Ejercicio 1

La única diferencia entre nuestro código y el código presente en los ejemplos del tema, es que no hemos tenido en cuenta las imágenes en gris, pues buscamos que estén a color.

El código genera y muestra imágenes con dos tipos de ruido diferentes: ruido sal y pimienta (s&p) y ruido gaussiano. Aquí hay algunas observaciones sobre los resultados obtenidos:

Ruido Sal y Pimienta (S&P):

- A medida que aumenta el valor de amount, se introduce más ruido en la imagen, lo que se traduce en una mayor cantidad de puntos dispersos en la imagen.
- El ruido de sal y pimienta se observarán puntos de colores dispersos aleatoriamente en la imagen, lo que puede afectar la calidad visual de la misma y hacer que parezca granulada.

Ruido Gaussiano:

- Valores de var más altos introducen más ruido en la imagen.

También podemos mencionar que los resultados obtenidos son bastante similares a los hechos en escala de grises.

Ejercicio 2

```python
@adapt_rgb(hsv_value)
def random_noise_hsv(image, *args, **kwargs):
    noisy_image = ski.util.random_noise(image, *args, **kwargs)
    return noisy_image
```

Con respecto al código anterior, hemos creado una función con el decorador rgb y el parámetro hsv_value. Al trabajar en el espacio de color HSV, el ruido se aplica específicamente al componente de valor de la imagen, manteniendo intactos los componentes de tono y saturación. Esto puede ser beneficioso para preservar los colores mientras se introduce el ruido, lo que permite un control más preciso sobre el efecto del ruido en la imagen.

En resumen, el decorador @adapt_rgb(hsv_value) permite aplicar el ruido de manera más controlada, centrada en el brillo de la imagen, lo que ayuda a preservar mejor los colores y mejorar la calidad visual general de la imagen.

Los resultados obtenidos en este ejercicio son considerablemente más suaves que los obtenidos en el anterior. Aquí podemos observar como los puntos de ruido s&p son blancos y negros, a diferencia de los puntos de colores primarios obtenidos anteriormente. Y como la imagen con ruido gaussiano es más ligera y se ve menos borrosa.

En este los colores se preservan mejor en comparación con el espacio de color RGB. Esto significa que los cambios introducidos por el ruido podrían ser menos perceptibles en términos de alteración del color.

Ejercicio 3

```python
# Listas para almacenar los tiempos de ejecución
tiempos_2D = []
tiempos_1D = []
```

El código en su gran mayoría es similar al ya visto en los ejemplos del tema, los principales cambios es que hemos creado 2 listas para poder almacenar los tiempos de ejecución de las 10 veces que se realizan las convulsiones en 1D y 2D.

```
    tiempo_promedio_2D = (fin_2D - inicio_2D) / repeticiones
    tiempos_2D.append(tiempo_promedio_2D)

    tiempo_promedio_1D = (fin_1D - inicio_1D) / repeticiones
    tiempos_1D.append(tiempo_promedio_1D)

tiempos_1D_float = [float(tiempo) for tiempo in tiempos_1D]  # Convierte los strings a float
tiempos_2D_float = [float(tiempo) for tiempo in tiempos_2D]  # Convierte los strings a float
tiempos2D = np.mean(tiempos_2D_float)   # Calcula la media
tiempos1D = np.mean(tiempos_1D_float)   # Calcula la media
```

El código realizado en el ejercicio lleva a cabo comparaciones de tiempos de ejecución entre convoluciones 1D y 2D utilizando máscaras Gaussianas de diferentes tamaños. Para explicar un mejor esta parte:

`tiempo_promedio_2D = (fin_2D - inicio_2D) / repeticiones`: Esto calcula el tiempo promedio que lleva realizar una convolución 2D para una máscara específica. Se toma la diferencia entre el tiempo de finalización y el tiempo de inicio de la convolución, y luego se divide por el número de repeticiones para obtener el tiempo promedio.

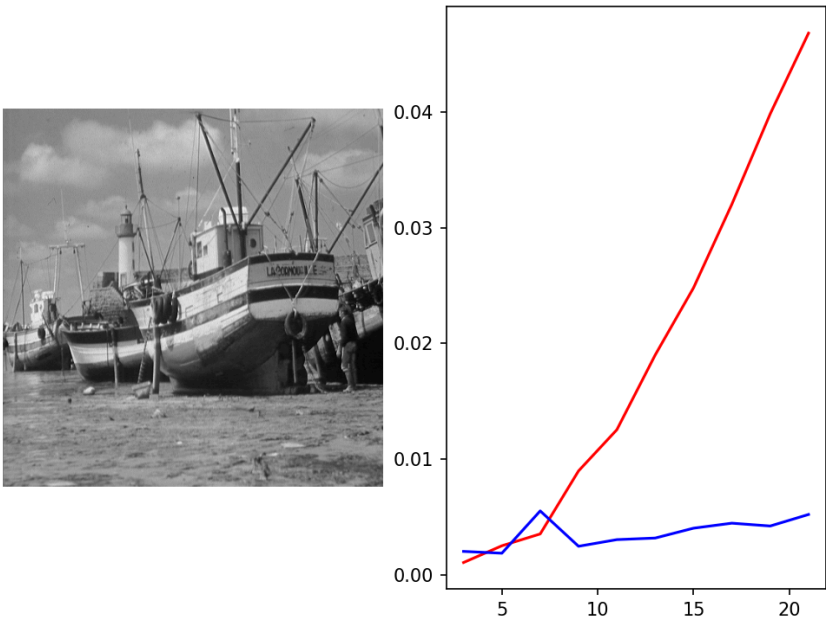
`tiempos_2D.append(tiempo_promedio_2D)`: Agrega el tiempo promedio calculado para la convolución 2D a la lista.

Se hace de igual forma con las convulsiones en 1D.

`tiempos_1D_float` y `tiempos_2D_float`: Convierten los tiempos almacenados en las listas de strings a float para poder realizar cálculos posteriormente.

`tiempos2D = np.mean(tiempos_2D_float)` y `tiempos1D = np.mean(tiempos_1D_float)`: Calculan el tiempo promedio general de ejecución para todas las convoluciones 2D y 1D, utilizando la función `np.mean()`. Estos valores representan el tiempo promedio de ejecución para cada tipo de convolución en todos los tamaños de máscara considerados.

```
¿Obtenemos el mismo resultado? True
Tiempo empleado con máscara  2D: 0.019110100
Tiempo empleado con máscaras 1D: 0.003593934
Factor 2D/1D: 5.32
```



Las gráficas muestran los tiempos de ejecución de las convoluciones 1D y 2D en función del tamaño de la máscara utilizada. Aquí está el análisis de las diferencias entre estos tiempos:

Como podemos observar, los tiempos de ejecución de las convoluciones bidimensionales tienen un tiempo muy superior a las 1D para tamaños de máscara a partir de las 4, aproximadamente. Esta diferencia se hace más notable a medida que aumenta el tamaño de la máscara. Cuanto mayor es el tamaño de la máscara, mayor es la diferencia en los tiempos de ejecución entre las convoluciones 1D y 2D.

Esto sugiere que las convoluciones 1D son más eficientes en términos de tiempo de ejecución en comparación con las convoluciones 2D, especialmente para máscaras de mayor tamaño, pues esta diferencia se hace mayor a partir del tamaño 6.

La relación entre los tiempos de ejecución se puede calcular utilizando el factor 2D/1D. Este indica cuánto más lenta es la convolución 2D en comparación con la convolución 1D para un tamaño de máscara dado.

Ejercicio 4

```
I = ski.io.imread("images/borrosa.png")
I = ski.util.img_as_float(I)

# Aplicar filtro gaussiano con sigma=3
F = ski.filters.gaussian(I, sigma=3)

# Definir el valor de alpha entre 0.5 y 1.5
alpha = 1.0

# Calcular la nueva imagen R
R = I + alpha * (I - F)
```

Este código sí que es un poco más diferente al presentado en los ejemplos. Este realiza un proceso de mejora de imagen utilizando el filtro gaussiano y la fórmula que nos dan.

Para ello cargamos la imagen llamada "borrosa.png" utilizando la función `ski.io.imread()`. La imagen se convierte a flotante utilizando `ski.util.img_as_float()` para permitir operaciones matemáticas. Después, se aplica un filtro gaussiano a la imagen original utilizando la función `ski.filters.gaussian()`. El parámetro `sigma=3` especifica el valor del parámetro sigma para el filtro gaussiano, que controla la desviación estándar de la distribución gaussiana utilizada para suavizar la imagen. Luego se define un valor para el parámetro alpha, que se encuentra en el rango de 0.5 a 1.5. Este parámetro controla la intensidad de la mezcla entre la imagen original y la imagen filtrada. Un valor más alto de alpha dará más peso a la diferencia entre las dos imágenes.

Por último, se calcula una nueva imagen llamada R utilizando la fórmula de mezcla lineal: `R = I + alpha * (I - F)`. Donde `I` representa la imagen original. `F` representa la imagen filtrada con el filtro gaussiano. Y `alpha` es el valor definido anteriormente. Esta fórmula puede ayudar a resaltar características de alto contraste mientras se preserva la estructura general de la imagen original.

Imagen Original (I)

Imagen Resultante (R)

Observando los resultados podemos comentar que la primera imagen mostrada es la imagen original cargada desde el archivo "borrosa.png". Esta imagen como ya lo dice el nombre, se encuentra borrosa, lo que sugiere que podría ser difícil discernir estructuras claras en la imagen.

La imagen resultante (R) es la segunda imagen mostrada, en ella se pueden observar mejoras significativas en la claridad y la nitidez de los detalles. Los bordes y las estructuras que estaban borrosas en la imagen original ahora están más definidos y se pueden distinguir con mayor claridad. Todo esto mientras se mantiene la estructura general y la apariencia global de la escena. Esta imagen hace el efecto opuesto al filtrado gaussiano.

Si modificamos el valor de alpha dentro del rango especificado entre 0.5 y 1.5 se podría ajustar la intensidad de la mejora en la imagen resultante. Por ejemplo, valores más altos de

alpha podrían enfatizar más los detalles recuperados del filtro gaussiano, mientras que valores más bajos podrían conservar más la apariencia original de la imagen.