

Laboratorio Nº 3- Detección y Extracción de Características			
Asignatura	Tópicos Especiales - Visión Artificial	Código	0756
Profesor	José Carlos Rangel Ortiz		

Introducción

En este laboratorio se mostraran los diferentes algoritmos disponibles en OpenCV para la detección de puntos clave dentro de las imágenes. Se abordaran los detectores de esquinas y regiones en diferentes imágenes. De igual manera se presenta al estudiante la manera de generar descriptores para puntos clave de las imágenes.

Dentro del contenido se emplearan de igual manera las librerías Numpy y MatPlotLib, las cuales seguirán siendo utilizadas a lo largo del semestre.

Fundamentos Teóricos

El ser humano tiene una capacidad sorprendente para unir las piezas o trozos de un objeto o elemento que está roto o desarmado. Si nos centramos en el ejemplo de un rompecabezas, los humanos desde tempranas edades demuestran la aptitud de armarlos basándose en la forma y contenido visual de cada pieza.

Este proceso que se puede decir está programado en nuestro cerebro, se enfoca en comparar las piezas y encontrar en ellas fragmentos de información que permitan determinar la continuidad de la imagen cuando dichas piezas se unan.

Instintivamente los seres humanos detectan estas “características” en cada pieza y usando este método son capaces de completar zonas de mayor tamaño cada vez.

La Visión Artificial al tratar de emular este comportamiento ha definido en sus procesos la manera de identificar estas características y utilizarlas en sus procesos. Para ello se basa en diversas funciones matemáticas que analizan la imagen de entrada y generan una lista de puntos que son considerados claves en la imagen y los cuales representan esta mediante un número reducido de parámetros.

Al trabajar con imágenes en el computador se hace difícil para muchos procesos emular el comportamiento humano, por lo cual, se han desarrollado una diversidad de algoritmos que detectan estos puntos clave (*keypoints*), según el tipo o contenido de las imágenes. De igual manera han surgido los algoritmos que buscan no solo un punto, sino un conjunto o región de estos ya sea una esquina, o alguna forma geométrica primitiva.

Además de la detección de los *keypoints*, se hace necesario una manera de almacenar información de estos para así realizar comparaciones que permitan encontrar similitudes entre imágenes

similares. En este punto surgen los descriptores, estos algoritmos se enfocan en analizar la vecindad de un *keypoint* para crear y almacenar una representación de la misma de tal manera que pueda ser utilizada posteriormente.

Los **detectores** y **descriptores** son un conjunto de algoritmos que se consideran del bloque de Visión Artificial Tradicional, pero que son útiles en una gran cantidad de escenarios como flujo óptico, reconocimiento de patrones, reconocimiento de objetos, etc.

Este laboratorio presenta de manera práctica los diferentes algoritmos para detectar y describir puntos claves. Se presentan solo los algoritmos que tienen acceso desde la versión estable de OpenCV, pero es importante que el estudiante conozca que existen algoritmos de pago y otros que solo están disponible en la versión *contrib* de OpenCV.

Objetivo

Extraer puntos claves y descriptores de diferentes imágenes utilizando OpenCV y el lenguaje Python.

Entregables

Al final de cada sección del laboratorio se indicará cuales son acciones que deberá cumplir el estudiante y los cuales serán evaluados en la actividad. El informe final debe contener todo lo solicitado y se debe redactar utilizando la Guía de Estilos para Memorias y Laboratorios Disponible en Moodle. El informe debe ser entregado en formato PDF y siguiendo las indicaciones para nombrar el archivo.

1. Detectores de Keypoints

1.1. Detector MSER

Este detector se enfoca en identificar regiones de puntos clave dentro de una imagen. Se construye aplicando filtros binarios en ambas direcciones a la imagen original de tal manera que cuando una región sigue presente en varios niveles, se le considera una región clave. Para mayor información de esta función se puede acceder a este enlace ¹.

1. Como primer paso crea un *script/notebook* y añade las importaciones a OpenCV, Numpy(np) y Pyplot(plt) de Matplotlib.
2. En el siguiente código cargamos nuestra imagen y se convierte a escala de grises, se crea la instancia del método y se aplica el mismo. Este, devuelve un conjunto de coordenadas y *bounding boxes* de las regiones identificadas dentro de la imagen.

```
# Leer la imagen y cambiar el espacio de color
imgname = 'img/mariposa.JPG'
imgMSER = cv2.imread(imgname)
grayMSER = cv2.cvtColor(imgMSER, cv2.COLOR_BGR2GRAY)

# Inicializar MSER y asignar parámetros
mser = cv2.MSER_create()

# Realizar la detección, obteniendo las coordenadas
# de las regiones y sus bounding boxes
coordinates, bboxes = mser.detectRegions(grayMSER)
```

3. El siguiente fragmento analiza los resultados del método y según el tamaño de las regiones selecciona algunas de las detectadas, de igual manera se define un arreglo con valor BGR de colores para pintar las regiones de colores diferentes en forma aleatoria. En este caso se estarán generando 3 imágenes (*canva#*) en la cual veremos las regiones identificadas. La imagen original, una en escala de grises y una máscara de las regiones detectadas.

```
coords = []
for coord, bbox in zip(coordinates, bboxes):
    x,y,w,h = bbox
    if w< 10 or h < 10 or w/h > 5 or h/w > 5:
        continue
    coords.append(coord)

print( "Regiones Detectadas usando MSER Detector : {}".format(len(coordinates)))

# lista de colores para asignar a las regiones
```

¹MSER

```

colors = [[43, 43, 200], [43, 75, 200], [43, 106, 200], [43, 137, 200],
          [43, 169, 200], [43, 200, 195], [43, 200, 163], [43, 200, 132],
          [43, 200, 101], [43, 200, 69], [54, 200, 43], [85, 200, 43],
          [116, 200, 43], [148, 200, 43], [179, 200, 43], [200, 184, 43],
          [200, 153, 43], [200, 122, 43], [200, 90, 43], [200, 59, 43],
          [200, 43, 64], [200, 43, 95], [200, 43, 127], [200, 43, 158],
          [200, 43, 190], [174, 43, 200], [142, 43, 200], [111, 43, 200],
          [80, 43, 200], [43, 43, 200]]
```

```

# Pintar las regiones con colores aleatorios
np.random.seed(0)
canvas1 = imgMSER.copy()
canvas2 = cv2.cvtColor(grayMSER, cv2.COLOR_GRAY2BGR)
canvas3 = np.zeros_like(imgMSER)
```

4. El siguiente fragmento de código dibuja en cada uno de los *canva#* creado las regiones, utilizando un color aleatorio. Se guarda la imagen y se muestra el resultado en pantalla.

```

for cnt in coords:
    xx = cnt[:,0]
    yy = cnt[:,1]
    color = colors[np.random.choice(len(colors))]
    canvas1[yy, xx] = color
    canvas2[yy, xx] = color
    canvas3[yy, xx] = color

cv2.imwrite("out/result1_mser.png", canvas1)
cv2.imwrite("out/result2_mser.png", canvas2)
cv2.imwrite("out/result3_mser.png", canvas3)

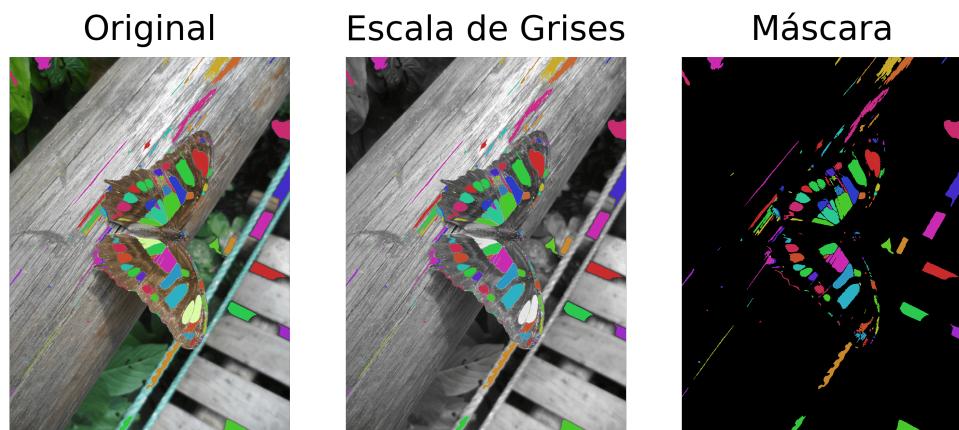
plt.subplot(131)
plt.imshow(canvas1[:, :, ::-1])
plt.title('Original')
plt.axis('off')

plt.subplot(132)
plt.imshow(canvas2[:, :, ::-1])
plt.title('Escala de Grises')
plt.axis('off')

plt.subplot(133)
plt.imshow(canvas3[:, :, ::-1])
plt.title('Máscara')
plt.axis('off')

plt.savefig("out/MSER_Result.png", dpi=600, transparent=True)
plt.show()
```

5. Lo cual producirá el siguiente resultado:



1.2. Detector FAST

En esta sección se utilizará el detector FAST². de OpenCV para la detección de *keypoints*. Para este detector el resultado o salida del método es un arreglo de valores que indican los *keypoints* identificados en la imagen.

1. Como primer paso crea un *script/notebook* y añade las importaciones a OpenCV, Numpy(np) y Pyplot(plt) de Matplotlib.
2. FAST puede recibir un conjunto de parámetros, los cuales están definidos de forma predefinida por el algoritmo y estos suelen funcionar de manera adecuada en la mayoría de los casos. Uno de los parámetros que es modificado de manera general es el *threshold*.
3. En nuestro ejemplo se modifica también la opción de supresión de máximos para evitar redundancia de puntos.

²FAST

4. El siguiente código aplica el detector FAST a una imagen y produce 2 salidas, con y sin supresión de máximos. Estas son almacenadas en disco.

```
imgFast = cv2.imread('img/tower2.png',0)

# Inicializar FAST con los valores por defecto, salvo el threshold

# Threshold
thre = 15
fast = cv2.FastFeatureDetector_create(thre)

# encontrar y dibujar los keypoints
kp = fast.detect(imgFast,None)
imgFast2 = cv2.drawKeypoints(imgFast, kp, None, color=(255,0,0))

# Imprimir los parámetros utilizados
print( "Threshold: {}".format(fast.getThreshold()) )
print( "nonmaxSuppression:{}".format(fast.getNonmaxSuppression()) )
print( "vecindario: {}".format(fast.getType()) )
print( "Keypoints Totales con nonmaxSuppression: {}".format(len(kp)) )
cv.imwrite('out/fast_true.png',imgFast2)

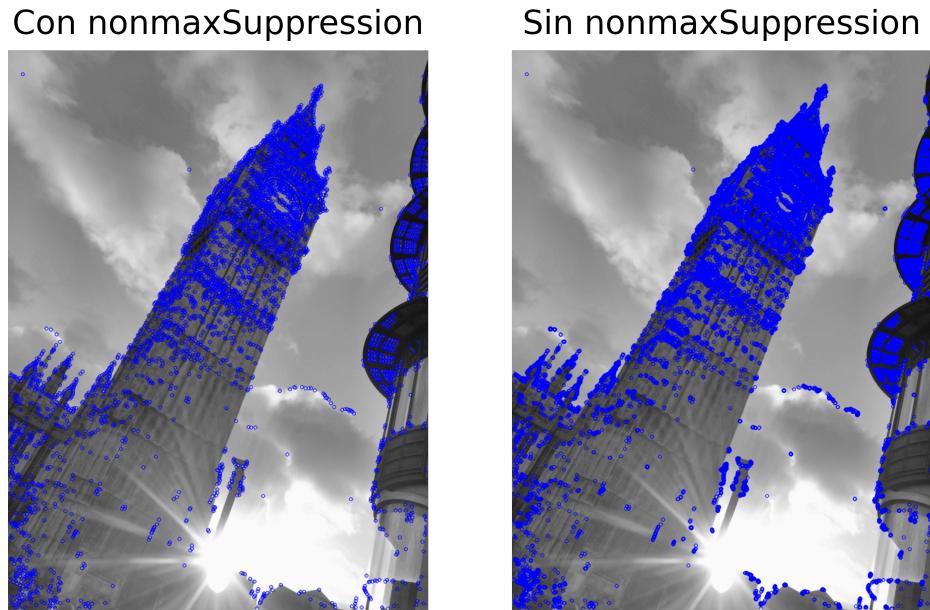
# Desactivar nonmaxSuppression
fast.setNonmaxSuppression(0)
kp = fast.detect(imgFast,None)
print( "Keypoints Totales sin nonMaxSuppression: {}".format(len(kp)) )

imgFast3 = cv2.drawKeypoints(imgFast, kp, None, color=(255,0,0))
cv.imwrite('out/fast_false.png',imgFast3)

plt.subplot(121)
plt.imshow(imgFast2[:, :, ::-1])
plt.title('Con nonMaxSuppression')
plt.axis('off')

plt.subplot(122)
plt.imshow(imgFast3[:, :, ::-1])
plt.title('Sin nonMaxSuppression')
plt.axis('off')

plt.show()
```



1.3. Detector ORB

ORB³ es un algoritmo detector y descriptor de *keypoints*. Consiste en una mejora realizada al algoritmo FAST. OpenCV permite utilizar las 2 partes del algoritmo de forma individual, por lo cual, podemos extraer los *keypoints* y también utilizar otro descriptor. ORB tiene parámetros predeterminados, el más comúnmente modificado por los usuarios es el *nfeatures* que determina la cantidad máxima de *keypoints* que devolverá el algoritmo.

1. Como primer paso crea un *script/notebook* y añade las importaciones a OpenCV, Numpy(np) y Pyplot(plt) de Matplotlib.
2. El siguiente código aplica el detector ORB a una imagen de entrada y almacena en disco una imagen donde marca los *keypoints* detectados.

```
imgORB = cv2.imread('img/tower2.png',0)

# Inicializar el Detector ORB
orb = cv2.ORB_create(nfeatures = 500)

# Encontrar los Keypoints
```

³ORB

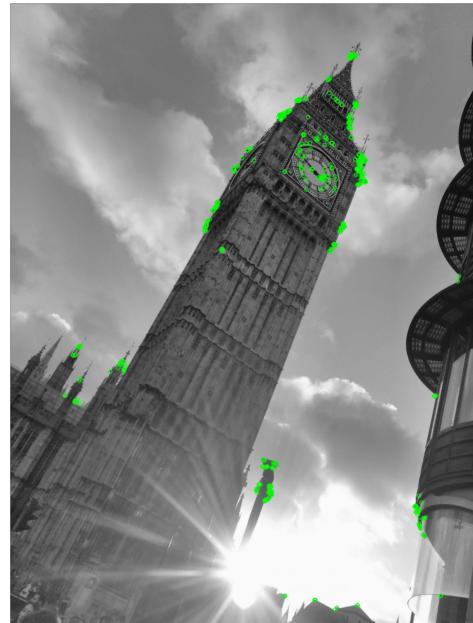
```
kp = orb.detect(imgORB,None)

# Pintar los Keypoints en la imagen original
imgORB2 = cv2.drawKeypoints(imgORB, kp, None, color=(0,255,0), flags=0)

print( "Keypoints Totales Usando ORB Detector : {}".format(len(kp)) )

plt.imshow(imgORB2[:, :, ::-1] )
plt.axis('off')
plt.savefig("out/ORB_KP.png", dpi=600, orientation='portrait', transparent=True)
plt.show()
```

-
3. Lo cual producirá el siguiente resultado:



Entregables para la Sección

Para esta sección en su informe debe incluir la modificación de una imagen de su propiedad. En dicha imagen se deben realizar las siguientes operaciones:

1. Seleccione una imagen para el detector MSER. Aplique el detector. Luego aplique un filtro bilateral a la imagen original. Posteriormente, aplique el detector a la imagen filtrada, grafique los resultados.
2. Para los detectores FAST y ORB, seleccione una imagen (debe ser la misma imagen para ambos detectores). Aplique los detectores, ajustando los parámetros para que se produzca una buena identificación en la imagen. Luego, aplique a la imagen original un giro de 90 utilizando la función:

```
ImageRot= cv2.rotate(ImageOrg, cv2.ROTATE_90_COUNTERCLOCKWISE)
```

La cual esta disponible en OpenCV. Seguidamente, aplique los algoritmos detectores utilizando los mismos parámetros que utilizó con las imágenes originales.

3. Presente gráficamente los resultados para cada detector señalando los puntos detectados **antes** y **después** de aplicar la modificación de la imagen original, se recomienda utilizar subplots de matplotlib.
4. Al final para esta sección deberá presentar un total de 3 pares de imágenes de **antes** y **después** en total, 1 para cada algoritmo detector presentado.

2. Descriptores

Los descriptores se encargan de analizar el vecindario de los *keypoints* y construir una representación numérica del mismo de tal manera que se pueda obtener información de imagen, pero solo de las zonas importantes de esta.

Existen diversos algoritmos para detección sin embargo no todos son accesibles mediante la versión estándar de OpenCV y en otros casos son algoritmos de pago. La mayoría de estos algoritmos integran un detector de *keypoints* en sus códigos, por lo cual se puede utilizar al extraer y describir puntos claves. De igual manera OpenCV permite su utilización de manera separada, es decir se puede usar solo el detector o el descriptor.

Desde el punto de vista de código para estos algoritmos OpenCV ha diseñado los mismos utilizando una misma plantilla, de tal manera que todos heredan características de una misma clase *Feature2D*, lo cual hace que su uso en código sea muy similar permitiendo cambiar fácilmente entre uno y otro algoritmo.

En esta parte de laboratorio nos enfocamos en utilizar algunos descriptores y visualizar los histogramas que producen cada uno de estos para algunos de los *keypoints* detectados.

2.1. Descriptor ORB

Este es un algoritmo desarrollado en los laboratorios de OpenCV y mejora el detector FAST y descriptor BRIEF. Para mayor información sobre este algoritmo se puede consultar el siguiente enlace⁴.

En este caso utilizaremos la siguiente combinación de algoritmos:

- Detector: **ORB**
 - Descriptor: **ORB**
1. Como primer paso crea un *script/notebook* y añade las importaciones a OpenCV, Numpy(np) y Pyplot(plt) de Matplotlib.
 2. El siguiente fragmento de código es una función que debemos declarar en nuestro script y permitirá generar un conjunto de gráficos de barras que representan el descriptor generado para algunas de los puntos clave identificados. Esta función se utilizará con todos los descriptores presentados en esta sección del laboratorio.

```
def show_img_ply(bins, val, pos):
    plt.subplot(3,4,pos)
    plt.bar(bins, val, width = 0.6, color='#0504aa',alpha=0.7)
    plt.xlim(min(bins), max(bins))
```

⁴ORB

3. Agregue el siguiente código, en este se crea una instancia del Detector ORB y se utiliza el método `detectAndCompute()` para detectar y describir los *keypoints* en un solo paso. Con este método el algoritmo detecta de manera inmediata los puntos claves y los describe. Es un método que se utiliza de igual manera para todo los algoritmos que sean detectores y descriptores disponibles en OpenCV.

```
imgORB = cv.imread('img/tower2.png')

# Inicializar Detector ORB
orb = cv2.ORB_create(nfeatures = 2500, edgeThreshold = 73, nlevels=18)

# Calcular Keypoints y Descriptores con ORB
kpORB, desORB = orb.detectAndCompute(imgORB, None)
```

4. El siguiente fragmento de código utiliza la función `show_img_ply()` para mostrar un histograma de los 12 primeros descriptores para la imagen en cuestión y guarda dicha representación en disco. Si lo desea puede modificar los histogramas que se visualizan.

```
#Preparar Salida
bins = list(range(desORB.shape[1]))
fig, axs = plt.subplots(3,4, figsize=(15,10), sharex='col', sharey='row')
fig.suptitle("ORB Detector -Descriptor.", fontsize=19, va='top')
for i in range(12):
    val= desORB[i]
    pos = i+1
    show_img_ply(bins, val, pos )
    if(i == 11):
        plt.savefig("out/ORB_Det_Desc_Histograma.png",
                    dpi=350, bbox_inches='tight')
plt.show()

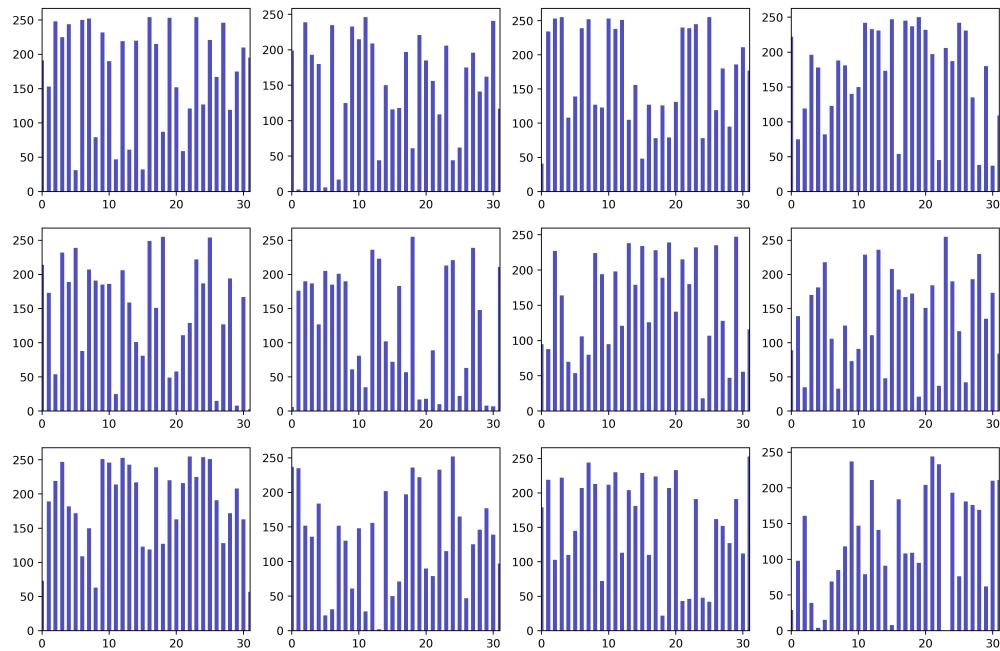
# Dibujar los Keypoints
imgORB2 = cv2.drawKeypoints(imgORB,kpORB,np.array([]),
                            color=(0,5,255), flags=0)
cv.imwrite("out/Tower_ORB.png", imgORB2)
plt.imshow(imgORB2[:, :, ::-1])
plt.show()
```

5. Lo cual producirá el siguiente resultado:

2.1 Descriptor ORB



ORB Detector -Descriptor.



2.2. Descriptor BRISK

Este es un algoritmo presentado en el 2011 por Stefan Leutenegger, Margarita Chli y Roland Y. Siegwart en el artículo *BRISK: Binary Robust invariant scalable keypoints*^[1]. Consiste al igual que ORB en un detector y descriptor, por lo cual su utilización es muy similar a este último. Para mayor información sobre este algoritmo se puede consultar el siguiente enlace⁵.

En este caso utilizaremos la siguiente combinación de algoritmos:

- Detector: **BRISK**
- Descriptor: **BRISK**

1. Como primer paso crea un *script/notebook* y añade las importaciones a OpenCV, Numpy(np) y Pyplot(plt) de Matplotlib. Agregue también la función `show_img_ply()` mostrada en el apartado anterior. De igual manera este código se puede colocar en el mismo *script/notebook* utilizado en el ejemplo anterior.
2. Agregue el siguiente código, en este se crea una instancia del Detector BRISK y se utiliza el método `detectAndCompute()` para detectar y describir los *keypoints* en un solo paso. Con este método el algoritmo detecta de manera inmediata los puntos claves y los describe. Es un método que se utiliza de igual manera para todo los algoritmos que sean detectores y descriptores disponibles en OpenCV.

```
imgBrisk = cv2.imread('img/tower2.png')

# Convertimos la imagen a escala de grises,
# puesto que es un prerequisito para usar descriptores binarios.
grayBrisk = cv2.cvtColor(imgBrisk, cv2.COLOR_BGR2GRAY)

# Instanciamos BRISK, el cual nos da los puntos
# clave así como los descriptores binarios.
detectorBrisk = cv2.BRISK_create()
keypointsBrisk, descriptorsBrisk = detectorBrisk.detectAndCompute(grayBrisk, None)

# Imprimimos el número de puntos clave hallados,
# así como las dimensiones del vector de features.
print(f'Número de puntos clave detectados.: {len(keypointsBrisk)})')
print(f'Dimensiones del vector de features: {descriptorsBrisk.shape}')
```

3. El siguiente fragmento de código utiliza la función `show_img_ply()` para mostrar un histograma de los 12 primeros descriptores para la imagen en cuestión y guarda dicha representación en disco. Si lo desea puede modificar los histogramas que se visualizan.

⁵BRISK

```

#Preparar Salida
bins = list(range(descriptorsBrisk.shape[1]))
fig, axs = plt.subplots(3,4, figsize=(15,10), sharex='col', sharey='row')
fig.suptitle("BRISK Detec BRISK Desc.", fontsize=19, va='top')
for i in range(12):
    val= descriptorsBrisk[i]
    pos = i+1
    show_img_ply(bins, val, pos )
    if(i == 11):
        plt.savefig("out/BRISK_Det_BRISK_Des_Histograma.png",
                    dpi=350, bbox_inches='tight')
plt.show()

# Dibujar los Keypoints
imgBrisk2 = cv2.drawKeypoints(imgBrisk, keypointsBrisk,np.array([]),
                             color=(0,5,255), flags=0)
cv2.imwrite("out/Tower_BRISK.png", imgBrisk2)
plt.imshow(imgBrisk2[:, :, ::-1])
plt.show()

```

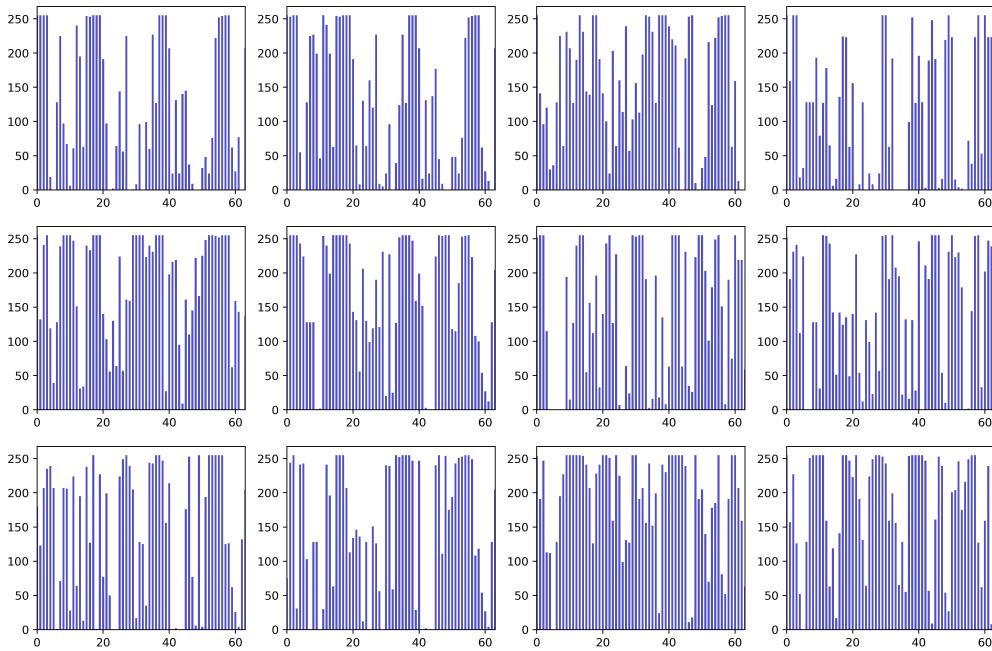
4. Lo cual producirá el siguiente resultado:



2.3 Descriptor KAZE

Visión Artificial / Laboratorio 3

BRISK Detec BRISK Desc.



2.3. Descriptor KAZE

Este es un algoritmo presentado en el 2012 por Pablo Fernández Alcantarilla, Adrien Bartoli y Andrew J. Davison en el artículo *KAZE Features*^[2]. Consiste al igual que ORB en un detector y descriptor, por lo cual su utilización es muy similar a este último. Para mayor información sobre este algoritmo se puede consultar el siguiente enlace⁶.

En este caso utilizaremos la siguiente combinación de algoritmos:

- Detector: **KAZE**
- Descriptor: **KAZE**

1. Como primer paso crea un *script/notebook* y añade las importaciones a OpenCV, Numpy(np) y Pyplot(plt) de Matplotlib. Agregue también la función `show_img_ply()` mostrada en el apartado anterior. De igual manera este código se puede colocar en el mismo *script/notebook* utilizado en el ejemplo anterior.
2. Agregue el siguiente código, en este se crea una instancia del Detector KAZE y se utiliza el método `detectAndCompute()` para detectar y describir los *keypoints* en un solo paso. Con este método el algoritmo detecta de manera inmediata los puntos claves y los describe. Es

⁶[KAZE](#)

un método que se utiliza de igual manera para todo los algoritmos que sean detectores y descriptores disponibles en OpenCV.

```
# Cargamos una imagen de prueba y la mostramos en pantalla.
imgKaze = cv2.imread('img/tower2.png')

# Convertimos la imagen a escala de grises,
# puesto que es un prerequisito para usar descriptores binarios.
grayKaze = cv2.cvtColor(imgKaze, cv2.COLOR_BGR2GRAY)

# Instanciamos KAZE, el cual nos da los puntos
# clave así como los descriptores binarios.
detectorKaze = cv2.KAZE_create()
keypointsKaze, descriptorsKaze = detectorKaze.detectAndCompute(grayKaze, None)

# Imprimimos el número de puntos clave hallados,
# así como las dimensiones del vector de features.
print(f'Número de puntos clave detectados.: {len(keypointsKaze)}')
print(f'Dimensiones del vector de features: {descriptorsKaze.shape}'')
```

3. El siguiente fragmento de código utiliza la función `show_img_ply()` para mostrar un histograma de los 12 primeros descriptores para la imagen en cuestión y guarda dicha representación en disco. Si lo desea puede modificar los histogramas que se visualizan.

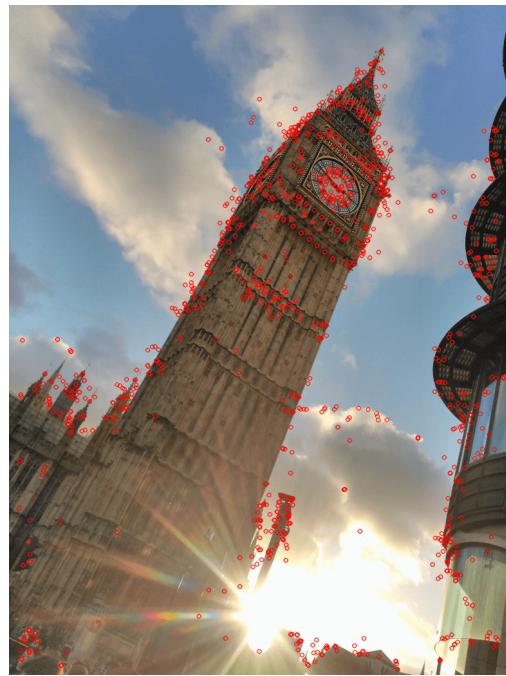
```
#Preparar Salida
bins = list(range(descriptorsKaze.shape[1]))
fig, axs = plt.subplots(3,4, figsize=(15,10), sharex='col', sharey='row')
fig.suptitle("KAZE Detec KAZE Desc.", fontsize=19, va='top')
for i in range(12):
    val= descriptorsKaze[i]
    pos = i+1
    show_img_ply(bins, val, pos )
    if(i == 11):
        plt.savefig("out/KAZE_Det_KAZE_Des_Histograma.png",
                    dpi=350, bbox_inches='tight')
plt.show()

# Dibujar los Keypoints
imgKaze2 = cv2.drawKeypoints(imgKaze,keypointsKaze,np.array([]),
                            color=(0,5,255), flags=0)
cv2.imwrite("out/Tower_KAZE.png", imgKaze2)
plt.imshow(imgKaze2[:, :, ::-1])
plt.show()
```

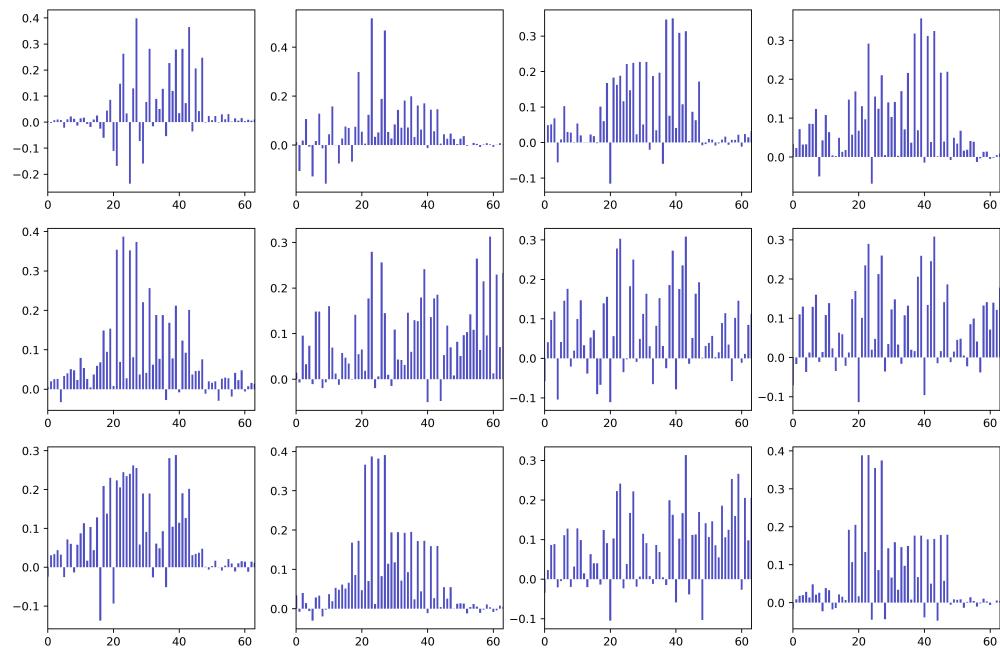
4. Lo cual producirá el siguiente resultado:

2.3 Descriptor KAZE

Visión Artificial / Laboratorio 3



KAZE Detec KAZE Desc.



2.4. Combinado Detectores y Descriptores

En esta ocasión sacamos provecho de una capacidad de OpenCV que permite utilizar un descriptor diferente al algoritmo detector empleado en la imagen. Para este ejemplos usaremos la siguiente configuración.

- Detector: **FAST**
 - Descriptor: **ORB**
1. Como primer paso crea un *script/notebook* y añade las importaciones a OpenCV, Numpy(np) y Pyplot(plt) de Matplotlib. Agregue también la función `show_img_ply()` mostrada en el apartado anterior. De igual manera este código se puede colocar en el mismo *script/notebook* utilizado en el ejemplo anterior.
 2. Agregue el siguiente código, en este se crea una instancia del Detector FAST y otra instancia del descriptor ORB. Se emplea el método `detect()` de FAST para encontrar los *keypoints* y la salida de este se pasa al método `compute()` de ORB para describir los *keypoints*. Este procedimiento es similar para la mayoría de las combinaciones de detectores y descriptores de OpenCV.
-
- ```
imgFast = cv2.imread('img/tower2.png',0)

Threshold
thre = 15
fast = cv2.FastFeatureDetector_create(thre)

encontrar y dibujar los keypoints
kpFAST = fast.detect(imgFast,None)

Inicializar Detector ORB
orb = cv2.ORB_create()

Calcular Descriptores con ORB
kpFAST, desORB = orb.compute(imgFast, kpFAST)
```
- 

3. El siguiente fragmento de código utiliza la función `show_img_ply()` para mostrar un histograma de los 12 primeros descriptores para la imagen en cuestión y guarda dicha representación en disco. Si lo desea puede modificar los histogramas que se visualizan.
- 

```
#Preparar Salida
bins = list(range(desORB.shape[1]))
fig, axs = plt.subplots(3,4, figsize=(15,10), sharex='col', sharey='row')
fig.suptitle("FAST Detec ORB Desc.", fontsize=19, va='top')
for i in range(12):
 val= desORB[i]
```

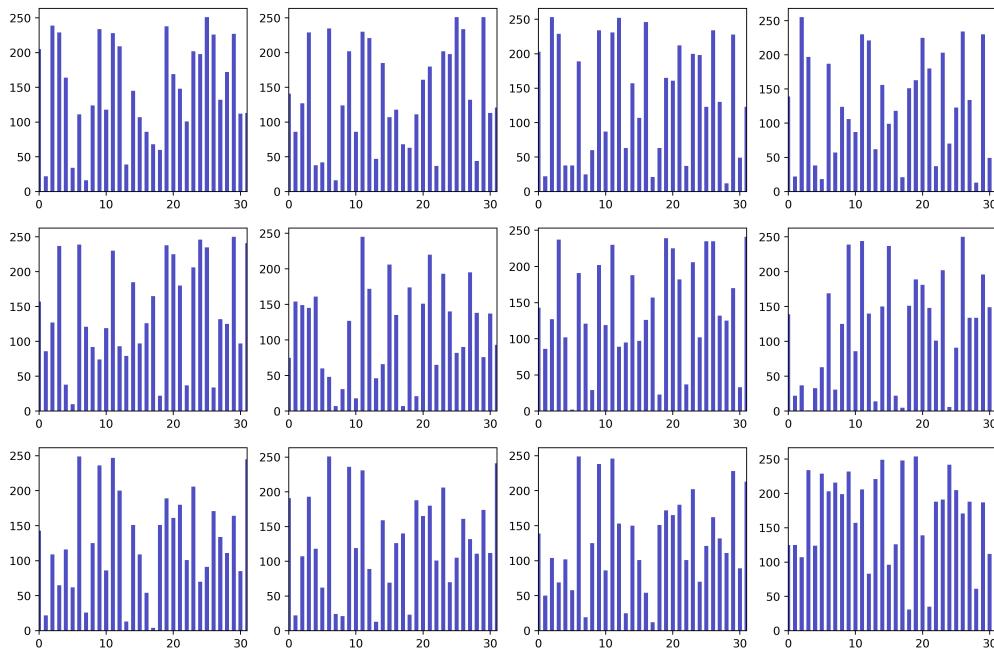
```
pos = i+1
show_img_ply(bins, val, pos)
if(i == 11):
 plt.savefig("out/FAST_Det_ORB_Des_Histograma.png",
 dpi=350, bbox_inches='tight')
plt.show()

Dibujar los Keypoints
imgFAST2 = cv2.drawKeypoints(imgFast,kpFAST,np.array([]),
 color=(0,5,255), flags=0)
cv2.imwrite("out/Tower_FAST.png", imgFAST2)
plt.imshow(imgFAST2[:, :, ::-1])
plt.show()
```

4. Lo cual producirá el siguiente resultado:



FAST Detec ORB Desc.



### Entregables para la Sección

Para esta sección en su informe debe incluir la modificación de una imagen de su propiedad. En dicha imagen se deben realizar las siguientes operaciones:

1. Seleccione una imagen de su propiedad, extraiga los *keypoints* utilizando el algoritmo FAST y describa estos puntos claves con el descriptor BRISK. Presente los histogramas para los 16 primeros *keypoints* utilizando `matplotlib`. Muestre también los *keypoints* detectados en la imagen original.
2. Seleccione una imagen de su propiedad, extraiga los *keypoints* utilizando el algoritmo STAR<sup>a</sup>, investigue como utilizar este detector y describa estos puntos claves con el descriptor BRIEF. Presente los histogramas para los 16 primeros *keypoints* utilizando `matplotlib`. Muestre también los *keypoints* detectados en la imagen original.
3. Al final de esta sección tendrá 1 par de imágenes donde una representa la imagen con los *keypoints* identificados y la otra una figura con los 16 histogramas de descriptores para la misma imagen.

<sup>a</sup>STAR

## Referencias

- [1] S. Leutenegger, M. Chli, and R. Y. Siegwart, "Brisk: Binary robust invariant scalable keypoints," in *2011 International Conference on Computer Vision*, 2011, pp. 2548–2555.
- [2] P. F. Alcantarilla, A. Bartoli, and A. J. Davison, "Kaze features," in *Computer Vision – ECCV 2012*, A. Fitzgibbon, S. Lazebnik, P. Perona, Y. Sato, and C. Schmid, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 214–227.