

OpenCV ROS - Detección Facial	
Facilitador	José Carlos Rangel Ortiz

1. Realidad Aumentada

1.1. Creación de Marcadores (Markers)

Un elemento importante que nos permitirá aplicar conceptos de realidad aumentada serán los marcadores o *markers* y los diccionarios de estos. Estos están compuestos de un conjunto de celdas internas y externas. Las externas se pintan de negro y forman un borde que puede ser detectado rápida y robustamente. Las celdas internas se utilizan para codificar el marcador. Los marcadores pueden ser de diversos tamaños y cuando se habla de su dimensión hace referencia al tamaño de las celdas internas de la matriz.

En ArUco, se maneja el concepto de diccionarios de marcadores, estos son el conjunto de marcadores que serán utilizados en una aplicación específica. Este módulo permite una manera automática de generación de marcadores con la cantidad de marcadores y la cantidad de celdas que deben tener. De igual manera incluye un conjunto de diccionarios predefinidos para utilizar en nuestras aplicaciones.

Para utilizar los marcadores en nuestro código, debemos, como primera instancia definir el diccionario que utilizaremos y en este ejemplo seleccionar un primer marcador para que se muestre en pantalla y se guarde en disco, se pueda imprimir y así usarse en otras secciones.

En esta sección utilizaremos el código `01_aruco_create_markers.py`. En este nos encontraremos con el código que permite usar un directorio y guardarlo en disco. Dentro de los posibles diccionarios se pueden mencionar:

- `DICT_4X4_50 = 0`
- `DICT_4X4_100 = 1`
- `DICT_4X4_250 = 2`
- `DICT_4X4_1000 = 3`
- `DICT_5X5_50 = 4`
- `DICT_5X5_100 = 5`
- `DICT_5X5_250 = 6`
- `DICT_5X5_1000 = 7`

- DICT_6X6_50 = 8
- DICT_6X6_100 = 9
- DICT_6X6_250 = 10
- DICT_6X6_1000 = 11
- DICT_7X7_50 = 12
- DICT_7X7_100 = 13
- DICT_7X7_250 = 14
- DICT_7X7_1000 = 15

1. Como primer paso seleccionamos el diccionario, en nuestro caso utilizaremos el diccionario **DICT_7X7_250**, este tiene una dimensión de 7×7 ($n = 7$) celdas y cuenta con un total de 250 marcadores diferentes. El número después del signo $=$, indica el índice del diccionario en la librería. Para más información de los diccionarios puede consultar la documentación de este módulo en la web¹.
2. Esta selección de diccionario se realiza en el siguiente fragmento de código:

```
aruco_dictionary = cv2.aruco.Dictionary_get(cv2.aruco.DICT_7X7_250)
```

3. Posterior a esto se puede utilizar la función `cv2.aruco.drawMarker()` para obtener un marcador listo para impresión. Utilizando el código a continuación, este recibe como parámetros el diccionario a utilizar, el *id* del marcador deseado, el tamaño en píxeles que se desea dar al elemento (*sidePixels*) y como último elemento la cantidad de celdas que se utilizarán en el borde del marcador.

```
aruco_marker_1 = cv2.aruco.drawMarker( dictionary=aruco_dictionary,
                                         id=2,
                                         sidePixels=600,
                                         borderBits=1)
```

4. Ejecute este script desde la consola, utilizando el siguiente código, o mediante su IDE elegido.

```
python 01_aruco_create_markers.py
```

¹[Diccionarios en ArUco](#)

5. Luego de su ejecución tendrá en su carpeta (donde esta el script) un conjunto de 3 imágenes que contienen un marcador. Para las siguientes secciones debe imprimir el marcador con el nombre `marker_DICT_7X7_250_600_1.png`, puede imprimirla en una hoja tamaño carta o utilizando solo la mitad de esta. El marcador debe verse como en la figura 1 y este caso corresponde con el marcador con $id = 2$ en el diccionario seleccionado. Puede cambiar los $id's$ e imprimir diversos marcadores y así probar los algoritmos con diferentes marcadores. También puede utilizar el siguiente enlace para obtener marcadores Aruco específicos <https://chev.me/arucogen/>

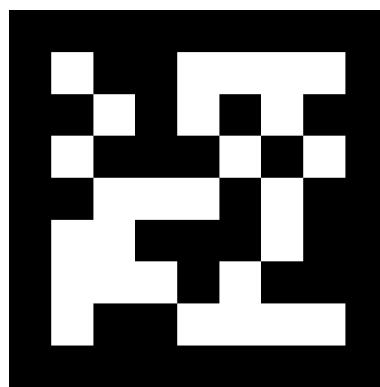


Figura 1: Marcador generado.

1.2. Detección de Marcadores

En esta sección utilizaremos el código `02_aruco_detect_markers.py`, para detectar marcadores en una imagen que se está obteniendo desde una cámara conectada al computador.

1. Como primer paso debemos indicar cual es el diccionario que utilizaremos en la aplicación, por lo cual, serán los marcadores de este diccionario los que podrá identificar nuestro código. Lo cual se hace mediante el siguiente código, se utiliza en este caso un diccionario de los pre-definidos en ArUco, debe coincidir con el diccionario utilizado para crear el marcador impreso en la sección anterior:

```
# Definimos nuestro diccionario
aruco_dictionary = cv2.aruco.Dictionary_get(cv2.aruco.DICT_7X7_250)
```

2. En este caso se utiliza la función `detectMarkers()` para procesar una imagen de entrada y buscar marcadores en la misma, esta función recibe como parámetros, una imagen en escala de grises, el objeto diccionario y un tercer parámetro que define todos los parámetros que se pueden ajustar en el proceso de reconocimiento.

```
# Detectamos los marcadores en una imagen de entrada
corners, ids, rejected_corners = cv2.aruco.detectMarkers(gray_frame,
                                                       aruco_dictionary,
                                                       parameters=parameters)
```

3. Este método devuelve:

- Una lista de las esquinas de los marcadores detectados
- Una lista de los ID's de los marcadores detectados
- Una lista de las esquinas que fueron detectadas, pero que no corresponden con ningún marcadores (lista de rechazados).

4. Luego de la ejecución del script, se identificaran los marcadores y sus ID's en la pantalla y se marcarán los elementos rechazados en la imagen. Para la ejecución escriba lo siguiente en una consola:

```
python 02_aruco_detect_markers.py
```

5. Para cerrar la ventana de visualización puede presionar la letra 'q' en su teclado.

1.3. Calibración de la Cámara

Un proceso de calibración de una cámara consiste en calcular los parámetros internos y externos de la misma. Estos definen el modelo de la misma tanto lineal como no lineal. En la actualidad existen numerosos métodos de calibración que utilizan diferentes tipos de patrones o plantillas, así como también existen los métodos que no hacen uso de las mismas [?]. Estos métodos se enfocan en utilizar patrones con dimensiones conocidas y tomando en cuenta dichas dimensiones calcular los parámetros del dispositivo. La calibración permite entre otras cosas obtener resultados más precisos cuando se realicen cálculos de distancias en las imágenes obtenidas por la cámara.

En nuestro caso ArUco cuenta con una función que permite calibrar las cámaras mediante el uso de un conjunto de marcadores de un diccionario seleccionado. Este es un procedimiento que se realiza solo una vez debido a que la óptica de la cámara no será modificada en ningún momento, por lo cual, el resultado obtenido se puede seguir utilizando en las demás partes del laboratorio donde se necesiten. El resultado de la calibración será un archivo serializado con la información obtenida por la función `cv2.aruco.calibrateCameraCharuco()`.

En ArUco para realizar la calibración se utilizan un conjunto de esquinas de varios puntos de vista, extraídos de un tablero con marcadores.

La función de calibración devuelve la matriz de la cámara (3×3 de punto flotante), esta matriz contiene la distancia focal y el centro de coordenadas de la cámara llamados también parámetros intrínsecos. Como segundo elemento la función devuelve un vector que contiene los coeficientes de distorsión, estos modelan la distorsión que es producida por la cámara.

En este caso utilizaremos el código `03_aruco_camera_calibration.py` disponible en los archivos del laboratorio.

1. Como primer paso para el proceso de calibración debemos construir el patrón de calibración que utilizaremos para nuestra cámara. Se define un tablero (similar al ajedrez) con unas dimensiones pre-definidas.
2. Esto se realiza mediante la función `cv2.aruco.CharucoBoard_create()`. En esta función definimos cuantos espacios/bloques/casillas tendrá nuestro patrón de calibración (3 filas y 3 columnas en nuestro caso), las longitud de cada celda de esta matriz (ancho en metros), la longitud de cada marcador en cada celda (ancho en metros) y también el diccionario de donde se obtendrán los marcadores para construir el patrón. Esta función toma los 4 primeros elementos del diccionario para crear la plantilla.
3. Para definir el diccionario, patrón y guardarlo en disco, se emplea el siguiente código:

```
# Definir el diccionario y crear el tablero para calibración
dictionary = cv2.aruco.Dictionary_get(cv2.aruco.DICT_7X7_250)
board = cv2.aruco.CharucoBoard_create(3, 3, .025, .0125, dictionary)

# Crear la imagen que contendrá el patrón
image_board = board.draw((200 * 3, 200 * 3))

# Guardar la imagen
cv2.imwrite('charuco.png', image_board)
```

4. La función para crear el tablero, recibe las dimensiones (cantidad de filas y columnas) que debe tener el tablero que crearemos, las dimensiones de un cuadrado del tablero (en metros), dimensión del marcador (en metros) y el diccionario que se utilizará.

5. Para guardar nuestro patrón debemos ejecutar este código de la siguiente manera:

```
python 03_aruco_camera_calibration.py
```

6. La primera ejecución guardará la imagen `charuco.png` en nuestra carpeta y a continuación luego de activar la cámara es probable que se produzca un error ya que no se está detectando ningún patrón.

7. La imagen en disco debe tener la apariencia similar a la mostrada en la figura 2.

8. Como siguiente punto debemos entonces imprimir este patrón cuidando que se mantengan las dimensiones definidas. Si al imprimir y medir las dimensiones reales del panel de calibración no concuerdan con las expresadas en el código, se recomienda modificar el código agregando los valores reales medidos en el patrón impreso.

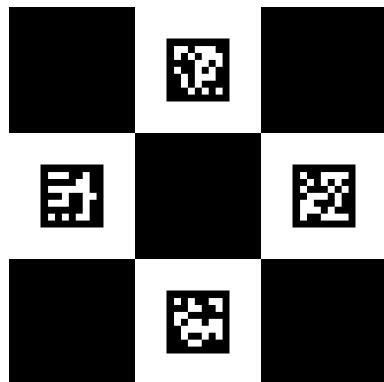


Figura 2: Patrón con marcadores generados para calibración.

9. A continuación con el patrón debemos ejecutar nuevamente el código de calibración y enfocarlo hacia nuestra cámara.
10. Durante esta ejecución se pueden presentar errores ya que pueden ocurrir muchas situaciones cuando se está captando el patrón y calculando los parámetros.
11. El código que almacena los parámetros es el siguiente, se emplea la librería de serialización pickle

```
# Obtener resultados de calibración:
retval, cameraMatrix, distCoeffs, rvecs, tvecs = cal

# Almacenar parámetros de la cámara:
f = open('calibration.pckl', 'wb')
pickle.dump((cameraMatrix, distCoeffs), f)
f.close()
```

12. Cuando el código se ha ejecutado sin problemas en su carpeta debe estar el archivo `calibration.pckl` el cual contendrá los parámetros calculados para la cámara.
13. Los parámetros de calibración son únicos para cada cámara, por lo cual, si usamos un archivo de calibración generado por otro equipo, es muy probable que se presenten errores al momento de hacer los cálculos.
14. Debe tomar en cuenta que cámara utilizará, en el caso de que tenga varias conectadas al computador. Nuestro código capture `frames` en tiempo real desde la cámara seleccionada y los proyecta en pantalla. Para seleccionar qué cámara utilizar se puede modificar el argumento en la siguiente función, el 0 indica la cámara empotrada para laptops, en este caso si usamos una segunda `webcam` debemos colocar el número 1 como argumento.

```
# Definir cámara
cap = cv2.VideoCapture(0)
```

15. Una vez ejecutado el programa 2 veces y obtenidos los parámetros, se puede proseguir con los siguientes ejemplos.

1.4. Detección de Pose de la Cámara

En la realidad aumentada (AR, por sus siglas en inglés) el procedimiento común consiste en superponer un objeto sobre un marcador que ha sido detectado en la imagen. Para ello OpenCV necesita conocer cual es la pose del marcador detectado respecto a la cámara que se esta utilizando. Se utilizará el archivo `04_aruco_detect_markers_pose.py`.

En este caso se utilizan los parámetros internos de la cámara calculados en el paso anterior y almacenados en el archivo `calibration.pckl`, para determinar la pose de un marcador que esta presente en una imagen captada desde una webcam.

Recuerde que si usa más de una cámara debe definir el índice de la cámara que utilizará y la cual fue calibrada anteriormente.

1. En este caso nuestra primera acción será indicarle a OpenCV los parámetros de calibración de nuestra cámara por lo cual se carga y des-serializa el archivo `calibration.pckl`.

```
# abrir archivo con la información de la cámara
f = open('calibration.pckl', 'rb')
cameraMatrix, distCoeffs = pickle.load(f)
f.close()
```

2. De igual manera al utilizar marcadores debemos definir el diccionario de marcadores que utilizaremos para que el programa sepa que esta buscando.

```
aruco_dictionary = cv2.aruco.Dictionary_get(cv2.aruco.DICT_7X7_250)
```

3. Luego el procesamiento de las imágenes de entrada incluye el código para detección de marcadores

```
corners, ids, rejectedImgPoints = cv2.aruco.detectMarkers( gray_frame,
                                                          aruco_dictionary,
                                                          parameters=parameters)
```

4. Finalmente el siguiente código analiza cada uno de los marcadores identificados y devuelve el vector de rotación(`rvecs`) y de traslación(`tvecs`) para cada marcador. Esta función recibe:

- a) Esquinas detectadas de los marcadores
- b) Tamaño de un lado de cada marcador (en metros), esto corresponde con el tamaño real de nuestro marcador, se puede dejar en `1m`, ya que se utiliza para ayudar a establecer el tamaño de los ejes.
- c) Parámetros de la cámara

d) Coeficiente de distorsión de la cámara

```
# Obtención de Vectores de rotación y traslación
# de los marcadores detectados
rvecs, tvecs, _ = cv2.aruco.estimatePoseSingleMarkers( corners,
                                                       1,
                                                       cameraMatrix,
                                                       distCoeffs)
```

5. De igual manera se procede a dibujar los ejes sobre cada marcador encontrado utilizando la siguiente instrucción, el ultimo parámetro es el tamaño que deseamos definir para los ejes, esta relacionado con el tamaño real de los marcadores y se define en metros:

```
# Dibujar los ejes
cv2.aruco.drawAxis(frame, cameraMatrix, distCoeffs, rvec, tvec, 1)
```

6. Para ejecutar este programa escribimos la siguiente orden en la consola

```
python 04_aruco_detect_markers_pose.py
```

7. Luego de su ejecución los ejes dibujados se verán de la siguiente manera en nuestra imagen (figura 3):



Figura 3: Ejes dibujados en los marcadores detectados.

1.5. Realidad Aumentada con Marcadores

En esta parte seguiremos utilizando los parámetros de calibración de la cámara para superponer un cuadrado a un marcador identificado en la imagen de entrada. Se utilizará el archivo `05_aruco_detect_markers_square.py`.

En este caso el código se basa en el anterior, pero con la diferencia que no dibujaremos un conjunto de ejes sobre el marcador, sino un cuadrado en el marcador. Este cuadrado deberá coincidir con las dimensiones del marcador detectado.

1. Luego de detectar los marcadores y sus esquinas, se procede a definir los puntos para los cuales deseó obtener la coordenadas para dibujarlos sobre el marcador, lo cual se hace con el siguiente código:

```
# Puntos deseados para superponer en el marcador
desired_points = np.float32([
    [-1 / 2, 1 / 2, 0],
    [1 / 2, 1 / 2, 0],
    [1 / 2, -1 / 2, 0],
    [-1 / 2, -1 / 2, 0]
]) * OVERLAY_SIZE_PER
```

2. El código anterior define las 4 esquinas del cuadrado que queremos dibujar, debemos tener en cuenta que estas no son las coordenadas, sino que representan la ubicación de los puntos con respecto al centro del marcador detectado, el cual es el origen del sistema de coordenadas del marcador.
3. El elemento `OVERLAY_SIZE_PER` permite establecer una escala entre el cuadrado a superponer y el tamaño del marcador, este se define al principio del código fuente. El valor predefinido de 1 significa que el cuadrado tendrá la misma dimensión que el marcador, un valor mayor indica que el cuadrado será de mayor tamaño que el marcador.
4. Una vez definidos los puntos deseados, procedemos a calcular estos valores en relación a las coordenadas de la imagen de entrada, lo cual se realiza mediante la siguiente función la cual recibe los puntos deseados, los vectores de rotación y traslación y los parámetros de calibración de la cámara para un correcto procedimiento.

```
# Projectar los puntos
projected_desired_points, jac = cv2.projectPoints(desired_points,
                                                    rvecs,
                                                    tvecs,
                                                    cameraMatrix,
                                                    distCoeffs)
```

5. Como último paso se envía la imagen y los puntos proyectados al método para dibujarlos en la imagen de salida, esta función fue creada al inicio del código para cumplir esta objetivo.

```
# Enviar los puntos para que se dibujen
draw_points(frame, projected_desired_points)
```

6. Contenido del método draw_points()

```
def draw_points(img, pts):
    """ Dibujar los puntos en la imagen"""

    pts = np.int32(pts).reshape(-1, 2)

    img = cv2.drawContours(img, [pts], -1, (255, 255, 0), -3)

    for p in pts:
        cv2.circle(img, (p[0], p[1]), 5, (255, 0, 255), -1)

    return img
```

7. Para ejecutar se debe escribir el siguiente comando en la consola

```
python 05_aruco_detect_markers_square.py
```

8. Luego de su ejecución nuestra imagen se verá de la siguiente manera (figura 4):

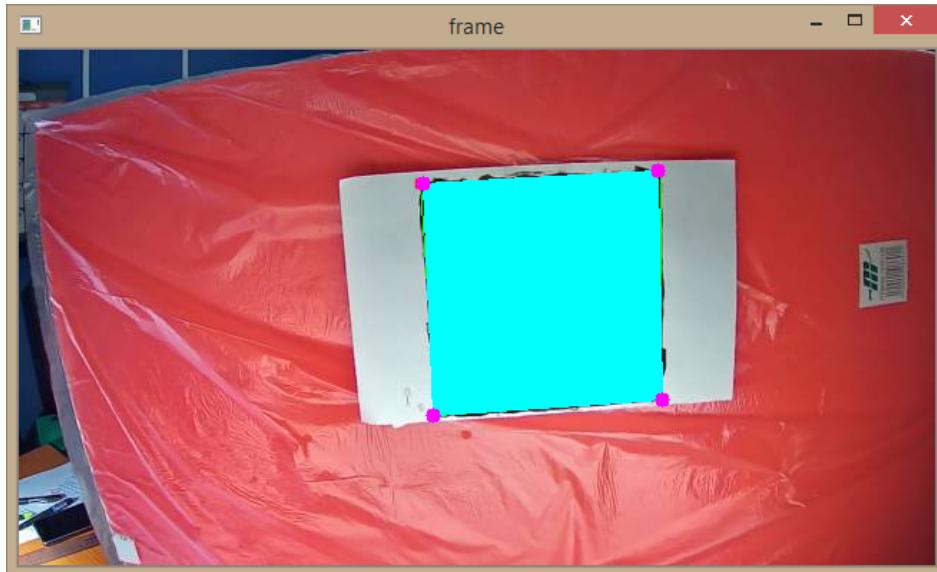


Figura 4: Cuadro superpuesto al marcador.

1.6. Realidad Aumentada con Marcadores 2

En esta sección se utilizará una versión modificada del código anterior para detectar los marcadores y en esta ocasión superponer una imagen al marcador detectado en la imagen de entrada. Se utilizará el archivo `06_aruco_detect_markers_augmented_reality.py`.

1. La imagen a superponer se carga al inicio del programa con el siguiente código y se puede ver en la figura 5.

```
# Cargar Imagen  
overlay = cv2.imread("tree_overlay.png")
```



Figura 5: Imagen para superponer en el marcador.

2. Para superponer una imagen a nuestra imagen de entrada, se ha creado el siguiente método `draw_augmented_overlay`, el cual recibe los puntos deseados, la imagen a superponer y la imagen de entrada.

```

def draw_augmented_overlay(pts_1, overlay_image, image):
    """Superponer la imagen 'overlay_image' en la imagen 'image'"""

    # Defino la cuadricula de la imagen a superponer :
    pts_2 = np.float32([[0, 0],
                        [overlay_image.shape[1], 0],
                        [overlay_image.shape[1], overlay_image.shape[0]],
                        [0, overlay_image.shape[0]]]
                       )

    # Dibujar un borde para apreciar los límites de la imagen:
    cv2.rectangle(overlay_image,
                  (0, 0),
                  (overlay_image.shape[1], overlay_image.shape[0]),
                  (255, 255, 0), 10)

    # Calcular la matriz de transformación entre los puntos:
    M = cv2.getPerspectiveTransform(pts_2, pts_1)

    # Transformar la imagen usando la matriz de transformación M:
    dst_image = cv2.warpPerspective(overlay_image, M, (image.shape[1], image.shape[0]))

    # Crear la máscara:
    dst_image_gray = cv2.cvtColor(dst_image, cv2.COLOR_BGR2GRAY)
    ret, mask = cv2.threshold(dst_image_gray, 0, 255, cv2.THRESH_BINARY_INV)

    # Calcular una operación lógica bitwise_and usando la máscara
    image_masked = cv2.bitwise_and(image, image, mask=mask)

    # Sumar las 2 imágenes para crear la imagen resultado:
    result = cv2.add(dst_image, image_masked)
    return result

```

3. Cada sección de código se explica a continuación:

4. En este fragmento se determina la transformación necesaria para que cada esquina la imagen `overlay_image` en `pts_2` coincida con los puntos del marcador identificado y almacenado en `pts_1`

```

# Calcular la matriz de transformación entre los puntos:
M = cv2.getPerspectiveTransform(pts_2, pts_1)

```

5. En este caso se aplica la matriz de transformación a la imagen `overlay_image`, se redefine su tamaño a un tamaño igual al de la imagen de entrada. Esta imagen tendrá todos los píxeles en negro(0), salvo donde se ubica la imagen `overlay_image` luego de aplicar la transformación. Ver Figura 7 imagen izquierda superior.

```

# Transformar la imagen usando la matriz de transformación M:

```

```
dst_image = cv2.warpPerspective(overlay_image, M, (image.shape[1], image.shape[0]))
```

6. Se procede con la creación de la máscara. Como primer paso se procede a crear una copia de la imagen `overlay_image` transformada pero en escala de grises. Utilizando la función de *Threshold*² de OpenCV y usando el método binario inverso³, se cambia a un valor de 0 solamente los píxeles con un valor mayor a 0. Los píxeles que no cumplen (son 0 o negro) esto se colocan en 255 (Blanco) y en este caso se corresponden con la ubicación de la `overlay_image`, ya que este es el espacio que necesitamos que se indique en nuestra máscara. Ver Figura 6.

```
# Crear la máscara:
dst_image_gray = cv2.cvtColor(dst_image, cv2.COLOR_BGR2GRAY)
ret, mask = cv2.threshold(dst_image_gray, 0, 255, cv2.THRESH_BINARY_INV)
```

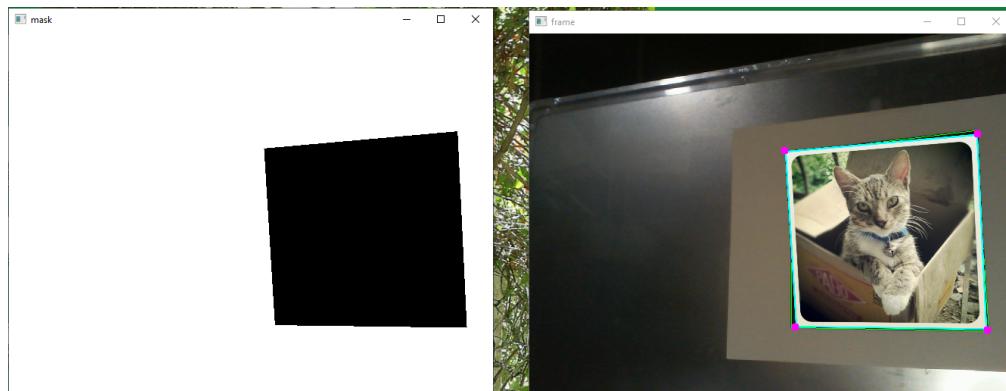


Figura 6: Resultados de la aplicación del *thresholding*.

7. En este caso nuestra máscara tiene valores 0 o negro donde estará ubicada la `overlay_image` y 255 o blanco donde no había datos, cuando se aplicó la transformación de perspectiva a la `overlay_image` los píxeles agregados se colocaron en negro. Se aplica entonces una operación lógica `bitwise_and`⁴⁵ en conjunto con la máscara. Esta operación devolverá la imagen de entrada original, pero con los píxeles en negro donde se ubicará `overlay_image`. Esta operación funciona como un *and* lógico y solo retorna Blanco, cuando la máscara y la imagen tienen un píxeles blancos en la misma posición. Cuando la máscara tiene un valor blanco entonces este no afecta los valores originales de la imagen y esta conserva su valor. Ver Figura 7 imagen derecha superior.

²Threshold en OpenCV

³THRESH_BINARY_INV

⁴Operación Bitwise en OpenCV

⁵Operaciones Lógicas OpenCV

```
# Calcular una operación lógica bitwise_and usando la máscara
image_masked = cv2.bitwise_and(image, image, mask=mask)
```

8. En esta operación se suman⁶ los píxeles de 2 imágenes. Los píxeles en negro (0) en una imagen, no modifican el valor del píxel en esa posición en la otra imagen. En este caso dst_image tiene valores en 0 donde no está overlay_image mientras que image_masked tiene valores en 0 donde estará ubicada la overlay_image por lo cual el resultado es la unión de ambas imágenes. Ver Figura 7 imagen inferior.

```
# Sumar las 2 imágenes para crear la imagen resultado:
result = cv2.add(dst_image, image_masked)
```

9. Las salidas parciales de este código se pueden apreciar en la figura 7.

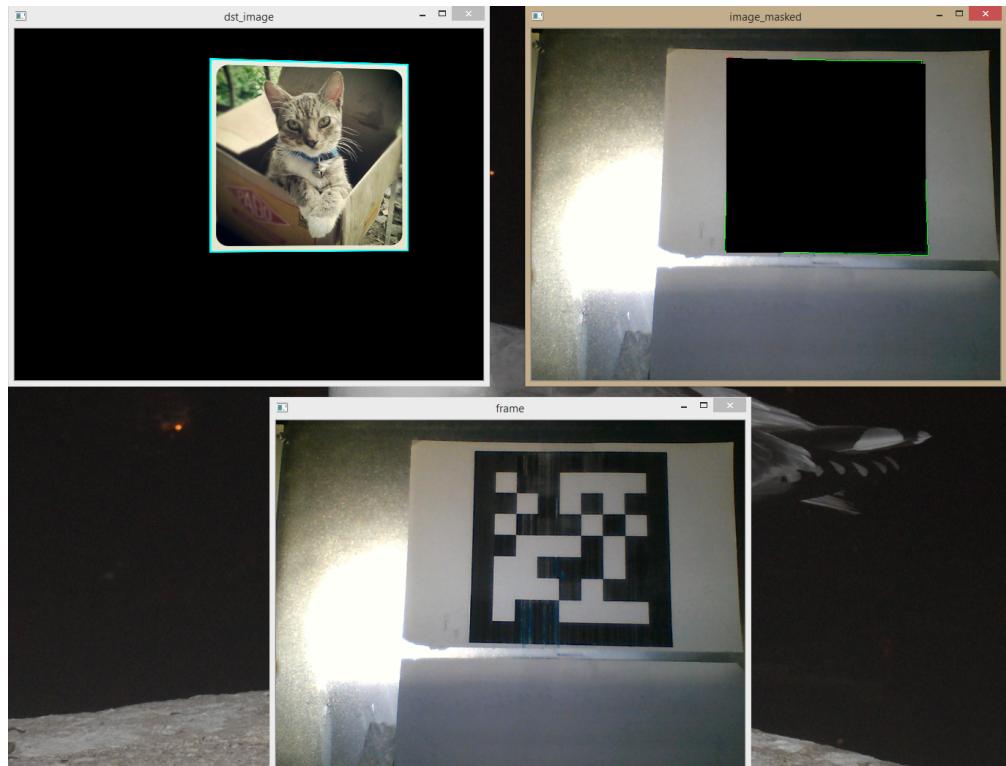


Figura 7: Diferentes resultados que se obtienen al aplicar los diversos pasos de la función.

10. Para ejecutar se debe escribir el siguiente comando en la consola

⁶Suma OpenCV

```
python 06_aruco_detect_markers_augmented_reality.py
```

11. Luego de su ejecución nuestra imagen se verá de la siguiente manera (figura 8):

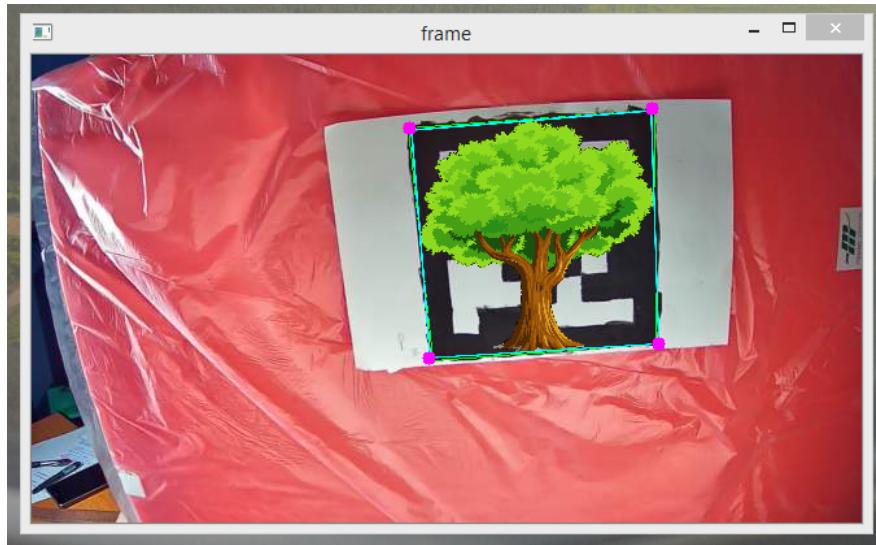


Figura 8: Imagen superpuesta al marcador.

2. Detección de Rostros y Realidad Aumentada

Dentro de los procedimientos de reconocimiento uno de los más comunes en la actualidad es el reconocimiento facial, el cual es aplicado en temas de seguridad, acceso, control biométrico, superposición de rostros, realidad aumentada, aplicación de filtros, entre otros.

En esta ocasión se creará utilizará una aplicación que en tiempo real detectará el rostro de una persona y colocará una imagen en una posición que se calculará según los elementos detectados en la cara del sujeto, similar a la aplicación de filtros de aplicaciones de mensajería y redes sociales como Snapchat, Facebook o Instagram. Este mismo principio es utilizado en aplicaciones como Deepface y aplicaciones de envejecimiento de rostros.

Para entrenar estos clasificadores se hace necesario contar con cierta cantidad de muestras positivas (objetos, caras, etc) y negativas. A estas se les extraen sus características y posteriormente se realiza un proceso de entrenamiento. Los clasificadores se basan en reconocer patrones presentes en las características o *features* detectadas en las imágenes. En este caso los detectores de características usados serán los filtros Haar.

Para los siguientes ejemplos utilizaremos los conceptos de Cascada de Clasificadores y los filtros de Haar. El método de cascada de clasificadores propuesto en el artículo [?] consiste en la aplicación sucesiva de varios clasificadores a una imagen. En este caso la imagen anteriormente ha sido analizada a través de sub-regiones de menor tamaño, cada una de estas regiones es evaluada por un clasificador y si el clasificador determina que una de esas sub-regiones corresponde con una cara (o cualquier objeto buscado), se envía dicha sub-región al siguiente nivel o clasificador, junto con todas las demás sub-regiones identificadas positivamente. Al final el resultado o la categoría determinada por la cascada se basa en la premisa de que esa sub-región ha sido identificada positivamente por todos los clasificadores dentro de la cascada. Este proceso se puede ver en la figura 9. Para más información se puede consultar el siguiente enlace⁷.

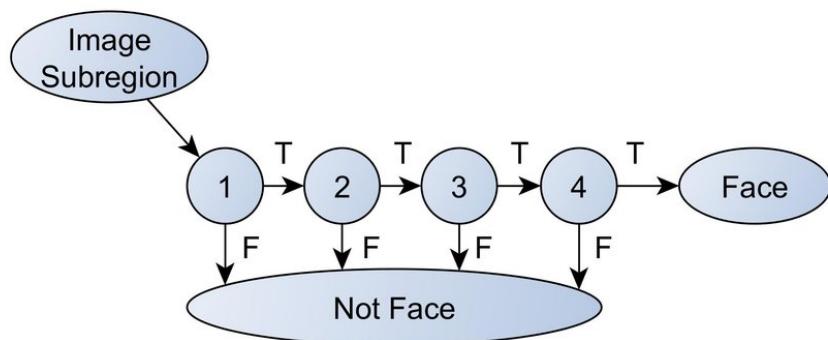


Figura 9: Esquema de una cascada de clasificadores.

⁷Cascada de Clasificadores

OpenCV posee entonces el clasificador HaarCascade⁸. Estos clasificadores permiten reconocer diferentes partes del cuerpo de humanos y animales y algunos objetos. Estos son modelos pre-entrenados y se encuentran dentro de la carpeta de instalación de OpenCV C:\opencv\sources\data\haarcascades. En este caso se han incluido dentro de los archivos del laboratorio ya que se utilizarán el detector de caras, nariz y ojos.

En este caso utilizaremos los códigos 01_snapchat_augmented_reality_glasses.py y 02_snapchat_augmented_reality_moustache.py. Los cuales agregan unas gafas y un bigote respectivamente a la imagen de entrada, siempre que se detecte un rostro en la misma.

A continuación se presenta la explicación de algunos fragmentos de códigos que se utilizan en estos códigos.

1. Iniciaremos con el código 01_snapchat_augmented_reality_glasses.py. Utilizando la cámara web para añadir unas gafas a la imagen de entrada.
2. El fragmento de código siguiente carga los modelos de clasificación de haarcascade para la detección de rostros y los ojos.

```
# Cargar clasificadores para la detección de rostro y ojos
face_cascade = cv2.CascadeClassifier("haarcascade_frontalface_default.xml")
eyepair_cascade = cv2.CascadeClassifier("haarcascade_mcs_eyepair_big.xml")
```

3. Como siguiente paso, las gafas se corresponden con una imagen que debe ser cargada en el código fuente. En este código utiliza el -1 como argumento al leer la imagen para indicar que se debe leer el canal Alfa(transparencia) si la imagen cuenta con uno.

```
# Carga la imagen de las gafas.
# El argumento -1 lee el canal alfa si este existe.
img_glasses = cv2.imread('glasses.png', -1)
```

4. En esta ocasión nuestra aplicación permite utilizar la cámara web o enviar constantemente una imagen que emula la entrada de la cámara. La cual se carga con el siguiente código.

```
# Definir una imagen para usar en pruebas y ajustar las ROIs:
test_face = cv2.imread("lena2.jpg")
```

5. En caso que deseemos utilizar esta imagen y no la de la cámara debemos comentar las líneas donde se inicia la captura de vídeo y descomentar la línea donde se asigna al frame nuestra imagen de prueba, tal como se muestra a continuación.

```
# Captura de vídeo desde la cámara
#ret, frame = video_capture.read()

# Utilizar la imagen de prueba como entrada
frame = test_face.copy()
```

⁸HaarCascade

6. Una vez terminada nuestra configuración inicial, procedemos a aplicar el detector de rostros en nuestra imagen de entrada, más precisamente en una copia en escala de grises de la misma. Esta función devuelve un conjunto de coordenadas (x, y) , el alto y ancho para cada una de las caras detectadas.

```
# Detectar las caras utilizando 'detectMultiScale()'  
faces = face_cascade.detectMultiScale(gray, 1.3, 5)
```

7. A continuación debemos detectar la ubicación de los ojos en cada una de las caras, lo cual se realiza con el detector Haar para los ojos. Para ello el primer paso es extraer las regiones donde se han detectado los rostros, tomando en cuenta su ubicación y dimensiones.

```
# Crear las ROISs basado en el tamaño  
roi_gray = gray[y:y + h, x:x + w]  
roi_color = frame[y:y + h, x:x + w]
```

8. Seguidamente, se utiliza el detector para identificar en cada ROI la ubicación de los ojos de en la imagen

```
# Detectar los ojos dentro de los rostros detectados  
eyepairs = eyepair_cascade.detectMultiScale(roi_gray)
```

9. En el siguiente paso procedemos a calcular la región donde se ubicarán nuestras gafas, esto cálculos toman en cuenta las dimensiones de la imagen de la gafas a utilizar. Para comprenderlos debemos recordar que la imagen de las gafas es un poco más grande que los ojos que detectemos, por lo tanto, con estos cálculos estamos estimando la ubicación de dicha imagen sobre nuestra imagen de entrada.

```
# Calcular las coordenadas donde se ubicaran las gafas.  
x1 = int(ex - ew / 10)  
x2 = int((ex + ew) + ew / 10)  
y1 = int(ey)  
y2 = int(ey + eh + eh / 2)
```

10. Una vez determinadas las coordenadas donde se ubicarán las gafas, se procede a realizar las operaciones necesarias para cambiar el tamaño de la imagen de las gafas, al tamaño de la zona en la cual se superpondrá la imagen de las gafas. De igual manera se crea la máscara de esta mediante la operación `bitwise_not()`⁹ la cual colocará en negro (0) los píxeles que tengan un valor blanco (255) y viceversa. En este punto la imagen a partir de la cual se creará la máscara, fue creada solo tomando el valor del canal alfa de la imagen original, por ende, tiene una mayoría de valores negros en el fondo y píxeles con valor mayor que 0 en donde se encuentran las gafas.

⁹`bitwise_not()`

11. Como resultado nuestra máscara solo tendrá valores bajos (negros y grises) en los píxeles donde están las gafas.

```
# Calcular las dimensiones de la región
img_glasses_res_width = int(x2 - x1)
img_glasses_res_height = int(y2 - y1)

# Cambiar el tamaño de la máscara al de la región
mask = cv2.resize(img_glasses_mask, (img_glasses_res_width, img_glasses_res_height))

# Crear la máscara de las gafas
mask_inv = cv2.bitwise_not(mask)
```

12. Cuando se han creado las máscaras, se procede a editar los ROIs para las cuales fueron creadas, primero se crea una ROI que tendrá los valores de color originales de la imagen de entrada. Posteriormente usando una operación `bitwise_and()` se modifican los valores de estas regiones.
13. Para el fondo se utiliza la región con colores originales y la máscara inversa (mayormente con píxeles blancos), como resultado de la operación `bitwise_and()` saldrá la imagen con colores originales, pero con un valor negro (0) en los píxeles donde la máscara tenía un valor de 0. Ver figura 10 columna izquierda.
14. Para las gafas en primer plano se utilizará una copia de la imagen original de las gafas redimensionada, aplicando un `bitwise_and()` con la máscara creada a partir de la imagen original, la cual tiene píxeles blancos (255) donde están ubicadas las gafas. Debido al funcionamiento de esta operación el resultado será una imagen de las gafas con valores píxeles negros donde la máscara tenía un valor negro. Dejando solo con un color diferente los píxeles donde se ubican las gafas, siendo en este caso el valor original de las mismas. Ver figura 10 columna derecha.

```
# Crear una ROI desde la image de color
roi = roi_color[y1:y2, x1:x2]

# Crear los ROIs de fondo y primer plano:
roi_bakground = cv2.bitwise_and(roi, roi, mask=mask_inv)
roi_foreground = cv2.bitwise_and(img, img, mask=mask)
```

15. Como ultimo paso se realiza una suma de las imágenes generadas a partir de las máscaras, en esta operación los píxeles negros tienen un valor de 0, por lo cual no modifican el píxel en esa coordenada de la otra imagen. Por lo cual, el resultado es una imagen donde los píxeles en negro ahora tienen un valor, terminando así el proceso y permitiendo su visualización. La suma se da entre las imágenes inferiores (izquierda y derecha) de la figura 10.

```
# Sumar las imágenes para producir el resultado
res = cv2.add(roi_bakground, roi_foreground)
```

```
# Colocar el resultado en la ROI original  
roi_color[y1:y2, x1:x2] = res
```

16. Las máscaras y los resultados de su aplicación se pueden apreciar en la figura 10:

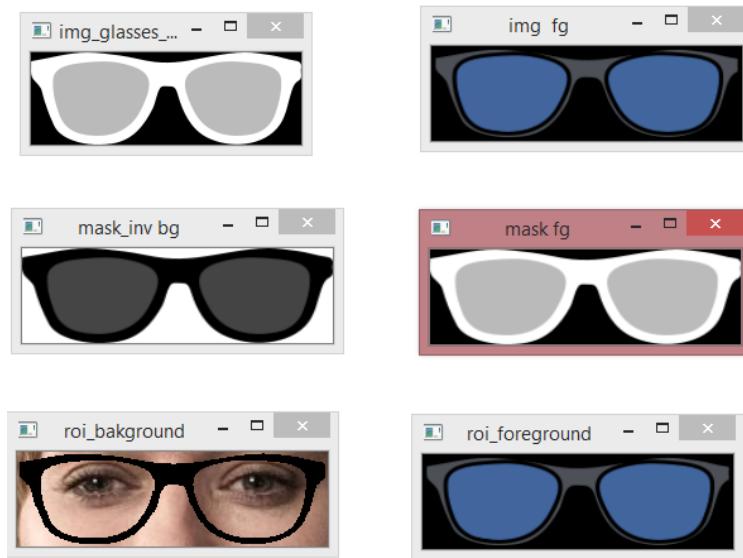


Figura 10: Máscaras y resultados de aplicación.

17. Para ejecutar se debe escribir el siguiente comando en la consola

```
python 01_snapchat_augmented_reality_glasses.py
```

18. Luego de su ejecución nuestra imagen se verá de la siguiente manera (figura 11):

19. En el caso del segundo ejemplo de esta sección consiste en un código similar pero con ligeras adaptaciones para que en lugar de agregar gafas, agregue un bigote a la imagen de entrada.

20. Para ejecutar se debe escribir el siguiente comando en la consola

```
python 02_snapchat_augmented_reality_moustache.py
```

21. Luego de su ejecución nuestra imagen se verá de la siguiente manera (figura 12):



Figura 11: Gafas superpuestas a una imagen de entrada.

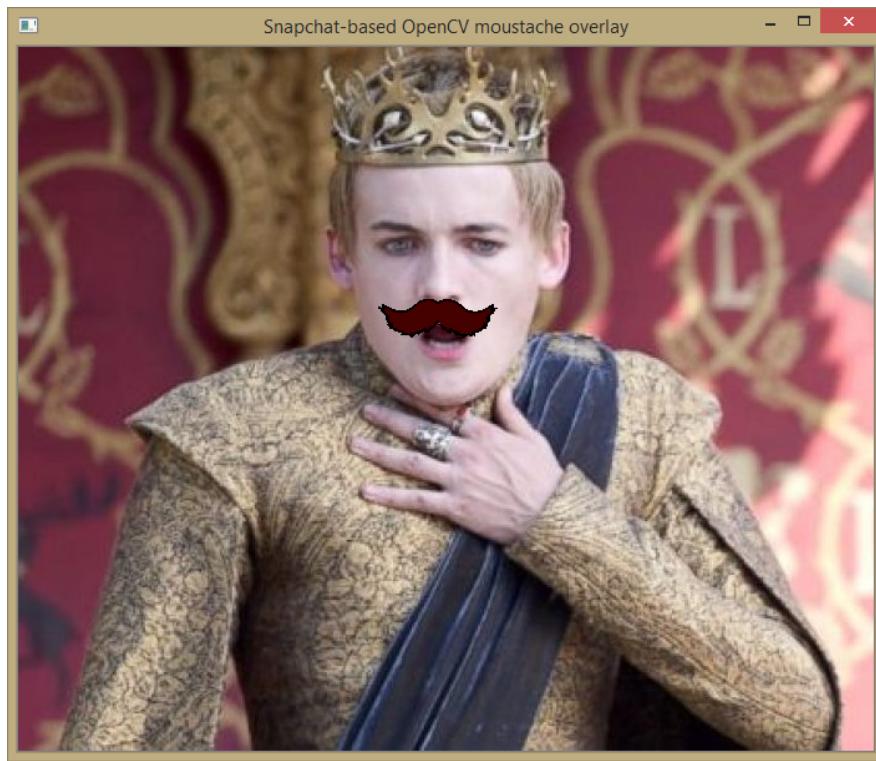


Figura 12: Bigote superpuesto a una imagen de entrada.

3. Reconocimiento de Rostros con OpenCV

Uno de los puntos fuertes de OpenCV es la capacidad de detección y las diferentes gamas de procedimientos que se pueden realizar con los rostros. Existen diversas librerías adicionales que permiten trabajar los diversos enfoques, sin embargo en esta ocasión nos centraremos en las opciones que trae OpenCV incluidas en su código. Esta sección del laboratorio ha sido extraída de [?] [?].

Los datos para esta sección se encuentran en un archivo comprimido disponible en Moodle.

En esta sección utilizaremos el enfoque de reconocimiento facial, en el cual, le enseñaremos a un algoritmo a diferenciar entre dos rostros, mediante un proceso de entrenamiento y prueba.

Este ejemplo aplica nociones básicas de aprendizaje automático. Se utiliza un conjunto (*dataset*) de imágenes para entrenar un reconocedor. La salida de este último consistirá en la etiqueta o categoría a la cual pertenece la imagen que se está evaluando.

El proceso de entrenamiento incluye una fase en la cual se deben procesar las imágenes y extraer las características de las mismas, para nuestro ejemplo dichas características se extraerán y describirán utilizando el enfoque *Local Binary Patterns Histograms* (LBPH).

OpenCV cuenta con 3 reconocedores faciales integrados. Estos se pueden utilizar de manera independiente en este código simplemente cambiando una línea de código. Estos son:

1. EigenFaces Face Recognizer Recognizer `cv2.face.EigenFaceRecognizer_create()`
2. FisherFaces Face Recognizer Recognizer `cv2.face.FisherFaceRecognizer_create()`
3. Local Binary Patterns Histograms (LBPH) Face Recognizer `cv2.face.LBPHFaceRecognizer_create()`

3.1. Local Binary Patterns Histograms (LBPH) Face Recognizer

Una explicación detallada de LBPH puede ser encontrada en [Face Detection](#).

Los reconocedores de Eigen y Fisher son afectados por la luz y esta es una condición que no se puede garantizar en situaciones de la vida real. El reconocedor usando LBPH es una mejora para superar esta desventaja. Su enfoque es utilizar descriptores locales en la imagen. LBPH trata de encontrar una estructura de la imagen y lo hace mediante la comparación de cada píxel con los de su vecindario.

Se toma una ventana de 3×3 y se mueve a través de la imagen, en cada movimiento se compara el píxel central con los vecinos. Los vecinos con una intensidad menor o igual al del píxel central se marcan utilizando un 1 y los demás con un 0. Estos valores dentro de la ventana se leen en el sentido de las agujas del reloj lo que creará un patrón binario como 11100011 el cual es específico para esta zona de la imagen. Haciendo esto a través de toda la imagen se tendrá una lista de patrones locales binarios.

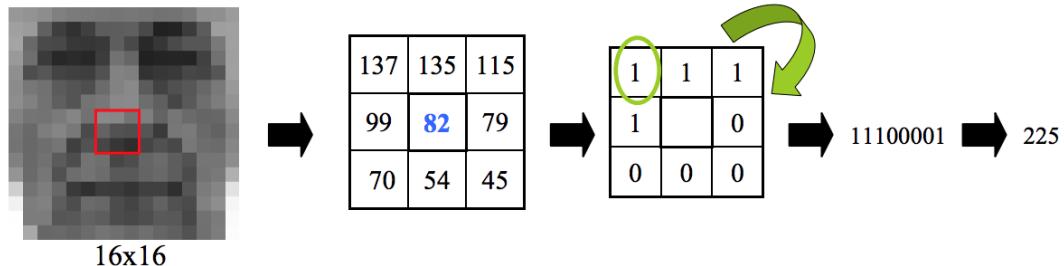


Figura 13: Construcción de Patrón Binario.

Con lo anterior se tiene la parte de los patrones binarios, para la creación del histograma, se convierte cada patrón en un número binario (binario → digital) y entonces se realiza un histograma de todos los valores decimales.

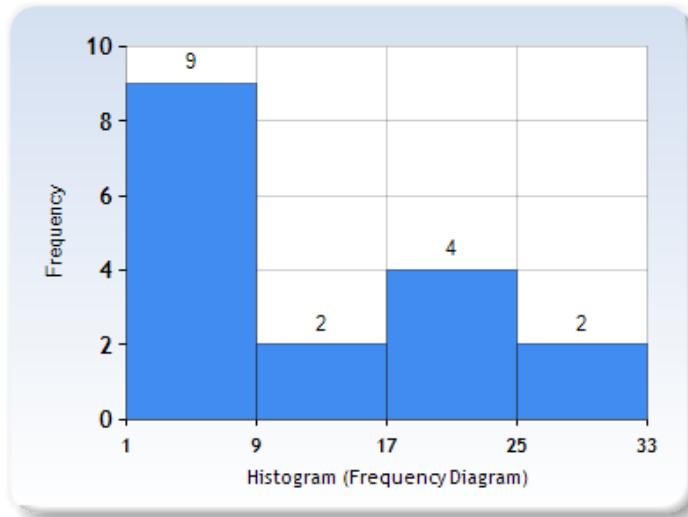


Figura 14: Histograma del Patrón.

Con este enfoque estaremos creando un histograma para cada cara en la imagen. Por lo cual, cuando tenemos un dataset de entrenamiento con 100 caras tendremos 100 histogramas diferentes que se almacenaran para realizar el proceso de reconocimiento posteriormente. El algoritmo sabe que cara pertenece cada histograma. Durante la etapa de reconocimiento se pasará una imagen al reconocedor, el cual calculará el histograma de la cara detectada en la imagen y lo comparará con los histogramas que tiene almacenados, para devolver la categoría que mejor coincide con la imagen en evaluación.

En esta imagen podemos ver como los descriptores LBPH no son afectados por los factores de iluminación. [Fuente](#).

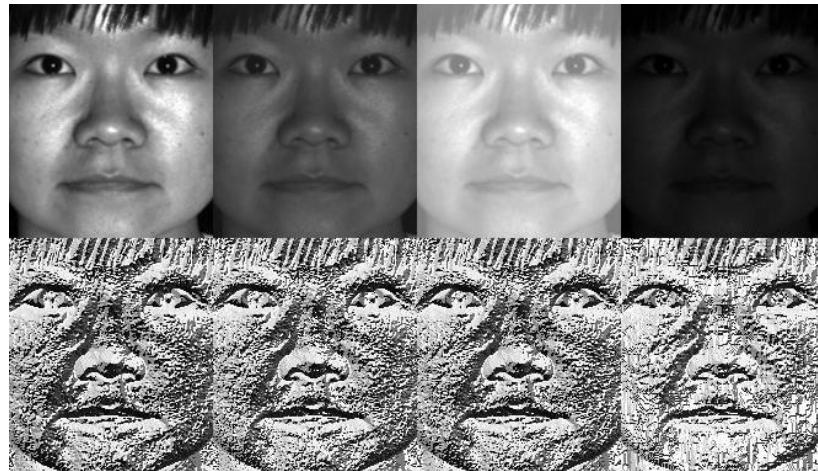


Figura 15: LBPH para diferentes iluminaciones.

3.1.1. Proceso de Reconocimiento Facial en OpenCV

El proceso de reconocimiento facial se puede dividir en 3 etapas:

1. **Preparar los datos de entrenamiento:** En este paso se leerán las imágenes de entrenamiento para cada persona con sus etiquetas, se detectaran las caras en cada imagen y se asignan a una etiqueta o label entero.
2. **Entrenar el reconocedor:** En este paso entrenaremos el reconocedor de caras de LBPH. enviándole/mostrándole la información que se ha preparado en el paso 1.
3. **Testing:** En esta etapa enviaremos algunas imágenes de prueba, para evaluar si la predicción se realiza de manera correcta.

3.2. Datos de entrenamiento

Entre mayor cantidad de imágenes por sujeto, los resultados serán mejores ya que el reconocedor será capaz de aprender datos de la misma persona desde diferentes puntos de vista. En este caso nuestro dataset tiene 12 imágenes de cada sujeto, los cuales se encuentran en el folder `training-data`, este contiene en su interior un folder para cada sujeto que deseamos reconocer cada folder tiene el formato `sLabel` (e.g. `s1, s2`)' donde el número es la etiqueta entera asignada a cada sujeto.

- training-data

- s1
 - 1.jpg
 - ...
 - 12.jpg
- s2
 - 1.jpg
 - ...
 - 12.jpg

El folder test-data contiene las imágenes que serán utilizadas para evaluar nuestro reconocedor luego de ser entrenado.

1. Se inicia con la importación de las librerías requeridas.

```
#import OpenCV module
import cv2
#import os module for reading training data directories and paths
import os
#import numpy to convert python lists to numpy arrays as
#it is needed by OpenCV face recognizers
import numpy as np
```

2. Las etiquetas en OpenCV deben ser de tipo entero, por lo cual se establece una forma de mapeado entre los números y los nombres de las personas. En nuestro caso no se utiliza el 0, por lo cual se deja vacío en la lista que contiene los nombres.

```
#there is no label 0 in our training data so subject name for index/label 0 is empty
subjects = ["", "Ruben Blades", "Elvis Presley"]
```

3. Como siguiente paso debemos preparar los datos de entrenamiento. Para entrenar el reconocedor OpenCV necesita dos arreglos, uno con los rostros (histograma de patrones) de los sujetos de en el conjunto de entrenamiento y el segundo vector contiene, en el mismo orden, las etiquetas de cada rostro

4. Por lo cual, si nuestro *dataset* contiene datos en esta forma:

PERSON-1	PERSON-2
img1	img1
img2	img2

5. Las listas producidas tendrán la siguiente estructura.

FACES	LABELS
person1_img1_face	1
person1_img2.face	1
person2_img1.face	2
person2_img2.face	2

6. Esta preparación se puede resumir como:

- a) Procesar la carpeta de entrenamiento, de donde se obtendrán la cantidad de personas que estarán en el reconocedor.
- b) Para cada sujeto se debe extraer la etiqueta que se le asignará y almacenarla en formato de entero.
- c) Leer las imágenes para cada personas y detectar la cara en cada una de estas.
- d) Añadir cada cara a la lista de caras y su etiqueta correspondiente a la lista de etiquetas.

```
#function to detect face using OpenCV
def detect_face(img):
    #convert the test image to gray image as opencv face detector expects gray images
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    #load OpenCV face detector, I am using LBP which is fast
    #there is also a more accurate but slow Haar classifier
    face_cascade = cv2.CascadeClassifier('opencv-files/lbpcascade_frontalface.xml')

    #let's detect multiscale (some images may be closer to camera than others) images
    #result is a list of faces
    faces = face_cascade.detectMultiScale(gray, scaleFactor=1.2, minNeighbors=5);

    #if no faces are detected then return original img
    if (len(faces) == 0):
        return None, None

    #under the assumption that there will be only one face,
    #extract the face area
    (x, y, w, h) = faces[0]

    #return only the face part of the image
    return gray[y:y+w, x:x+h], faces[0]
```

7. El detector LBP necesita trabajar con imágenes en escala de grises, de igual manera se debe realizar como primer paso recorte de la ubicación del rostro dentro de la imagen. En este

fragmento de código se realiza la conversión de la imagen a escala de grises y se utilizan el cv2.CascadeClassifier mediante un modelo de detección frontal de rostros, para realizar el recorte de la cara en la imagen. Este método devuelve el (x, y, width, height) de la zona en la cual se encuentra el rostro, para que pueda ser extraída utilizando OpenCV

```
#this function will read all persons' training images, detect face from each image
#and will return two lists of exactly same size, one list
# of faces and another list of labels for each face
def prepare_training_data(data_folder_path):

    #-----STEP-1-----
    #get the directories (one directory for each subject) in data folder
    dirs = os.listdir(data_folder_path)

    #list to hold all subject faces
    faces = []
    #list to hold labels for all subjects
    labels = []

    #let's go through each directory and read images within it
    for dir_name in dirs:

        #our subject directories start with letter 's' so
        #ignore any non-relevant directories if any
        if not dir_name.startswith("s"):
            continue;

        #-----STEP-2-----
        #extract label number of subject from dir_name
        #format of dir name = slabel
        #, so removing letter 's' from dir_name will give us label
        label = int(dir_name.replace("s", ""))

        #build path of directory containin images for current subject subject
        #sample subject_dir_path = "training-data/s1"
        subject_dir_path = data_folder_path + "/" + dir_name

        #get the images names that are inside the given subject directory
        subject_images_names = os.listdir(subject_dir_path)

        #-----STEP-3-----
        #go through each image name, read image,
        #detect face and add face to list of faces
        for image_name in subject_images_names:

            #ignore system files like .DS_Store
            if image_name.startswith("."):
                continue;

            #build image path
```

```

#sample image path = training-data/s1/1.pgm
image_path = subject_dir_path + "/" + image_name

#read image
image = cv2.imread(image_path)

#display an image window to show the image
cv2.imshow("Training on image...", image)
cv2.waitKey(100)

#detect face
face, rect = detect_face(image)

#-----STEP-4-----
#for the purpose of this tutorial
#we will ignore faces that are not detected
if face is not None:
    #add face to list of faces
    faces.append(face)
    #add label for this face
    labels.append(label)

cv2.destroyAllWindows()
cv2.waitKey(1)
cv2.destroyAllWindows()

return faces, labels

```

8. Esta función cumple la misión de preparar los datos de entrenamiento, recibiendo la ruta de la carpeta de entrenamiento y devolviendo las listas de caras y etiquetas de cada cara.

```

#let's first prepare our training data
#data will be in two lists of same size
#one list will contain all the faces
#and other list will contain respective labels for each face
print("Preparing data...")
faces, labels = prepare_training_data("training-data")
print("Data prepared")

#print total faces and labels
print("Total faces: ", len(faces))
print("Total labels: ", len(labels))

```

3.2.1. Entrenamiento del Reconocedor

9. En este paso procedemos a instanciar el reconocedor y posteriormente se realiza el entrenamiento del mismo utilizando el método train(faces-vector, labels-vector) el cual recibe la

lista de caras y etiquetas del conjunto de entrenamiento.

```
#train our face recognizer of our training faces
face_recognizer.train(faces, np.array(labels))
```

3.2.2. Predicción

10.

```
#function to draw rectangle on image
#acording to given (x, y) coordinates and
#given width and height
def draw_rectangle(img, rect):
    (x, y, w, h) = rect
    cv2.rectangle(img, (x, y), (x+w, y+h), (0, 255, 0), 2)

#function to draw text on give image starting from
#passed (x, y) coordinates.
def draw_text(img, text, x, y):
    cv2.putText(img, text, (x, y), cv2.FONT_HERSHEY_PLAIN, 1.5, (0, 255, 0), 2)
```

11. Estas funciones dibujarán un rectángulo sobre el rostro detectado y escribirán la etiqueta que ha definido el detector que corresponde con el rostro.

```
#this function recognizes the person in image passed
#and draws a rectangle around detected face with name of the
#subject
def predict(test_img):
    #make a copy of the image as we don't want to change original image
    img = test_img.copy()
    #detect face from the image
    face, rect = detect_face(img)

    #predict the image using our face recognizer
    label= face_recognizer.predict(face)
    #print(label[0])
    #get name of respective label returned by face recognizer
    label_text = subjects[label[0]]

    #draw a rectangle around face detected
    draw_rectangle(img, rect)
    #draw name of predicted person
    draw_text(img, label_text, rect[0], rect[1]-5)

return img
```

12. En este caso utilizamos el reconocedor entrenado para definir una etiqueta para un rostro en una imagen de prueba. Para esto se utiliza el método predict(face), este retorna una tupla que contendrá el label(entero) al cual el reconocedor asignó la imagen y también un valor de confianza/probabilidad de dicho resultado.

```
print("Predicting images...")

#load test images
test_img1 = cv2.imread("test-data/test0.jpg")
test_img2 = cv2.imread("test-data/test6.jpg")

#perform a prediction
predicted_img1 = predict(test_img1)
predicted_img2 = predict(test_img2)
print("Prediction complete")

#display both images
cv2.imshow(subjects[1], predicted_img1)
cv2.imshow(subjects[2], predicted_img2)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

4. Tracking Facial

El *tracking* o seguimiento de objetos en visión artificial se enfoca en tratar de estimar la trayectoria de un objetivo a través de una secuencia de vídeo, donde solo se conoce la ubicación inicial de este objetivo. Esta tarea es realmente desafiante tomando en cuenta factores como las variaciones de apariencia, occlusiones, movimientos rápidos, desenfoque por movimiento y variaciones de escalas.

En este sentido Los seguidores basados en el filtro de correlación discriminativa (**DCF**, por sus siglas en inglés) proveen el estado del arte en la ejecución de estos métodos. Adicionalmente estos seguidores son computacionalmente eficientes, lo qué es crítico en aplicaciones que se ejecutan en tiempo real. De hecho, el estado del arte en la ejecución de estos seguidores se puede ver en los resultados del reto de seguimiento de objetos visuales VOT2014 en este, los este tipo de seguidores se encontraban en las propuestas de los tres primeros lugares del evento.

En este caso estaremos utilizando la librería DLIB la cual implementa un seguidor basado filtros DCF, el cual es fácil de utilizar tanto para objetos como para personas. Se utilizará en este caso para seguir una cara, pero puede ser implementado de manera sencilla para seguir objeto arbitrario que sean seleccionado por el usuario.

En el caso de la librería DLIB, esta es un complemento a OpenCV desarrollada principalmente en C++. En nuestro caso utilizaremos la versión para Python de esta librería, pero anteriormente a su instalación se requiere instalar el programa CMake que permite la compilación de esta al momento de su instalación. La descarga de CMake se puede realizar en [CMake](#).

- Como primer paso debemos instalar dlib utilizando el siguiente comando en una terminal. Esto puede tardar un poco.

```
pip install dlib
```

- Una vez finalizada la instalación procedemos a crear el código de nuestro archivo para *tracking*. Siendo el primer paso importar las librerías requeridas. En este caso se debe contar con una cámara conectada al computador.

```
# Import required packages:  
import cv2  
import dlib
```

- Posteriormente se crea una función para dibujar las instrucciones de texto en la pantalla del visualizador.

```
def draw_text_info():  
    """Dibujar información textual"""  
  
    # Coordenadas para dibujar el texto informativo:  
    menu_pos_1 = (10, 20)  
    menu_pos_2 = (10, 40)
```

```
# Escribir texto:
cv2.putText(frame, "Use '1' para reiniciar tracking",
            menu_pos_1,
            cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255))

if tracking_face:
    cv2.putText(frame,
                "tracking =",
                menu_pos_2,
                cv2.FONT_HERSHEY_SIMPLEX,
                0.5, (0, 255, 0))
else:
    cv2.putText(frame,
                "detectando una cara para inicializar tracking...",
                menu_pos_2, cv2.FONT_HERSHEY_SIMPLEX, 0.5,
                (0, 0, 255))
```

- Como siguiente paso se crea el objeto de vídeo que enlaza con la cámara web, de igual manera se crea una instancia del detector de rostros de DLIB y se crea la instancia del *tracker* basado en DCF

```
# Crear enlace para captura desde la cámara web:
capture = cv2.VideoCapture(0)

# Crear detector frontal de caras de dlib:
detector = dlib.get_frontal_face_detector()

# Inicializar el seguidor de correlacion.
tracker = dlib.correlation_tracker()

# Esta variable se encarga de indicar si actualmente se esta siguiendo una cara:
tracking_face = False
```

- La ultima parte del código se encarga de obtener el *frame* de la cámara, aplicar el detector facial. En este caso el algoritmo de *tracking* espera si el detector ha identificado una cara en la imagen. De ser cierto esto, recibe el *frame* y la zona donde se encuentra el rostro, lo cual inicia el trabajo de esta instancia de objeto, siguiendo lo que se encuentre dentro de la zona detectada. Como siguiente punto cuando se esta dando seguimiento a un rostro, se utiliza el método *tracker.update(frame)* para actualizar la ubicación del rostro, se obtiene la posición del mismo y se utiliza esta para marcar dicha ubicación en el imagen, a tiempo real.

```
while True:
    # Captura un frame desde la camara:
    ret, frame = capture.read()

    # Dibujar instrucciones
    draw_text_info()
```

```

if tracking_face is False:
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    # Trata de detectar una cara para inicializar el seguidor:
    rects = detector(gray, 0)
    # Verifica si se puede iniciar el seguimiento:
    if len(rects) > 0:
        # Iniciar tracking:
        tracker.start_track(frame, rects[0])
        tracking_face = True

if tracking_face is True:
    # Actualiza el seguimiento e imprime el ratio peak to sidelobe
    # (mide qué tan seguro está el rastreador) :
    print(tracker.update(frame))
    # Se obtiene la posición del objeto:
    pos = tracker.get_position()
    # Se dibuja posición:
    cv2.rectangle(frame, (int(pos.left()), int(pos.top())),
                  (int(pos.right()), int(pos.bottom())),
                  (0, 255, 0), 3)

# Captura de los eventos de teclado
key = 0xFF & cv2.waitKey(1)

# '1' para reiniciar el seguimiento:
if key == ord("1"):
    tracking_face = False

# 'q' para salir:
if key == ord('q'):
    break

# Mostrar los resultados:
cv2.imshow("Face tracking usando el detector facial de dlib y
            filtros de correlación para el tracking", frame)

# Release everything:
capture.release()
cv2.destroyAllWindows()

```

6. En el código anterior se agregan funcionalidades para reiniciar el proceso de seguimiento con otro rostro, al presionar la tecla 1 y también para cerrar la aplicación al presionar la tecla *q*.