Procesamiento digital de imagen y vídeo en *Python* con *OpenCV*

José Luis Garrido Labrador

10 de enero de 2019

Resumen

El objetivo de este documento es facilitar el comienzo de proyectos de procesamiento de imagen y vídeo en el entorno de Python, basándose en las herramientas que aporta MatLab en su *Image Processing Toolbox*. En el presente informe se exploran la instalación del entorno Python+OpenCV, las herramientas de visualizado y procesado, las funciones básicas y complejas inspiradas en el manual para Matlab de César Represa [1]. Cada tema tiene un *Jupyter Notebook* asociado y se pueden encontrar en el repositorio: https://github.com/joselucross/Python-OpenCV-Guide

0. Instalación

Lo primero que se debe hacer es instalar Python, la recomendación personal es usar conda, en especial la distribución miniconda [2] para poder tener varios entornos Python según las necesidades.

Una vez instalado miniconda con el instalador correspondiente o usando la linea de comandos en entornos GNU/Linux podemos pasar a crear un entorno donde tendremos las librerías necesarias sin comprometer el resto de aplicaciones Python. Para ello usamos el siguiente comando[1].

```
$ conda create --name HAE python=3.7
```

Esto comenzará la instalación del entorno Python que vamos a usar, tras esto entramos en el mismo usando.

```
$ conda activate HAE
```

Dentro del entorno pasamos a instalar las cuatro librerías que necesitamos, *numpy* que permitirá almacenar las imágenes en matrices numéricas, *matplotlib* librería basada en MatLab para el dibujado de gráficos y *opencv-python* para tener las herramientas para el procesamiento de las imágenes. Podemos también instalar *jupyter* si queremos utilizar ese entorno de programación. El comando total será:

```
(HAE) $ pip install numpy matplotlib opencv-python jupyter
```

En el caso de utilizar *jupyter* para poder utilizar el entorno necesitaremos hacer uso del comando:

```
(HAE) $ python -m ipykernel install --user --name HAE --display-name
"Python (HAE)"
```

Tras instalar todas las herramientas básicas tenemos que instalar la librería de OpenCV [3]. En sistemas Linux es tan fácil como utilizar la línea de comando[2] y en el caso de utilizar Windows se puede descargar directamente el ejecutable desde su página principal. Una vez hecho todo esto podemos empezar

[1] En el caso de usar en sistemas Windows antes se debe ejecutar desde la consola el fichero activate.bat de la carpeta *Scripts* en el directorio de instalación

[2] pacman, apt, yum, rpm...

1. Introducción a *MatPlotLib* y *NumPy*

Partiendo de la base de que ya se conocen las estructuras de control básicas de Python pasamso a explicar el funcionamiento de *MatPlotLib*[4] y *NumPy*[5].

1.1. Operaciones de matrices

Las matrices son objetos que se crean a partir de listas o a partir de una cadena como en *MatLab*. Todos los elementos de la lista han de ser del mismo tipo.

```
A = np.array([[1,2,3],[4,5,6],[7,8,9]]) #Matriz del tipo array
B = np.matrix('1 2 3;4 5 6;7 8 9') #El tipo matrix ha de ser
    bidimensional
```

La suma y resta de matrices se hace de igual manera que dos variables cualquiera

```
C = A + B
D = A - B
```

Para una multiplicación de los elementos por el homólogo bastaría con el operador *, lo mismo con la división usando /.

```
E = A * B # E[0,0] = A[0,0]*B[0,0]
F = A / B # F[0,0] = A[0,0]/B[0,0]
```

Si queremos hacer la multiplicación de matrices debemos utilizar la función *dot*

```
G = np.dot(A,B)
```

1.2. Operaciones booleanas

Otro tipo de operaciones importantes son las operaciones a nivel de bit o booleanas. Estas son la negación (*not*), disyunción (*or*), disyunción exclusiva (*xor*) y conjunción (*and*). Las funciones en *NumPy* son:

```
np.logical_not(H)
np.logical_or(H,I)
np.logical_xor(H,I)
np.logical_and(H,I)
```

Además puede contar con un flag `where` como un array booleano que funciona como máscara aplicando la función cuando sea cierta en cada elemento.

1.3. Transformación de matrices

Si se quiere hacer más grande[3] una matriz, obtener una submatriz, trasponer o alguna operación parecida *NumPy* tiene varias funciones para ello.

```
#Trasponer
np.transpose(A)
#Una fila
A[2,:] # Fila 3
# Una columna
A[:,1] # Columna 2 en forma de fila
# Una columna
A[:,[1]] # Columna 2 en forma de columna
#Sumatriz de la columna 2 a la 3 y fila 1 a 3
A[0:3:1,1:3:1] # El primer numero es el comienzo, el segundo es el
    destino excedido en 1 y el tercer el incremento
#Negacion
-A
#Concatenar
np.concatenate((A,B),axis=1) #Axis es el eje, comenzando en 0 filas, 1
    columnas
#Concatenar en pila
np.stack((A,B),axis=2) #En el eje z
#Cambiar forma reordenando los valores
A.reshape(9,1)
```

1.4. Generación de datos

NumPy tiene una colección de funciones para generar matrices aleatorias, de ceros, lógicas etc.

```
dim=3
dim1 = 5
dim2 = 2
#Funciones de generacion
np.zeros(dim) #Genera una matriz de ceros cuadrada de dimension dimXdim
np.zeros((dim1,dim2)) #Genera una matriz de ceros de dimension dim1Xdim2
#Tambien existe ones para lo mismo
np.identity(dim) #Genera una matriz identidad de dimension dim
np.linspace(start,stop,num) #Genera un vector con los num entre start y
    stop
np.arange(dim) #Genera un vector con los valores ordenados de de 0 a
    dim-1 (similar a np.array(list(range(dim))))
```

[3] No recomendable hacer en un bucle, ralentiza mucho la ejecución

1.5. Modificación del tipo

Las matrices tienen un tipo y este es modificable a otros, sin embargo, no es semejante a MatLab. Las funciones semejantes a *im2double* están en *OpenCV* y no en numpy.

El tipo se puede definir en la creación (tanto por generación como con el constructor) añadiendo el flag `dtype=np.TYPE` siendo TYPE una varidad grande como *bool_, int8, int 16, uint8, float16, complex64...*[4]

1.6. Funciones de *MatPlotLib*

MatPlotLib es una librería basada en MatLab por lo que la mayoría de las funciones que permite tienen una nomenclatura similar a la de MatLab

```
#Dibujar una funcion
X = np.linspace(0,100,100)
Y = X*X #Cuadrado
#Dibujado
plt.plot(X,Y)
plt.show()
```

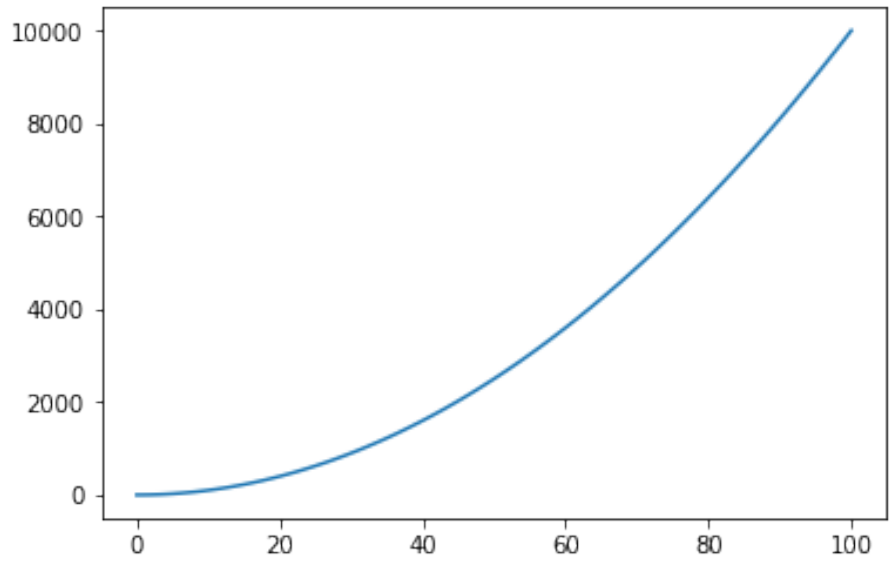

Figura 1: Función cuadrado

[4]Todos los tipos en `https://docs.scipy.org/doc/numpy-1.15.4/user/basics.types.html`

No nos centraremos en las posibilidades en funciones (cambios de colores, tamaño de ejes etc) sino en la figura y las funciones que se le aplican. A destacar la función `title` que introduce un título a la ventana.

Otro detalle importante es la función `subplot`

```
plt.subplot(1,3,1)
plt.plot(X)
plt.title("X")
plt.subplot(1,3,2)
plt.plot(Y/2)
plt.title("Y/2")
plt.subplot(1,3,3)
plt.plot(X,Y)
plt.title("Y = X*X")
plt.heigth = "200px"
plt.show()
```

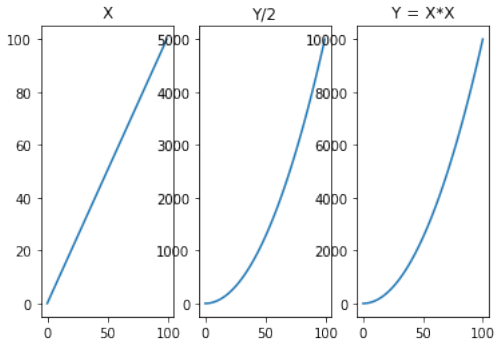

Figura 2: Ejemplo de subplot

En el caso de que queramos modificar el tamaño de la figure debemos crearla con anterioridad

```
#Figura
plt.figure(figsize=(10,10))
plt.plot(X,Y)
plt.title("Titulo")
plt.show()
```

Por último está la función `stem` que representa valores discretos. Recibe como parámetros X e Y y otros como el color o la representación de la linea.

```
X = np.linspace(-2*np.pi,0,10)
Y = np.cos(X)
plt.stem(X,Y,'g-.')
```

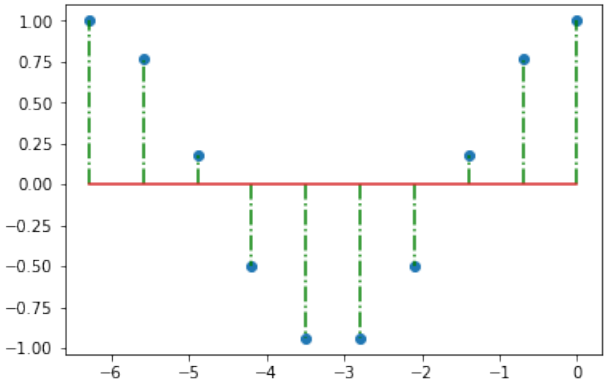

Figura 3: Ejemplo de stem

2. Introducción a *OpenCV*

Una vez entendido el funcionamiento de *NumPy* y *MatPlotLib* de manera general podemos pasar a gestionar las imágenes. Debemos importar la librería `cv2` junto con las de `numpy` y `matplotlib.pyplot`

2.1. Lectura de imágenes y espacios de color

Las imágenes se cargan como matrices *NumPy*, se leen con la función `imread` y tiene dos parámetros de entrada, el fichero a cargar y el modo de lectura, si es 0 es en blanco y negro, si es 1 es en color.

```
img = cv2.imread('imagen.png',0) #Blanco y negro
img = cv2.imread('imagen.png',1) #Color
```

Para saber si una imagen está en blanco y negro o en color basta con visualizar el atributo `shape` de la imagen cargada y si tiene tres dimensiones entonces es en color, si solamente es 2 entonces es en blanco y negro.

A la hora de visualizar la imagen *OpenCV* tiene su propia función, esta es `imshow`.

```
cv2.imshow('Titulo',img)
cv2.waitKey(0) #Esperar a pulsar cualquier tecla
cv2.destroyAllWindows() #Cerrar ventana
```

Otra alternativa es usar *MatPlotLib* usando la función `imshow`[5]. Sin embargo, de usar esta función se notará un error en la visualización.

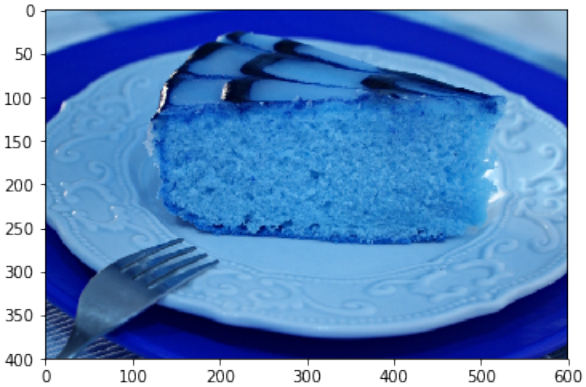

Figura 4: Imagen visualizada erróneamente

[5]Es importante que la importación de `matplotlib.pyplot` y `cv2` no se haga usando * por conflictos como este

Esto es debido a la codificación del color con que openCV lee las imágenes en BGR mientras que `imshow` de *MatPlotLib* espera una imagen en RGB. Aquí existe una discrepancia muy importante y es que mientras que la función de visualización de *OpenCV* espera imágenes en BGR en *MatPlotLib* se usa RGB, es por tanto que según se quiera visualizar mejor utilizar el modelo deseado. De manera particular lo recomendable es utilizar RGB por conveniencia.

Otro detalle importante es como se visualizan las imágenes en blanco y negro. Si usamos las funciones de *OpenCV* no importa, pero si utilizamos la de *MatPlotLib* debemos utilizar el flag `cmap=gray`[6].

Por tanto, una función importante es la conversión de codificaciones de color, esto se hace con la función `cvtColor` de `cv2` y las constantes de dirección (`cv2.COLOR_*`). Algunos ejemplos son:

```
im = cv2.cvtColor(im,cv2.COLOR_BGR2RGB) #De BGR a RGB
im = cv2.cvtColor(im,cv2.COLOR_RGB2HSV) #De RGB a HSV}
im = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)#De BGR a escala de grises
im = cv2.cvtColor(im,cv2.COLOR_BGR2YUV) #De BGR a YUV
```

2.2. Escritura de imagen

La escritura en disco de una imagen se realiza con la función `cv2.imwrite(ruta,imagen)`. Es importante destacar que la imagen ha de estar en BGR.

2.3. Corte y concatenación de capas de color

Además de las funciones que se vieron en *NumPy* para hacer extraer capas de una matriz o para concatenar matrices *OpenCV* posee un par de funciones que realizan lo mismo. Son `cv2.split(img)` y `cv2.merge((r,g,b))` semejantes a corte por *slice* (`img[:,:,0]`) y a `np.stack((r,g,b),axis=2)` respectivamente.

2.4. Tipo de datos

Las imágenes se codifican por defecto en números naturales (uint) de 8 bits (0 a 255)[7]. Sin embargo, que los valores esten en esos rangos dificulta el procesamiento, por lo que es interesante utilizar una codificación double ente 0 y 1. La función semejante en MatLab sería `im2double` pero en *OpenCV* la función es `normalize`

```
imgDouble = cv2.normalize(img.astype('float'),None, 0.0, 1.0,
    cv2.NORM_MINMAX)
```

[6] También veremos más de este flag en los mapas de color

[7] Un caso especial es HSV donde H va de 0 a 179 al representar el círculo cromático de 0 a 180º

Sin embargo, esta función no es siempre adecuada, primero porque no existen garantías que vaya a ver un valor 255 ó 0 en la matriz y luego, porque en los sistemas HSV no se considera 179 como el máximo, sino 255[8] por lo que el valor normalizado es erróneo. Por ende, aunque es una dificultad añadida a aquellos acostumbrados a trabajar con imágenes en *double* con MatLab, es recomendable en este caso utilizar solamente el tipo uint8.

2.5. Operaciones básicas sobre imágenes

En esta sección se explicarán las técnicas de modificación de una imagen, su brillo, contraste y filtrado.

2.5.1. Brillo

El brillo se define como la cantidad de luz que tiene una imagen, y se representa con valores más altos en las tres capas en RGB o en la capa valor en HSV. Hay que tener en cuenta que no es suficiente con sumar el valor, en el caso de que la suma sobrepase el valor del rango permitido por la codificación (0-255) por lo que una suma que provoque salir de ese rango degeneraría en hacer el pixel más oscuro o más claro cuando no se quiere, a este fenómeno se le llama desbordamiento y *OpenCV* no lo repara automáticamente. Es por tanto se recomienda aplicar esta función:

```
#Cambio de brillo
hsv = cv2.cvtColor(rgb, cv2.COLOR_RGB2HSV) #Convertir la imagen a hsv
h, s, v = cv2.split(hsv) #Extraer las capas en variables distintas
valor = int(input("Introduzca un brillo entre -255 y 255: "))
if valor > 0:
   lim = 255 - valor #valor es el brillo a aumentar
   v[v>lim] = 255
   v[v<=lim] += valor
else:
   lim = 0 + np.abs(valor) #valor es el brillo a disminuir
   v[v<lim] = 0
   v[v>=lim] -= np.abs(valor)

final = cv2.merge((h,s,v))
rgb = cv2.cvtColor(final, cv2.COLOR_HSV2RGB)
```

Una estructura similar se puede utilizar cuando hagamos operaciones que pueden provocar este desbordamiento.

Como resultado de esta operación tenemos el siguiente ejemplo, al que se le ha incrementado un brillo de 45:

[8] Si se utiliza `COLOR_RGB2HSV_FULL` en la conversión si es adecuado

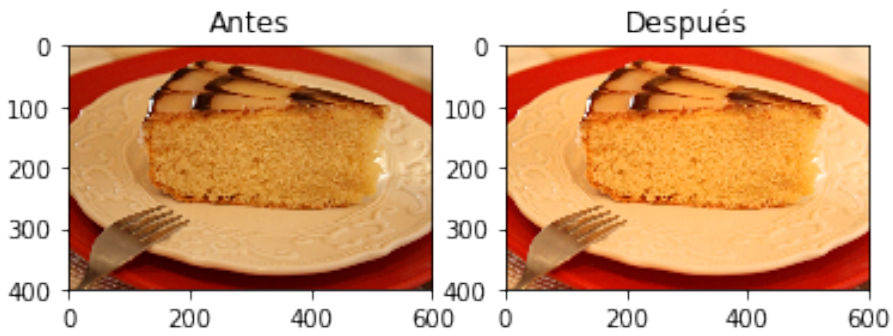

Figura 5: Ejemplo Brillo

2.5.2. Histograma

El histograma es una representación de cuantos píxeles hay en cada intensidad. En el se puede ver información muy útil de la imagen, en particular el las que están en escala de grises[9]. En Python existen varias formas de hacer histogramas, se comentarán las tres principales para la instalación, la de *OpenCV*, *NumPy* y *MatPlotLib*.

NumPy y *OpenCV* generan un array de tamaño 255 con la cantidad de píxeles con por cada intensidad. Se representan con la simple función `plot`. Se calcula de la siguiente manera:

```
#OpenCV
histCV = cv2.calcHist([img],[0],None,[256],[0,256])
#NumPy
histNP,bins = np.histogram(img.ravel(),256,[0,256])
```

Ambos métodos genera el mismo resultado. Lo que varía es la escala vertical.

[9]En RGB habría que realizarlo por cada capa

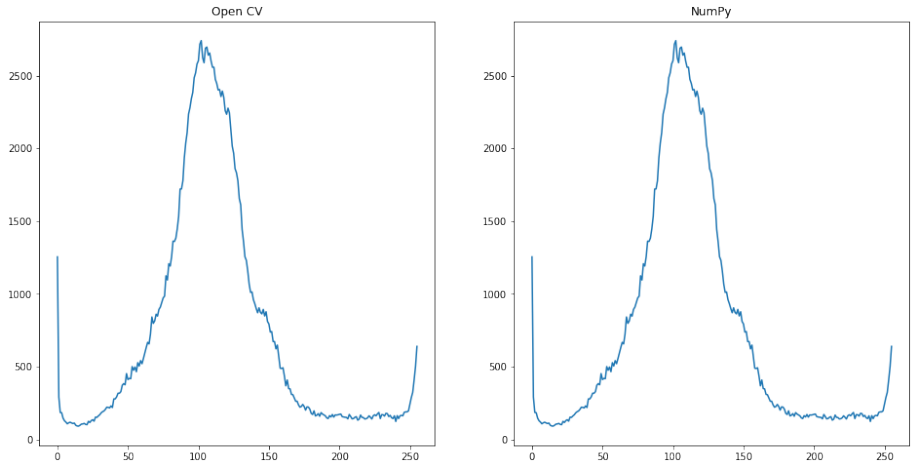

Figura 6: Histogramas

La última forma es la de *MatPlotLib*, esto con la función `hist`, los parámetros son los mismos que para *NumPy* y muestra directamente el histograma, pero esta vez, como un diagrama de barras, lo que aporta información mucho más relevante al no intentar hacer un interpolador segmentario[10]

```
plt.hist(img.ravel(),256,[2,256])
```

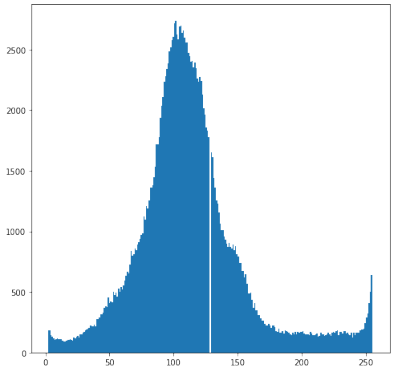

Figura 7: Histograma con *MatPlotLib*

Se puede observar más claramente los valores reales de cada intensidad, incluso observar valores vacíos al no calcular una línea que una dos puntos. La técnica de *NumPy* y *MatPlotLib* se pueden combinar quedando como resultado:

[10]Se dice de la técnica en programación matemática de calcular una función lineal que una los puntos consecutivos de una serie de resultados para una función desconocida

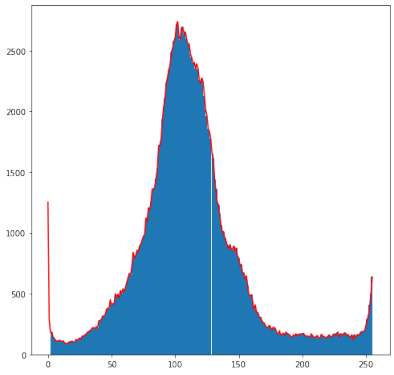

Figura 8: Histograma con *MatPlotLib* y *NumPy*

2.5.3. Ecualización

Una operación interesante sobre la imagen es la ecualización que viene a representar un .^estiramiento"del histograma aportando mayor contraste.

```
ecualizado = cv2.equalizeHist(img);
```

El resultado lo podemos ver a continuación:

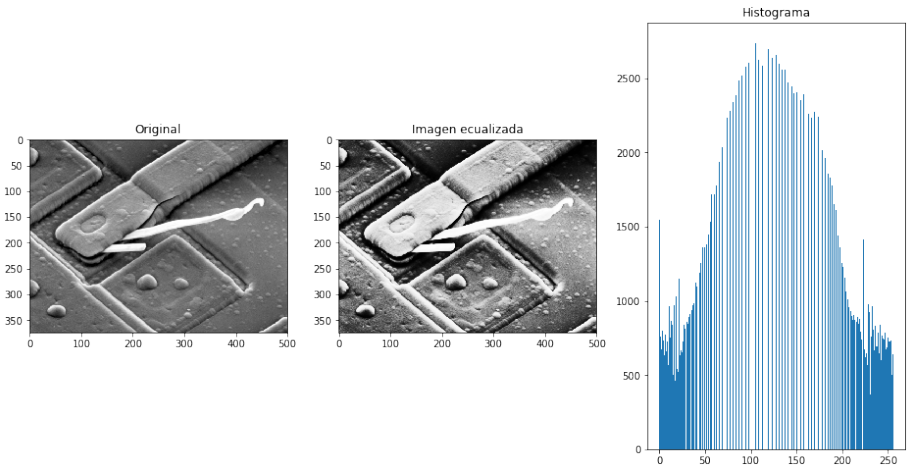

Figura 9: Ecualización

2.6. Filtrado

El filtrado a una imagen se realiza mediante la operación de convolución[11]. Los detalles técnicos se pueden explorar más en el manual [1].

Para hacer un filtrado se utiliza la función:
`cv2.filter2D(src, ddepth, kernel[, dst[, anchor[, delta[, borderType]]]])`.
Siendo *src* la imagen, *ddepth* la "profundidad"que viene a ser el tipo de datos, lo recomendable es que tome el valor de -1 para mantener el mismo tipo. Por último está *kernel* que es la matriz que contiene el núcleo de la convolución. El valor opcional importante que es interesante matizar es *anchor* que define la posición del pixel frente al *kernel*, de manera predeterminada vale (-1,-1) que viene a significar que es el centro del *kernel*.

Realmente no hace una convolución, sino una correlación. La función es la siguiente:

$$\mathrm{dst}(x,y) = \sum_{x'=0\ y'=0}^{x'<\mathrm{cols}\ y'<\mathrm{rows}} \mathrm{kernel}(x',y') * \mathrm{src}(x+x'-\mathrm{anchor}.x, y+y'-\mathrm{anchor}.y)$$

Un ejemplo de filtro genérico con un núcleo de 5x5 sería:

```
#Filtro generico
kernel = np.ones((5,5),np.float32)/50
dst = cv2.filter2D(img,-1,kernel)
```

El resultado sería una imagen con el filtro de media (el nombre propio del filtro aplicado).

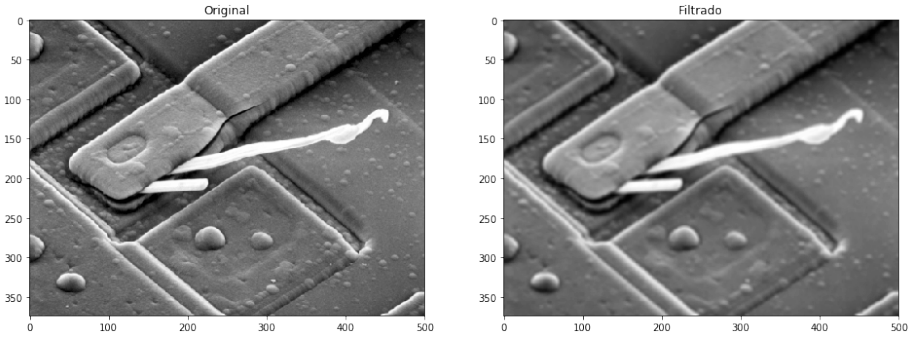

Figura 10: Filtro AVG

[11] .ªnálisis funcional operador matemático que transforma dos funciones f y g en una tercera función que representa la magnitud en la que superponen f y una versión trasladada e invertida de g"[6]

La mayoría de filtros se pueden aplicar con esta función, sin embargo, algunos necesitan un núcleo complejo por lo que es preferible utilizar funciones más específicas.

Entra las funciones específicas están la de media (`blur(img,(N,N))`), gaussiano (`GaussianBlur(img,(N,N),`σ`X,`σ`Y)`.

2.6.1. Gradientes

El gradiente es la función que representa la variación luminosa de una imagen. Es la derivada en ambos ejes (componente x y componente y). De manera práctica se utilizan como un filtro.

Existen tres formas de calcularlo, *prewitt*, *sobel* y el *laplaciano*. Las dos últimas tienen su propia función en *OpenCV* y *prewitt* requiere una generación del filtro manualmente.

De usar *prewitt* el núcleo sería una matriz 3x3 de la forma:

$$H_x = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

La componente H_y sería la transpuesta. La suma de las dos imágenes resultantes al utilizar ambos núcleos sería la imagen con *prewitt*.

Si usamos *sobel* también tendremos como resultado dos imágenes que habrá que sumar (x e y) pero no requeriremos de un núcleo, podremos generar los dos filtros utilizando al función `Sobel(img,-1,1,0,ksize=5)`[12] para la componente x `Sobel(img,-1,0,1,ksize=5)` para la componente y.

El caso del *laplaciano* es diferente, no requiere un kernel específico para cada componente ya que es simétrico:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

El operador laplaciano se calcula como:

$$\nabla^2 f(x,y) = \frac{\delta^2 f}{\delta x^2 f} + \frac{\delta^2 f}{\delta t^2 f}$$

La función necesaria es: `Laplacian(img,-1)`, se puede especificar el valor de δ utilizando el flag `delta`[13]. En definitiva el código sería:

[12] El `ksize` determina el tamaño del kernel

[13] Es importante recordar que tiene que ir entre el valor mínimo y máximo que vale cada píxel (0-255)

```
# Prewitt
Hx = np.array([[1,1,1],[0,0,0],[-1,-1,-1]])
Hy = np.transpose(Hx)
Px = cv2.filter2D(img,-1,Hx)
Py = cv2.filter2D(img,-1,Hy)
prewitt = Px+Py
# Sobel
Sx = cv2.Sobel(img,-1,1,0,ksize=5)
Sy = cv2.Sobel(img,-1,1,0,ksize=5)
sobel = Sx+Sy
# Laplaciano
laplaciano = cv2.Laplacian(img,-1)
```

Y como resultado tendríamos:

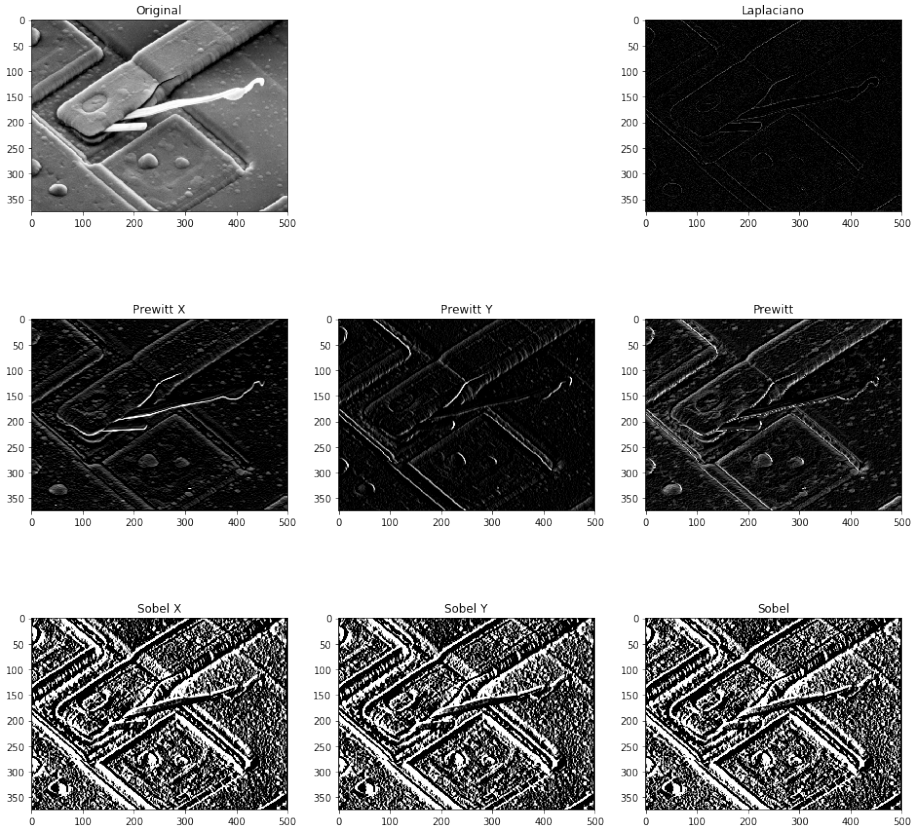

Figura 11: Gradientes

Estos gradientes los podemos utilizar para realzar imágenes. De esta manera podemos utilizar *sobel* y *prewitt* por $\frac{1}{2}$ sumada a la imagen original o restar el

laplaciano y tendríamos la imagen realzada:

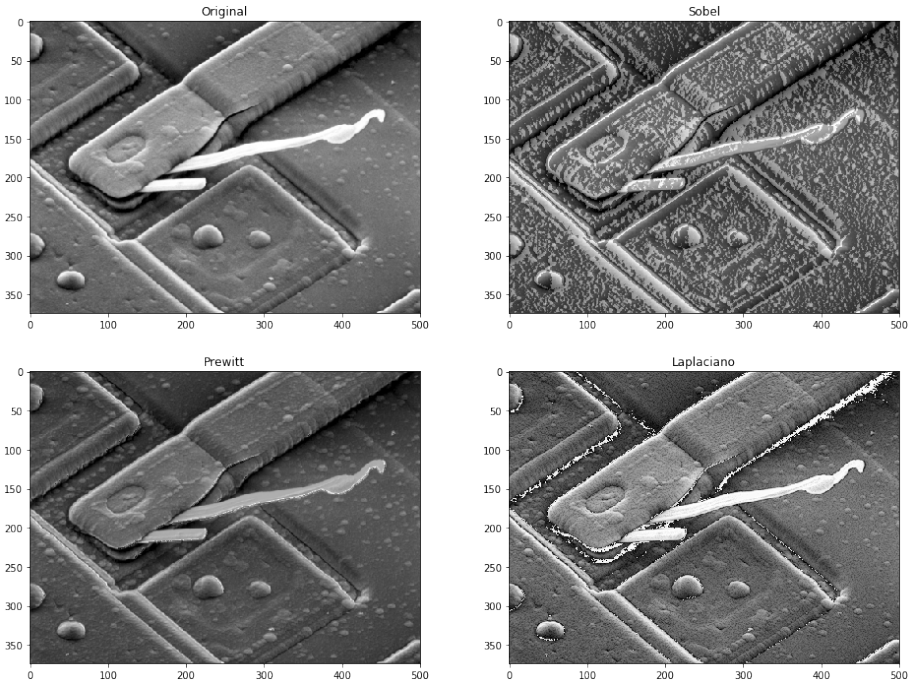

Figura 12: Realzado

3. Procesamiento de color

La mayoría de imágenes, especialmente en los últimos tiempos son en color. Como se especificó al principio de la sección anterior, las imágenes se codifican en tres capas, por defecto se abren en BGR y se puede cambiar utilizando `cvtColor` y la dirección querida. Las más comunes son[14]:

- cv2.COLOR_BGR2RGB
- cv2.COLOR_RGB2HSV_FULL[15]
- cv2.COLOR_RGB2YUV
- cv2.COLOR_RGB2HSI_FULL[16]
- cv2.COLOR_RGB2GRAY

Si lo separamos por capas podemos ver como son codificadas las imágenes.

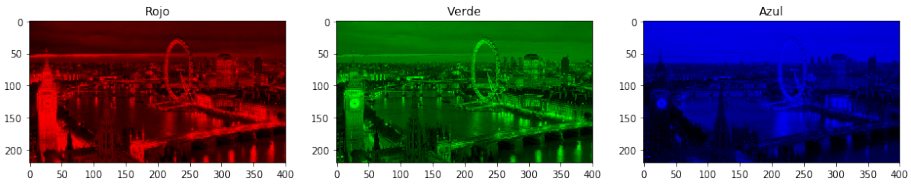

Figura 13: RGB

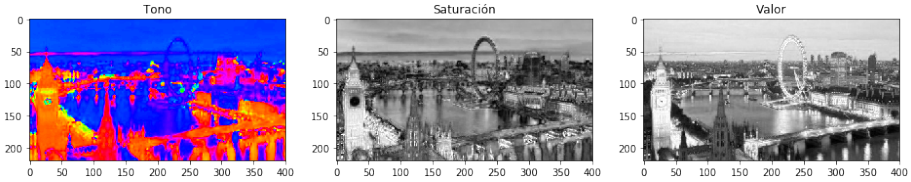

Figura 14: HSV

3.1. Falso color

El falso color es la generación de capas *RGB* mediante la información de luminosidad (valor en *HSV* o lo que es lo mismo, la única capa de blanco y negro). Se puede aplicar con al función `applyColorMap(imagen,colorMap)`, el mapa de color es un entero accesible mediante `COLORMAP_{nombre}`, entre ellos

[14] Todas tienen conversión entre ellas y son invertibles
[15] Se recomienda utilizar esta codificación para usar siempre la misma escala
[16] Se recomienda utilizar esta codificación para usar siempre la misma escala

está `JET, AUTUMN, BONE, COOL` etc. Sin embargo, también podemos aplicarlo sin necesidad de generar un nuevo objeto con la propia función de *MatPlotLib*, `imshow`, como vimos anteriormente, las imágenes en blanco y negro necesitaban el flag `cmap='gray'` para imprimirla correctamente, si cambiamos ese valor por `jet, autumn, bone` etc tendríamos el mismo resultado[17].

De manera adicional se puede mostrar la barra de color con la función `colorbar` de *MatPlotLib* pero solamente funcionará correctamente al utilizar la conversión de color mediante *MatPlotLib*.

```
#Mediante OpenCV
im2 = cv2.applyColorMap(imagenBN,cv2.COLORMAP_JET)
plt.imshow(cv2.cvtColor(im2,cv2.COLOR_BGR2RGB))

#Mediante MatPlotLib
plt.imshow(imagenBN,cmap='jet')
plt.colorbar()
```

El resultado sería:

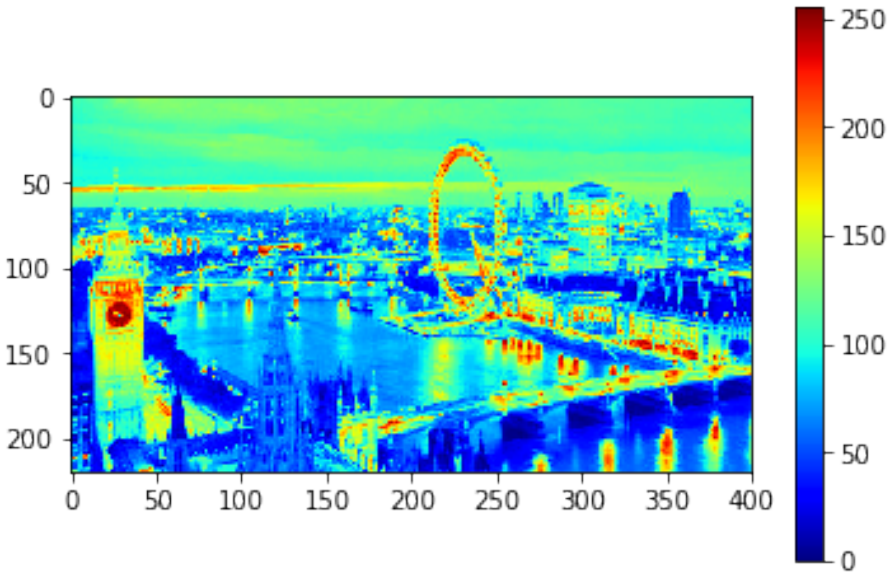

Figura 15: Aplicado el mapa de color JET

[17]Recordar que si usamos las funciones de openCV tendríamos la imagen en *BGR*

4. *Chroma Key*

El *Chroma Key* es la sustitución de un color por un fondo separando el objeto del fondo. Para eso utilizamos una máscara alfa que vale 0 para el fondo y 1 para el objeto, de tal manera que al multiplicar por la imagen solamente quede el objeto. Lo mismo con la versión invertida de la mascarilla para el fondo donde se pondrá la imagen, utilizando la operación suma al final.

La mascarilla se puede calcular utilizando las técnicas de distancia de color y diferencia de color[18].

4.1. Distancia de color

Para calcular la máscara alfa se parte de los valores r, g y b del fondo y se calcula mediante una función de distancia como la distancia *euclídea* o la *manhattan*, esta última más utilizada debido a su bajo costo *computacional*.

$$\alpha = \sqrt{(R - r_f)^2 + (G - g_f)^2 + (B - b_f)^2}$$

$$\alpha \approx |R - r_f| + |G - g_f| + |B - b_f|$$

De cualquier manera, el fondo valdrá 0, por tanto todos los píxeles que sean 0 (más un margen que probablemente será necesario[19]) valdrán 0 y el resto, independientemente de su valor, será 255. Es importante, que las tres capas estén en tipo float.

```
#Fondo
r = R[x,y]
b = B[x,y]
g = G[x,y]

alfa = np.abs(R-r)+np.abs(B-b)+np.abs(G-g)
#Considerando un margen de hasta 5
alfa[alfa>5] = 255
alfa[alfa<=5] = 0
```

[18] Es importante destacar que el fondo debe de ser de un color que no esté en el objeto a sustraer

[19] Se verán técnicas avanzadas para el cálculo del margen en la siguiente sección

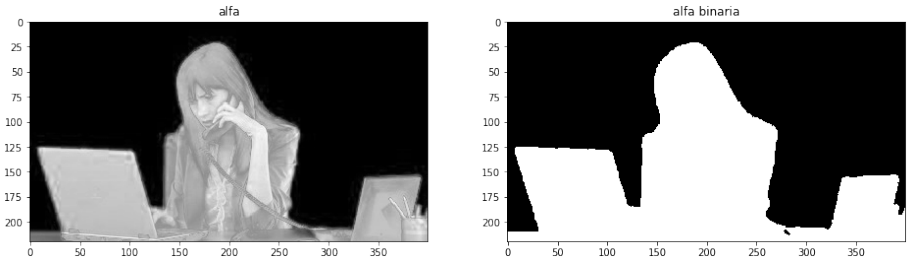

Figura 16: Máscara en distancia de color

4.2. Diferencia de color

Esta técnica consiste en obtener alfa evaluando la intensidad del canal del fondo frente al resto de canales. Tenemos que saber cual es el color predominante del fondo, algo que podemos saber a simple vista o analizando los pixeles del fondo. Primero calculamos M:

$$M = G - max(R, B)$$

En este caso el fondo suele ser más alto que el objeto por lo que lo recomendable es primero normalizar (función `cv2.normalice`), y luego $\alpha = 1 - M$.

```
M = G - np.maximum(R,B)
M = cv2.normalize(img.astype('float'),None, 0.0, 1.0, cv2.NORM_MINMAX)
alfa=1-M
```

Hay que recordar que en *OpenCV* no se corrige el desbordamiento, por lo que no funciona de manera igual. Por tanto M sería errónea y alfa también. En este caso la operación se complica por muchos factores. Para empezar tenemos que redefinir como hacer la resta de la capa prominente del resto de capas.

El primer paso sería convertir la imagen (antes de extraer las capas) al tipo *float*:

```
imgF = img.astype('float')
```

De esta manera no evitamos el desbordamiento pero es más fácil de regular. Tras esto realizamos la operación $G -$ np.maximum(R, B). Y corregimos los desbordamientos.

```
M[M<0]=0
M[M>255]=255
M = M.astype('uint8')

#Considerando un margen de hasta 5
alfa = cv2.bitwise_not(M)
```

```
alfa[alfa>5] = 255
alfa[alfa<=5] = 0
```

Es importante remarcar que al usar la diferencia de color tendremos la máscara invertida, es decir, el fondo será blanco, mientras que con distancia de color será al revés, por eso una solución rápida es invertir la imagen negando los bits (`v2.bitwise_not(M)`)

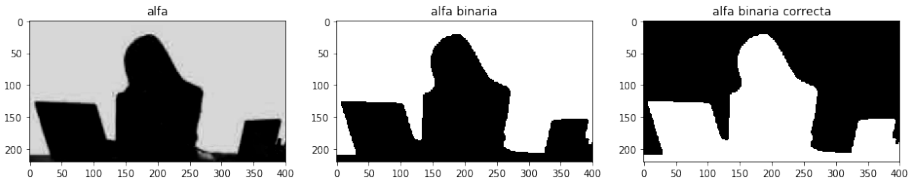

Figura 17: Máscara en diferencia de color

4.3. Adición de fondo

Tras obtener la máscara alfa por cualquier sistema realizamos la binarización de la máscara convirtiéndola al tipo *bool*, después generamos la máscara inversa que será la máscara para el fondo. Finalmente generamos dos partes, una la imagen sin el fondo *chroma* multiplicando la máscara por la imagen, lo mismo se realiza con el nuevo fondo con la máscara invertida. Se suman los resultados y ya tenemos el resultado final.

```
dsI = np.stack((alfa,alfa,alfa),axis=2)*img
dsF = np.stack((alfaF,alfaF,alfaF),axis=2)*fondo
dsFinal = dsF+dsI
```

Como detalle final es importante saber que debemos utilizar la función de concatenación en el eje z debido a que *NumPy* solo hace operaciones de dos matrices si estas son de las mismas dimensiones. No se puede utilizar la función `merge` debido a que el tipo de datos booleano no es soportado por *OpenCV* en esta función.

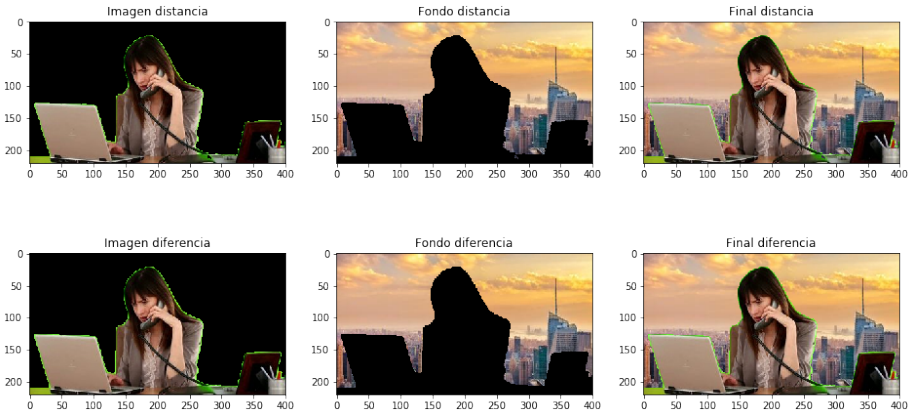

Figura 18: Resultado final

Se puede observar una linea verde, para corregir esto bastaría con subir el margen entre el valor 0 y el valor 255.

5. Imágenes binarias

Para los problemas comunes dentro de la visión por computador solemos tener un exceso de información, como color o muchos niveles de grises que son innecesarios. Por tanto, la mayoría de problemas se pueden resolver usando solamente niveles, el negro y el blanco, y en esto consisten las imágenes binarias.

5.1. Binarización de una imagen

La binarización es simplemente convertir un conjunto de valores como el valor máximo y el otro conjunto en el valor mínimo (255 y 0 en unit8, True y Falso en bool...). Esto lo podemos hacer de muchas maneras.

Utilizar un valor umbral y modificandolo utilizando las herramientas de *NumPy*:

```
imbn[imbn>100] = 255
imbn[imbn<=100] = 0
```

O utilizando la función de *OpenCV* `threshold`[20]

```
l,imbn2 = cv2.threshold(img,127,255,cv2.THRESH_BINARY)
```

Recibe como parámetros la imagen a binarizar, el umbral, el valor máximo y el tipo de binarización. Este puede ser `THRESH_BINARY, THRESH_BINARY_INV, THRESH_TRUNC, THRESH_TOZERO` y `THRESH_TOZERO_INV`. Las diferencias serían:

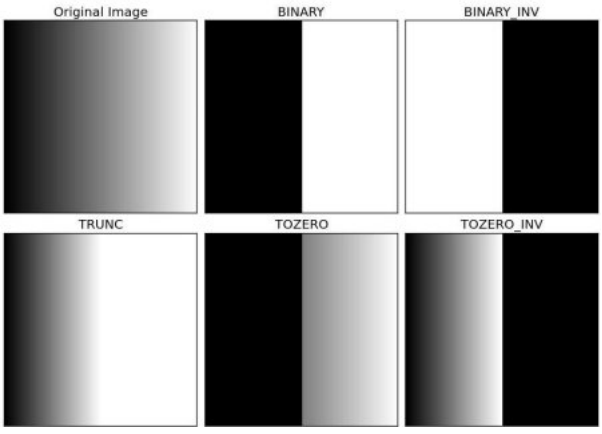

Figura 19: Tipos de Thresholds

El problema que ocurre con este sistema manual es que muchas veces no se

[20] Es importante destacar que devuelve dos valores en una tupla, el primer valor(l) no es importante de momento

conoce cual es el umbral que necesitamos para separar el objeto del fondo. Para eso tenemos el método *Otsu* que calcula, cual es el umbral para la separación.

```
l,imbnO = cv2.threshold(img,0,255,cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)
```

Como se puede observar el valor que antes era el umbral ahora es el valor 0 y el modo se le suma `THRESH_OTSU`[21]. Si la imagen tiene mucho ruido es buena idea utilizar un filtrado gaussiano antes de aplicarlo.

Otro modo de hacer el *threshold* es hacerlo de manera adaptativa, es especialmente útil cuando estamos antes cambios de luz muy marcados, sin embargo, si se usan ventanas[22] muy pequeñas y los objetos son grandes se conseguirá un efecto similar a un gradiente. Existen dos modos, el de media y el gaussiano. La función que se aplica es `cv2.adaptiveThreshold(src, maxValue, adaptiveMethod, thresholdType, blockSize, C`, recibiendo la imagen a calcular, el valor máximo, el método, el tipo de *threshold*, el tamaño de la máscara y un valor a sustraer del resultado (generalmente positivo). Por tanto los dos tipos sería:

```
imbnM = cv2.adaptiveThreshold(img,255,
                        cv2.ADAPTIVE_THRESH_MEAN_C,
                        cv2.THRESH_BINARY_INV,101,25)
imbnG = cv2.adaptiveThreshold(img,255,
                        cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                        cv2.THRESH_BINARY_INV,101,25)
```

[21] En este caso, el valor de l nos determina que hemos utilizado OTSU
[22] Se llama ventana a un tamaño de información

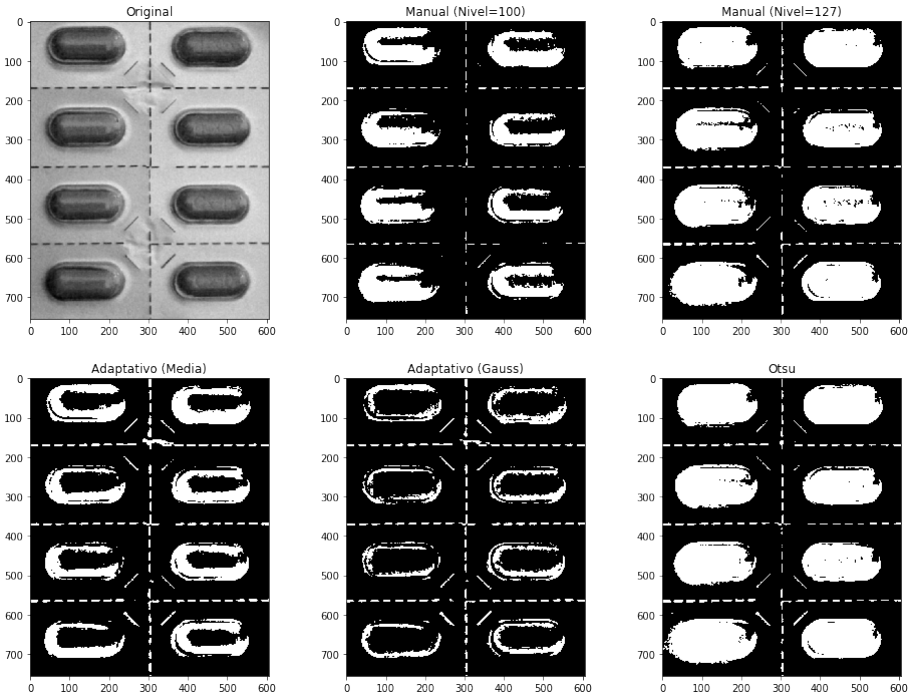

Figura 20: Comparación de métodos de binarización

5.2. Rellenado de agujeros

Una vez tengamos nuestra imagen binaria la mayoría de las veces no nos valdrá completamente, en muchos casos nuestro mayor problema será la iluminación con la generación de sombras no deseadas y brillos excesivos, esto provoca que, aún usando sistemas de binarización como OTSU, tendremos huecos en nuestros objetos. Para esto existe un método para rellenarlos. El proceso es algo complejo teóricamente, pero es simple de ejecutar. Consiste en crear una máscara que contenga los pixeles de los agujeros y mediante la operación `OR` juntarla con la imagen binaria. El proceso sería el siguiente:

```
#Mascara
h, w = imbn0.shape[:2]
mask = np.zeros((h+2, w+2), np.uint8)

#Rellenado
imbnF = imbn0.copy()

cv2.floodFill(imbnF,mask, (0,0), 255) #Calculo de rellenado
imbnI = cv2.bitwise_not(imbnF) #Inversion para que los pixeles blancos
    sean los del hueco
```

```
imbnHF = imbnO | imbnI #Operacion or
```

El resultado sería el siguiente:

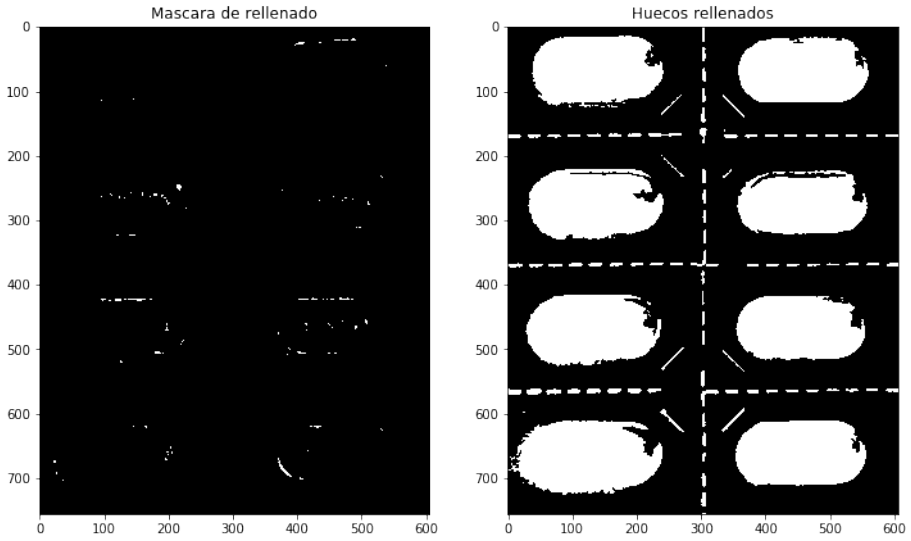

Figura 21: Rellenado

5.3. Detección de bordes

Matemáticamente existen tres técnicas de detección de bordes, son *prewitt*, *sobel* y *cany*. Las dos primeras se han visto. Añadir estarái *Cany* y es bastante simple, unicamente requiere de la imagen original dos valores de *threshold*

```
imbnCa = cv2.Canny(img,100,50)
```

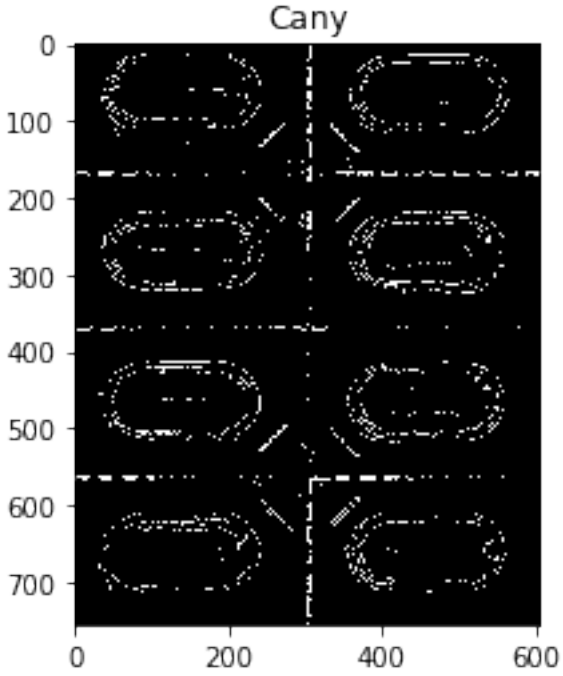

Figura 22: Cany

5.4. Extracción de características

Una gran ventaja de trabajar con imágenes binarias es que son muy útiles para extraer características. El primer paso es extraer los contornos, que son vectores con todos los puntos que rodean los objetos. La función es `findContours` y devuelve la misma imagen, los *contornos* y una matriz de herencia que no necesitaremos. Necesitaremos el modo y el método, que los definimos por constantes de *OpenCV* y, para no complicar los distintos métodos dependiendo del tipo de dato usaremos unicamente `cv2.RETR_TREE` para el modo y `cv2.CHAIN_APPROX_SIMPLE` como método que son los recomendables si la imagen binaria la hemos conseguido con OTSU.

Finalmente tendremos la función de esta manera:

```
im2, contours, hierarchy =
    cv2.findContours(imbnHF,cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
```

El tamaño de la variable *contours* será el número de objetos detectados.

A partir de estos objetos detectados podemos obtener características como el rectángulo que rodea al objeto (`cv2.boundingRect`), el área (`cv2.contourArea`)

y el perímetro (`cv2.arcLength`). Además podemos a partir de un conjunto de valores llamado momentos[23] podemos calcular el centro del objeto.

Considerando entonces un contorno `obj` los datos se extraería de esta manera:

```
#Con funciones propias
x,y,w,h = cv2.boundingRect(obj) #BoundingBox
area = cv2.contourArea(obj) #Area
peri = cv2.arcLength(obj,True) #Perimetro

#Sin funciones propias
M = cv2.moments(obj)

#Semejante a Centroid
cX = int(M["m10"] / M["m00"])
cY = int(M["m01"] / M["m00"])
center = (cX,cY) #Centro
```

5.4.1. Dibujado sobre imagen

Una herramienta muy útil para la visualización de resultados el la impresión de un rectángulo para rodear a un objeto detectado sobre la imagen original. Para ello necesitamos de la librería `matplotlib.patches` que llamaremos `ptc`. Esta librería tiene el objeto `Rectangle` que podemos definir con los valores de x,y,w y h que calculamos para el *BoundingBox*, y como cualquier objeto de *MatPlitLib* podremos añadirle flags como el color de relleno, tamaño de borde, color del borde etc.

Para poder integrarlo con la imagen debemos generar unos *axes* a partir de la función `plt.subplots` (no confundir con `subplot`), esta función nos devuelve una figura y los *axes*. Sobre esos se puede aplicar la función `add_patch` y pasarle el rectángulo.

```
import matplotlib.patches as ptc

fig,ax = plt.subplots(1)
fig.set_size_inches(12,8)
ax.imshow(img,cmap='gray')

for c in contours:
x,y,w,h = cv2.boundingRect(c)
rect =
    ptc.Rectangle((x,y),w,h,linewidth=1,edgecolor='r',facecolor='none')
ax.add_patch(rect)
```

[23] El momento de una imagen es una media ponderada de la intensidad de los pixeles

Un ejemplo sería:

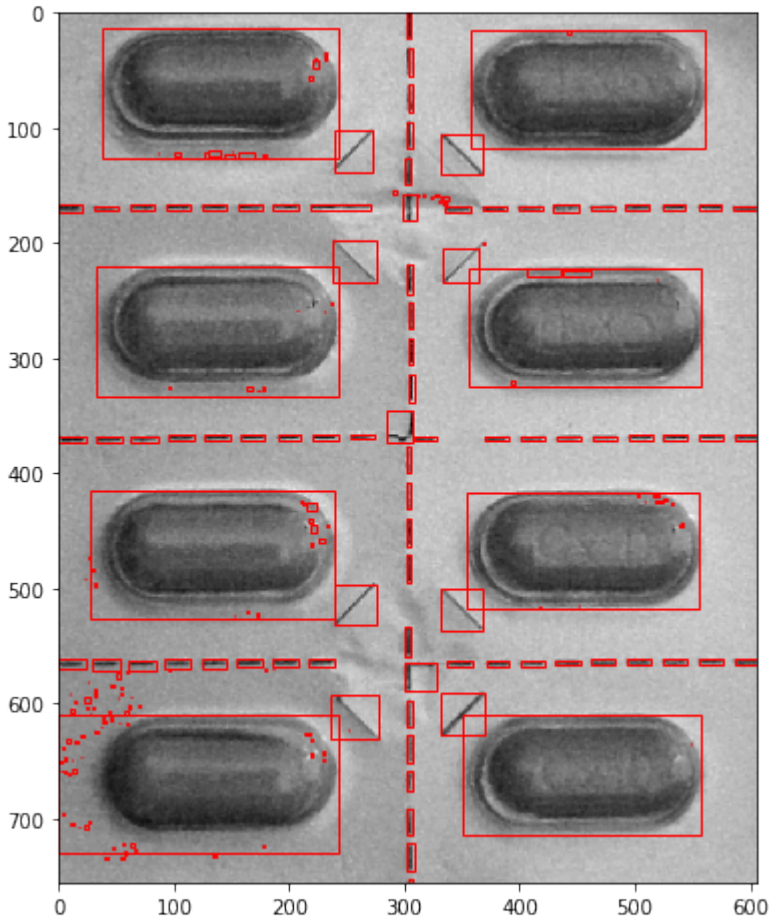

Figura 23: Rectángulos sobre la imagen

Podemos hacer lo mismo con textos, aunque sobre la librería *PyPlot* sin necesidad de añadírselo a los *axes* y por tanto funciona semejante a la función `text` de *MatLab*.

```
plt.text(cX,cY,'Obj '+str(counter),color='w')
```

5.5. Operaciones morfológicas

En el apartado anterior pudimos observar como podemos llegar a tener muchos objetos innecesarios al ser muy pequeños o que los objetos están incompletos, para esto podemos utilizar las operaciones morfológicas de erosión y dilatación así como el gradiente, apertura y cierre, estas dos últimas alternan erosión y dilatación y viceversa respectivamente.

Para estas operaciones se utilizan máscaras que pueden ser fabricadas manualmente o generandola nosotros con la función `cv2.getStructuringElement` que recibe uno de los tres tipos (`MORPH_RECT, MORPH_ELLIPSE` y `MORPH_CROSS`) y el tamaño como una tupla.

```
sr = cv2.getStructuringElement(cv2.MORPH_RECT,(9,9))
se = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(9,9))
sc = cv2.getStructuringElement(cv2.MORPH_CROSS,(9,9))
```

La apariencia de estas máscaras en 9x9 sería:

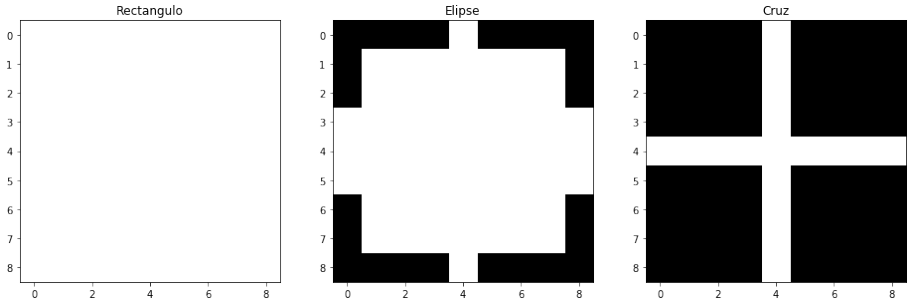

Figura 24: Máscaras morfológicas

Las operaciones de erosión y dilatación tienen sus propias funciones, con el nombre en inglés y reciben la imagen a dilatar, la máscara y las iteraciones que se van a hacer.

```
dil = cv2.dilate(imbnM,st,iterations = 1)
ero = cv2.erode(imbnM,st,iterations = 1)
```

Y la operación de gradiente, apertura y cierre con la misma función cambiando el argumento del tipo.

```
gra = cv2.morphologyEx(imbnM,cv2.MORPH_GRADIENT,st)
opn = cv2.morphologyEx(imbnM,cv2.MORPH_OPEN,st)
clo = cv2.morphologyEx(imbnM,cv2.MORPH_CLOSE,st)
```

Para comparar estos distintos resultados podemos ver la siguiente imagen.
La conclusión obvia es que la apertura sirve para eliminar objetos pequeños,

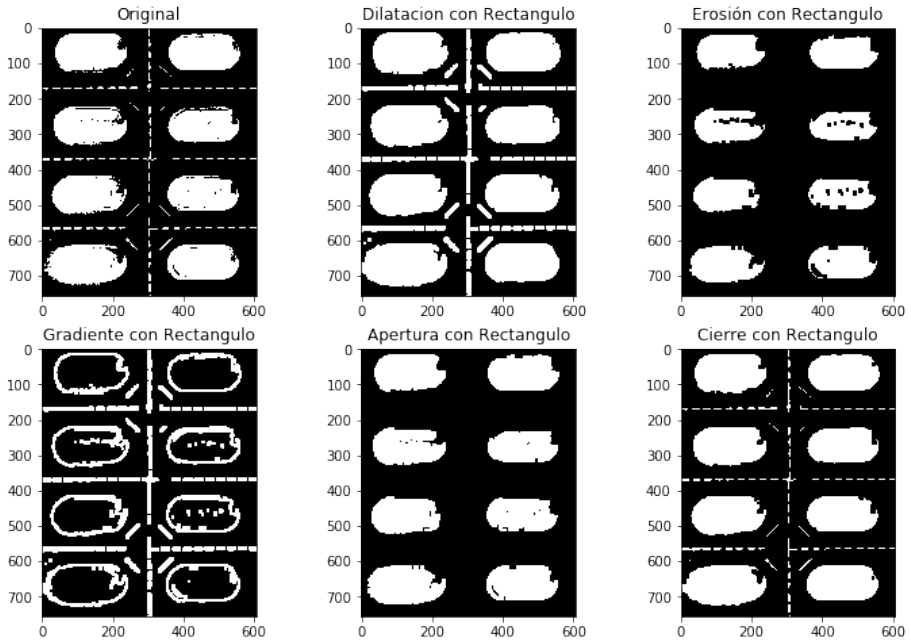

Figura 25: Operaciones morfológicas

el cierre para rellenar objetos y el gradiente para los bordes.

Otra operación interesante, pero que no está en *openCV* es la eliminación de ruido mediante el borrado de aquellos objetos con una cantidad de pixeles menor a la indicada. Esto no existe, sin embargo, una técnica que podemos aplicar es usar el área a partir de los contornos y aquellos que tengan un área menor que la indicada serían borrados de la colección ya que es sobre esta que hacemos las operaciones.

6. Procesamiento de vídeo

El vídeo es, por definición, un conjunto de imágenes consecutivas a una velocidad determinada, a cada una de las imágenes se les llama cuadro, fotograma o *frame*, la velocidad se mide en cuadros por segundo, *fps* por sus siglas en inglés. Es por tanto que resulta fácil deducir que procesar vídeo no es más que procesar un conjunto de imágenes.

Para ello en *OpenCV* existen varios objetos para el procesamiento de vídeo donde se almacenan los datos de los mismos e, independientemente de si es un objeto de lectura o escritura, tienen varias características importantes a destacar.

- Ancho (en píxeles) `CV_CAP_PROP_FRAME_WIDTH`
- Alto (en píxeles) `CV_CAP_PROP_FRAME_HEIGHT`
- Cuadros por segundo `CV_CAP_PROP_FPS`
- Cantidad de cuadros `CV_CAP_PROP_FRAME_COUNT`

Estos datos se pueden leer y modificar (como el *frame-rate*) con las funciones `set` y `get` sobre los objetos de vídeos que se verán a continuación.[24]

6.1. Lectura y visualización

Para leer un vídeo se requiere crear un objeto de tipo `cv2.VideoCapture` que es instanciado con la ruta del fichero de vídeo a abrir. Automáticamente abre el *stream* pero en caso contrario siempre se puede verificar el estado del objeto con la función `isOpened`.

Sobre los objetos de captura de vídeo se utiliza la función `read` que devuelve dos valores, un flag que determina si se ha recibido correctamente el siguiente *frame* y el siguiente frame. También cada llamada provoca que se mueva el puntero del frame leído (independientemente de que se haya leído bien). Este valor se accede con `get` y modifica con *set* y tiene el flag `CV_CAP_PROP_POS_FRAMES`.

Con todo esto un código sencillo de lectura de un vídeo sería:

```
cap = cv2.VideoCapture("./out.mp4")
while not cap.isOpened():
    print("Reitentando")
   cap = cv2.VideoCapture("./out.mp4")
   cv2.waitKey(1000)

pos_frame = cap.get(1)
```

[24] Los valores constantes CV_CAP... no están en la librería, en el apéndice 1 se pueden ver los valores enteros (8)

```
while True:
   flag, frame = cap.read()
   if flag:
      # El siguiente frame se ha leido
      pos_frame = cap.get(1)
        print(str(pos_frame)+" frames leidos")
   else:
      # El frame no se ha leido y se reintenta la lectura
      cap.set(1, pos_frame-1)
      print("No esta listo")
      # Esperamos
      cv2.waitKey(1000)

   if cap.get(1) == cap.get(7):
      # Cuando el total de frames y los frames leidos coinciden se
          termina
      break

#Cerramos el video
cap.release()
```

Cada objeto *frame* es una imagen con las mismas propiedades que lo visto en el resto del documento.

A la hora de visualizar nos encontraremos un problema si queremos utilizar *MatPlotLib* y es que, debido a su funcionamiento, no mostrará cada *frame* tras cada llamada a su función `imshow`, es por eso, que para este caso debemos utilizar la función `imshow` de *openCV*, esto implica que el *frame* ha de estar en BGR y que deberemos poner un control para cerrar la ventana como vimos en el apartado 2.1

6.2. Escritura de vídeo

Para poder guardar un vídeo el proceso es semejante pero invertido. Se crea un objeto de tipo `cv2.VideoWriter` que recibe como parámetros el *códec*, la frecuencia de los cuadros y las dimensiones. Para añadir un nuevo frame basta con la función `write` que recibe como parámetro la imagen adecuada.

El parámetro especial es el *códec* o `fourcc`. Se consigue utilizando el objeto `cv2.VideoWriter_fourcc()` y recibiendo una cadena con el codec (`*'MJPG'`) o los cuatro caracteres que lo forman (`'M','J','P','G'`).

Como ejemplo tenemos este código que pasa un vídeo al doble de velocidad:

```
cap = cv2.VideoCapture("./videos/video04.avi")

while not cap.isOpened():
   cap = cv2.VideoCapture("./videos/video04.avi")
```

```
   cv2.waitKey(1000)

pos_frame = cap.get(1)

fourcc = cv2.VideoWriter_fourcc('M','J','P','G')

#fourcc = cv2.VideoWriter_fourcc(*'XVID')
out = cv2.VideoWriter("./videos/videoOut.avi",fourcc, cap.get(5)*2,
    (int(cap.get(3)),int(cap.get(4))))

while True:
   flag, frame = cap.read()
   if flag:
      # El siguiente frame se ha leido
      pos_frame = cap.get(1)
   else:
      # El frame no se ha leido y se reintenta la lectura
      cap.set(1, pos_frame-1)
      # Esperamos
      cv2.waitKey(1000)

   out.write(frame)
   if cap.get(1) == cap.get(7):
      # Cuando el total de frames y los frames leidos coinciden se
          termina
      break

cap.release()
out.release()
```

7. Captura de imagen

Para concluir esta guía terminamos con la captura de imágenes mediante dispositivos tales como cámaras web. El funcionamiento es el mismo que el de un objeto de vídeo solamente que en vez de utilizar una ruta para abrir el fichero se pasa un número que indica que cámara, la cámara por defecto del sistema es 0.

La diferencia principal sobre los vídeos es que en este tipo de objetos no vamos a poder modificar el alto y ancho, ni la frecuencia de captura ni otros detalles que vienen implícitos en la cámara. La lectura de un frame se realiza de la misma manera (función read) y no tiene más detalles relevantes.

7.1. *Timer*

Una utilidad que se le pueden dar a las cámaras es que realicen capturas de imágenes cada cierto tiempo, para eso están los temporizadores, objetos `Timer` que llaman a una función cada cierto intervalo una cantidad de veces. El método para hacer esto es utilizar tres variables globales que no dependan de la captura y usar la librería `threading`. Las variables que necesitaremos serán el plazo entre dos capturas, la cantidad de capturas a realizar y la cantidad de capturas realizadas, además tendrán que ser globales la cámara y el objeto de vídeo de tal manera que el código sería:

```
def shot():
   global cam,write,fps,total,veces

   _,frame = cam.read()
   if _:
      write.write(frame)
      veces += 1

   if veces != total:
      threading.Timer(fps, shot).start()
   else:
      cam.release()
      write.release()
```

Referencias

[1] C. Represa Pérez, *Procesamiento Digital de Imagen y Vídeo*. Departamento de Ingeniería Electromecánica, Universidad de Burgos.

[2] Wikipedia contributors, "Anaconda (python distribution) — Wikipedia, the free encyclopedia," 2018. [Online; accessed 27-December-2018].

[3] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[4] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in science & engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[5] T. E. Oliphant, *A guide to NumPy*, vol. 1. Trelgol Publishing USA, 2006.

[6] L. M. Cabrejas Arce, J. L. Garrido Labrador, and J. M. Ramírez Sanz, "Rna aplicadas a robótica,"

Apéndice I - Constantes de características de vídeo

- CV_CAP_PROP_POS_MSEC —— 0
- CV_CAP_PROP_POS_FRAMES —— 1
- CV_CAP_PROP_POS_AVI_RATIO —— 2
- CV_CAP_PROP_FRAME_WIDTH —— 3
- CV_CAP_PROP_FRAME_HEIGHT —— 4
- CV_CAP_PROP_FPS —— 5
- CV_CAP_PROP_FOURCC —— 6
- CV_CAP_PROP_FRAME_COUNT —— 7
- CV_CAP_PROP_FORMAT —— 8
- CV_CAP_PROP_MODE —— 9
- CV_CAP_PROP_BRIGHTNESS —— 10
- CV_CAP_PROP_CONTRAST —— 11
- CV_CAP_PROP_SATURATION —— 12
- CV_CAP_PROP_HUE —— 13
- CV_CAP_PROP_GAIN —— 14
- CV_CAP_PROP_EXPOSURE —— 15
- CV_CAP_PROP_CONVERT_RGB —— 16
- CV_CAP_PROP_WHITE_BALANCE_BLUE_U —— 17
- CV_CAP_PROP_RECTIFICATION —— 18
- CV_CAP_PROP_MONOCHROME —— 19
- CV_CAP_PROP_SHARPNESS —— 20
- CV_CAP_PROP_AUTO_EXPOSURE —— 21
- CV_CAP_PROP_GAMMA —— 22
- CV_CAP_PROP_TEMPERATURE —— 23
- CV_CAP_PROP_TRIGGER —— 24
- CV_CAP_PROP_TRIGGER_DELAY —— 25
- CV_CAP_PROP_WHITE_BALANCE_RED_V —— 26