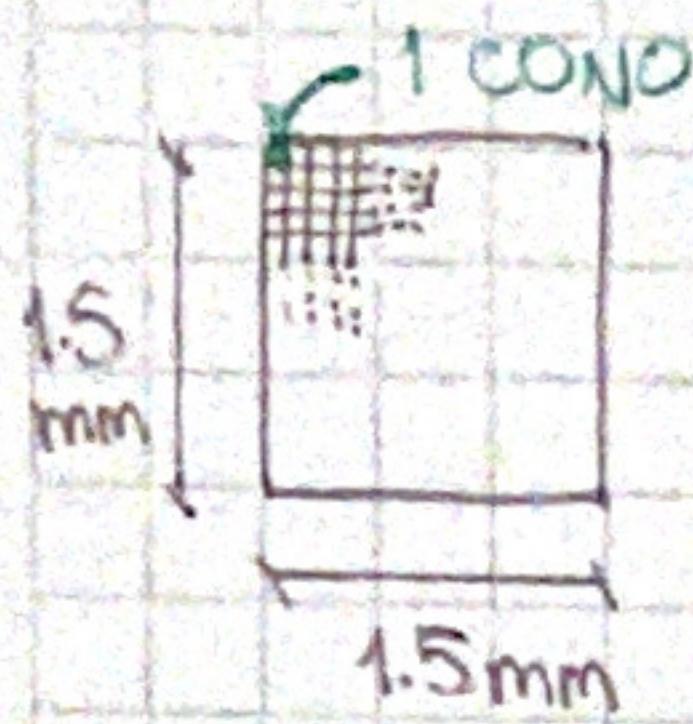


## EJERCICIO 1

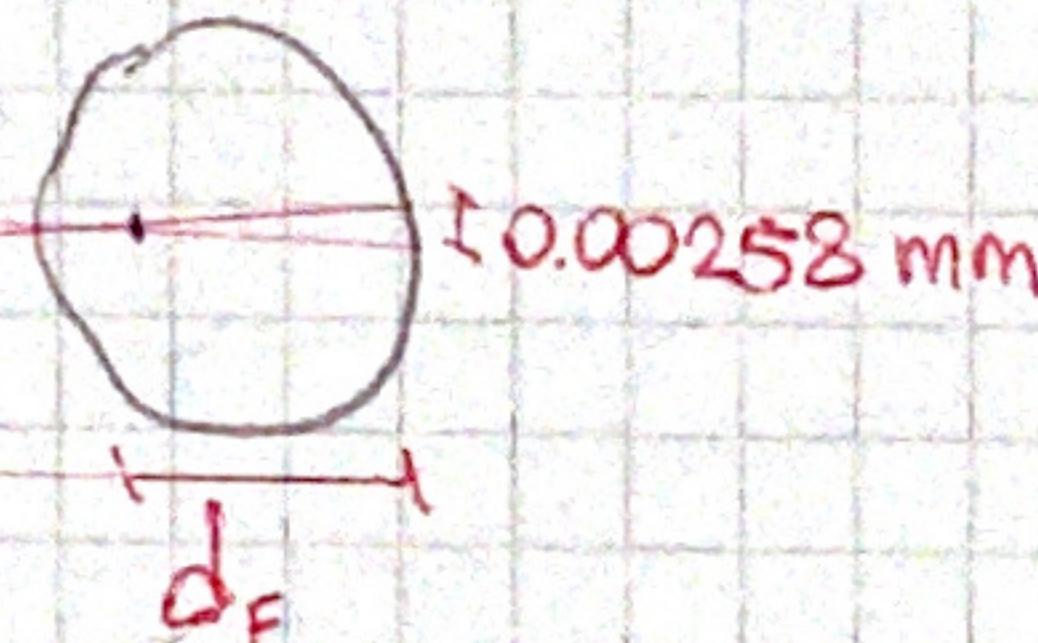
- Se modela al sensor como un cuadrado compuesto por conos de base cuadrada.



$$\text{CANTIDAD DE CONOS: } 150\,000 \frac{1}{\text{mm}^2} \cdot 1.5\text{ mm} \cdot 1.5\text{ mm} = 337\,500$$

$$\text{CANTIDAD DE CONOS EN UN LADO DEL SENSOR: } \sqrt{337\,500} \approx 581$$

$$\text{LADO DE CADA CONO: } \frac{1.5\text{ mm}}{581} \approx 0.00258\text{ mm}$$



$$\frac{h}{30\text{ cm}} = \frac{0.00258\text{ mm}}{d_f}$$

El celular está 30 cm  $\Rightarrow$  De asume foco cercano como  $d_f = 14\text{ mm}$

$$\Rightarrow h \approx \frac{0.00258}{14} \cdot 300\text{ cm} \approx 0.0553\text{ mm}$$

- Un celular moderno (ejemplo iPhone 12) tiene pixeles de tamaño:

$$\frac{1}{460\text{ ppi}} = 0.002174\text{ inch} \approx 0.0552\text{ mm} \leftarrow \begin{array}{l} \text{MUY CERCANO Y POR DEBAJO} \\ \text{DEL RESULTADO ANTERIOR} \end{array}$$

## EJERCICIO 2

- Se asume lo mismo que en el ejercicio anterior.

$$\text{CANT. DE PIXELES EN UN LADO DEL CCD: } \sqrt{1\,000\,000} = 1000$$

$$\text{LADO DE CADA PIXEL: } \frac{10\text{ mm}}{1000} = 0.01\text{ mm}$$

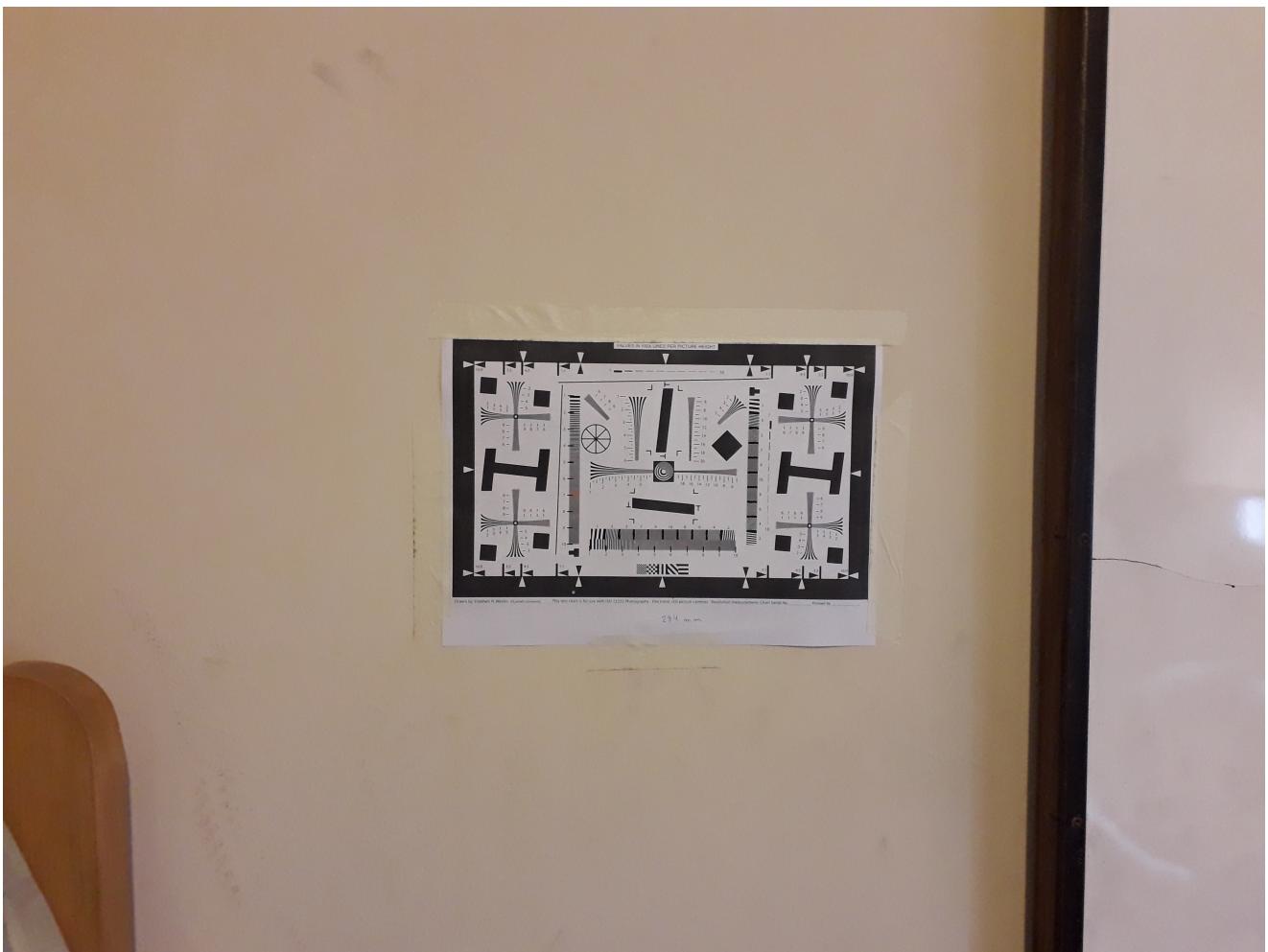
$$\bullet \frac{h}{1\text{ m}} = \frac{0.01\text{ mm}}{d_f} \Rightarrow h = \frac{0.01\text{ mm}}{35\text{ mm}} \cdot 1000\text{ mm} \approx 0.2857\text{ mm}$$

## Ejercicio 03

Para la medición de la resolución real de una cámara se utilizó un celular Samsung Galaxy J4, y según sus especificaciones posee una resolución de 13MP en la cámara trasera. Además, para las mediciones se empleó un calibre de precisión 0,02mm.

### Sosteniendo la cámara con las manos

Se saca una fotografía con la cámara a un metro de distancia del abaco de referencia, sosteniendo la cámara con las manos, es decir sin utilizar ningún tipo de apoyo estable. Observando la imagen resultante, se observa que la cámara deja de resolver aquellas líneas que se encuentran en la zona delimitada por las marcas 4 y 5, y se procede a medir el espesor de tales líneas.



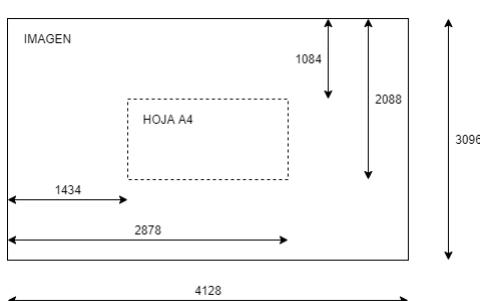
Entonces, se midió que el espesor de la línea es,

$$l = 0,22\text{mm}$$

Por otro lado, la hoja A4 mide 210mm x 297mm, por ende asumiendo que los píxeles son cuadrados, se puede estimar la cantidad de píxeles que cubren la hoja dentro de la imagen.

$$\text{pixeles en hoja} = \frac{210\text{mm} \cdot 297\text{mm}}{(0,22\text{mm})^2} = 1,289\text{MP}$$

Normalmente, para calcular la resolución de la cámara, debería utilizarse el tamaño que abarca la imagen, no obstante como ocupaba un área mucho mayor que la hoja utilizada, buscar medir el área abarcada podía resultar en una tarea poco precisa y tediosa. Por ende, se decidió estimar qué área de la imagen ocupa la hoja, de esta forma se puede realizar una regla de tres simples para poder conocer la cantidad de píxeles de la imagen a partir de los píxeles de la hoja.



Entonces, el área de la hoja A4 con respecto al área que abarca la imagen es,

$$\alpha = \frac{(2088 - 1084) \cdot (2878 - 1434)}{4128 \cdot 3096} = 0,1134$$

Finalmente, como la cantidad de píxeles de la hoja cubren el 11,34% de la imagen, se deduce que la imagen posee,

$$\text{pixeles en imagen} = \frac{1,289\text{MP}}{0,1134} = 11,367\text{MP}$$

### Sosteniendo la cámara con apoyo estable

Se saca una fotografía con la cámara a un poco más de un metro de distancia del abaco de referencia, sosteniendo la cámara con un soporte para estabilizar la cámara y empleando un temporizador. Observando la imagen resultante, se observa que la cámara deja de resolver aquellas líneas que se encuentran en la zona delimitada por las marcas 3 y 4. Se pueden notar dos diferencias esenciales con respecto al experimento

anterior, por un lado las líneas las deja de resolver en una marca más gruesa, eso se debe a que la ubicación de la cámara estaba a una distancia mayor. Por otro lado, si se observa dónde deja de resolver las líneas horizontalmente y verticalmente, los lugares coinciden, y por ende resulta razonable asumir que los pixeles son cuadrados. Esto no sucedía cuando se sostenía con las manos la cámara, lo cual se debe a que la cámara no estaba estable y el resultado se vió afectado por el movimiento ligero de la misma.



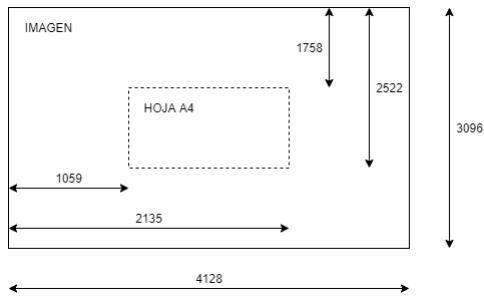
Entonces, se midió que el espesor de la línea es,

$$l = 0,28\text{mm}$$

Por otro lado, la hoja A4 mide 210mm x 297mm, por ende asumiendo que los pixeles son cuadrados, se puede estimar la cantidad de pixeles que cubren la hoja dentro de la imagen.

$$\text{pixeles en hoja} = \frac{210\text{mm} \cdot 297\text{mm}}{(0,28\text{mm})^2} = 795,54kP$$

Normalmente, para calcular la resolución de la cámara, debería utilizarse el tamaño que abarca la imagen, no obstante como ocupaba un área mucho mayor que la hoja utilizada, buscar medir el área abarcada podía resultar en una tarea poco precisa y tediosa. Por ende, se decidió estimar qué área de la imagen ocupa la hoja, de esta forma se puede realizar una regla de tres simples para poder conocer la cantidad de pixeles de la imagen a partir de los pixeles de la hoja.



Entonces, el área de la hoja A4 con respecto al área que abarca la imagen es,

$$\alpha = \frac{(2135 - 1059) \cdot (2522 - 1758)}{4128 \cdot 3096} = 0,0643$$

Finalmente, como la cantidad de pixeles de la hoja cubren el 11,34% de la imagen, se deduce que la imagen posee,

$$\text{pixeles en imagen} = \frac{795,54kP}{0,0643} = 12,37MP$$

## Conclusión

La medición de la resolución fue más cercana al valor especificado por el fabricante del celular cuando se empleó un soporte que mantuviera estable la cámara, que resulta razonable dado que de esta forma no hay movimientos que afecten a la imagen capturada e introduzcan errores. Por otro lado, el valor resultante es menor que el valor nominal de la resolución, esto se podría deber a que en realidad la cámara no posee esa resolución ya sea por desperfectos en la manufactura o bien por desgaste durante su uso, pero también podría ser un problema del experimento. Esto último se debe a que, por ejemplo, al medir el espesor de la línea se haya cometido error porque aunque se utilizó un elemento de precisión, el tamaño de la línea es tal que se pudo haber producido error de medición humano.

## Ejercicio 04

```
In [2]: import numpy as np  
In [3]: import matplotlib.pyplot as plt
```

### Funciones

En las siguientes celdas se desarrollan las funciones para crear las imágenes.

```
In [4]: def create_image(internal_left: int, internal_right: int, external_left: int, external_right: int) -> np.array:  
    """ Creates an image with two boxes with internal squares using gray scale  
    @param internal_left Color of the internal pixel of the left box  
    @param internal_right Color of the internal pixel of the right box  
    @param external_left Color of the external pixels of the left box  
    @param external_right Color of the external pixels of the right box  
    @return Image numpy array  
    """  
  
    # Constant parameters used to create the image  
    IMAGE_COLS = 2  
    BOX_SHAPE = 3  
  
    # Create the image numpy array and fill it with zeros  
    image = np.zeros((BOX_SHAPE, BOX_SHAPE * IMAGE_COLS))  
    for k in range(IMAGE_COLS):  
        # Coordinate of the current box center  
        central_coordinates = (BOX_SHAPE // 2, BOX_SHAPE // 2 + k * BOX_SHAPE)  
  
        for i in range(BOX_SHAPE):  
            for j in range(BOX_SHAPE):  
                # Pixel coordinates  
                x_coord = i  
                y_coord = j + BOX_SHAPE * k  
                # Pixel colors  
                internal_color = internal_left * (1 - k) + internal_right * k  
                external_color = external_left * (1 - k) + external_right * k  
                # Draw the image pixel  
                image[x_coord, y_coord] = internal_color if (x_coord, y_coord) == central_coordinates else external_color  
  
    # Return the resulting image  
    return image  
  
In [5]: def create_images_from_array(descriptors: list) -> list:  
    """ Creates images from the descriptors given for each of them, these descriptors  
    should contain the internal and external colors for the images.  
    @param descriptors List of (internal_left, internal_right, external_left, external_right) colors for the image  
    @return List of image numpy arrays  
    """  
  
    return [create_image(*descriptor) for descriptor in descriptors]  
  
In [6]: def show_experiment(title, descriptor_one, descriptor_two):  
    """ Creates two images from their descriptors and plots them together side by side  
    to compare the effects on the human visual system.  
    @param title Title for the plot  
    @param descriptor_one Descriptor of the first image  
    @param descriptor_two Descriptor of the second image  
    """  
  
    # Create both images  
    image_one = create_image(*descriptor_one)  
    image_two = create_image(*descriptor_two)  
    # Create subplot and plot the title  
    fig, ax = plt.subplots(1, 2, figsize=(20, 5))  
    fig.suptitle(title, fontsize=18)  
    # Plot the first image  
    ax[0].imshow(image_one, cmap='gray', vmin=0, vmax=255)  
    ax[0].axis('off')  
    ax[0].set_title(f'[1º Escenario] Interior Izquierda {descriptor_one[0]} - Interior Derecha {descriptor_one[1]}', fontsize=15)  
    # Plot the second image  
    ax[1].imshow(image_two, cmap='gray', vmin=0, vmax=255)  
    ax[1].axis('off')  
    ax[1].set_title(f'[2º Escenario] Interior Izquierda {descriptor_two[0]} - Interior Derecha {descriptor_two[1]}', fontsize=15)
```

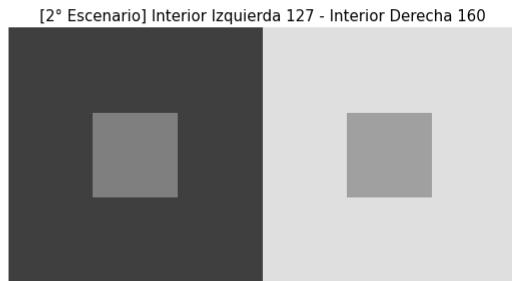
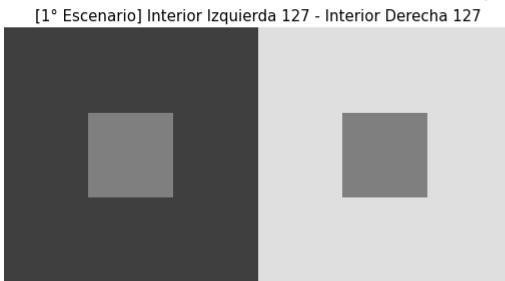
### Experimentos

En las siguientes celdas se prueban diferentes casos de luminancia interna y externa para ambos cuadrados, y se generan las imágenes para visualizar los efectos. Para entender los casos presentados y cómo leerlos, es necesario tener en cuenta que,

- Un escenario es una imagen donde se ponen lado a lado dos bloques de 9x9 píxeles, cuyos píxeles internos son distintos de los vecinos
- Cada caso presentado es la comparación de dos escenarios, donde ambos escenarios poseen los mismos fondos o píxeles vecinos
- El primer escenario muestra cuando las **luminancias físicas** son iguales en los píxeles internos
- El segundo escenario muestra cuando las **luminancias percibidas** son iguales en los píxeles internos

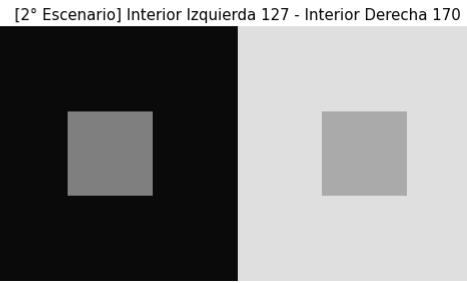
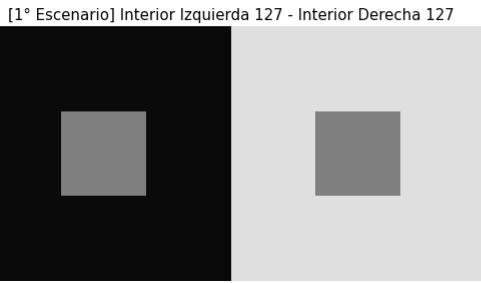
```
In [7]: show_experiment('Caso 1 - Exterior izquierda 63 - Exterior derecha 223', (127, 127, 63, 223), (127, 160, 63, 223))
```

Caso 1 - Exterior izquierda 63 - Exterior derecha 223



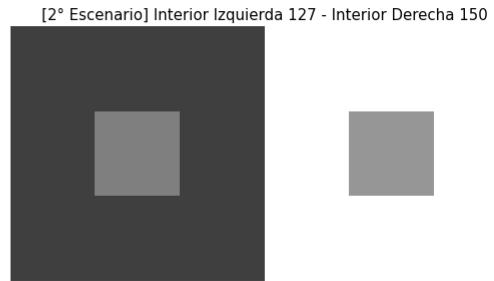
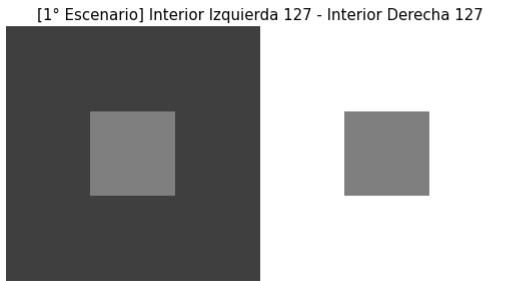
```
In [8]: show_experiment('Caso 2 - Exterior izquierda 10 - Exterior derecha 223', (127, 127, 10, 223), (127, 170, 10, 223))
```

Caso 2 - Exterior izquierda 10 - Exterior derecha 223



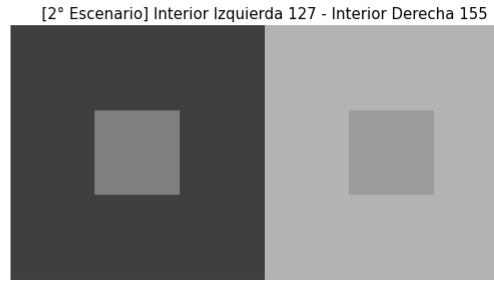
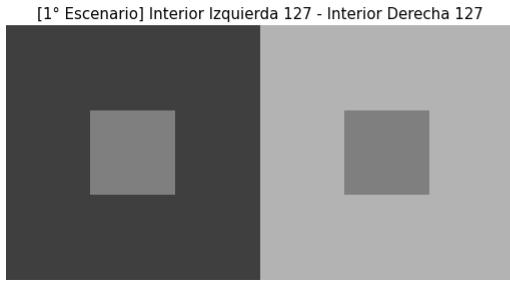
In [9]: `show_experiment('Caso 3 - Exterior izquierda 63 - Exterior derecha 255', (127, 127, 63, 255), (127, 150, 63, 255))`

Caso 3 - Exterior izquierda 63 - Exterior derecha 255



In [24]: `show_experiment('Caso 4 - Exterior izquierda 63 - Exterior derecha 180', (127, 127, 63, 180), (127, 155, 63, 180))`

Caso 4 - Exterior izquierda 63 - Exterior derecha 180



## Conclusiones

En general, se puede observar que por poseer diferente color de fondo, el contraste provoca que la luminancia percibida de los bloques interiores sea diferente, así es como el sistema visual del ser humano percibe que dos bloques que físicamente tienen la misma luminancia, tienen diferente intensidad. En cada experimento, se buscó modificar la luminancia interior del bloque de la derecha hasta percibirlo igual que el bloque izquierdo, de esta forma cuando las intensidades percibidas coinciden, las intensidades física son diferentes.

Por otro lado, se replicó este experimento para diferentes escenarios en los cuales se fue utilizando un fondo distinto. Resultó ser que cuando se cambiaba el fondo, el contraste entre los pixeles era distinto y cambiaba cómo se percibía la diferencia de luminancia entre los bloques. Esto último se puede notar a partir del hecho de que para poder percibir las mismas luminancias, el ajuste realizado cambiaba cuando se cambiaba el fondo. Cuando el fondo de la izquierda se hace más oscuro, entonces es necesario aumentar más la luminancia del bloque de la derecha para que sean percibidos iguales.

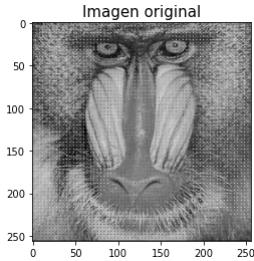
Vale aclarar, que los resultados de estos experimentos se basan en la percepción visual del ser humano, es decir, tienen una base subjetiva.

## Ejercicio 05

```
In [2]: import numpy as np  
In [3]: import matplotlib.pyplot as plt  
In [4]: import math
```

### Lectura de la imagen original

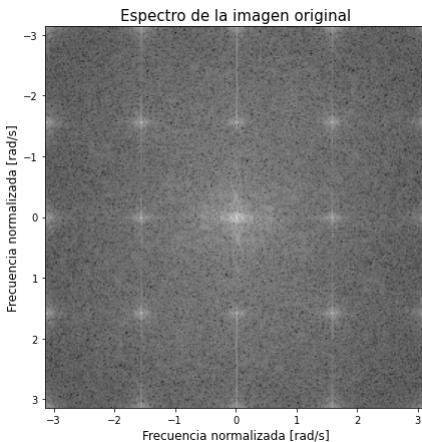
```
In [5]: from skimage.io import imread  
In [6]: from skimage.io import imsave  
In [7]: original_image = imread('../resources/mono.bmp')  
In [8]: plt.imshow(original_image, cmap='gray')  
plt.title('Imagen original', fontsize=15)  
plt.show()
```



### Espectro de la imagen original

Se calcula la transformada rápida de Fourier en dos dimensiones para la imagen original.

```
In [10]: spectrum_original = np.fft.fftshift(np.fft.fft2(original_image, s=(256, 256)))  
  
In [11]: plt.figure(figsize=(15, 7))  
plt.imshow(np.log10(np.abs(spectrum_original)), cmap='gray', extent=[-np.pi, np.pi, -np.pi, np.pi])  
plt.xlabel('Frecuencia normalizada [rad/s]', fontsize=12)  
plt.ylabel('Frecuencia normalizada [rad/s]', fontsize=12)  
plt.title('Espectro de la imagen original', fontsize=15)  
plt.show()
```

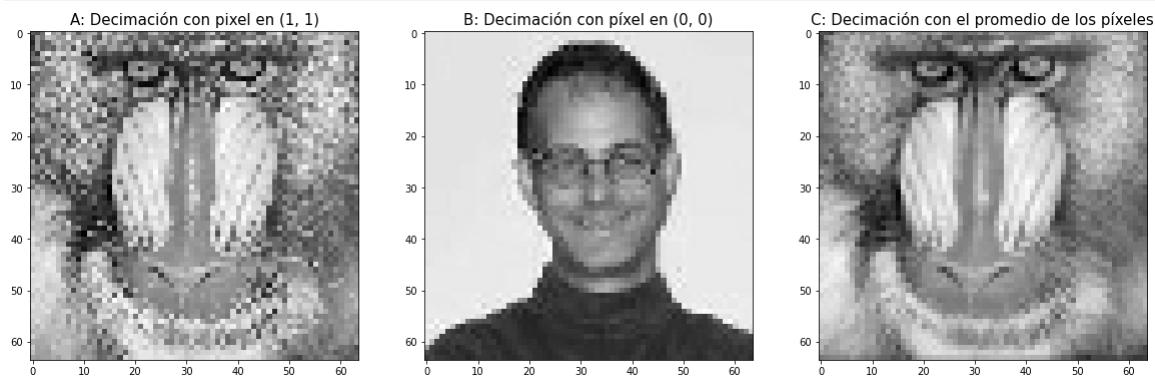


### Decimación de la imagen

La **decimación** consiste en disminuir la frecuencia de muestreo, es decir, disminuir la cantidad de muestras que en este caso corresponden a píxeles de la imagen. Además, se puede aplicar algún procesamiento particular para determinar cómo generar los píxeles nuevos. En esta sección, se aplicará decimación escogiendo píxeles de posiciones particulares, o encontrando el valor promedio de los píxeles.

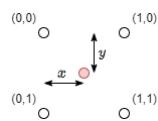
```
In [12]: from skimage.measure import block_reduce  
  
In [13]: def extract_position(block: np.array, x: int, y: int):  
    """ Returns a specific position from the given pixel block.  
    @param block Pixel block of arbitrary shape  
    @param x Coordinate for the selected pixel in the pixel block  
    @param y Coordinate for the selected pixel in the pixel block  
    @return Returns the pixel content  
    """  
    if x >= block.shape[0] or y >= block.shape[1]:  
        raise ValueError('Coordinates out of the block')  
    return block[x, y]  
  
def extract_position_from_blocks(blocks: np.array, x: int, y: int) -> np.array:  
    """ Receives a numpy array of arbitrary shape containing pixel blocks of a given size  
    and extracts a particular position or pixel. Returns the same numpy array of arbitrary  
    shape replacing each pixel block with the corresponding extracted pixel.  
    @param blocks Numpy array of pixel blocks  
    @param x Coordinate for the selected pixel in the pixel block  
    @param y Coordinate for the selected pixel in the pixel block  
    @return Returns a numpy array  
    """  
    new_blocks = np.zeros((blocks.shape[0], blocks.shape[1]))  
    for i in range(blocks.shape[0]):  
        for j in range(blocks.shape[1]):  
            new_blocks[i, j] = extract_position(blocks[i, j], x, y)  
    return new_blocks  
  
In [14]: image_a = block_reduce(original_image, block_size=(4, 4), func=lambda blocks, axis: extract_position_from_blocks(blocks, 1, 1))  
In [15]: image_b = block_reduce(original_image, block_size=(4, 4), func=lambda blocks, axis: extract_position_from_blocks(blocks, 0, 0))  
In [16]: image_c = block_reduce(original_image, block_size=(4, 4), func=np.mean)
```

```
In [17]: fig, ax = plt.subplots(1, 3, figsize=(20, 10))
ax[0].imshow(image_a, cmap='gray')
ax[0].set_title('A: Decimación con pixel en (1, 1)', fontsize=15)
ax[1].imshow(image_b, cmap='gray')
ax[1].set_title('B: Decimación con pixel en (0, 0)', fontsize=15)
ax[2].imshow(image_c, cmap='gray')
ax[2].set_title('C: Decimación con el promedio de los pixeles', fontsize=15)
plt.show()
```



## Interpolación bilineal

La interpolación bilineal consiste en encontrar el valor de los píxeles agregados por el proceso de upsampling en una imagen, utilizando los cuatro píxeles conocidos más cercanos que lo rodean. Los píxeles conocidos se emplean realizando una interpolación lineal en dos pasos, primero en el eje de coordenadas x y luego en el eje de coordenadas y. Para ello, se transforman las coordenadas según se muestra en la siguiente ilustración,



Así, el interpolador se puede pensar como un campo escalar que determina el valor del píxel de acuerdo a la ubicación a interpolar, según se muestra en la siguiente expresión,

$$f(x, y) = f(0, 0) \cdot (1 - x) \cdot (1 - y) + f(1, 0) \cdot x \cdot (1 - y) + f(0, 1) \cdot (1 - x) \cdot y + f(1, 1) \cdot x \cdot y$$

Escrito matricialmente, se obtiene que,

$$f(x, y) = X \cdot F \cdot Y = (1 - x \quad x) \cdot \begin{pmatrix} f(0, 0) & f(0, 1) \\ f(1, 0) & f(1, 1) \end{pmatrix} \cdot \begin{pmatrix} 1 - y \\ y \end{pmatrix}$$

```
In [18]: def bilinear_interpolator(image, ratio):
    """ Upsamples an image applying the bilinear interpolation.
        @param image Original image
        @param ratio Upsampling factor
        @return Interpolated image
    """
    # Fetch the original shape and create the new shape
    input_x, input_y = image.shape
    output_x = int(input_x * ratio)
    output_y = int(input_y * ratio)
    # Create the new image with zeros
    output_image = np.zeros((output_x, output_y))
    for i in range(output_x):
        for j in range(output_y):
            # Coordinates measured from the original image
            aux_x = i / ratio
            aux_y = j / ratio
            # Search for the 4 points used for the interpolation
            x1 = int(aux_x)
            y1 = int(aux_y)
            x2 = x1
            y2 = y1 + 1
            x3 = x1 + 1
            y3 = y1
            x4 = x1 + 1
            y4 = y1 + 1
            x = aux_x - int(aux_x)
            y = aux_y - int(aux_y)
            # When the iteration is already in the limit of the image
            if x4 > input_x:
                x4 = input_x - 1
                x3 = x4
                x2 = x4 - 1
                x1 = x2
            if y4 > input_y:
                y4 = input_y - 1
                y3 = y4
                y2 = y4 - 1
                y1 = y2
            # Formula for bilinear interpolation
            X = np.array([[1-x, x]])
            F = np.array([[image[x1, y1], image[x2, y2], [image[x3, y3], image[x4, y4]]]])
            Y = np.array([[1-y, y]]).T
            output_image[i, j] = X @ F @ Y
    return output_image
```

```
In [19]: image_a_bilinear = bilinear_interpolator(image_a, 4)
```

```
In [20]: image_b_bilinear = bilinear_interpolator(image_b, 4)
```

```
In [21]: image_c_bilinear = bilinear_interpolator(image_c, 4)
```

```
In [22]: fig, ax = plt.subplots(1, 3, figsize=(20, 10))
ax[0].imshow(image_a_bilinear, cmap='gray')
ax[0].set_title('Imagen A con interpolación bilineal', fontsize=15)
ax[1].imshow(image_b_bilinear, cmap='gray')
ax[1].set_title('Imagen B con interpolación bilineal', fontsize=15)
ax[2].imshow(image_c_bilinear, cmap='gray')
ax[2].set_title('Imagen C con interpolación bilineal', fontsize=15)
plt.show()
```



## Espectro de interpolaciones bilineales

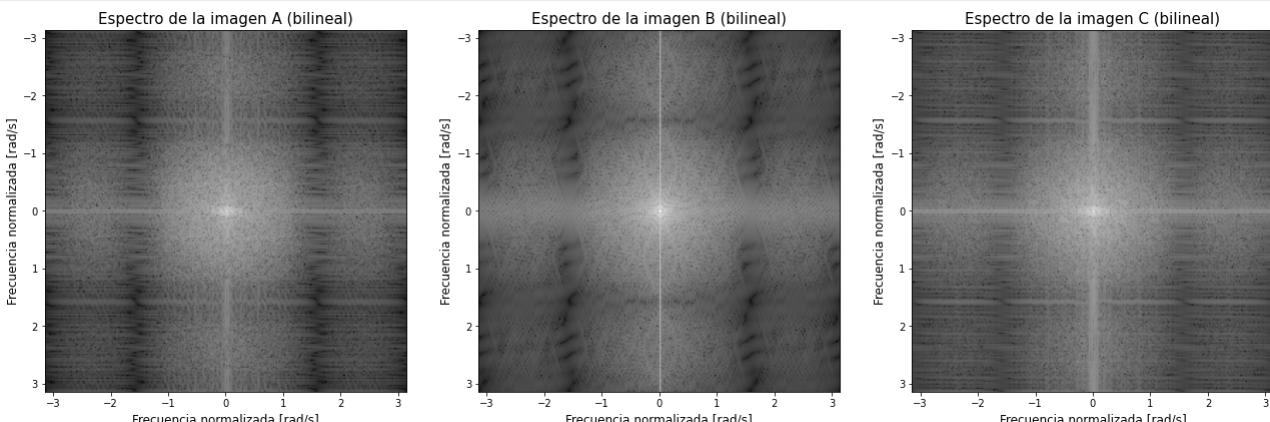
Se calcula la transformada rápida de Fourier para las imágenes resultantes de la interpolación bilineal.

```
In [23]: spectrum_bilinear_a = np.fft.fftshift(np.fft.fft2(image_a_bilinear, s=(256, 256)))
```

```
In [24]: spectrum_bilinear_b = np.fft.fftshift(np.fft.fft2(image_b_bilinear, s=(256, 256)))
```

```
In [25]: spectrum_bilinear_c = np.fft.fftshift(np.fft.fft2(image_c_bilinear, s=(256, 256)))
```

```
In [26]: fig, ax = plt.subplots(1, 3, figsize=(22, 15))
ax[0].set_title('Espectro de la imagen A (bilineal)', fontsize=15)
ax[0].imshow(np.log10(np.abs(spectrum_bilinear_a)), cmap='gray', extent=[-np.pi, np.pi, -np.pi, -np.pi])
ax[0].set_xlabel('Frecuencia normalizada [rad/s]', fontsize=12)
ax[0].set_ylabel('Frecuencia normalizada [rad/s]', fontsize=12)
ax[1].set_title('Espectro de la imagen B (bilineal)', fontsize=15)
ax[1].imshow(np.log10(np.abs(spectrum_bilinear_b)), cmap='gray', extent=[-np.pi, np.pi, -np.pi, -np.pi])
ax[1].set_xlabel('Frecuencia normalizada [rad/s]', fontsize=12)
ax[1].set_ylabel('Frecuencia normalizada [rad/s]', fontsize=12)
ax[2].set_title('Espectro de la imagen C (bilineal)', fontsize=15)
ax[2].imshow(np.log10(np.abs(spectrum_bilinear_c)), cmap='gray', extent=[-np.pi, np.pi, -np.pi, -np.pi])
ax[2].set_xlabel('Frecuencia normalizada [rad/s]', fontsize=12)
ax[2].set_ylabel('Frecuencia normalizada [rad/s]', fontsize=12)
plt.show()
```



## Interpolación bicúbica

A diferencia de la bilineal, la interpolación bicubica ofrece en general mejores resultados debido a que tiene en cuenta una mayor cantidad de datos; en particular, utiliza los 16 puntos más cercanos para estimar el valor del punto a interpolar. El método elegido para realizarla es mediante un kernel definido por la siguiente función:

$$W(x) = \begin{cases} (a+2)|x|^3 - (a+3)|x|^2 + 1 & \text{for } |x| \leq 1, \\ a|x|^3 - 5a|x|^2 + 8a|x| - 4a & \text{for } 1 < |x| < 2, \\ 0 & \text{otherwise,} \end{cases}$$

En el cual  $x$  es una distancia desde el punto a interpolar, con alguno de los 16 puntos de referencia, y  $a$  es un parámetro de ajuste del kernel, definido por el usuario.

Las distancias en  $x$  e  $y$  son utilizadas para calcular, mediante la función del kernel, los valores que multiplicaran matricialmente a las intensidades de los 16 pixeles más cercanos, como se muestra a continuación:

$$\begin{pmatrix} k(x_0) & k(x_1) & k(x_2) & k(x_3) \end{pmatrix} \cdot \begin{pmatrix} f(x_0, y_0) & f(x_0, y_1) & f(x_0, y_2) & f(x_0, y_3) \\ f(x_1, y_0) & f(x_1, y_1) & f(x_1, y_2) & f(x_1, y_3) \\ f(x_2, y_0) & f(x_2, y_1) & f(x_2, y_2) & f(x_2, y_3) \\ f(x_3, y_0) & f(x_3, y_1) & f(x_3, y_2) & f(x_3, y_3) \end{pmatrix} \cdot \begin{pmatrix} k(y_0) \\ k(y_1) \\ k(y_2) \\ k(y_3) \end{pmatrix}$$

```
In [27]: def k(x):
    """ Computes the kernel operation for an input x which is the distance from the interpolation point to the reference point.
    @param x    Distance from the interpolation point to the reference point.
    @return     Result of the computation of the kernel operation.
    """
    a = -0.75
    ret = 0
    absx = abs(x)

    if absx <= 1:
        ret = (a+2)*absx**3 - (a+3)*absx**2 + 1
    elif absx < 2:
        ret = a*absx**3 - 5*a*absx**2 + 8*a*absx - 4*a

    return ret

def padding(img, H, W):
    """ Adds 2 rows and 2 cols to each side of the image with the same values as the edges, for the kernel computations to be possible.
    @param img  Original image.
    @param H    Height of the original image.
    @param W    Width of the original image.
    @return    Padded image.
    """

```

```

"""
p_img = np.zeros((H+4, W+4))
p_img[2:H+2, 2:W+2] = img

# Pad the first/last two col and row
p_img[2:H+2, 0:2] = img[:, 0:1]
p_img[2:H+2, W+2:W+4] = img[:, W-1:W]
p_img[H+2:H+4, 2:W+2] = img[H-1:H, :]
p_img[0:2, 2:W+2] = img[0:1, :]

# Pad the missing sixteen points
p_img[0:2, 0:2] = img[0, 0]
p_img[0:2, W+2:W+4] = img[0, W-1]
p_img[H+2:H+4, 0:2] = img[H-1, 0]
p_img[H+2:H+4, W+2:W+4] = img[H-1, W-1]

return p_img
"""

def bicubic_interpolator(image, ratio):
    """ Upsamples an image applying the bicubic interpolation.
        @param image Original image
        @param ratio Upsampling factor
        @return Interpolated image
    """
    # Getting image size
    H, W = image.shape

    # Padding image
    img = padding(image, H, W)

    # Getting size for new image
    dH = math.floor(H * ratio)
    dW = math.floor(W * ratio)

    # Creating new image
    dst = np.zeros((dH, dW))

    h = 1/ratio

    for j in range(dH):
        for i in range(dW):
            x, y = i * h + 2, j * h + 2

            dx1 = 1 + x - math.floor(x)
            dx2 = x - math.floor(x)
            dx3 = math.floor(x) + 1 - x
            dx4 = math.floor(x) + 2 - x

            dy1 = 1 + y - math.floor(y)
            dy2 = y - math.floor(y)
            dy3 = math.floor(y) + 1 - y
            dy4 = math.floor(y) + 2 - y

            x_vect = np.matrix([[k(dx1), k(dx2), k(dx3), k(dx4)]])
            points_mat = np.matrix([[img[int(y-dy1)], int(x-dx1)],
                                   [img[int(y-dy2)], int(x-dx1)],
                                   [img[int(y-dy3)], int(x-dx1)],
                                   [img[int(y-dy4)], int(x-dx1)],
                                   [img[int(y-dy1)], int(x-dx2)],
                                   [img[int(y-dy2)], int(x-dx2)],
                                   [img[int(y-dy3)], int(x-dx2)],
                                   [img[int(y-dy4)], int(x-dx2)],
                                   [img[int(y-dy1)], int(x-dx3)],
                                   [img[int(y-dy2)], int(x-dx3)],
                                   [img[int(y-dy3)], int(x-dx3)],
                                   [img[int(y-dy4)], int(x-dx3)],
                                   [img[int(y-dy1)], int(x-dx4)],
                                   [img[int(y-dy2)], int(x-dx4)],
                                   [img[int(y-dy3)], int(x-dx4)],
                                   [img[int(y-dy4)], int(x-dx4)]])

            y_vect = np.matrix([
                [k(dy1)], [k(dy2)], [k(dy3)], [k(dy4)]])

            # Computing each value of the interpolated image as two dot matix products between
            # the kernelised x and y vectors, and the reference 16 points
            dst[j, i] = np.dot(np.dot(x_vect, points_mat), y_vect)

    return dst

```

In [28]: `image_a_bicubic = bicubic_interpolator(image_a, 4)`

In [29]: `image_b_bicubic = bicubic_interpolator(image_b, 4)`

In [30]: `image_c_bicubic = bicubic_interpolator(image_c, 4)`

In [31]: `fig, ax = plt.subplots(1, 3, figsize=(20, 10))
ax[0].imshow(image_a_bicubic, cmap='gray')
ax[0].set_title('Imagen A con interpolación bicúbica', fontsize=15)
ax[1].imshow(image_b_bicubic, cmap='gray')
ax[1].set_title('Imagen B con interpolación bicúbica', fontsize=15)
ax[2].imshow(image_c_bicubic, cmap='gray')
ax[2].set_title('Imagen C con interpolación bicúbica', fontsize=15)
plt.show()`



## Espectro de interpolaciones bicúbicas

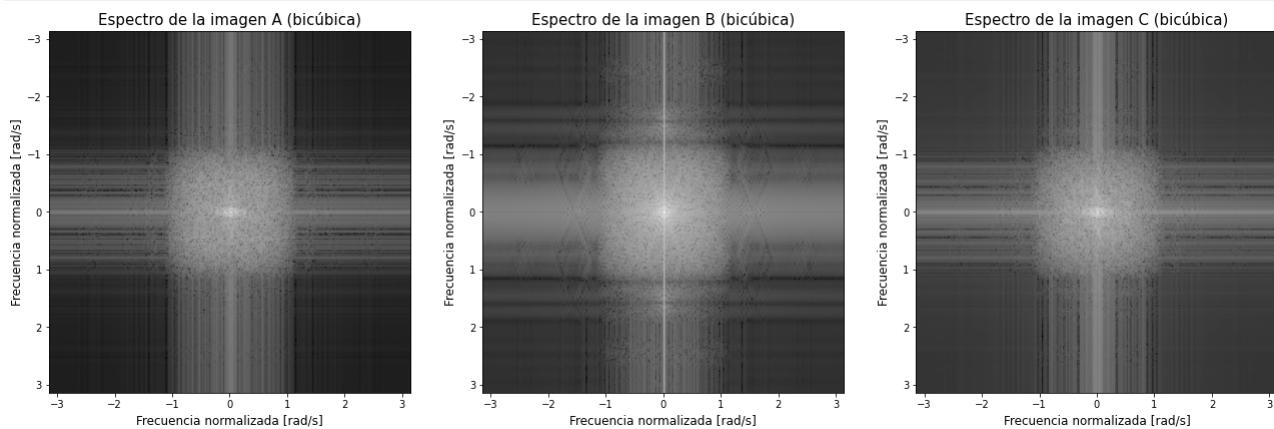
Se calcula la transformada rápida de Fourier para las imágenes resultantes de la interpolación bicúbica.

In [32]: `spectrum_bicubic_a = np.fft.fftshift(np.fft.fft2(image_a_bicubic, s=(256, 256)))`

In [33]: `spectrum_bicubic_b = np.fft.fftshift(np.fft.fft2(image_b_bicubic, s=(256, 256)))`

```
In [34]: spectrum_bicubic_c = np.fft.fftshift(np.fft.fft2(image_c_bicubic, s=(256, 256)))
```

```
In [35]: fig, ax = plt.subplots(1, 3, figsize=(22, 15))
ax[0].set_title('Espectro de la imagen A (bicúbica)', fontsize=15)
ax[0].imshow(np.log10(np.abs(spectrum_bicubic_a)), cmap='gray', extent=[-np.pi, np.pi, np.pi, -np.pi])
ax[0].set_xlabel('Frecuencia normalizada [rad/s]', fontsize=12)
ax[0].set_ylabel('Frecuencia normalizada [rad/s]', fontsize=12)
ax[1].set_title('Espectro de la imagen B (bicúbica)', fontsize=15)
ax[1].imshow(np.log10(np.abs(spectrum_bicubic_b)), cmap='gray', extent=[-np.pi, np.pi, np.pi, -np.pi])
ax[1].set_xlabel('Frecuencia normalizada [rad/s]', fontsize=12)
ax[1].set_ylabel('Frecuencia normalizada [rad/s]', fontsize=12)
ax[2].set_title('Espectro de la imagen C (bicúbica)', fontsize=15)
ax[2].imshow(np.log10(np.abs(spectrum_bicubic_c)), cmap='gray', extent=[-np.pi, np.pi, np.pi, -np.pi])
ax[2].set_xlabel('Frecuencia normalizada [rad/s]', fontsize=12)
ax[2].set_ylabel('Frecuencia normalizada [rad/s]', fontsize=12)
plt.show()
```



## Comparación con OpenCV

En esta sección, se realiza la interpolación bicúbica utilizando el algoritmo provisto por la biblioteca OpenCV, para validar el funcionamiento de la interpolación bicúbica desarrollada.

Si bien no hay demasiado para validar con las imágenes interpoladas con el algoritmo bicúbico desarrollado en este trabajo, y aquel proporcionado por OpenCV, los espectros sirven aquí como herramienta adicional. En las imágenes originales, sencillamente se puede observar que ambos algoritmos funcionan, y que producen resultados cualitativamente muy similares. Sin embargo, la comparación favorable de los espectros logrados (aún cuando en ellos sí se pueden notar las pequeñas diferencias), proporciona más información respecto de la similitud de los resultados obtenidos por ambos algoritmos.

```
In [36]: import cv2
```

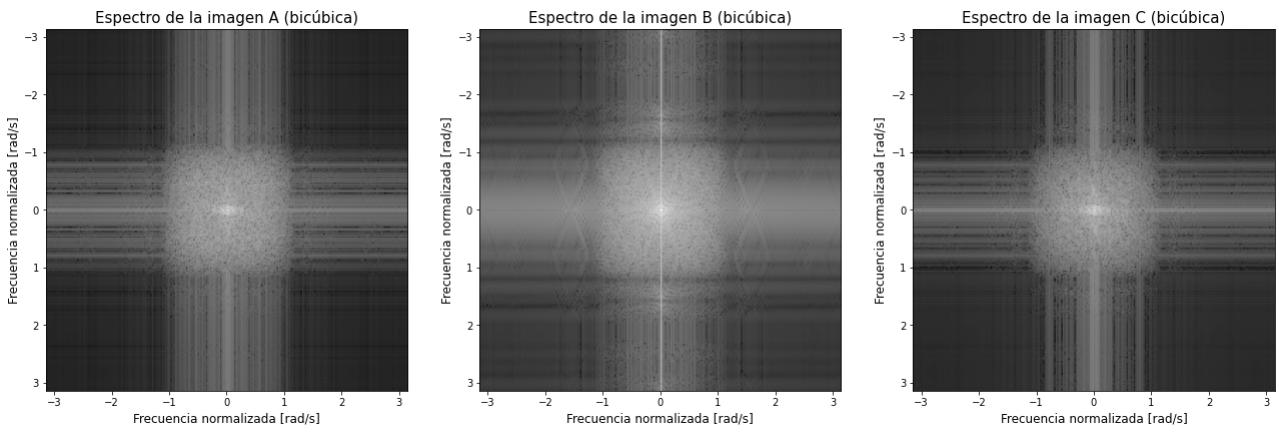
```
In [37]: image_a_cv = cv2.resize(image_a.astype('float32'), original_image.shape, interpolation=cv2.INTER_CUBIC)
image_b_cv = cv2.resize(image_b.astype('float32'), original_image.shape, interpolation=cv2.INTER_CUBIC)
image_c_cv = cv2.resize(image_c.astype('float32'), original_image.shape, interpolation=cv2.INTER_CUBIC)
```

```
In [38]: fig, ax = plt.subplots(1, 3, figsize=(20, 10))
ax[0].imshow(image_a_cv, cmap='gray')
ax[0].set_title('Imagen A con interpolación bicúbica', fontsize=15)
ax[1].imshow(image_b_cv, cmap='gray')
ax[1].set_title('Imagen B con interpolación bicúbica', fontsize=15)
ax[2].imshow(image_c_cv, cmap='gray')
ax[2].set_title('Imagen C con interpolación bicúbica', fontsize=15)
plt.show()
```



```
In [39]: spectrum_cv_a = np.fft.fftshift(np.fft.fft2(image_a_cv, s=(256, 256)))
spectrum_cv_b = np.fft.fftshift(np.fft.fft2(image_b_cv, s=(256, 256)))
spectrum_cv_c = np.fft.fftshift(np.fft.fft2(image_c_cv, s=(256, 256)))
```

```
In [40]: fig, ax = plt.subplots(1, 3, figsize=(22, 15))
ax[0].set_title('Espectro de la imagen A (bicúbica)', fontsize=15)
ax[0].imshow(np.log10(np.abs(spectrum_cv_a)), cmap='gray', extent=[-np.pi, np.pi, np.pi, -np.pi])
ax[0].set_xlabel('Frecuencia normalizada [rad/s]', fontsize=12)
ax[0].set_ylabel('Frecuencia normalizada [rad/s]', fontsize=12)
ax[1].set_title('Espectro de la imagen B (bicúbica)', fontsize=15)
ax[1].imshow(np.log10(np.abs(spectrum_cv_b)), cmap='gray', extent=[-np.pi, np.pi, np.pi, -np.pi])
ax[1].set_xlabel('Frecuencia normalizada [rad/s]', fontsize=12)
ax[1].set_ylabel('Frecuencia normalizada [rad/s]', fontsize=12)
ax[2].set_title('Espectro de la imagen C (bicúbica)', fontsize=15)
ax[2].imshow(np.log10(np.abs(spectrum_cv_c)), cmap='gray', extent=[-np.pi, np.pi, np.pi, -np.pi])
ax[2].set_xlabel('Frecuencia normalizada [rad/s]', fontsize=12)
ax[2].set_ylabel('Frecuencia normalizada [rad/s]', fontsize=12)
plt.show()
```



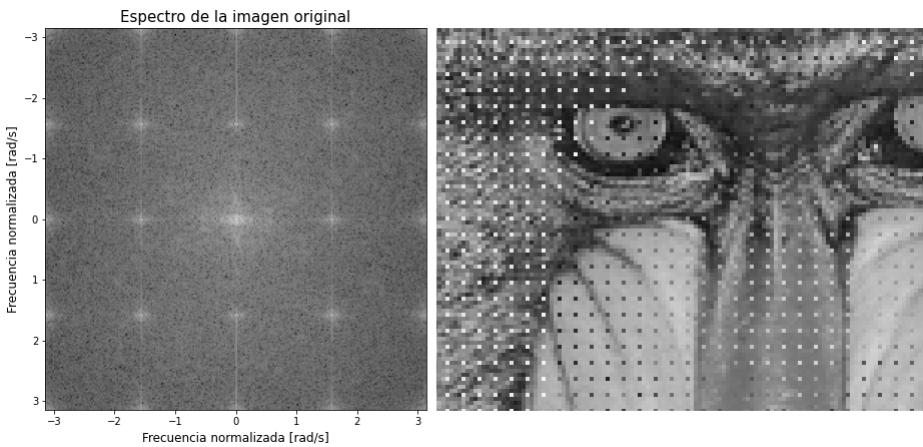
## Conclusiones

### Interpretación del espectro de la imagen original

Cuando se observa el espectro de la imagen original, se observan puntos brillantes ubicados a intervalos regulares, y podrían ser pensados como el equivalente a una delta de dirac en esa frecuencia, aunque por estar aplicando la FFT de una ventana espacial de muestras realmente no se observa la delta. Este resultado es razonable si se observa de cerca la imagen, y se ve que existe un patrón periódico con un período espacial de  $T = 4px$ , es decir, una frecuencia espacial de  $f = \frac{\pi}{2} \approx 1,57$ . Justamente, al tener un patrón periódico con este período y frecuencia en ambas direcciones de la imagen, el resultado es una delta desde ambas orientaciones y provoca un punto brillante en el espectro. Más aún, al ser una forma periódica no senoidal ni cosenoidal, se descompone en una serie de frecuencias armónicas que son las que vemos a intervalos regulares (múltiples de la fundamental).

La razón de ser de este patrón, es justamente el hecho de que dentro de un bloque de 4 píxeles por 4 píxeles, el pixel ubicado en la posición (0,0) pertenece a una imagen diferente que se encuentra escondida dentro de la del mono.

Vale aclarar, que en verdad no es una señal completamente periódica, de hecho en cada repetición del patrón los valores de los píxeles cambian dado el contenido de la imagen del mono y la imagen oculta, no obstante si bien la información del período es diferente, sí hay periodicidad en el patrón con el cual se encuentran. Un análogo con señales temporales sería el multiplexado temporal aplicado en sistemas de comunicaciones para transmitir múltiples canales por un mismo medio, alternando intervalos temporales asignados a cada canal.



### Comparación de las interpolaciones bilineal y bicúbica

Si se comparan las imágenes obtenidas a partir de interpolación bilineal y bicúbica, a primera vista no se puede percibir una diferencia, no obstante, cuando se comparan los espectros sí se puede evidenciar una notable diferencia entre las imágenes. La interpolación bicúbica obtuvo una imagen resultante cuyo espectro está acotado o limitado, mientras que con la interpolación bilineal se ve contenido espectral en casi todas las frecuencias. Esto podría ser interpretado de diversas formas. En primer lugar, la interpolación bicúbica permite estimar píxeles intermedios obteniendo cambios más suaves que con la interpolación bilineal, eso provoca que el contenido de alta frecuencia espacial sea menor que en la bilineal. En segundo lugar, se podría pensar que ese contenido adicional de alta frecuencia que posee la interpolación bilineal es ruido que representa el error en la estimación de los píxeles intermedios al utilizar la bilineal, es decir, que la bicúbica no los presenta porque es más precisa en su estimación.

Resulta interesante pensar que con respecto a aquello que puede percibir el ser humano, las imágenes no presentan diferencias notables, no obstante la imagen obtenida por interpolación bicúbica presenta un espectro considerablemente menor. A priori, uno no conoce el verdadero espectro de la imagen, ya que tanto con la bilineal como la bicúbica se obtienen estimaciones, pero la bicúbica da un resultado idéntico (en este caso) en lo perceptible, y que desde el enfoque de la teoría de la información es más eficiente.

### Comparación de los espectros de cada imagen

Cuando se comparan las tres imágenes obtenidas, la imagen A, B y C, se observa que la imagen B presenta en su espectro una línea vertical centrada en el origen horizontal con una intensidad mayor que en las demás imágenes. Esta línea representa en la orientación vertical un contenido de bajas frecuencias (por eso el espesor pequeño de la línea) y en la orientación horizontal un contenido de altas frecuencias (por eso la extensión de la línea), y está asociada a los bordes de la imagen. Por ende, tiene sentido que sea más intensa en la imagen B, ya que hay un cambio abrupto en la luminosidad de la imagen en el borde de la persona y el fondo. Mientras que en las demás imágenes los cambios son más suaves, sin tanto contraste.