dot(np.dot(self.P, self.H.T), np.linalg.inv(S))

        # Calcul des coordonnes maj
        self.x = np.round(self.x + np.dot(K, (z - np.dot(self.H, self.x))))

        I = np.eye(self.H.shape[1])
        # Maj de P
        self.P = (I - (K * self.H)) * self.P
        return self.x[0:2]
```

Code complet du programme principal :

```
from cv2 import cv2
from FiltreKalman import FiltreKalman
import numpy as np

chemin = "haarcascade_frontalface_default.xml"
cascade_de_visages = cv2.CascadeClassifier(chemin)

# Lecture de la video (0 pour la webcam, "video.mp4" pour la video)
#video = cv2.VideoCapture(0)
video = cv2.VideoCapture('video.mp4')

largeur = 0
hauteur = 0

# Initialisation du filtre
filtre_k = FiltreKalman(1/30, 1, 0.1,0.1)

# Boucle infinie pour traiter la vid o jusqu'   ce qu'elle soit finie
while True:
    # Une liste de couples (x,y) correspondant aux positions des visages
    centres_visages=[]
    # Lecture de la vid o image par image
    retval, image = video.read()

    # Conversion de l'image en nuances de gris (pour faciliter le traitement)
```

```
    image_nvgris = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Detection des visages dans l'image en nv de gris
    visages = cascade_de_visages.detectMultiScale(
        image_nvgris,
        minNeighbors=5,
        minSize=(30, 30)
    )

    # Dessine un rectangle autour de chaque visage
    for (x, y, w, h) in visages:
        cv2.rectangle(image, (x, y), (x+w, y+h), (0, 255, 0), 2)
        centres_visages.append(np.array([[x], [y]]))
        largeur = w #la hauteur et largeur nous sont utiles pour plus tard
        hauteur = h

    # Prediction gr ce au filtre de Kalman
    (x_pred, y_pred) = filtre_k.prediction()
    # Dessine un rectangle    la position pr dite
    cv2.rectangle(image, (x_pred, y_pred), (x_pred + largeur, y_pred + hauteur), (255, 0, 0), 2)
    # Ajout d'une l gende
    cv2.putText(image, "Position predite", (x_pred + 15, y_pred), 0, 0.5, (255, 0, 0), 2)

    # Fase de maj du filtre Kalman
    if centres_visages != []:
        (x_estim, y_estim) = filtre_k.maj(centres_visages[0])
        # Dessine un rectangle    la position estim e gr ce Kalman
        cv2.rectangle(image, (x_estim, y_estim), (x_estim + largeur, y_estim + hauteur), (0, 0, 255), 2)
        # Ajout d'une l gende
        cv2.putText(image, "Position estimee", (x_estim + 15, y_estim + 10), 0, 0.5, (0, 0, 255), 2)

    # Pour sortir de la boucle en appuyant sur 'q'
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break


    # Affichage du r sultat
    cv2.imshow('Traitement de la video', image)


# Quand on sort de la boucle, on ferme tout
video.release()
cv2.destroyAllWindows()
```