CAPÍTULO 6.
Procesamiento de Imágenes

Como capítulo final de este libro, trabajaremos con señales en dos dimensiones, específicamente con imágenes. Esto te permitirá obtener las bases conceptuales y de programación para abordar cursos más avanzados en procesamiento de imagen, por ejemplo, de visión por computador.

Al finalizar el capítulo, deberás estar en capacidad de:

1. Explicar las diferencias entre imágenes blanco-negro, escala de grises, e imágenes a color.
2. Explicar en qué consisten los modelos de color RGB y HSV, así como realizar conversiones utilizando la librería de OpenCV de Python.
3. Realizar ecualización de imagen utilizando la librería de OpenCV de Python.
4. Distinguir diferentes tipos de ruido en imágenes.
5. Reconocer qué tipo de filtro es adecuado para reducir un tipo de ruido específico en la imagen.
6. Explicar el concepto de convolución en imágenes.
7. Realizar detección de bordes a partir de diferentes tipos de kernels.
8. Explicar las diferencias entre DCT y DFT en imágenes.
9. Aplicar la DCT o la DFT en imágenes utilizando la librería de OpenCV de Python.
10. Explicar el concepto de compresión de imágenes.

6.1. CONCEPTOS BÁSICOS DE IMÁGENES

En las primeras secciones del libro hemos trabajado con señales uni-dimensionales, y gran parte de los ejemplos se han enfocado en audio. En este capítulo, nos enfocaremos en imágenes, que corresponden a señales en 2D, cuyos ejes corresponden a filas y columnas.

Lo primero que debemos saber es que no todas las imágenes tienen las mismas características. Por ejemplo, pueden variar entre ellas el tamaño y el color utilizado.

En relación con el tamaño, la unidad de medida es el píxel, y la resolución de la imagen está dada por la cantidad de filas y columnas. Entonces, una imagen de 100 x 100 tendrá 10,000 píxeles de resolución, mientras que, una imagen de 1,000 x 1,000 tendrá 1M píxeles de resolución. En las cámaras digitales actuales es típico encontrar resoluciones de varios mega píxeles. Entonces, hemos identificado la primera diferencia entre las señales 1D correspondientes a audio y las imágenes, en el primer caso hablábamos de muestras de la señal, y ahora hablaremos de píxeles de la imagen.

La segunda característica de la imagen corresponde a su color. Podemos encontrar imágenes a blanco-negro (BW: *black and white*), a escala de grises y a color de tres bandas (aunque también existen imágenes con mayor número de canales, las cuales no abordaremos en este libro).

Las primeras, BW, tienen solamente un bit asociado a cada píxel de la imagen, de tal forma que, si la imagen tiene 1M píxeles, entonces tendrá 1M bits. El valor de "1" corresponde al blanco, mientras que, el valor de "0" corresponde al negro. Las segundas, imágenes a escala de grises, tienen 8 bits por cada píxel, y el rango de color va del negro ("00000000") al blanco ("11111111") pasando por distintas tonalidades de gris, para un total de 256 colores. Entonces, una imagen de 1M píxeles tendrá 8Mbits, o de forma equivalente 1MB. Finalmente, tenemos las imágenes a color, que típicamente se denominan RGB (Red, Green, Blue), aunque realmente ese es uno de los espacios de color que existen. En este caso, por cada píxel de la imagen tenemos 8 bits asociados a cada uno de los tres canales de color, para un total de 24 bits por píxel. Retomando el mismo ejemplo, para la imagen de 1M píxeles, tendremos 3MB.

Para ilustrar de mejor forma las diferencias en términos de color de las imágenes BW, a escala de grises y a color, se presenta en la Figura 70 una imagen del repositorio personal de los autores del libro.

Figura 70. Ejemplo de imagen: a) BW, b) Escala de grises, c) Color. Fuente: repositorio personal de los autores.

6.2. ESPACIOS DE COLOR

El espacio de color más ampliamente conocido se denomina RGB, donde la imagen se representa en tres canales o "bandas" de color, una correspondiente al rojo, otra al verde, y la última al azul. Cada color tiene 256 tonalidades distintas (2^8), y en total se tiene una paleta de16,776,216 colores (es decir, 2^{24}). Retomando el ejemplo de la sección anterior, se presentan las tres bandas de color en la Figura 71.

(a) (b) (c)

Figura 71. Ejemplo de imagen RGB: a) banda R, b) banda G, c) banda B. Fuente: repositorio personal de los autores.

(a) (b) (c)

Figura 72. Ejemplo de imagen HSB: a) banda H, b) banda S, c) banda B. Fuente: repositorio personal de los autores.

Otro espacio de color corresponde a HSV (*Hue, Saturation,Value*) o también conocido como HSB (*Hue, Saturation, Brightness*). En este espacio de color, la primera banda corresponde al tono de la imagen, la segunda a la saturación de la imagen, y la tercera al *brillo de la imagen*. Para nuestra foto de la playa, las tres bandas se presentan en la Figura 72. En este espacio de color, la banda de brillo (Figura 72b) es muy similar a la imagen a escala de grises (que presentamos en la Figura 70b).

6.3. INTRODUCCIÓN A LA LIBRERÍA OPENCV

Bueno, en este punto te preguntarás como se puede leer la imagen en lenguaje Python, convertir una imagen a color en una imagen a escala de grises y/o BW, así como transformar una imagen de un espacio a color a otro. Para ello, vamos a utilizar la librería OpenCV de Python, la cual es especializada en procesamiento de imágenes[4].

4 https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_tutorials.html

Figura 73. Logo de OpenCV.

Entonces, manos a la obra con el código en Python.

Paso 1: importar la librería de OpenCV, leer la imagen que previamente hemos subido a nuestro entorno de trabajo en Colaboratory, y conocer el tamaño de la imagen.

```
import cv2
img = cv2.imread("/content/Fig74.jpg")
img.shape
```

Para la imagen de prueba, el resultado es:

```
(300, 400, 3)
```

Paso 2: visualización de la imagen. Para ello se debe importar un patch en Colaboratory.

```
from google.colab.patches import cv2_imshow
cv2_imshow(img)
```

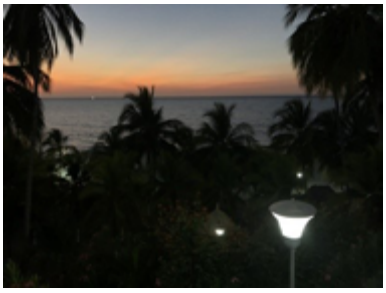


Figura 74. Imagen a color – foto playa.

Nota: si trabajas en Jupyter Notebook no es necesario que importes el patch, y puedes utilizar cv2.imshow.

Paso 3: conversión de imagen RGB a escala de grises

```
img_gray=cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
cv2_imshow(img_gray)
```

Figura 75. Imagen a escala de grises – foto playa.

Paso 4: conversión de imagen a escala de grises en BW

```
r,img_bw = cv2.threshold(img_gray,45,255,cv2.THRESH_BINARY)
cv2_imshow(img_bw)
```

Figura 76. Imagen a blanco y negro – foto playa.

Lo que hemos realizado en este paso 4 se conoce como *umbralización* de la imagen (o *thresholding*, en inglés), proceso en el cual a los píxeles que superan el umbral se les asigna el color blanco, y a los que no superan el umbral se les asigna el color negro. Si modificamos el valor del umbral, la imagen va a lucir más clara (umbral bajo) o más oscura (umbral alto). Podemos apreciar que las palmeras tienen el color negro, mientras que el mar y el cielo el color blanco, dado que, en la imagen a escala de grises la tonalidad de gris tanto del cielo como del mar es mucho más clara que la de las palmeras.

La instrucción `cv2.threshold`[5] requiere de dos valores numéricos, el primero corresponde al umbral, y el segundo al valor que se asigna en caso de que el píxel supere el umbral. En el ejemplo, el umbral es 45 y el valor asignado a los píxeles que superen el umbral es 255.

5 https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_thresholding/py_thresholding.html

Paso 5: guardar las imágenes en tu entorno de trabajo

```
cv2.imwrite('image_color.jpg',img)
cv2.imwrite('image_gray.jpg',img_gray)
cv2.imwrite('image_bw.jpg',img_bw)
```

Paso 6: conversión de RGB a HSV

```
H, S, V = cv2.split(cv2.cvtColor(img, cv2.COLOR_RGB2HSV))
cv2_imshow(H)
```

Figura 77. Imagen canal H – foto playa.

```
cv2_imshow(S)
```

Figura 78. Imagen canal S – foto playa.

```
cv2_imshow(V)
```

Figura 79. Imagen canal V – foto playa.

Hasta este punto, ya sabemos cómo leer imágenes con la librería OpenCV, convertir una imagen a color en una imagen a escala de grises y BW, y convertir del espacio de color RGB a HSV. Puedes ampliar la información de conversión de espacios de color en la documentación de OpenCV de `cv2.cvtColor`[6].

6.4. ECUALIZACIÓN DE IMÁGENES

¿Alguna vez te ha pasado que tomas una foto con poca luz y la imagen te quedó muy oscura? ¿Sabes cómo funcionan los ajustes de brillo en los celulares, por ejemplo, para aclarar fotos oscuras? ¡Eso lo aprenderás en esta sección y adiós a borrar fotos porque quedaron muy oscuras!

Lo primero que debemos conocer es el concepto de histograma de una imagen y como calcularlo y graficarlo en lenguaje Python. Pues bueno, la definición general del histograma es que es una representación gráfica de la ocurrencia de los datos. En el caso de imágenes, el histograma muestra cuántos píxeles de la imagen tienen color 0, cuántos tienen color 1, y así sucesivamente hasta cuantos píxeles tienen color 255 (en imágenes a escala de grises). En el caso de imágenes a color, presentará la cantidad de píxeles para cada uno de los 256 niveles de color por banda, es decir, es necesario dibujar tres histogramas, uno para la banda R, otro para la banda G, y otro para la banda B. Si la imagen es BW, entonces el histograma solamente tendrá dos niveles de color, el 0 correspondiente al negro, y el 1 correspondiente al blanco.

A continuación, se presentan los pasos.

Paso 1: Lectura de la imagen a color

```
import cv2
img = cv2.imread("/content/Fig80.jpg")
```

Y obtengo esta hermosa imagen. Si, ya sé que está un poco oscura, pero más adelante aprenderemos a aclararla.

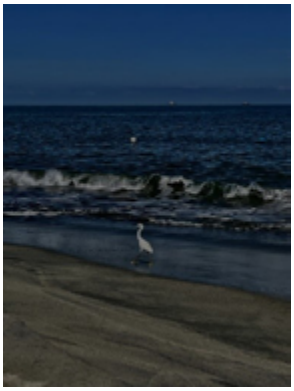


Figura 80. Imagen a color – foto mar. Fuente: repositorio personal de los autores.

6 https://opencv24-python-tutorials.readthedocs.io/en/stable/py_tutorials/py_imgproc/py_colorspaces/py_colorspaces.html

Por ahora, vamos a conocer el tamaño de la imagen con `img.shape`. El resultado es (4032, 3024, 3. Es decir que, nuestra imagen tiene 4032 filas, 3024 columnas y 3 bandas de color (por defecto en el espacio BGR). El total de píxeles de la imagen es 4032 x 3024, que es igual a 12,192,768. En términos de bytes, el total se calcula así: 4032 x 3024 x 3, que es igual a 36,578,304, dado que en cada banda de color un píxel tiene 1B, y la imagen tiene tres bandas de color. Ahora, te preguntarás si ese tamaño que acabamos de encontrar es el mismo que te aparece en tu PC en relación con esa imagen, y si revisas te darás cuenta de que solo pesa 1,54 KB. La diferencia entre el cálculo que acabamos de realizar y el peso real de la imagen radica en su tipo de formato, (en este caso es *.jpg), el cual es un formato de compresión de imágenes que reduce su peso, pero conserva su resolución espacial. Si la imagen estuviese en formato bmp de 24 bits, el espacio en disco sería el calculado previamente (alrededor de 36 MB).

Para facilitar la visualización de la imagen en Colaboratory, vamos a realizar un proceso de redimensionamiento de la imagen, para que quede de tamaño 400 filas y 300 columnas, para ello utilizaremos el siguiente código:

Paso 2: Redimensionamiento de la imagen

```
img=cv2.resize(img, (300, 400), interpolation = cv2.INTER_AREA)
```

Ten en cuenta que primero incluimos la cantidad de columnas que deseamos que la imagen tenga, y después la cantidad de filas. Entonces, la cantidad de píxeles por banda es ahora de 400 x 300, que es igual a 120,000. El siguiente paso, es convertir la imagen a escala de grises.

Paso 3: Conversión imagen a color en escala de grises

```
img_gray=cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
cv2_imshow(img_gray)
```

Obteniendo esta imagen:

Figura 81. Imagen a escala de grises – foto mar.

Bueno, ahora sí, vamos a dibujar nuestro histograma y a mejorar la apariencia de la imagen.

Paso 4: Histograma de la imagen a escala de grises

```
import matplotlib.pyplot as plt
pixels=img_gray.shape[0]*img_gray.shape[1]
print('la cantidad de píxeles de la imagen es:', pixels)
hist = cv2.calcHist([img_gray],[0],None,[256],[0,256])
plt.plot(hist)
plt.show()
```

Obteniendo el siguiente resultado:

la cantidad de píxeles de la imagen es: 120000

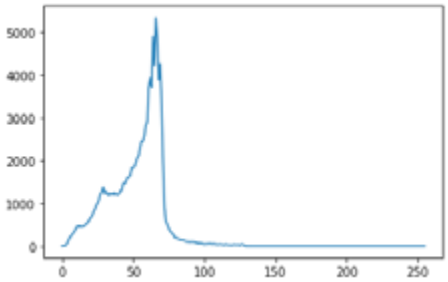


Figura 82. Histograma de la imagen a escala de grises – foto mar.

A partir del histograma se identifica que la imagen está altamente concentrada en intensidades de píxel alrededor de 60 (en escala 0 a 255), y que existen muy pocos píxeles con intensidades superiores a 128 (mitad de escala). Esto es coherente con la "apariencia oscura" de la imagen.

A continuación, mejoraremos la apariencia de la imagen a escala de grises.

Paso 5: Ecualización del histograma de la imagen a escala de grises

```
img_gray_eq = cv2.equalizeHist(img_gray)
cv2_imshow(img_gray_eq)
```

Como resultamos, obtenemos:

Figura 83. Imagen ecualizada a escala de grises – foto mar.

Si comparas esta imagen con la imagen a escala de grises original (*Paso 3*), notarás una gran diferencia. Es más clara. ¿Cómo crees entonces que es el histograma de la imagen ecualizada?

Paso 6: Histograma de la imagen a escala de grises ecualizada

```
hist2 = cv2.calcHist([img_gray_eq],[0],None,[256],[0,256])
plt.plot(hist2)
```

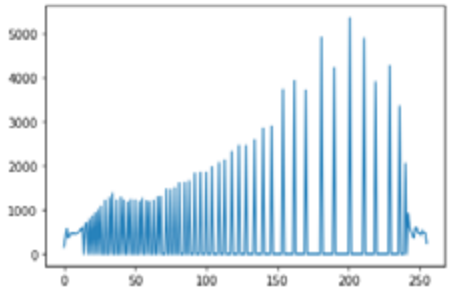


Figura 84. Histograma de la imagen ecualizada a escala de grises – foto mar.

Este histograma es significativamente diferente al obtenido en el *Paso 4*. Ahora, una gran parte de los píxeles de la imagen tienen niveles de color mayores a 128, y, por lo tanto, la imagen tiene una apariencia clara. Por otro lado, es típico en los histogramas ecualizados que se tengan numerosos picos de ocurrencia, y que no se tengan curvas suavizadas como en los histogramas de imágenes naturales (sin ecualizar).

A continuación, dibujaremos el histograma por banda de color y ecualizaremos la imagen a color.

Paso 7: Histograma de la imagen a color (histograma por cada banda de color)

```
img_RGB=cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
color = ('r','g','b')
for i,col in enumerate(color):
    histr = cv2.calcHist([img_RGB],[i],None,[256],[0,256])
    plt.plot(histr,color = col)
    plt.xlim([0,256])
plt.show()
```

En este punto es pertinente explicar que cuando leemos imágenes con OpenCV, las bandas de color quedan en orden contrario al del espacio RGB. Es decir, primero la banda B (azul), después la banda G (verde), y finalmente, la banda R (roja). Por ello, se hace necesario convertir de BGR a RGB, y posteriormente dibujar el histograma de cada una de las bandas (se puede realizar en gráficas independientes, o en la misma gráfica como con este código).

El histograma que obtenemos es:

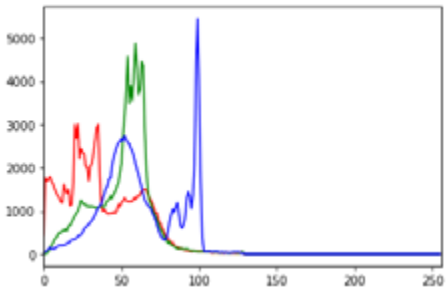


Figura 85. Histograma por banda de la imagen a color – foto mar.

Cada histograma se ha dibujado con el color correspondiente a su banda. El histograma de la banda roja tiene la mayor parte de sus píxeles por debajo del color 50. El histograma de la banda verde tiene la mayor parte de sus píxeles con color cercano a 50. Mientras que, el histograma de la banda azul tiene dos zonas de color que sobresalen, alrededor de 50 y alrededor de 100, esta última con mayor cantidad de píxeles. Aunque los histogramas son diferentes entre sí, tienen en común que en los tres casos la cantidad de píxeles por encima del color 128 es prácticamente cero.

Paso 8: Ecualización del histograma de la imagen a color

```
H, S, V = cv2.split(cv2.cvtColor(img, cv2.COLOR_RGB2HSV))
V_equ = cv2.equalizeHist(V)
img_equ = cv2.cvtColor(cv2.merge([H, S, V_equ]), cv2.COLOR_HSV2RGB)
cv2_imshow(img_equ)
```

El proceso de ecualización de la imagen lo realizaremos en la banda V del espacio de color HSV. Para lo cual, primero convertiremos la imagen de RGB a HSV, posteriormente realizaremos un split en las tres bandas, para así ecualizar únicamente la banda V. Finalmente, volvemos a unir las tres bandas del espacio HSV (con la banda V ecualizada), y convertimos de HSV a RGB. La imagen a color ecualizada es:

Figura 86. Imagen ecualizada a color – foto mar.

Mejoró con relación a la imagen del Paso 1, ¿cierto? Bueno, ya has aprendido un concepto de procesamiento de imágenes que tiene una aplicación práctica. Cuando vuelvas a cambiar el brillo de una imagen, recuerda que lo que estás haciendo es un proceso de ecualización de su histograma.

Espero que te haya gustado esta temática. Si quieres ampliar la información de histogramas en OpenCV, te invito a consultar la documentación de la librería[7].

6.5. RUIDO EN IMÁGENES

En esta sección aprenderemos a reconocer tres tipos diferentes de ruido presentes en imágenes: guassiano (gaussian), uniforme (uniform), y sal y pimienta (salt and pepper).

6.5.1. Ruido gaussiano:

Este ruido se caracteriza porque su distribución (histograma) tiene la forma de una campana de gauss, en la que existe un valor central (con gran parte de los píxeles del ruido), y pocos píxeles en los colores extremos. La forma y comportamiento está definida por el promedio y la varianza. Si la varianza es baja, la campana de gauss es angosta; mientras que, si la varianza es alta, la campana de gauss es ancha. El promedio es el valor central de la campana.

Vamos ahora a generar este tipo de ruido para adicionarlo a una imagen a color y visualizar su efecto. Para ello utilizaremos el siguiente código en Python:

```
import cv2
import numpy as np
from google.colab.patches import cv2_imshow
img = cv2.imread("/content/Fig89.jpg")
noise = np.zeros((img.shape[0], img.shape[1]),dtype=np.uint8)
gaussian_noise = np.zeros((img.shape[0], img.shape[1], img.sha-
pe[2]),dtype=np.uint8)
gaussian_noise[:,:,0]=cv2.randn(noise, 128, 30)
gaussian_noise[:,:,1]=cv2.randn(noise, 128, 30)
gaussian_noise[:,:,2]=cv2.randn(noise, 128, 30)
cv2_imshow(gaussian_noise)
```

Lo primero que hacemos es crear una matriz de ceros del mismo tamaño de la imagen a la cual le adicionaremos el ruido. Posteriormente, con la instrucción `cv2.randn`[8] vamos a generar ruido gaussiano. Debemos seleccionar el valor central de la distribución gaussiana (μ), y la desviación estándar (σ); para nuestro caso $\mu = 128$, y $\sigma = 30$. Este ruido gaussiano lo creamos para cada una de las bandas a color (banda 0, banda 1 y banda 2, de gaussian_noise). El resultado se presenta a continuación:

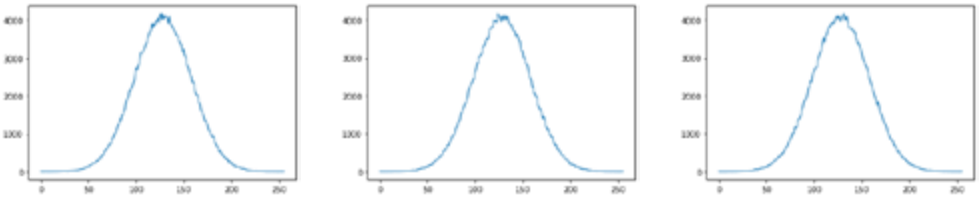
7 https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_histograms/py_table_of_contents_histograms/py_table_of_contents_histograms.html#table-of-content-histograms

8 https://docs.opencv.org/4.5.3/d2/de8/group__core__array.html#gaeff1f61e972d133a04ce3a5f81cf6808

Figura 87. Imagen a color – ruido gaussiano.

Para verificar que el ruido obtenido es de tipo gaussiano, utilizamos el siguiente código:

```
import matplotlib.pyplot as plt
# repetir este paso por canal
hist = cv2.calcHist([gaussian_noise],[0],None,[256],[0,256])
plt.plot(hist)
plt.show()
```



(a) (b) (c)

Figura 88. Histograma por banda de la imagen a color – ruido gaussiano.

Se verifica que los histogramas de la Figura 88 de cada uno de los canales, efectivamente tienen forma de campana de gauss.

Como siguiente paso, leemos una imagen en Colaboratory[9]:

```
img = cv2.imread("/content/Fig89.jpg")
img.shape
img=cv2.resize(img, (640, 480), interpolation = cv2.INTER_AREA)
cv2_imshow(img)
```

Figura 89. Imagen a color – villa de leyva. Fuente: repositorio personal de los autores.

9 Esta imagen hace parte del repositorio personal del autor de este libro

Y adicionamos el ruido que previamente hemos creado, así:

```
noisy_img_gn = cv2.add(img, (gaussian_noise*0.5).astype(np.uint8))
cv2_imshow(noisy_img_gn)
```

El ruido gaussiano se multiplica por 0.5 para no saturar a la imagen, y se convierte en formato entero de 8 bits con `astype(np.uint8)`. Posteriormente, se adiciona a la imagen a color con la instrucción `cv2.add`, obteniendo el siguiente resultado:

Figura 90. Imagen a color con ruido gaussiano – villa de leyva.

¿Cuál es el efecto de este tipo de ruido en la imagen?

Rta: La foto luce "envejecida".

6.5.2. Ruido uniforme:

Otro ruido típico en imágenes es el ruido uniforme. A diferencia del ruido anterior, este tiene una distribución *uniforme* de sus colores, es decir que no existe un color central, sino que todos los colores (o tonos) tienen la misma cantidad de píxeles (o aproximadamente la misma cantidad).

El procedimiento para crear este tipo de ruido es similar al caso anterior. Debemos crear una matriz de ceros del mismo tamaño de la imagen, y posteriormente para cada una de las bandas de color creamos el ruido. Sólo que en este caso utilizamos la instrucción `cv2.randu`, en lugar de `cv2.randn`. Podemos utilizar el siguiente código en Python:

```
noise = np.zeros((img.shape[0], img.shape[1]),dtype=np.uint8)
uniform_noise = np.zeros((img.shape[0], img.shape[1], img.sha-
pe[2]),dtype=np.uint8)
uniform_noise[:,:,0]=cv2.randu(noise, 0, 256)
uniform_noise[:,:,1]=cv2.randu(noise, 0, 256)
uniform_noise[:,:,2]=cv2.randu(noise, 0, 256)
cv2_imshow(uniform_noise)
```

Obteniendo el siguiente resultado:

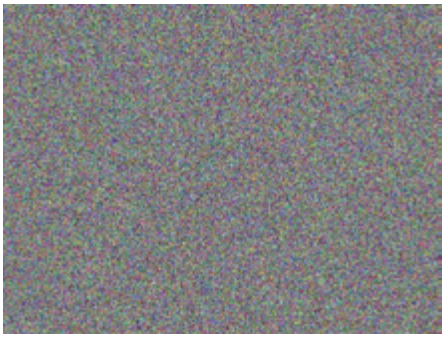
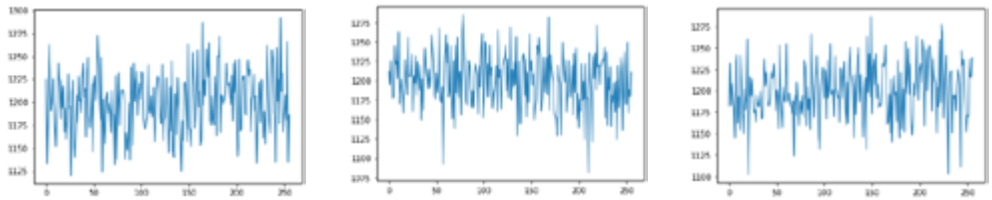


Figura 91. Imagen a color – ruido uniforme.

Como siguiente paso dibujamos el histograma, banda a banda, así:

```
# repetir este paso por canal
hist = cv2.calcHist([uniform_noise],[0],None,[256],[0,256])
plt.plot(hist)
plt.show()
```

Obteniendo:



(a) (b) (c)

Figura 92. Histograma por banda de la imagen a color – ruido uniforme.

Aunque el histograma que obtenemos no tiene una *perfecta distribución uniforme*, la cantidad de píxeles para cada uno de los tonos (0 a 255) es muy similar entre sí. Para el tamaño de imagen que hemos utilizado en este ejemplo, la cantidad de píxeles por canal es 480 x 640 = 307.200. Esta cantidad de píxeles dividido en los 256 tonos es igual a 1.200. Si se revisa con detalle los histogramas, precisamente las ocurrencias oscilan alrededor de ese valor.

Finalmente, adicionamos el ruido a la imagen a color, con el siguiente código en Python:

```
noisy_img_un = cv2.add(img, (uniform_noise*0.5).astype(np.uint8))
cv2_imshow(noisy_img_un)
```

Cuyo resultado es:

Figura 93. Imagen a color con ruido uniforme – villa de leyva.

¿En qué se diferencia esta imagen de la obtenida al adicionar el ruido gaussiano?

Rta: aparte de envejecida, tienen pequeños "gránulos" o "puntos de arena" en toda la imagen. El efecto es notorio en zonas amplias y de pocos detalles, como el cielo o las nubes.

6.5.3. Ruido sal y pimienta:

Este tipo de ruido tiene dos tonos, uno correspondiente a la sal y el otro a la pimienta. Para crear ruido de este tipo, lo primero que debemos hacer es crear ruido uniforme para cada una de las bandas de color, y posteriormente aplicar un proceso de umbralización (similar al que utilizamos cuando convertimos una imagen a escala de grises en una imagen BW). Dependiendo del valor del umbral seleccionado, tendremos más o menos píxeles correspondientes a sal y a pimienta.

Para el siguiente código en Python el umbral seleccionado es 10, y a los píxeles que superen el umbral se les asigna el color 255 (máxima escala).

```
sp_noise=np.zeros((img.shape[0],  img.shape[1],  img.shape[2]),d-
type=np.uint8)
ret,impulse_noise0=cv2.threshold(uniform_noise[:,:,0],10,255,cv2.
THRESH_BINARY)
ret,impulse_noise1=cv2.threshold(uniform_noise[:,:,1],10,255,cv2.
THRESH_BINARY)
ret,impulse_noise2=cv2.threshold(uniform_noise[:,:,2],10,255,cv2.
THRESH_BINARY)
sp_noise[:,:,0]=impulse_noise0
sp_noise[:,:,1]=impulse_noise1
sp_noise[:,:,2]=impulse_noise2
cv2_imshow(sp_noise[:,:,0])
```

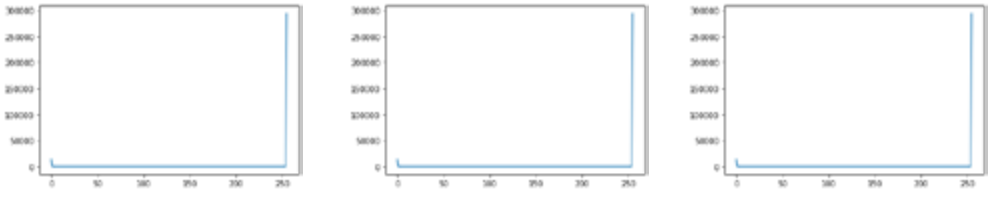
¡En este caso obtendremos poca pimienta y mucha sal!

Figura 94. Imagen a color con ruido sal y pimienta, con $th = 10$.

Y sus histogramas por banda, son:

```
# repetir este paso por canal
hist = cv2.calcHist([sp_noise],[2],None,[256],[0,256])
plt.plot(hist)
plt.show()
```

Obteniendo:



(a) (b) (c)

Figura 95. Histograma por banda de la imagen a color – ruido sal y pimienta con $th = 10$.

Ahora, vamos a fijar un umbral alto, por ejemplo, de 200:

```
sp_noise=np.zeros((img.shape[0],  img.shape[1],  img.shape[2]),d-
type=np.uint8)
ret,impulse_noise0=cv2.threshold(uniform_noise[:,:,0],200,255,-
cv2.THRESH_BINARY)
ret,impulse_noise1=cv2.threshold(uniform_noise[:,:,1],200,255,-
cv2.THRESH_BINARY)
ret,impulse_noise2=cv2.threshold(uniform_noise[:,:,2],200,255,-
cv2.THRESH_BINARY)
sp_noise[:,:,0]=impulse_noise0
sp_noise[:,:,1]=impulse_noise1
sp_noise[:,:,2]=impulse_noise2
cv2_imshow(sp_noise[:,:,0])
```

¡En este caso obtendremos poca sal y mucha pimienta!

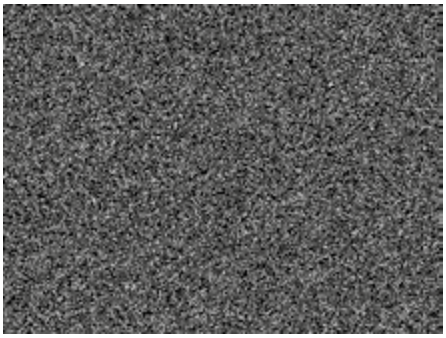
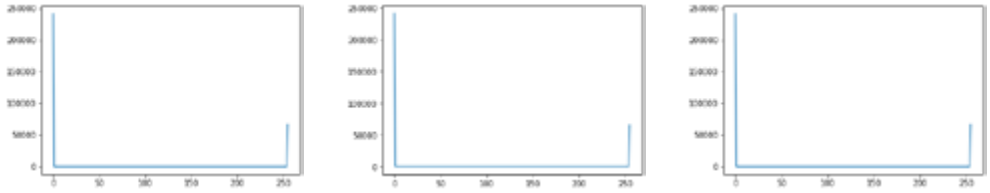


Figura 96. Imagen a color con ruido sal y pimienta, con $th = 200$.

Y sus histogramas, son:



(a) (b) (c)

Figura 97. Histograma por banda de la imagen a color – ruido sal y pimienta con $th = 200$.

Y finalmente adicionamos este ruido a la imagen, así:

```
noisy_img_sp = cv2.add(img, (sp_noise*0.5).astype(np.uint8))
cv2_imshow(noisy_img_sp)
```

Figura 98. Imagen a color con ruido sal y pimienta, $th = 200$ – villa de leyva.

¿En qué se diferencia esta imagen de la obtenida al adicionar el ruido uniforme?

Rta: es mucho más notorio el efecto granular que en la imagen con ruido uniforme.

6.6. FILTROS ESPACIALES

En esta sección entenderemos y aplicaremos el concepto de filtro espacial. Matemáticamente lo abordaremos en la Sección 6.7, pero por ahora, de forma conceptual y práctica realizaremos el filtrado de ruido en imágenes.

Lo primero que vamos a realizar es comparar el efecto que tiene en una imagen los tres diferentes tipos de ruido que se explicaron en la Sección 6.5. La Figura 99 presenta un ejemplo.

(a) (b) (c)

Figura 99. Imagen de playa con tres tipos distintos de ruido: (a) sal y pimienta, (b) guassiano, (c) uniforme. Fuente: repositorio personal de los autores.

Pero, ¿cómo las diferenciamos?

- Empecemos con la imagen central, el efecto que tiene es de "envejecida", entonces esa imagen contiene ruido gaussiano.
- Ahora, comparemos las imágenes de los extremos, ambas tienen un efecto "granular". La imagen de la izquierda tiene ruido sal y pimienta, por ser más evidente el efecto granular; mientras que, la de la derecha tiene ruido uniforme.

A continuación, por medio de ejemplos se ilustrará el efecto que tienen diferentes filtros espaciales en imágenes con diferentes tipos de ruido.

Empezaremos con la imagen que tienen ruido sal y pimienta, a la cual le aplicamos un filtro tipo promedio. Este filtro 2D es similar al filtro de promedio 1D que conocimos al inicio de este libro, pero en este caso es una matriz con todos sus valores iguales a uno dividido en su tamaño (igual a filas x columnas). Por ejemplo, si el tamaño del filtro es (5×5), entonces cada posición del filtro tendrá el peso de $1/25$, como se presenta en la siguiente Figura.

$$\frac{1}{25}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Figura 100. Filtro de promedio (5×5).

Una vez hemos leído la imagen en Python, aplicamos el filtro con el siguiente código:

```
f1_sp = cv2.blur(noisy_img_sp, (5,5), 0)
cv2_imshow(f1_sp)
```

Este filtro lo aplicamos a la imagen denominada *noisy_img_sp*, cuyo resultado es la *imagen f1_sp*, la cual se presenta a continuación.

Figura 101. Imagen filtrada con filtro de promedio – ruido sal y pimienta.

El segundo tipo de filtro que vamos a evaluar es el filtro gausisano. En este caso, los valores de la matriz varían entre sí, teniendo mayor peso la posición central del filtro, y de menor peso las posiciones de los extremos. En este tipo de filtro los pesos decrecen de forma gaussiana a medida que se alejan de la posición central, como se presenta en la siguiente figura:

$$\frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

Figura 102. Filtro gaussiano (5×5). Se ha encerrado en un recuadro rojo la posición central del filtro.

Para el filtro gausisano se utiliza el siguiente código en Python:

```
f2_sp = cv2.GaussianBlur(noisy_img_sp, (5,5), 0)
cv2_imshow(f2_sp)
```

Cuyo tamaño del filtro es también (5×5), y la salida en este caso se denomina *f2_sp*. La imagen filtrada se presenta a continuación:

Figura 103. Imagen filtrada con filtro de gaussiano – ruido sal y pimienta.

Finalmente, filtraremos la imagen con un filtro de mediana (no confundir con el filtro de promedio). Este tipo de filtro difiere a los dos anteriores en que no existe una matriz de pesos del filtro. Se realiza un proceso de ordenamiento de los valores de los píxeles de la imagen de una región de igual tamaño al del filtro, y se selecciona el valor correspondiente a la posición central de los píxeles ordenados.

El código en Python es:

```
f3_sp = cv2.medianBlur(noisy_img_sp, 5)
cv2_imshow(f3_sp)
```

La imagen filtrada corresponde a *f3_sp,* como se presenta en la siguiente figura.

Figura 104. Imagen filtrada con filtro de mediana – ruido sal y pimienta.

La forma en que cada uno de estos filtros opera sobre la imagen, se explicará en detalle en la Sección 6.7.

Por ahora, quiero que respondas la siguiente pregunta.

¿Cuál imagen filtrada consideras que presenta mejor calidad?, es decir, ¿qué filtro seleccionarías para eliminar ruido tipo sal y pimienta?
Rta: Para este tipo de ruido, el filtro de mediana es la mejor opción.

En la segunda parte de esta sección, buscaremos un filtro para una imagen que contiene ruido tipo gaussiano. Partiremos con el filtro de promedio (Figura 72), y seguiremos con otro tipo de filtro denominado filtro bilateral (Figura 73).

Figura 105. Imagen filtrada con filtro de promedio – ruido gaussiano.

El filtro bilateral tiene en cuenta tres parámetros para calcular el valor de salida: diámetro de la vecindad (d), varianza a nivel de color (σ_{color}), y varianza a nivel de ubicación espacial (σ_{space}).

- d es el diámetro de cada vecindad de píxeles. Si es negativo, se calcula a partir de σ_{space}.
- Cuando σ_{color} es alto, entonces, los colores más alejados dentro de la vecindad se mezclan, obteniendo largas áreas de color casi-homogéneo.
- Cuando σ_{space} es alto, entonces, los píxeles más alejados entre sí se mezclan (espacialmente hablando).

Este tipo de filtro es similar al filtro gaussiano, en términos de la cercanía en ubicación espacial, pero incluye el concepto de cercanía de color también.

El siguiente es el código en Python para el filtro bilateral con $d = 9$, $\sigma_{color} = 10$, $\sigma_{space} = 10$

```
blur1 = cv2.bilateralFilter(noisy_img_sp,15,50,100)
cv2_imshow(blur1)
```

Figura 106. Imagen filtrada con filtro bilateral – ruido gaussiano.

¿Cuál imagen filtrada consideras que presenta mejor calidad?, es decir, ¿Qué filtro seleccionarías para eliminar ruido tipo gaussinoa?
Rta: Para este tipo de ruido, el filtro bilateral es la mejor opción.

Puedes complementar la información de los filtros espaciales de esta sección en https://docs.opencv.org/4.x/d4/d13/tutorial_py_filtering.html

6.7. CONVOLUCIÓN

En esta sección comprenderemos el concepto de "*convolución*" en imágenes. Estrictamente hablando, realmente realizaremos una operación de *correlación* entre un filtro y una imagen, pero, teniendo en cuenta que, en la comunidad académica de visión por computador y de aprendizaje profundo el nombre utilizado para esa operación es el de *convolución*, utilizaremos ese nombre en este libro.

Lo primero a tener en cuenta es que la *convolución* es una operación que requiere dos matrices, una de las cuales es la imagen, y la otra es el filtro. Típicamente, los filtros tienen la misma cantidad de filas que de columnas, por ejemplo, de 3 x 3, pero se podrían diseñar filtros con dimensiones que no sean iguales entre sí. Cada una de las posiciones del filtro se denominan "pesos". Conceptualmente, el filtro debe tener una dimensión menor a la de la imagen para poder realizar un proceso de "barrido" sobre ella.

Con un ejemplo ilustraremos el proceso:

Figura 107. Imagen y filtro para operación de convolución.

El primero paso consiste en adicionarle un borde a la imagen con valores de ceros, ampliando su dimensión en 2 filas y dos columnas. Es decir, para nuestra Imagen de ejemplo, la cual es de (5 x 5), al incluirle el borde quedará de (7 x 7).

0	0	0	0	0	0	0
0	10	10	30	30	10	0
0	10	10	30	30	10	0
0	10	10	30	30	10	0
0	10	10	30	30	10	0
0	10	10	30	30	10	0
0	0	0	0	0	0	0

Figura 108. Imagen de entrada con borde.

El propósito de adicionarle el borde a la Imagen de entrada es que el resultado de la convolución (Imagen filtrada) contenga la misma cantidad de filas y de columnas que de la Imagen de entrada. Cuando el tamaño del filtro es de 3 x 3, el borde es de 2 filas (una superior y una inferior) y dos columnas (una a la izquierda y una a la derecha); cuando el filtro es de tamaño 5x5, el borde es de 4 filas (dos superiores y dos inferiores) y 4 columnas (dos a la derecha y dos a la izquierda), y así sucesivamente.

Como segundo paso, el filtro se superpone sobre la Imagen de entrada, ubicándolo en el extremo superior izquierdo. Posteriormente, se realiza la multiplicación de los píxeles de la Imagen con los pesos del filtro. Si el filtro es de tamaño 3 x 3, entonces se realizan 9 multiplicaciones. Finalmente, se suma el resultado de las multiplicaciones, y el valor obtenido se asigna al primer pixel de la imagen (primera fila, primera columna). Hay que tener en cuenta que, si el resultado de la operación anterior es negativo, se escribe un cero en el pixel de salida correspondiente. Por otro lado, si el resultado es superior a 255, se escribe 255.

El proceso se presenta a continuación:

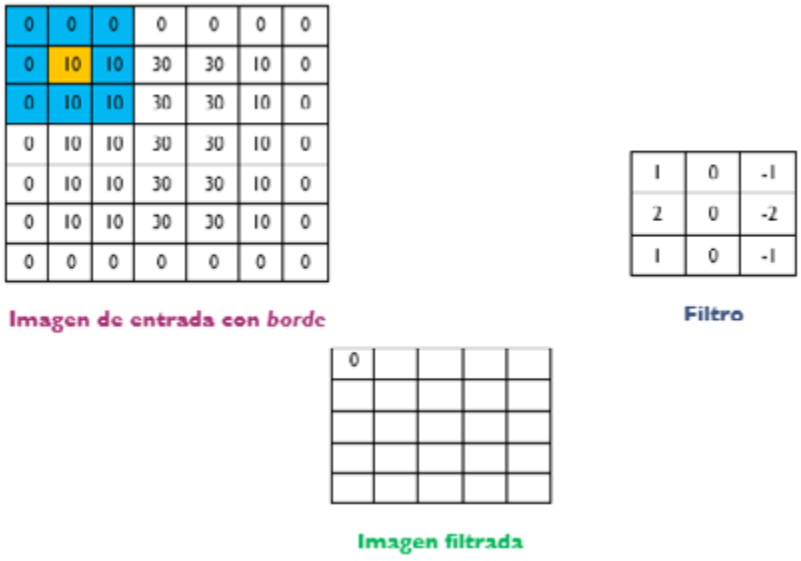


Figura 109. Proceso de convolución: Paso 2. Se sombrea en amarillo el píxel central de la imagen, para el paso correspondiente.

Como tercer paso, el filtro se desplaza una posición a la derecha, y se repite de nuevo el proceso de realizar las multiplicaciones, sumar su resultado y asignar al pixel correspondiente de la imagen de salida (primera fila, segunda columna). El proceso se presenta en la siguiente figura.

Figura 110. Proceso de convolución: paso 3. Se sombrea en amarillo el píxel central de la imagen, para el paso correspondiente.

Como cuarto paso, y así sucesivamente, se desplaza de nuevo el filtro una posición a la derecha, se realizan las correspondientes multiplicaciones, se suman sus valores y se asigna al píxel de la imagen de salida que corresponda. Una vez el filtro se desplaza y llega al borde de la imagen, debe desplazarse de nuevo, empezando por la segunda fila de la imagen, primera columna. El proceso de desplazamiento se realiza de forma iterativa, hasta que se recorra por completo a la imagen de entrada. La ubicación del píxel central para cada uno de los pasos del proceso de convolución y la dirección del desplazamiento se presentan a continuación:

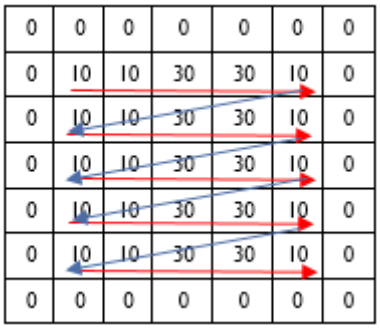


Figura 111. Pixel central en el proceso de convolución: barrido de la imagen de izquierda a derecha, y de arriba abajo.

Para el presente ejemplo, el resultado de la convolución es:

0	0	0	60	90
0	0	0	80	120
0	0	0	80	120
0	0	0	80	120
0	0	0	60	90

Figura 112. Imagen filtrada.

Para saber cuál es el tamaño del borde a adicionarle a la imagen, utilizaremos las siguientes ecuaciones:

$$W_0 = W_1 - W_k + 1 \qquad \text{Ecuación 50}$$

$$H_0 = H_1 - H_k + 1 \qquad \text{Ecuación 51}$$

Donde W_1, W_K, W_0, corresponden a la cantidad de columnas de la imagen de entrada con borde, del filtro y de la imagen filtrada (output), respectivamente. Mientras que, H_1, H_k, H_0, corresponden a la cantidad de filas de la imagen de entrada con borde, del filtro y de la imagen filtrada, respectivamente.

Entonces, si queremos que la imagen de salida tenga 5 x 5 y estamos utilizando un filtro de 3 x 3, necesitamos que la imagen de entrada con borde sea de 7 x 7, teniendo en cuenta que al reemplazar los valores anteriores en la ecuación 50 o ecuación 51, tenemos que $5 = 7 - 3 + 1$. A la imagen de entrada sin borde (cuyas dimensiones son iguales a la de la imagen de salida), debemos adicionarle 2 filas y 2 columnas, o, en otras palabras, un borde de 1 rodeando a la imagen.

6.8. DETECCIÓN DE BORDES

En esta subsección abordaremos el tema de detección de bordes en imágenes. Lo primero, es saber que, así como existen filtros cuyo propósito consiste en reducir el ruido de una imagen (como los vistos en el Capítulo 6.6.), también tenemos filtros cuyo objetivo es detectar el borde de una imagen. Mientras los primeros cumplen que la sumatoria de sus *pesos* es igual a 1, en los segundos (detección de bordes) se cumple que la sumatoria de sus *pesos* es igual a 0.

Adicionalmente, se pueden detectar bordes en una sola dirección o multi-dirección. Dentro de los filtros más conocidos en la literatura tenemos *Prewitt, Sobel y Laplaciano*. Y como algoritmo de detección de bordes (que incluye etapa de pre-procesamiento, filtrado y pos-procesamiento), tenemos el algoritmo Canny[10].

Empecemos con el filtro *Prewitt*. Es una clase de detector de bordes aplicando la diferencia entre píxeles de primer orden. Puede detectar bordes en el eje horizon-

10 Öztürk, Ş., & Akdemir, B. (2015). Comparison of edge detection algorithms for texture analysis on glass production. Procedia-Social and Behavioral Sciences, 195, 2675-2682.

tal o en el eje vertical. Este filtro utiliza un tamaño de 3x3, donde la fila o columna central son de valor 0, y las filas o columnas de los extremos son de valor 1 y -1. A continuación, se presenta el filtro *Prewitt* para cada dirección de detección de borde.

Figura 113. Filtro Prewitt (3 x 3).

En el caso del filtro *Sobel*, también se detectan bordes en la dirección vertical y horizontal, pero en este caso, se realiza un énfasis en el pixel central de las filas o columnas cuyos pesos son distintos de cero, realizando una detección más fuerte de los cambios de la imagen utilizando la primera derivada. En la siguiente figura se presenta el filtro *Sobel*.

Figura 114. Filtro Sobel (3 x 3).

Por otro lado, el filtro *Laplaciano* se basa en la segunda derivada de la imagen (o diferencia de segundo orden)[11]. Existen dos versiones del filtro *Laplaciano*, en la primera, se computa la diferencia entre el pixel central y el promedio de sus vecinos directos (arriba, abajo, izquierda, derecha), y en la segunda, se computa la diferencia entre el pixel central y el promedio de todos sus vecinos (incluidas las esquinas). La versión básica y la alternativa se presentan a continuación[12].

Figura 115. Filtro Laplaciano (3 x 3).

Finalmente, tenemos el algoritmo o filtro Canny, el cual realiza varias etapas, las cuales se resumen a continuación[13y14]:

11 https://www.sciencedirect.com/topics/engineering/laplacian-filter.
12 Nixon, M. S., & Aguado, A. S. (2008). Low-level feature extraction (including edge detection). Feature Extraction and Image Processing. 3rd edi. Linacre House/Jordan Hill/Oxford: Elsevier, 115-79.
13 https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html
14 https://docs.opencv.org/3.4/da/d5c/tutorial_canny_detector.html

a. Reducción de ruido: es una etapa de pre-procesamiento que consiste en reducir el ruido presente en la imagen, por medio de un filtro Gaussiano de tamaño 5 x 5.

b. Identificación del gradiente de intensidad de la imagen: se filtra la imagen obtenida en el paso anterior tanto con un filtro Sobel de detección de bordes horizontales, como de detección de bordes verticales, obteniendo G_x y G_y, respectivamente. A partir de las dos imágenes resultantes (una por cada filtro Sobel), se calcula la imagen gradiente, tanto en magnitud como en fase, aplicando las siguientes ecuaciones:

$$|G| = \sqrt{G_x + G_y} \qquad \textbf{Ecuación 52}$$

$$\measuredangle = \mathbf{tg}^{-1}\left({}^{G_y}/_{G_x} \right) \qquad \textbf{Ecuación 53}$$

La dirección del gradiente siempre es perpendicular a los bordes. Se aproxima a uno de los cuatro posibles ángulos: horizontal, vertical, diagonal derecha, diagonal izquierda.

c. Supresión de los no máximos: esta etapa y la siguiente hacen parte del pos-procesamiento. Consiste en remover los píxeles no deseados, que no correspondan con el borde de la imagen. Si existen varios píxeles vecinos en la dirección del gradiente que son potenciales bordes, se identifica cuál de ellos es un máximo local, y ese es el píxel que se conversa para la siguiente etapa del algoritmo.

d. Umbralización con histéresis: en esta última fase se eliminan falsos bordes, a partir de un proceso de histéresis con dos umbrales. Se define un umbral alto y un umbral bajo. Si el potencial borde supera al umbral alto, entonces se considera un borde real. Si, por el contrario, es menor que el umbral bajo, se descarta. Para los potenciales bordes cuya intensidad se encuentra entre el umbral bajo y el umbral alto, la decisión de incluirse como un verdadero borde o de eliminarse depende de sus píxeles vecinos. Si éstos son bordes, se considera también como borde; en caso contrario, se descarta.

Una de las ventajas del *algoritmo Canny* es que detecta de forma simultánea bordes en cuatro direcciones (vertical, horizontal, diagonal derecha y diagonal izquierda). Adicionalmente, el borde detectado es delgado, gracias a sus etapas de pos-procesamiento posteriores al filtrado (supresión de los no-máximos y umbralización con histéresis).

A continuación, aplicaremos los filtros anteriores a una imagen, para comparar las diferencias de forma visual entre los bordes detectados en cada caso.

```
import cv2
import numpy as np
from google.colab.patches import cv2_imshow
img = cv2.imread('coctel.jpg')
cv2_imshow(img)
```

```
prewitt_x  = np.array([[1, 1, 1],
                       [0,  0,  0],
                       [-1,  -1,  -1]], dtype=np.float32)
print(prewitt_x )

fig1= cv2.filter2D(img, -1, prewitt_x, borderType=0)
cv2_imshow(fig1)

fig1g = cv2.cvtColor(fig1, cv2.COLOR_BGR2GRAY )
ret, fig1bw = cv2.threshold(fig1g,50,255,cv2.THRESH_BINARY)
cv2_imshow(255-fig1bw) # imagen filtrada con Prewitt_x
```

```
prewitt_y  = np.array([[1,  0,  -1],
                       [1,  0,  -1],
                       [1,  0,  -1]], dtype=np.float32)
print(prewitt_y )

fig2= cv2.filter2D(img, -1, prewitt_y, borderType=0)
cv2_imshow(fig2)

fig2g = cv2.cvtColor(fig2, cv2.COLOR_BGR2GRAY )
ret, fig2bw = cv2.threshold(fig2g,50,255,cv2.THRESH_BINARY)
cv2_imshow(255-fig2bw) # imagen filtrada con Prewitt_y
```

```
fig3bw  = fig1bw  + fig2bw
cv2_imshow(255-fig3bw) # imagen filtrada con Prewitt_x + Prewitt_y
```

```
sobel_x  = np.array([[1,  2,  1],
                     [0,  0,  0],
                     [-1,  -2,  -1]], dtype=np.float32)
print(sobel_x )

fig4= cv2.filter2D(img, -1, sobel_x, borderType=0)
cv2_imshow(fig4)

fig4g = cv2.cvtColor(fig4, cv2.COLOR_BGR2GRAY )
ret, fig4bw = cv2.threshold(fig4g,50,255,cv2.THRESH_BINARY) # imagen
filtrada con Sobel_x
cv2_imshow(255-fig4bw)
```

```
sobel_y  = np.array([[1,  0,  -1],
                     [2,  0,  -2],
                     [1,  0,  -1]], dtype=np.float32)
print(sobel_y )

fig5= cv2.filter2D(img, -1, sobel_y, borderType=0)
cv2_imshow(fig5)

fig5g = cv2.cvtColor(fig5, cv2.COLOR_BGR2GRAY )
ret, fig5bw = cv2.threshold(fig5g,50,255,cv2.THRESH_BINARY) # imagen
filtrada con Prewitt_y
cv2_imshow(255-fig5bw)
```

```
fig6bw  = fig4bw  + fig5bw
cv2_imshow(255-fig6bw) # imagen filtrada con Sobel_x + Sobel_y
```

```
laplaciano1 = np.array([[0,  -1,  0],
                        [-1,  4,  -1],
                        [0,  -1,  0]], dtype=np.float32)
print(laplaciano1)

fig7= cv2.filter2D(img, -1, laplaciano1, borderType=0)
cv2_imshow(fig7)

fig7g = cv2.cvtColor(fig7, cv2.COLOR_BGR2GRAY )
ret, fig7bw = cv2.threshold(fig7g,50,255,cv2.THRESH_BINARY)
cv2_imshow(255-fig7bw) # imagen filtrada con Laplaciano básico
```

```
laplaciano2 = np.array([[-1,  -1v -1],
                        [-1,  8,  -1],
                        [-1,  -1,  -1]], dtype=np.float32)
print(laplaciano2)

fig8= cv2.filter2D(img, -1, laplaciano2, borderType=0)
cv2_imshow(fig8)

fig8g = cv2.cvtColor(fig8, cv2.COLOR_BGR2GRAY )
ret, fig8bw = cv2.threshold(fig8g,50,255,cv2.THRESH_BINARY)
cv2_imshow(255-fig8bw) # imagen filtrada con Laplaciano alternativo
```

```
edges_canny = cv2.Canny(img,220,55)
cv2_imshow(255-edges_canny) # imagen filtrada con algoritmo Canny
```

Empezaremos analizando las imágenes filtradas con *Prewitt*. La obtenida con *Prewitt_x* detecta bordes especialmente en la dirección horizontal, como la altura de la bebida dentro de la copa, o el soporte horizontal del techo del restaurante. En el caso de la imagen filtrada con *Prewitt_y,* no se detectan los bordes mencionados anteriormente, pero sí los bordes correspondientes a las columnas verticales de soporte del techo. Finalmente, la imagen obtenida al sumar las dos anteriores es más completa que sus antecesoras por separado, mostrando bordes en ambas direcciones.

En el caso de las imágenes obtenidas con Sobel, los resultados son similares a las obtenidas con *Prewitt*. Sin embargo, se puede apreciar mayor demarcación en algunos bordes.

Por otro lado, las imágenes obtenidas con el filtro *Laplaciano* (en sus dos versiones) muestran el borde vertical de la copa, aunque es más notorio con el Laplaciano alternativo. En ambos casos, las imágenes filtradas tienen bordes delgados, a diferencia de sus antecesoras.

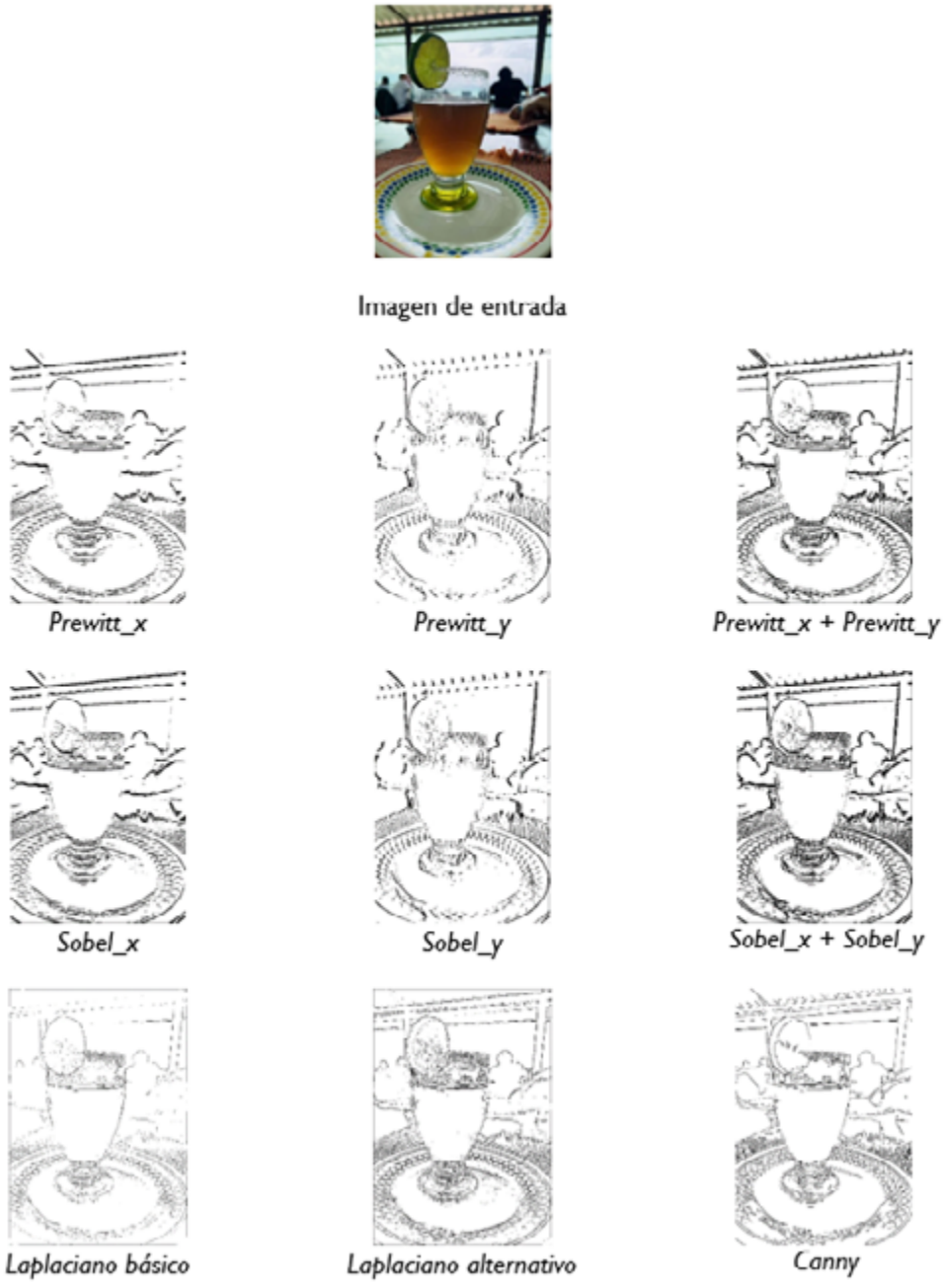


Figura 116. Imagen de entrada y detección de bordes con diferentes tipos de filtros.
Fuente: repositorio personal de los autores.

Finalmente, con el algoritmo Canny se tienen bordes delgados en todas las direcciones, y aparecen bordes en zonas de la imagen que con los otros filtros no se visualizaban, por ejemplo, las ondulaciones en el tejado.

6.9. TRANSFORMADA DFT Y DCT

En esta sección se abordan dos transformadas en imágenes, del dominio espacial al dominio frecuencial. Específicamente, las correspondientes con la Transformada de Fourier Discreta (DFT) y la Transformada Consenoidal Discreta (DCT).

6.9.1. DFT (Discrete Fourier Transform)

La DFT de una imagen se calcula a partir de la siguiente ecuación:

$$F(k,l) = \sum_{a=0}^{M-1} \sum_{b=0}^{N-1} f(a,b) e^{-i2\pi\left(\frac{ka}{M}+\frac{lb}{N}\right)} \qquad \textbf{Ecuación 54}$$

Teniendo en cuenta que,

$$e^{ix} = \{cos(x)\} + \{i * sin(x)\} \qquad \textbf{Ecuación 55}$$

Donde $F(k,l)$ es la Transformada Discreta de Fourier, mientras que *f(a,b)* es la imagen en el dominio espacial de tamaño (M,N). Es decir, el resultado de la DFT se obtiene al multiplicar la imagen en el dominio espacial *f(a,b)* por la función base (que en este caso es una señal exponencial compleja) y sumar el resultado para cada pareja (k,l). Se resalta que tanto los valores *(a,b)* como los valores (k,l) son enteros.

Cuando se grafica la DFT de una imagen, no se puede relacionar fácilmente el resultado obtenido con la imagen original. Típicamente, si existen cambios significativos de dirección en la imagen, éstos se verán reflejados en la DFT (patrones de líneas blancas). Si la imagen se invierte en el eje vertical (*flip* vertical), el efecto que se tiene en su DFT es precisamente el de inversión. De forma similar, si la imagen si invierte respecto al aje horizontal (*flip* horizontal), también se tendrá el efecto en su DFT de inversión. En ambos casos, la inversión en la DFT es en relación con el eje vertical, de tal forma que la DFT de la imagen invertida horizontal es igual a la DFT de la imagen invertida vertical. Por otro lado, si a la imagen se le aplica doble inversión (una por cada eje), su DFT es igual al de la imagen original (sin invertir).

La Figura 84 presenta un ejemplo de una imagen y su correspondiente DFT para diferentes tipos de manipulaciones de la imagen. Se resalta que la DFT de la imagen original es igual a la DFT de doble *flip*; mientras que, la DFT de *flip* vertical es igual a la DFT de *flip* horizontal.

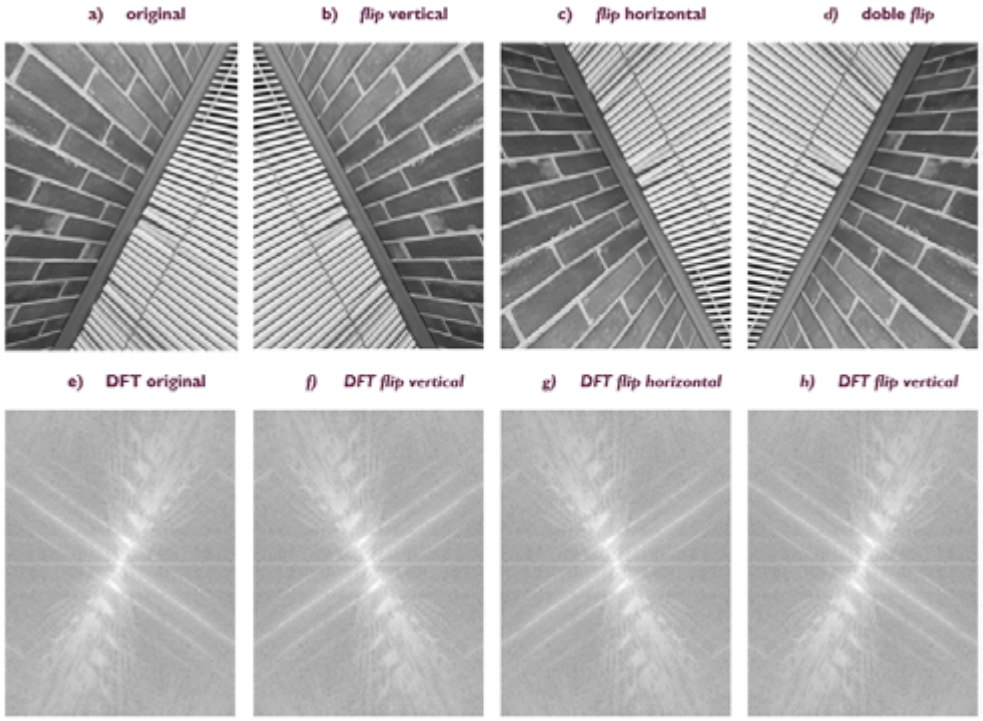


Figura 117. Imagen con su respectiva DFT.

Para calcular la DFT de una imagen en Python, utilizamos el siguiente código:

Paso 1) Cargue de librerías de lectura de la imagen

```
import numpy as np
import cv2
from google.colab.patches import cv2_imshow
url= "/content/oficina.png"
img = cv2.imread(url)
```

Paso 2) Convertir la imagen RGB a escala de grises y representarla en punto flotante de 32 bits

```
img_gray=cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
cv2_imshow(img_gray)
img_float32 = np.float32(img_gray)
```

Paso 3) Calcular la DFT y visualizar el resultado en escala logarítmica

```
dft = cv2.dft(img_float32, flags = cv2.DFT_COMPLEX_OUTPUT)
dft_shift = np.fft.fftshift(dft)
magnitude_spectrum = 20*np.log(cv2.magnitude(dft_shift[:,:,0],dft_
shift[:,:,1]))
cv2_imshow(magnitude_spectrum)
```

Paso 4) Invertir la imagen en el eje vertical, calcular su DFT y graficar

```
flipVertical = cv2.flip(img_float32, 1)
cv2_imshow(flipVertical)
dft = cv2.dft(flipVertical, flags = cv2.DFT_COMPLEX_OUTPUT)
dft_shift = np.fft.fftshift(dft)
magnitude_spectrum = 20*np.log(cv2.magnitude(dft_shift[:,:,0],dft_
shift[:,:,1]))
cv2_imshow(magnitude_spectrum)
```

Paso 5) Invertir la imagen en el eje horizontal, calcular su DFT y graficar

```
flipHorizontal = cv2.flip(img_float32, 0)
cv2_imshow(flipHorizontal)
dft = cv2.dft(flipHorizontal, flags = cv2.DFT_COMPLEX_OUTPUT)
dft_shift = np.fft.fftshift(dft)
magnitude_spectrum = 20*np.log(cv2.magnitude(dft_shift[:,:,0],dft_
shift[:,:,1]))
cv2_imshow(magnitude_spectrum)
```

Paso 6) Doble inversión de la imagen (horizontal y vertical), calcular su DFT y graficar

flipBoth = cv2.flip(img_float32, -1)

```
cv2_imshow(flipBoth)
dft = cv2.dft(flipBoth, flags = cv2.DFT_COMPLEX_OUTPUT)
dft_shift = np.fft.fftshift(dft)
magnitude_spectrum = 20*np.log(cv2.magnitude(dft_shift[:,:,0],dft_
shift[:,:,1]))
cv2_imshow(magnitude_spectrum)
```

6.9.2. DCT (Discrete Cosine Transform)

Esta transformada es muy útil para la compresión de imágenes, dado que gran parte de la información de la imagen (la más significativa o representativa) se concentra en pocos coeficientes espectrales. Hace parte del algoritmo de compresión de imágenes conocido como JPEG (*Joint Photographic Experts Group*).

A diferencia de la DFT, en este caso todos sus coeficientes son reales, calculados a partir de la siguiente ecuación:

$$C(k,l) = \sum_{a=0}^{M-1} \sum_{b=0}^{N-1} f(a,b) \cos\left(\frac{\pi}{M}\left(a+\frac{1}{2}\right)k\right) \cos\left(\frac{\pi}{N}\left(b+\frac{1}{2}\right)l\right) \qquad \textbf{Ecuación 56}$$

Donde $C(k,l)$ corresponde a la DCT de la imagen $f(a,b)$ de tamaño (M,N).

Típicamente, la DCT se calcula por bloques de la imagen, es decir, la imagen se divide en zonas y a cada zona se la aplica la DCT. A continuación, se presenta un ejemplo de la DCT para la imagen completa, y para diferentes tamaños de bloque.

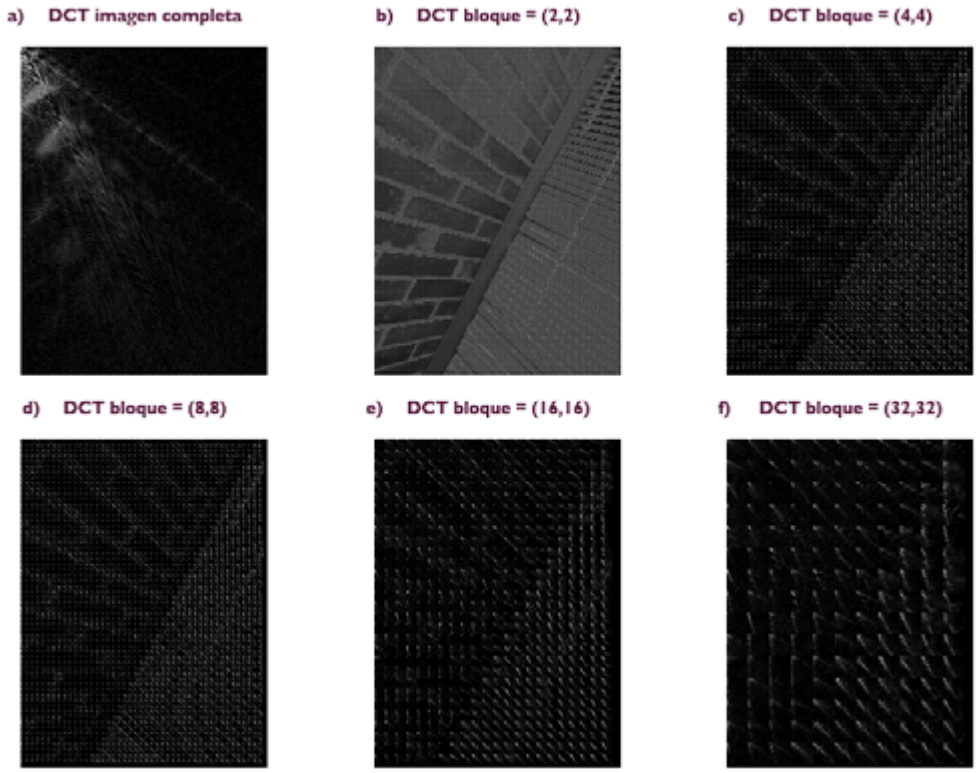


Figura 118. DCT de la imagen de la Figura 84.a.

A diferencia de la DFT, sí es posible encontrar una relación directa entre la DCT y la imagen de entrada, cuando el tamaño del bloque es pequeño. Por ejemplo, en la Figura 118b, se alcanza a apreciar la pared y la persiana de la oficina; mientras que, en la Figura 118c y Figura 118d, se visualizan líneas diagonales correspondientes a la separación entre filas de ladrillos. Cuando el tamaño del bloque es de (32,32) o superior, ya no se alcanzan a identificar los patrones de la imagen.

En este caso, el código de Python para obtener las gráficas anteriores, se presenta a continuación:

Paso 1) Cargue de librerías de lectura de la imagen

```
import numpy as np
import cv2
from google.colab.patches import cv2_imshow
url= "/content/oficina.png"
img = cv2.imread(url)
```

Paso 2) Convertir la imagen RGB a escala de grises y representarla en punto flotante de 32 bits

```
img_gray=cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
cv2_imshow(img_gray)
img_float32 = np.float32(img_gray)
```

Paso 3) Calcular la DCT y visualizar el resultado

```
dct = cv2.dct(img_float32)
cv2_imshow(dct)
```

Nota: en este caso no se necesitan re-ordenar los coeficientes, como sí se realizó en el caso de la DFT. Adicionalmente, no se calcula la magnitud, dado que los valores son reales. Tampoco, se grafica en escala logarítmica.

Paso 4) Definir el tamaño del block, calcular la cantidad de bloques y crear un DCT de salida de valores cero.

```
B=2 #blocksize
img1 = img_float32
h= img1.shape[0]
w =img1.shape[1]
blocksV=np.int(h/B)
blocksH=np.int(w/B)
transformed=np.zeros([h, w])
```

Nota: este ejemplo está diseñado para bloques cuadrados. En este caso es de (2,2).

Paso 4) Aplicar la DCT por bloque y escribir el resultado en la zona de salida correspondiente.

```
for row in range(blocksV):
    for col in range(blocksH):
      currentblock = cv2.dct(img1[row*B:(row+1)*B,col*B:(col+1)*B])
        transformed[row*B:(row+1)*B,col*B:(col+1)*B]= currentblock
cv2_imshow((transformed))
```

6.9.3. Comprensión de imágenes con la DCT

Como se había mencionado previamente, una de las aplicaciones de la DCT es en la comprensión de imágenes, específicamente en el estándar JPEG. A continuación, se explicará brevemente en que consiste ese método de comprensión.

Lo primero a resaltar es que JPEG es un método de comprensión con pérdida de información (o *lossy*), que significa que parte de los datos se pierden en el proceso de compresión y no se puede recuperar la imagen exactamente igual a la original; no obstante, de forma visual, no se apreciarán diferencias significativas entre la imagen original y su versión comprimida. Su principal ventaja sobre métodos de compresión sin pérdida de información (o *lossless*) es que permite obtener una tasa de compre-

sión mayor, conocida como CR (*compression rate*), la cual corresponde a la relación entre el tamaño de la imagen sin comprimir y el tamaño de la imagen comprimida.

Los principales bloques que hacen parte del método JPEG son: DCT, cuantización inteligente, y codificación RL y *Huffman*. De forma muy resumida, los pasos son los siguientes[15]:

a. Aplicar DCT por bloques de la imagen, por ejemplo, de tamaño (8,8). El resultado es otra imagen del mismo tamaño, cuyos datos corresponden a coeficientes espectrales.

b. Aplicar cuantización a los coeficientes espectrales, dividiendo su valor entre un factor de cuantización. De esta manera, se reduce la cantidad de valores de salida (y la precisión de los datos). Adicionalmente, el proceso es inteligente, dado que el factor de cuantización no es constante, sino que, depende de la amplitud del coeficiente a cuantizar. A los coeficientes que representan frecuencias mayores se les aplica un factor de cuantización mayor.

c. A los coeficientes cuantizados se les aplica el método de codificación run-length (RL). Este método aprovecha la gran cantidad de ceros consecutivos que se obtienen al combinar la DCT con la cuantización inteligente. El barrido sobre los coeficientes cuantizados se realiza en forma de zig-zag, empezando en el extremo superior izquierdo de la matriz (DCT cuantizada). La longitud de la trama de salida es mucho menor a la cantidad de coeficientes cuantizados del paso b.

d. Finalmente, se aplica codificación Huffman. La idea principal de este método es representar los "símbolos" de mayor ocurrencia de la trama con la menor cantidad de bits, mientras que, los de menor ocurrencia con la mayor cantidad de bits. Entonces, los coeficientes espectrales cuantizados y codificados con RL tendrán una representación binaria que es significativamente menor a multiplicar el tamaño de la imagen por 8 bits (en el caso de imágenes a escala de grises) o por 24 bits (en el caso de imágenes a color de 3 canales). Los valores de compresión pueden llegar a 100 veces.

15 https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossy/jpeg/index.htm