



INSTITUTO POLITECNICO NACIONAL  
ESCUELA SUPERIOR DE COMPUTO

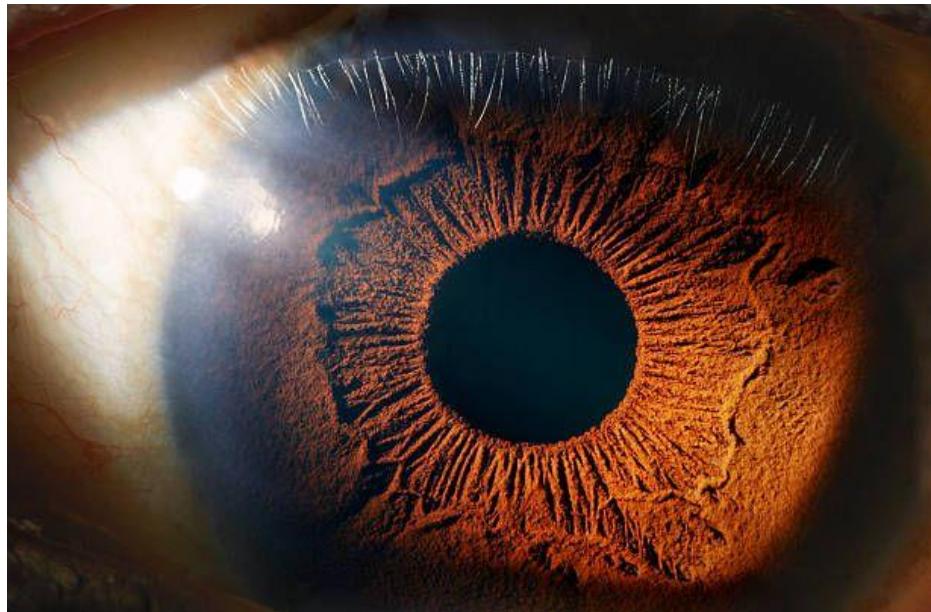


IMAGE ANALYSIS  
MODELOS DE COLOR

ESCUADRON 241:

- PORTOCARRERO RODRIGUEZ HABID
- MUZQUIS PALACIO ERNESTO
- MARTINEZ LOPEZ GERARDO ESTEBAN
- SOLANO MUÑOZ ARTURO

Fecha de entrega: 5.10.2025



## Contenido

Objetivo:	4
Introducción	4
Pillow	4
NumPy	4
Matplotlib	5
Energía	5
Entropía	5
Asímetria	5
Media	6
Varianza	6
Los siguientes son propiedades del histograma:	7
Resultados Obtenidos:	9
Desarrollo	10
1) Resumen Ejecutivo	10
2) Objetivo(s) y Alcance	10
3) Contexto y Motivación	10
4) Dataset y Fuentes de Datos	10
5) Metodología / Pipeline	11
Diagrama simple	11
6) Arquitectura del Proyecto (Estructura de Carpetas)	11
7) Instalación y Requisitos	11
8) Algoritmos utilizados, fragmentos de código en Python y descripción de que hacen las líneas de código	12
Sesión 1. Preparación del entorno e importaciones	12
Sesión 2. Utilidades generales — <code>to_uint8</code>	14
Sesión 3. Métricas e histogramas con resumen en figura — <code>_stats_*</code> y <code>plot_hist_on_axes</code>	15
Sesión 4. Ventana única de histogramas — <code>open_hist_grid</code>	17
Sesión 5. Figura con botón de histogramas — <code>figure_with_hist_button</code>	19
Sesión 6. Pasos base del pipeline — carga, separación, gris BT.601 y binarización	21

Sesión 7. Conversión a flotante y modelo YIQ — to_float01 y convertir_yiq.....	23
Sesión 8. Modelos de color sustractivo y perceptual — convertir_cmy y convertir_hsv .....	25
Sesión 9. Binarización interactiva con slider — figure_binarization_with_slider.....	27
Sesión 10. Demostración secuencial del pipeline — demo(path, umbral).....	29
Sesión 11. Selección de imagen y ejecución principal — seleccionar_imagen y bloque _main_ .....	31
<b>13) Resultados / Visualizaciones .....</b>	<b>33</b>
7. Comparación general y conclusiones parciales .....	41
<b>19) Cómo Reproducir .....</b>	<b>42</b>
Conclusión.....	42

## Objetivo:

El objetivo de esta práctica es implementar un programa en Python capaz de realizar el análisis básico de imágenes digitales mediante el uso de distintos modelos de color (RGB, YIQ, CMY y HSV). A través de la librería *Pillow* para la carga de imágenes, *NumPy* para el manejo matricial, *OpenCV* para las conversiones de color y *Matplotlib* para la visualización de histogramas, se busca comprender cómo cada modelo representa la información cromática y de luminancia. El sistema permite separar canales, convertir entre espacios de color, aplicar umbralización (binarización) y visualizar histogramas para analizar la distribución de intensidades. Con ello, se pretende desarrollar una comprensión práctica de la estructura y comportamiento de las imágenes digitales y de las herramientas fundamentales empleadas en análisis y procesamiento visual computacional.

## Introducción

El análisis de imágenes tiene como objetivo extraer información significativa a partir del contenido de una imagen, lo que facilita la comparación y clasificación de imágenes. Este enfoque es valioso en aplicaciones donde la evaluación humana resulta insuficiente por factores como la necesidad de procesamiento masivo, la incapacidad del ojo humano, entre otros.

La importancia de este análisis radica en su capacidad para convertir percepciones visuales en datos matemáticos manipulables. Mientras que el ojo humano puede percibir cualidades como "brillante", "contrastada" o "detallada", el análisis computacional permite asignar valores numéricos específicos a estas características, estableciendo patrones y umbrales que pueden ser utilizados para la toma de decisiones automatizada.

Anteriormente se ha resuelto con librerías el código presentado en la práctica en las cuales se incluyen las siguientes:

### [Pillow.](#)

Algunas de sus capacidades principales abarcan la lectura y escritura de más de 30 formatos de imagen diferentes, operaciones geométricas como manipulación de píxeles a nivel básico. Pillow es ideal para la etapa de preprocesamiento, permitiendo la extracción de metadatos y la conversión entre modos de color.

### [NumPy.](#)

Proporciona estructuras de datos eficientes para representar imágenes como arreglos multidimensionales. La importancia de NumPy está en su implementación vectorizada de

operaciones matemáticas que permite procesar millones de píxeles simultáneamente sin necesidad de bucles explícitos.

También facilita el cálculo de estadísticas descriptivas mediante funciones optimizadas para media, varianza, desviación estándar, entre otras.

### Matplotlib.

Matplotlib se especializa en la visualización de resultados, ofreciendo herramientas comprehensivas para la generación de gráficos. Su módulo pyplot permite crear histogramas interactivos, visualizar distribuciones de probabilidad y comparar múltiples datasets de manera intuitiva.

En las propiedades de las imágenes se tiene lo siguiente:

### Energía.

Representa la concentración de la distribución de intensidades. Se calcula como la suma de los cuadrados de las probabilidades de cada nivel de intensidad en el histograma. Una energía elevada indica que la imagen está dominada por pocos tonos específicos. Por el contrario, una energía baja sugiere una distribución equilibrada entre múltiples tonos.

$$E = \sum_{g=0}^{L-1} (P(g))^2$$

### Entropía.

La entropía es el desorden presente en una imagen demostrando la impredecibilidad. Se calcula mediante la sumatoria de las probabilidades multiplicadas por el logaritmo de dichas probabilidades. Una entropía alta es una escena rica en detalles y variabilidad. Una entropía baja señala redundancia y predictibilidad.

$$e = -\sum_{g=0}^{L-1} P(g) \log_2 [P(g)]$$

### Asimetría

Es el balance de la distribución del histograma respecto a su valor central. Una asimetría cercana a 0 denota una distribución simétrica, donde los valores se distribuyen equilibradamente

alrededor de la media. Una asimetría positiva indica que la cola de la distribución se extiende hacia valores altos de intensidad. Una asimetría negativa sugiere concentración en tonos brillantes, común en imágenes sobreexpuestas.

$$\alpha = \sum_{g=0}^{L-1} (g - \bar{g})^3 P(g)$$

### Media.

Representa el brillo promedio de la imagen. Valores bajos de media establecida son imágenes oscuras, mientras que valores altos de media son imágenes brillantes. La media sirve como referencia primaria para operaciones de normalización de iluminación y corrección gamma.

$$\bar{g} = \sum_{g=0}^{L-1} g P(g) = \sum_i \sum_j \frac{I(i, j)}{M}$$

### Varianza.

La varianza significa la dispersión de los valores de intensidad alrededor de la media, funcionando como indicador de contraste intrínseco. Una varianza alta señala una amplia distribución de valores de intensidad. Una varianza baja indica compresión del rango tonal o bajo contraste.

$$\sigma^2 = \sum_{g=0}^{L-1} (g - \bar{g})^2 P(g)$$

El proceso comienza cargando la imagen con OpenCV, que por defecto la interpreta en formato BGR. Para mostrarla correctamente suele convertirse a RGB, además de revisar que el archivo no esté dañado, ajustar sus dimensiones y, si es necesario, transformarlo a escala de grises.

Cuando la imagen se mantiene en color, el análisis se realiza por separado en cada canal (rojo, verde y azul). Para cada uno se genera un histograma que refleja la frecuencia de los niveles de intensidad entre 0 y 255. Estos histogramas luego se normalizan para obtener distribuciones de probabilidad que servirán como base para los cálculos estadísticos.

Los valores numéricos se interpretan contrastándolos con umbrales previamente definidos. Esto permite clasificar imágenes según sus características.

Un histograma es una representación gráfica de la distribución de frecuencias de un conjunto de datos. Su estructura consiste en dividir el rango total de valores en intervalos llamados, luego se cuentan los elementos de cada conjunto que caen dentro de cada intervalo.

El resultado se visualiza como un conjunto de barras: la base de cada barra corresponde al intervalo de valores, y la altura representa la frecuencia o cantidad de ocurrencias. En este caso, una barra corresponde a la luminiscencia, una al color rojo, otra al color verde y la última al color azul.

Un histograma muestra la distribución de intensidades de los píxeles. En una imagen en escala de grises, cada valor de 0 a 255 (negro y blanco respectivamente) tiene asociado una frecuencia que indica cuántos píxeles presentan ese nivel de brillo. En imágenes a color, el histograma puede calcularse de forma independiente para cada canal en RGB ofreciendo una visión más detallada de la composición cromática de la imagen.

### Los siguientes son propiedades del histograma:

- *Distribución*: muestra cómo se reparten los datos en el rango de valores disponibles. Indica si predominan los tonos oscuros o claros. Una concentración hacia la izquierda significa que la imagen es más oscura; hacia la derecha, que es más clara.
- *Rango*: muestra valores desde 0 hasta 255.
- *Frecuencia*: es la distribución de los datos en todo el rango del histograma. Una distribución uniforme sugiere intensidades balanceadas; una sesgada significa predominancia de tonos oscuros o claros; varios picos representan zonas irregulares como lo son muy brillantes o muy oscuras.
- *Eje X*: es el eje horizontal del histograma. Contiene valores. Se establece en función del rango de los valores representados.
- *Eje Y*: es el eje vertical del histograma. Contiene valores. Se establece en función del rango de los valores representados.

En el análisis de imágenes, un histograma sirve para la evaluación de brillo y contraste mostrándose simplemente si la imagen está oscura, clara o balanceada. Eso se puede ver en el canal de luminiscencia que, de acuerdo con ello, mostrará las propiedades de brillo de una imagen. Lo mismo ocurre con la exposición de una imagen si es que la concentración de frecuencias está a la izquierda o a la derecha siendo subexpuestas o sobreexpuesta respectivamente.

Un histograma se puede comparar con otros histogramas para conocer las diferentes frecuencias y concentraciones de datos en distintos histogramas después de someter a la imagen a varias modificaciones (ya sea RGB, binarización, entre otros). También se pueden detectar anomalías como desviaciones inusuales o falta de datos en un rango específico.

## Resultados Obtenidos:

- YIQ: La imagen fue separada en sus componentes de luminancia y crominancia. El componente Y mostró la estructura en escala de grises, mientras que I y Q representaron los componentes de color en los rangos de crominancia. Este modelo es útil para la compresión de imágenes y la transmisión de video, ya que puede reducir la cantidad de información sin perder detalles importantes en la luminancia.
- CMY: La conversión a este modelo mostró cómo los colores se pueden representar de manera sustractiva. Al ser un modelo de color usado en impresoras, observamos cómo se genera el color al mezclar los tres colores básicos (Cian, Magenta y Amarillo). Este modelo, aunque no es intuitivo para la visualización, es esencial para la tecnología de impresión.
- HSV: La conversión a HSV permitió observar de manera más visualmente comprensible cómo se estructuran los colores en términos de matiz, saturación y valor. El modelo HSV es ampliamente utilizado en software de edición de imágenes y es muy intuitivo para los usuarios debido a su alineación con la percepción humana del color.
- Binarización: Aplicando un umbral, la imagen fue convertida en una imagen binaria (blanco y negro), lo que permite realizar segmentaciones y análisis de objetos en la imagen de manera más fácil.

# Desarrollo

## 1) Resumen Ejecutivo

Este proyecto carga una imagen de ejemplo (imagen1.jpg, una rosa), la convierte a arreglos NumPy y aplica transformaciones básicas de visión: visualización RGB, conversión a escala de grises, conversión a HSV, cálculo de histogramas y binarización por umbral configurable (por defecto 128). Los resultados se muestran con Matplotlib y sirven como base para prácticas posteriores (segmentación y mejoras de contraste). Se documenta el entorno usado (Anaconda/Nirvana) y la solución a errores típicos de dependencias.

---

## 2) Objetivo(s) y Alcance

- Objetivo general: Implementar y documentar un flujo mínimo de análisis de imágenes en Python para cargar, transformar y segmentar una imagen mediante umbralización.
  - Objetivos específicos:
    - Cargar imágenes con Pillow y convertirlas a numpy.ndarray.
    - Visualizar canales y resultados con Matplotlib.
    - Implementar conversión RGB→Grayscale y RGB→HSV.
    - Aplicar umbral fijo (parámetro umbral) y discutir alternativas (Otsu).
    - Dejar instrucciones reproducibles del entorno y dependencias.
  - Alcance: Se trabaja con una imagen local (imagen1.jpg). No incluye entrenamiento de modelos ni procesamiento en lote.
- 

## 3) Contexto y Motivación

- Problema a resolver (por qué es relevante).
  - Casos de uso (laboratorio, seguridad, industria, etc.).
- 

## 4) Dataset y Fuentes de Datos

- Fuente: archivo local

- Formato: JPEG (960×1280 aprox.).
  - Notas: Para reproducir, colocar cualquier imagen JPG con ese nombre en la misma carpeta del script o ajustar la ruta en el código (ver sección Ejecución/ Uso).
- 

## 5) Metodología / General

1. Carga: PIL.Image.open(path).convert("RGB") → numpy.array.
2. Visualización inicial: mostrar RGB y canales individuales.
3. Conversión a Grayscale: luminancia BT.601 con OpenCV (cv2.cvtColor).
4. Binarización: umbral fijo (cv2.threshold).
5. Modelos de color: YIQ, CMY y HSV con skimage.color.
6. Histogramas interactivos: cada figura incluye un botón "Histogramas" que abre una ventana única con los histogramas de los ítems relevantes de esa figura.
7. Salida: figuras y ventanas de histogramas (bloqueantes) para cada etapa.

## Diagrama simple

imagen1.jpg → Carga (Pillow) → Arrays (NumPy) → RGB/Gris → Umbral → {YIQ, CMY, HSV} → Visualización + Botón de histogramas

---

## 6) Arquitectura del Proyecto (Estructura de Carpetas)

vision-proyecto/

```
|-- src/
|   |-- análisis_imagenes.py
|   |-- Documentacio_practica1_escuadron241.docx
|   |-- imagen1.jpg
```

---

## 7) Instalación y Requisitos

Hardware: PC con Windows.

Software: VS Code + Anaconda. Entorno: Nirvana (Python 3.9.18).

Paquetes clave y versiones (probadas):

```
numpy==1.26.4  
matplotlib==3.7.3  
pillow==11.1.0  
# opcionales útiles  
opencv-python==4.9.0.80  
scikit-image==0.21.0  
scipy==1.10.1
```

Instalación (usando el Python del entorno Anaconda):

---

## 8) Algoritmos utilizados, fragmentos de código en Python y descripción de que hacen las líneas de código

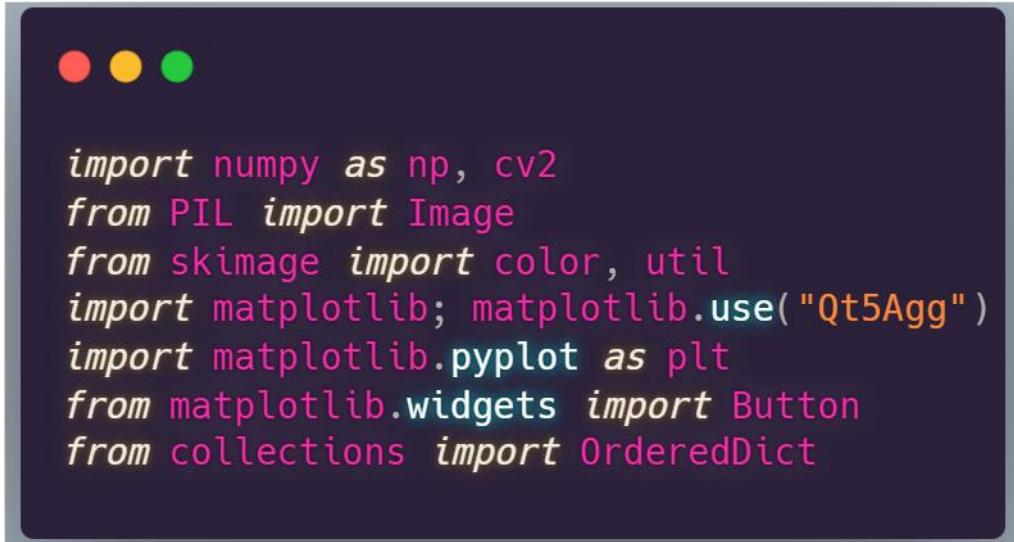
### Sesión 1. Preparación del entorno e importaciones

#### Objetivo.

Dejar listo el entorno de trabajo y fijar el backend de Matplotlib en **Qt5Agg** para abrir ventanas interactivas con botones.

#### Algoritmo / ideas clave.

- Importar librerías: **Pillow** (carga de imágenes), **NumPy** (arreglos), **OpenCV** (gris BT.601 y binarización), **scikit-image** (modelos de color), **Matplotlib** (visualización con widgets).
- Establecer backend **Qt5Agg** (requiere **PyQt5**) antes de pyplot.
- Habilitar GUI para selección de archivos con **QFileDialog** (PyQt5).
- Usar **OrderedDict** para preservar el orden al mostrar imágenes e histogramas.



```
import numpy as np, cv2
from PIL import Image
from skimage import color, util
import matplotlib; matplotlib.use("Qt5Agg")
import matplotlib.pyplot as plt
from matplotlib.widgets import Button
from collections import OrderedDict
```

Este bloque configura la base del proyecto: podremos **abrir una imagen con Pillow, tratarla como matriz con NumPy, convertirla** (OpenCV para gris/umbral; scikit-image para YIQ/HSV), y **visualizar** resultados en ventanas de Matplotlib que incluyen un **botón** (de matplotlib.widgets.Button) para abrir **una sola ventana** con los histogramas de lo mostrado. Qt5Agg se fija explícitamente para que Matplotlib use PyQt5; por eso importamos QApplication y QFileDialog. También se importa Tkinter como alternativa

Código (fragmento):

## Sesión 2. Utilidades generales — to\_uint8

### Objetivo.

Normalizar cualquier imagen numérica a formato **uint8 [0–255]** sin cambiar su apariencia visual (misma escala percibida), para poder mostrarla y calcular histogramas de forma consistente.



```
# ===== utilidades generales =====
def to_uint8(img: np.ndarray) -> np.ndarray:
    """Convierte cualquier imagen a uint8 sin alterar su apariencia visual."""
    if img.dtype == np.uint8:
        return img
    if img.dtype.kind == "f" and 0.0 <= float(img.min()) and float(img.max()) <= 1.0:
        return (img * 255).clip(0, 255).astype(np.uint8)
    m, M = float(img.min()), float(img.max())
    if M - m < 1e-12:
        return np.zeros_like(img, dtype=np.uint8)
    return ((img - m) / (M - m) * 255.0).astype(np.uint8)
```

### Algoritmo / ideas clave.

- Si ya es uint8, **regresa tal cual**.
- Si es flotante y ya está en **[0,1]**, **escala**:  $img * 255$ , recorta a  $[0,255]$  y castea a uint8.
- Si está en cualquier otro rango/tipo:
  - Calcula **mínimo m y máximo M**.
  - Si  $M-m \approx 0$ , devuelve **matriz de ceros** (evita dividir entre  $\sim 0$ ).
  - Si no, aplica **min-max**:  $(img - m) / (M - m) * 255$  y castea a uint8.

### Descripción general del código.

La función detecta el **tipo y rango** de la imagen y aplica la **conversión más segura** para conservar la apariencia: pasa de flotantes normalizados o de rangos arbitrarios (p. ej.,  $[-1,1]$ ,  $[0,65535]$ , valores HDR) a uint8. Esto es clave para **visualización en Matplotlib, guardado y cálculo de histogramas** con un dominio uniforme (0–255).

## Sesión 3. Métricas e histogramas con resumen en figura — `_stats_*` y `plot_hist_on_axes`

### Objetivo.

Calcular **métricas estadísticas** a partir del histograma (energía, entropía, media, varianza y asimetría) y mostrarla junto con el **histograma** de cada imagen/canal, tanto en la **figura** (resumen) como en la **consola** (detalle). (resumen) como en la **consola** (detalle).

```
def _stats_from_hist(hist: np.ndarray):
    """
    Calcula energía, entropía, media, varianza y skewness a partir de un histograma (256 bins).
    Asum. intensidades 0..255.
    """
    total = hist.sum()
    if total == 0:
        return {"energy": 0.0, "entropy": 0.0, "mean": 0.0, "var": 0.0, "skew": 0.0}
    p = hist.astype(np.float64) / float(total)
    energy = float(np.sum(p ** 2))

    mask = p > 0
    entropy = float(-np.sum(p[mask] * np.log2(p[mask])))

    intens = np.arange(256, dtype=np.float64)
    mean = float(np.sum(intens * p))
    var = float(np.sum(((intens - mean) ** 2) * p))
    std = np.sqrt(var) if var > 0 else 0.0
    skew = float(np.sum(((intens - mean) / std) ** 3) * p) if std > 0 else 0.0

    return {"energy": energy, "entropy": entropy, "mean": mean, "var": var, "skew": skew}

def _stats_from_image(img_u8: np.ndarray):
    """Calcula histograma (256 bins) y estadísticas para una imagen uint8 2D."""
    hist, bins = np.histogram(img_u8.ravel(), bins=256, range=(0, 255))
    stats = _stats_from_hist(hist)
    return hist, bins, stats

def _stats_text_block(stats: dict) -> str:
    return (
        f"Energia: {stats['energy']:.4f}\n"
        f"Entropia: {stats['entropy']:.4f}\n"
        f"Media: {stats['mean']:.2f}\n"
        f"Varianza: {stats['var']:.2f}\n"
        f"Asimetria: {stats['skew']:.4f}"
    )

def _print_stats_console(name: str, stats: dict):
    print(f"\n[Estadísticas] {name}")
    print(f"  Energia : {stats['energy']:.6f}")
    print(f"  Entropia : {stats['entropy']:.6f}")
    print(f"  Media : {stats['mean']:.6f}")
    print(f"  Varianza : {stats['var']:.6f}")
    print(f"  Asimetria : {stats['skew']:.6f}")

def plot_hist_on_axes(ax, img_u8: np.ndarray, name: str):
    """Dibuja un histograma: 1 curva si es gris, 3 curvas si es RGB. Muestra estadísticas."""
    ax.set_title(name)
    ax.set_xlim(0, 255)
    ax.set_xlabel("Intensidad")
    ax.set_ylabel("Frecuencia")
    ax.grid(True, alpha=.3)

    if img_u8.ndim == 2:
        hist, bins, stats = _stats_from_image(img_u8)
        ax.plot(bins[:-1], hist)
        txt = _stats_text_block(stats)
        ax.text(
            0.98, 0.98,
            transform=ax.transAxes,
            fontsize=9, va='top', ha='right',
            bbox=dict(facecolor='white', alpha=0.8, edgecolor='none')
        )
        _print_stats_console(name, stats)
    else:
        colors = (("red", 0), ("green", 1), ("blue", 2))
        stats_texts = []
        for cname, i in colors:
            hist, bins = np.histogram(img_u8[..., i].ravel(), bins=256, range=(0, 255))
            ax.plot(bins[:-1], hist, label=cname, color=cname)
            stats = _stats_from_hist(hist)
            stats_texts.append((f'{name} - {cname.upper()}', stats))
        _print_stats_console(f'{name} - {cname.upper()}', stats)

        summary_lines = []
        for cname, st in stats_texts:
            line = f'{cname}: μ={st["mean"]:.1f}, σ²={st["var"]:.1f}, H={st["entropy"]:.2f}'
            summary_lines.append(line)
        txt = " ".join(summary_lines) + "\nVer consola para más detalles"
        ax.text(
            0.5, 0.98,
            transform=ax.transAxes,
            fontsize=9, va='top', ha='center',
            bbox=dict(facecolor='white', alpha=0.8, edgecolor='none')
        )
    ax.legend()
```

## Algoritmo / ideas clave.

- `_stats_from_hist(hist)`: recibe un histograma (256 bins en 0–255) y devuelve:
  - **Energía**  $\sum p^2 \rightarrow$  concentración tonal.
  - **Entropía**  $-\sum p \log_2 p \rightarrow$  desorden/variabilidad.
  - **Media**  $\mu \rightarrow$  brillo promedio.
  - **Varianza**  $\sigma^2 \rightarrow$  contraste.
  - **Asimetría (skew)**  $\rightarrow$  sesgo a claros/oscuros (usa normalización por  $\sigma^3$ ). Maneja casos vacíos para evitar divisiones por cero.
- `_stats_from_image(img_u8)`: construye el **histograma 0–255** de una imagen **uint8 2D** y llama a `*_from_hist` para obtener métricas.
- `_stats_text_block(stats)`: formatea un **bloque corto** con los cinco indicadores (para incrustar en la figura).
- `_print_stats_console(name, stats)`: imprime en **consola** las métricas con mayor precisión (útil para el reporte de resultados).
- `plot_hist_on_axes(ax, img_u8, name)`:
  - Configura ejes (0–255, grilla suave).
  - Si es **gris (2D)**: calcula histograma + métricas y **dibuja 1 curva**; agrega un **cuadro de texto** con las cifras y manda el **detalle a consola**.
  - Si es **RGB (3D)**: calcula histograma **por canal** (R, G, B), **dibuja 3 curvas** con leyenda y muestra un **resumen por canal** ( $\mu$ ,  $\sigma^2$ , H) en la figura; el **detalle completo** de cada canal va a consola.

## Descripción general del código.

Este bloque convierte los histogramas en **evidencia cuantitativa**: cada gráfica incluye el histograma y un **resumen compacto** de métricas, mientras que la consola guarda el **registro preciso** por imagen/canal. Con esto, la comparación entre **RGB vs Gris vs Binarizada** y entre **Y/I/Q o H/S/V** es inmediata y sustentada con números.

## Sesión 4. Ventana única de histogramas — open\_hist\_grid.

### Objetivo.

Mostrar **en una sola ventana** un grid de histogramas para las imágenes seleccionadas, manteniendo orden, tamaño adecuado y bloqueo de la interfaz hasta que el usuario la cierre.



```
def open_hist_grid(images_dict: OrderedDict, title: str = "Histogramas", hist_whitelist=None):
    """
    Abre UNA sola ventana con un grid de histogramas SOLO para los nombres
    incluidos en 'hist_whitelist'. Si es None, usa todos.
    """
    # Filtra por whitelist si se especifica
    if hist_whitelist is not None:
        items = [(k, v) for k, v in images_dict.items() if k in hist_whitelist]
    else:
        items = list(images_dict.items())

    if not items:
        print("No hay imágenes seleccionadas para histogramas.")
        return

    names, imgs = zip(*items)
    n = len(imgs)
    cols = min(3, n)
    rows = int(np.ceil(n / cols))

    fig, axes = plt.subplots(rows, cols, figsize=(5 * cols, 3.8 * rows))
    if not isinstance(axes, np.ndarray):
        axes = np.array([axes])
    axes = axes.ravel()

    for ax, name, img in zip(axes, names, imgs):
        u8 = to_uint8(img)
        plot_hist_on_axes(ax, u8, name)

    # Apaga ejes sobrantes si el grid no es exacto
    for ax in axes[len(imgs):]:
        ax.axis("off")

    fig.suptitle(title, y=0.98, fontsize=12)
    fig.tight_layout(rect=[0, 0, 1, 0.95])
    plt.show(block=True)
```

## Algoritmo / ideas clave.

- **Filtro selectivo:** si hist\_whitelist está definido, solo grafica esas claves; si no, grafica todas.
- **Grid dinámico:** calcula rows y cols en función del número de imágenes (cols ≤ 3) y aplana ejes para iterar.
- **Normalización:** convierte cada imagen a uint8 con to\_uint8 para comparar histogramas en el rango 0–255.
- **Trazado:** usa plot\_hist\_on\_axes(ax, u8, name) para dibujar 1 curva (gris) o 3 (RGB).
- **Limpieza visual:** apaga ejes sobrantes cuando el grid no es exacto; ajusta títulos y márgenes con tight\_layout.
- **Interacción:** plt.show(block=True) abre una única ventana y bloquea hasta que se cierre (evita que se acumulen múltiples ventanas).

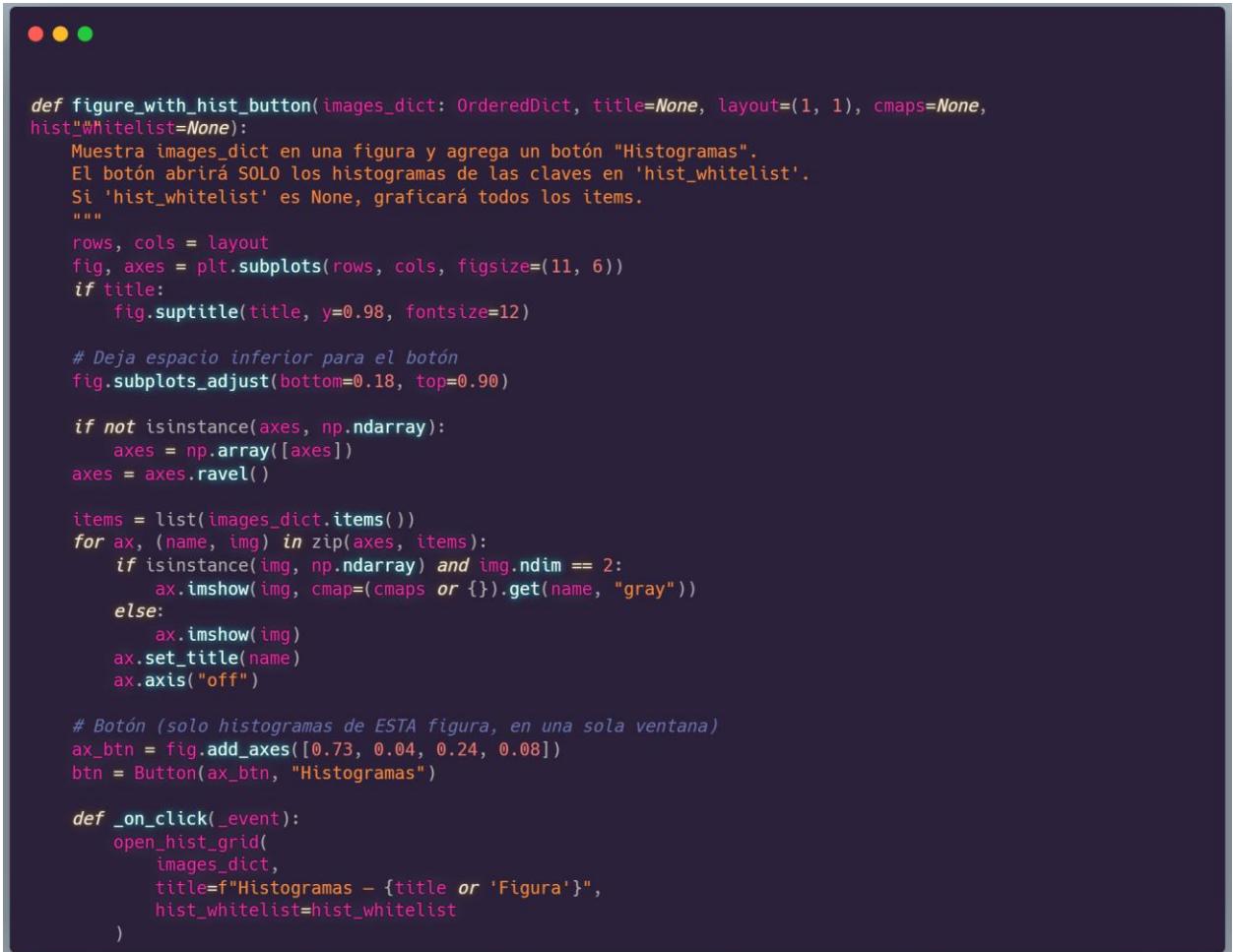
## Descripción general del código.

La función recibe un diccionario ordenado images\_dict (nombre→imagen), lo filtra con hist\_whitelist si corresponde y arma un subplot proporcional al número de elementos. Para cada imagen, primero la normaliza a uint8 y luego delega el dibujo del histograma a plot\_hist\_on\_axes. Finalmente agrega un título global, corrige el layout y muestra la ventana de manera modal (bloqueante). Esto permite comparar de forma clara y consistente las distribuciones de intensidad de las imágenes o de sus canales.

## Sesión 5. Figura con botón de histogramas — figure\_with\_hist\_button

### Objetivo.

Mostrar uno o varios **resultados/imágenes** en una misma figura (grid configurable) y añadir un **botón “Histogramas”** que abre **una sola ventana** con los histogramas de los elementos de esa figura (filtrados por hist\_whitelist si se indica).



```
● ● ●

def figure_with_hist_button(images_dict: OrderedDict, title=None, layout=(1, 1), cmaps=None,
hist_whitelist=None):
    Muestra images_dict en una figura y agrega un botón "Histogramas".
    El botón abrirá SOLO los histogramas de las claves en 'hist_whitelist'.
    Si 'hist_whitelist' es None, graficará todos los items.
    """
    rows, cols = layout
    fig, axes = plt.subplots(rows, cols, figsize=(11, 6))
    if title:
        fig.suptitle(title, y=0.98, fontsize=12)

    # Deja espacio inferior para el botón
    fig.subplots_adjust(bottom=0.18, top=0.90)

    if not isinstance(axes, np.ndarray):
        axes = np.array([axes])
    axes = axes.ravel()

    items = list(images_dict.items())
    for ax, (name, img) in zip(axes, items):
        if isinstance(img, np.ndarray) and img.ndim == 2:
            ax.imshow(img, cmap=(cmaps or {}).get(name, "gray"))
        else:
            ax.imshow(img)
        ax.set_title(name)
        ax.axis("off")

    # Botón (solo histogramas de ESTA figura, en una sola ventana)
    ax_btn = fig.add_axes([0.73, 0.04, 0.24, 0.08])
    btn = Button(ax_btn, "Histogramas")

    def _on_click(_event):
        open_hist_grid(
            images_dict,
            title=f"Histogramas - {title or 'Figura'}",
            hist_whitelist=hist_whitelist
        )

```

### Algoritmo / ideas clave.

- Crea la figura y ejes según layout=(rows, cols) y opcionalmente coloca un **título**.
- **Reserva espacio inferior** (subplots\_adjust) para alojar el botón.
- Asegura que axes sea un arreglo 1D (aplana) para iterar sin errores.
- Recorre images\_dict (orden garantizado por OrderedDict):

- Si la imagen es 2D, usa mapa de color de `cmaps[name]` o “**gray**” por defecto; si es 3D, la muestra tal cual.
  - Coloca **título por imagen** y oculta ejes para una vista limpia.
- Crea el **botón** en un axes adicional y conecta el callback `_on_click`:
  - Llama a `open_hist_grid(...)` con el **título contextual** (“Histogramas — {title}”) y `hist_whitelist`.
  - Así, el botón muestra **solo** los histogramas relevantes de **esa** figura en **una ventana única**.
- (En el script completo) Guarda referencias del botón/callback en `fig` para evitar **garbage collection**, y **devuelve** (`fig, axes`); el `plt.show(block=True)` lo hace el **llamador**.

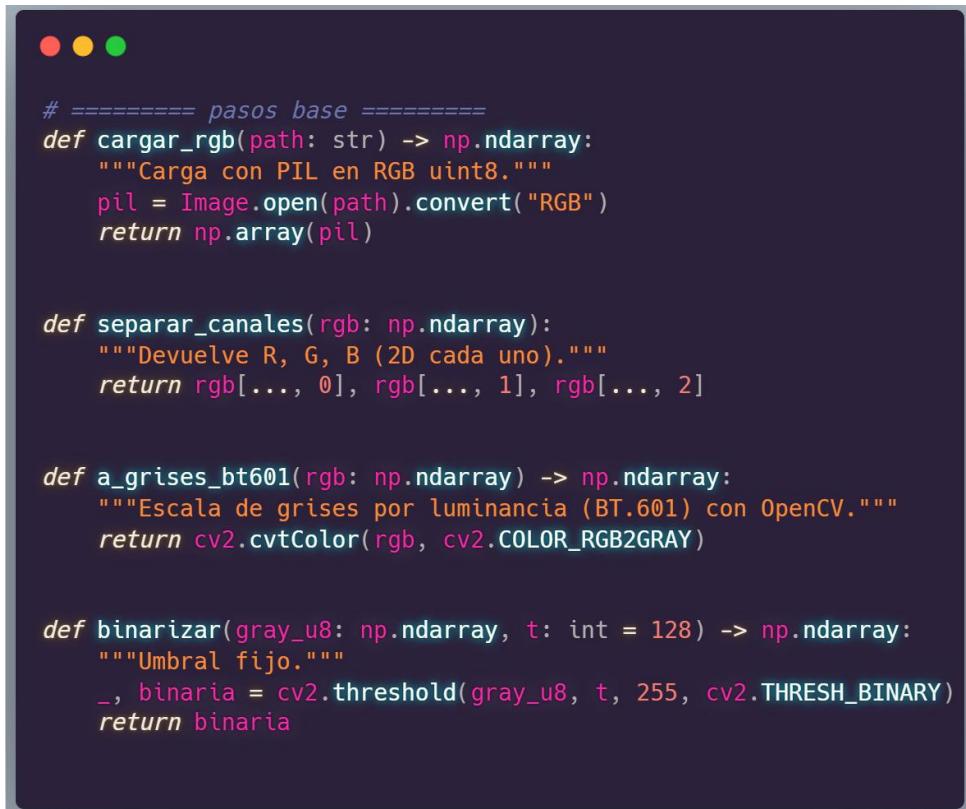
#### **Descripción general del código.**

Esta función actúa como un **contenedor interactivo**: renderiza los resultados de una etapa (p. ej., canales RGB, gris, YIQ/CMY/HSV) y centraliza la apertura de histogramas en un **único panel** por figura. El parámetro `hist_whitelist` controla qué claves del diccionario generan histograma (útil para omitir composiciones “visual” que no deben graficarse). Con `layout` eliges la cuadrícula y con `cmaps` asignas colormaps por nombre de imagen. Esta separación permite encadenar etapas de la demo: cada `figure_with_hist_button(...)` muestra la vista y deja el análisis de distribución a un botón consistente y no intrusivo.

## Sesión 6. Pasos base del pipeline — carga, separación, gris BT.601 y binarización

### Objetivo.

Implementar el flujo mínimo para preparar una imagen: cargar en RGB, separar canales, convertir a escala de grises por luminancia (BT.601) y segmentar con umbral fijo.



```
# ===== pasos base =====
def cargar_rgb(path: str) -> np.ndarray:
    """Carga con PIL en RGB uint8."""
    pil = Image.open(path).convert("RGB")
    return np.array(pil)

def separar_canales(rgb: np.ndarray):
    """Devuelve R, G, B (2D cada uno)."""
    return rgb[..., 0], rgb[..., 1], rgb[..., 2]

def a_grises_bt601(rgb: np.ndarray) -> np.ndarray:
    """Escala de grises por luminancia (BT.601) con OpenCV."""
    return cv2.cvtColor(rgb, cv2.COLOR_RGB2GRAY)

def binarizar(gray_u8: np.ndarray, t: int = 128) -> np.ndarray:
    """Umbral fijo."""
    _, binaria = cv2.threshold(gray_u8, t, 255, cv2.THRESH_BINARY)
    return binaria
```

### Algoritmo / ideas clave.

- **Carga (Pillow → NumPy):** abrir archivo y convertirlo explícitamente a **RGB uint8**.
- **Separación de canales:** obtener **R, G, B** como matrices 2D independientes para análisis por canal.
- **Grises BT.601 (OpenCV):** usar la conversión estándar de luminancia (pondera más el verde).
- **Binarización (umbral fijo):** segmentar la imagen gris con  $t$ (por defecto **128**) usando `cv2.threshold`.

### **Descripción general del código.**

Primer bloque de código: Abre la imagen con **Pillow**, la fuerza a **RGB** y la pasa a **numpy.ndarray uint8** (0–255).

Segundo bloque de código: Extrae **R**, **G** y **B** como matrices 2D; útil para ver contribuciones individuales y sus histogramas.

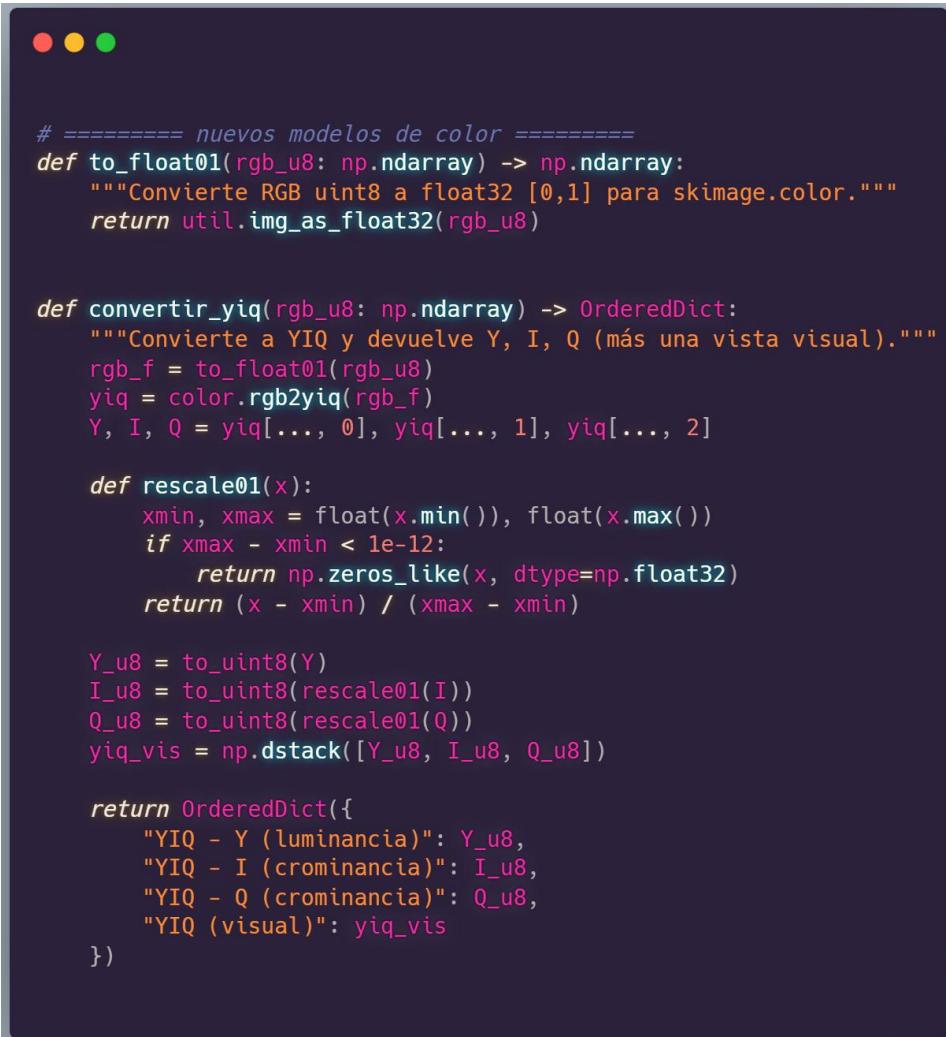
Tercer bloque de código: Convierte a **gris** siguiendo **BT.601** (aprox.  $0.299R + 0.587G + 0.114B$ ); base para histogramas de luminancia y umbralización.

Cuarto bloque de código: Aplica **umbral fijo**: píxeles  $\geq t$  pasan a **255**, el resto a **0**. Es la forma más directa de **segmentar** regiones brillantes/oscuras y comparar cómo cambian **energía/entropía** tras la umbralización.

## Sesión 7. Conversión a flotante y modelo YIQ — to\_float01 y convertir\_yiq

### Objetivo.

Preparar la imagen RGB para conversiones de espacio de color en **scikit-image** y obtener sus componentes **YIQ** (luminancia y crominancias) en formato visualizable y comparable (uint8).



```
# ===== nuevos modelos de color =====
def to_float01(rgb_u8: np.ndarray) -> np.ndarray:
    """Convierte RGB uint8 a float32 [0,1] para skimage.color."""
    return util.img_as_float32(rgb_u8)

def convertir_yiq(rgb_u8: np.ndarray) -> OrderedDict:
    """Convierte a YIQ y devuelve Y, I, Q (más una vista visual)."""
    rgb_f = to_float01(rgb_u8)
    yiq = color.rgb2yiq(rgb_f)
    Y, I, Q = yiq[..., 0], yiq[..., 1], yiq[..., 2]

    def rescale01(x):
        xmin, xmax = float(x.min()), float(x.max())
        if xmax - xmin < 1e-12:
            return np.zeros_like(x, dtype=np.float32)
        return (x - xmin) / (xmax - xmin)

    Y_u8 = to_uint8(Y)
    I_u8 = to_uint8(rescale01(I))
    Q_u8 = to_uint8(rescale01(Q))
    yiq_vis = np.dstack([Y_u8, I_u8, Q_u8])

    return OrderedDict({
        "YIQ - Y (luminancia)": Y_u8,
        "YIQ - I (crominancia)": I_u8,
        "YIQ - Q (crominancia)": Q_u8,
        "YIQ (visual)": yiq_vis
    })
```

### **Algoritmo / ideas clave.**

- **to\_float01**: pasa **RGB uint8 → float32 en [0,1]** usando skimage.util.img\_as\_float32, requisito para skimage.color.
- **convertir\_yiq**:
  - Convierte **RGB→YIQ** con color.rgb2yiq.
  - Separa **Y** (luminancia), **I** y **Q** (crominancias).
  - **Reescala I y Q a [0,1]** con rescale01 (min-max por canal) solo para **visualización**; luego las pasa a uint8 con **to\_uint8**.
  - Apila **Y\_u8, I\_u8, Q\_u8** en un **composite visual** (**yiq\_vis**) para ver “coloreadas” las tres componentes.
  - Devuelve un OrderedDict con **Y, I, Q** (para histogramas/métricas) y **YIQ (visual)** (solo para mostrar).

### **Descripción general del código.**

Primer def del código: Convierte la imagen a **float32 [0,1]**, formato estándar para funciones de skimage.color.

### **Notas clave del Segundo def del código.**

- **Y** conserva la estructura y texturas (luminancia) → ideal para **gris/umbralización**.
- **I/Q** codifican **crominancia**; se reescalan para verlos correctamente, pero sus histogramas se calculan sobre las versiones uint8 reescaladas (consistentes con 0–255).
- El **composite YIQ (visual)** es solo para **visualización** y normalmente **se excluye** de hist\_whitelist cuando se grafican histogramas.

## Sesión 8. Modelos de color sustractivo y perceptual — convertir\_cmy y convertir\_hsv

### Objetivo.

Obtener representaciones alternativas de la imagen para análisis y visualización:

- **CMY** (modelo sustractivo, útil para impresión).
- **HSV** (modelo perceptual, útil para segmentación por color e interfaces de edición).



```
def convertir_cmy(rgb_u8: np.ndarray) -> OrderedDict:
    """CMY: C=255-R, M=255-G, Y=255-B (más composite para visualizar)."""
    cmy_u8 = 255 - rgb_u8
    C, M, Y = cmy_u8[..., 0], cmy_u8[..., 1], cmy_u8[..., 2]
    return OrderedDict({
        "CMY - C (255-R)": C,
        "CMY - M (255-G)": M,
        "CMY - Y (255-B)": Y,
        "CMY (composite)": cmy_u8
    })

def convertir_hsv(rgb_u8: np.ndarray) -> OrderedDict:
    """HSV con skimage; devuelve H, S, V y un RGB reconstruido para ver colores."""
    rgb_f = to_float01(rgb_u8)
    hsv = color.rgb2hsv(rgb_f)
    H, S, V = hsv[..., 0], hsv[..., 1], hsv[..., 2]
    H_u8, S_u8, V_u8 = to_uint8(H), to_uint8(S), to_uint8(V)
    hsv_to_rgb_u8 = to_uint8(color.hsv2rgb(hsv))
    return OrderedDict({
        "HSV - H (matiz)": H_u8,
        "HSV - S (saturación)": S_u8,
        "HSV - V (valor)": V_u8,
        "HSV→RGB (reconstruido)": hsv_to_rgb_u8
    })
```

## Algoritmo / ideas clave.

- **CMY**: aplicar la relación sustractiva directa **C=255-R, M=255-G, Y=255-B** y devolver cada canal + un **composite** para visualizar.
- **HSV**: convertir **RGB→HSV** con skimage.color.rgb2 hsv (valores en [0,1]), extraer **H** (matiz), **S** (saturación) y **V** (valor/iluminación), normalizar a **uint8** y generar un **RGB reconstruido** desde HSV para validar visualmente la conversión.

## Descripción general del código.

1. En **CMY**, picos altos en C/M/Y indican predominio de **bajas** intensidades en R/G/B respectivamente (lógica sustractiva).
2. En **HSV**, **H** permite segmentar por color independientemente de la iluminación; **S** refleja cuán “vivo” es el color; **V** captura brillo global.
3. Para tus figuras, incluye **H, S, V** en **hist\_whitelist** y omite el **reconstruido/composite** para mantener histogramas limpios.

## Sesión 9. Binarización interactiva con slider — figure\_binarization\_with\_slider

```
# ===== NUEVO: ventana con slider de umbral para binarización =====
def figure_binarization_with_slider(gray_u8: np.ndarray, initial_t: int = 128):
    """
    Abre una ventana con:
        - Imagen binarizada
        - Slider para cambiar 't' (0..255) y actualizar en vivo
        - Botón 'Histogramas' (del estado actual)
    Al cerrar la ventana, el programa continúa normalmente.
    """
    # Estado mutable del umbral actual
    current_t = {"t": int(np.clip(initial_t, 0, 255))}

    # Imagen inicial
    bin_img = binarizar(gray_u8, current_t["t"])

    fig, ax = plt.subplots(1, 1, figsize=(10, 6))
    fig.suptitle("4) Binarización con slider de umbral", y=0.98, fontsize=12)

    # Dejar espacio para slider y botón
    fig.subplots_adjust(bottom=0.25, top=0.90)

    im = ax.imshow(bin_img, cmap="gray")
    title_text = ax.set_title(f"Binarización (t={current_t['t']})")
    ax.axis("off")

    # Slider (posición: [x, y, ancho, alto] en coords de figura)
    ax_slider = fig.add_axes([0.10, 0.10, 0.55, 0.06])
    slider = Slider(
        ax=ax_slider, label="Umbral t",
        valmin=0, valmax=255, valinit=current_t["t"], valstep=1
    )

    # Botón de histogramas (usará el estado actual de la imagen)
    ax_btn = fig.add_axes([0.72, 0.08, 0.22, 0.10])
    btn = Button(ax_btn, "Histogramas")

    def update_threshold(val):
        # El slider entrega float; cast a int y recorta a [0,255]
        t = int(np.clip(val, 0, 255))
        if t == current_t["t"]:
            return
        current_t["t"] = t
        updated = binarizar(gray_u8, t)
        im.set_data(updated)
        title_text.set_text(f"Binarización (t={t})")
        fig.canvas.draw_idle()

    def show_hist(_event):
        # Construye el grid de histograma con el estado actual
        t = current_t["t"]
        current_bin = binarizar(gray_u8, t)
        images = OrderedDict({f"Binarización (t={t})": current_bin})
        open_hist_grid(
            images,
            title=f"Histogramas - Binarización (t={t})",
            hist_whitelist={f"Binarización (t={t})"}
        )

    slider.on_changed(update_threshold)
    btn.on_clicked(show_hist)

    # Guardar referencias para evitar GC
    fig._bin_slider = slider
    fig._bin_btn = btn
    fig._bin_image_artist = im
    fig._bin_state = current_t

plt.show(block=True)
```

## Objetivo.

Permitir **ajustar el umbral de binarización en tiempo real** sobre la imagen en gris y, desde el mismo panel, **abrir los histogramas** del estado actual.

## Algoritmo / ideas clave.

- Mantener un **estado mutable** del umbral (current\_t) en [0,255].
- Generar la **imagen binaria inicial** con binarizar(gray\_u8, initial\_t).
- Crear una figura con:
  - **Imagen binaria** (actualizable),
  - **Slider** 0..255 para cambiar t y refrescar la vista,
  - **Botón “Histogramas”** que invoca open\_hist\_grid(...) con la binaria **del umbral vigente**.
- En update\_threshold: castear el valor del slider a int, **re-binarizar** y **refrescar** (set\_data + draw\_idle).
- En show\_hist: reconstruir la imagen binaria con el **umbral actual** y abrir **una sola ventana** de histograma.
- Mantener **referencias** a slider/botón/artista en fig para evitar *garbage collection*.
- La ventana es **bloqueante** (plt.show(block=True)): al cerrarla, el flujo continúa.

## Descripción general del código.

La función arma una **UI mínima** para experimentar con la **umbralización** sin recompilar ni re-ejecutar: el usuario mueve el slider y observa al instante cómo cambia la segmentación; cuando desea analizar, pulsa **“Histogramas”** y obtiene el histograma de la binaria **correspondiente a ese t** (ideal para comentar cambios en **energía/entropía** según el umbral). Es una extensión interactiva coherente con el resto del pipeline.

## Sesión 10. Demostración secuencial del pipeline — demo(path, umbral)

```
# ===== demo secuencial (una ventana a la vez) =====
def demo(path="imagen.jpg", umbral=128):
    # 1) Original
    rgb = cargar_rgb(path)
    figure_with_hist_button(
        OrderedDict({"Original RGB": rgb}),
        title="1) Original",
        layout=(1, 1),
        hist_whitelist={"Original RGB"}
    )
    plt.show(block=True)

    # 2) Separada en canales RRB
    r, g, b = separar_canales(rgb)
    figure_with_hist_button(
        OrderedDict({"Canal R": r, "Canal G": g, "Canal B": b}),
        title="2) Canales RGB",
        layout=(1, 3),
        cmaps={"Canal R": "Reds", "Canal G": "Greens", "Canal B": "Blues"},
        hist_whitelist={"Canal R", "Canal G", "Canal B"}
    )
    plt.show(block=True)

    # 3) Escala de grises (BT.601)
    gray = a_grises_bt601(rgb)
    figure_with_hist_button(
        OrderedDict({"Escala de grises (BT.601)": gray}),
        title="3) Escala de grises",
        layout=(1, 1),
        hist_whitelist={"Escala de grises (BT.601)"}
    )
    plt.show(block=True)

    # 4) Binarización con slider interactivo (NUEVO)
    figure_binarization_with_slider(gray, umbral)

    # 5) YIQ
    yiq_imgs = convertir_yiq(rgb)
    figure_with_hist_button(
        yiq_imgs,
        title="5) Modelo YIQ (aplicado a RGB original)",
        layout=(2, 2),
        hist_whitelist={
            "YIQ - Y (luminancia)",
            "YIQ - I (crominancia)",
            "YIQ - Q (crominancia)",
        }
    )
    plt.show(block=True)

    # 6) CMY
    cmy_imgs = convertir_cmy(rgb)
    figure_with_hist_button(
        cmy_imgs,
        title="6) Modelo CMY (aplicado a RGB original)",
        layout=(2, 2),
        hist_whitelist={
            "CMY - C (255-R)",
            "CMY - M (255-G)",
            "CMY - Y (255-B)",
        }
    )
    plt.show(block=True)

    # 7) HSV
    hsv_imgs = convertir_hsv(rgb)
    figure_with_hist_button(
        hsv_imgs,
        title="7) Modelo HSV (aplicado a RGB original)",
        layout=(2, 2),
        hist_whitelist={
            "HSV - H (matiz)",
            "HSV - S (saturación)",
            "HSV - V (valor)",
        }
    )
    plt.show(block=True)
```

### **Objetivo.**

Ejecutar, en **ventanas consecutivas**, todo el flujo de análisis: cargar la imagen, separar canales, pasar a gris, **binarizar con slider interactivo**, y convertir a **YIQ, CMY, HSV**, cada uno con su **botón de histogramas**.

### **Algoritmo / ideas clave.**

- **Paso 1 — Original (RGB):** muestra la imagen original y permite ver el histograma RGB.
- **Paso 2 — Canales R/G/B:** despliega R, G y B en un grid (colormaps “Reds/Greens/Blues”) + histogramas por canal.
- **Paso 3 — Gris (BT.601):** convierte a luminancia y muestra su histograma.
- **Paso 4 — Binarización interactiva:** abre la ventana con **slider** (0–255) para ajustar el umbral en vivo y botón de histogramas del estado actual.
- **Paso 5 — YIQ:** obtiene **Y, I, Q** y muestra histogramas de cada componente (excluyendo la vista “visual”).
- **Paso 6 — CMY:** calcula **C=255-R, M=255-G, Y=255-B** y traza histogramas por canal (el composite es solo visual).
- **Paso 7 — HSV:** extrae **H/S/V**, muestra histogramas de cada uno y un RGB reconstruido solo para verificación visual.

### **Descripción general del código.**

La función orquesta el pipeline completo en **una secuencia de figuras**. Tras cada figura, plt.show(block=True) **bloquea** hasta que el usuario la cierre, evitando que se acumulen ventanas. En el paso de binarización se usa figure\_binarization\_with\_slider(gray, umbral) para experimentar con el umbral sin re-ejecutar el script. Para cada etapa, figure\_with\_hist\_button(...) coloca un **botón “Histogramas”** que abre **una única ventana** con los histogramas **relevantes** (controlados por hist\_whitelist). De esta forma, el flujo guía al lector desde la representación original hasta modelos de color alternativos, con análisis cuantitativo disponible en cada punto.

## Sesión 11. Selección de imagen y ejecución principal — seleccionar\_imagen y bloque \_\_main\_\_

```
# Abrimos una imagen
def seleccionar_imagen():
    app = QApplication(sys.argv)
    ruta, _ = QFileDialog.getOpenFileName(
        None,
        "Selecciona una imagen",
        "",
        "Archivos de imagen (*.jpg *.jpeg *.png *.bmp *.tiff)"
    )
    return ruta

if __name__ == "__main__":
    ruta = seleccionar_imagen()

    if ruta:
        entrada = input("Escribe el umbral inicial (ENTER = 128): ")
        if entrada.strip() == "":
            umbral = 128
        else:
            try:
                umbral = int(entrada)
            except ValueError:
                print("⚠️ No escribiste un número válido, usaré 128.")
                umbral = 128

        demo(ruta, umbral)
    else:
        print("No seleccionaste ninguna imagen.")
```

### **Objetivo.**

Permitir que el usuario **elija una imagen desde un diálogo del sistema** (PyQt5) y lanzar la **demo completa** con un **umbral inicial** configurable desde consola.

### **Algoritmo / ideas clave.**

- Crear una **QApplication** y abrir un `QFileDialog.getOpenFileName` con filtro para formatos comunes (jpg, jpeg, png, bmp, tiff).
- Si hay ruta válida, **pedir el umbral inicial** por consola (ENTER → 128 por defecto).
- Validar la entrada (entero) con manejo de error → **fallback a 128**.
- Ejecutar `demo(ruta, umbral)` para recorrer **todo el pipeline** en ventanas secuenciales.
- Si no se selecciona archivo, notificar en consola y **terminar**.

### **Descripción general del código.**

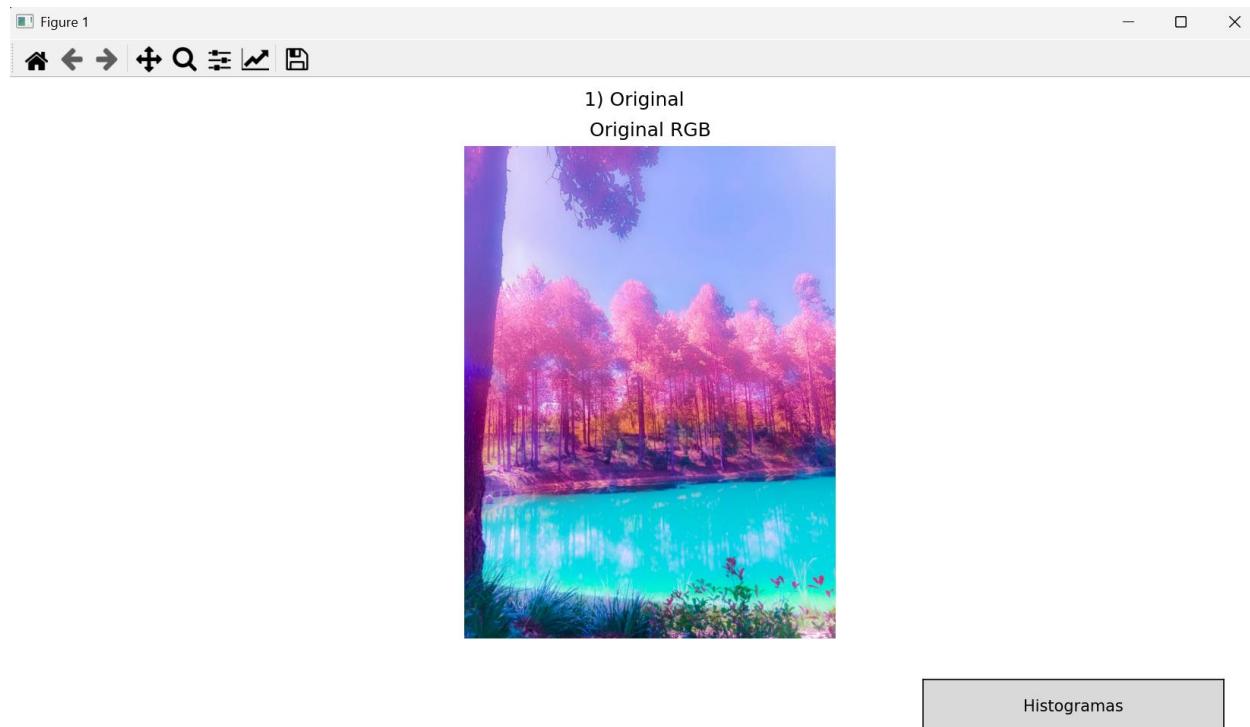
Inicializa la **app Qt**, abre un diálogo nativo para escoger imagen y devuelve la **ruta** seleccionada (o cadena vacía si se cancela).

Control de flujo **simple y robusto**: solicita umbral, **parsea a int** con try/except, aplica **default 128** y llama a `demo(...)`. Si el usuario cancela, imprime un mensaje y termina sin errores.

---

## 13) Resultados / Visualizaciones

Durante la práctica se ejecutó el programa **análisis\_imágenes2.py**, que permitió observar y analizar paso a paso el comportamiento de una imagen al transformarla a distintos **modelos de color** y aplicar **procesos de análisis estadístico e histográfico**.



### 1. Imagen original y canales RGB

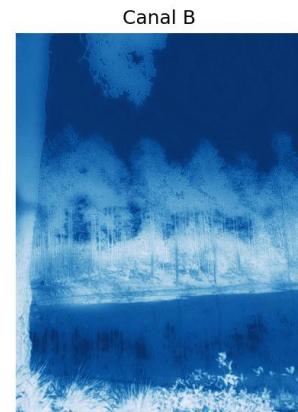
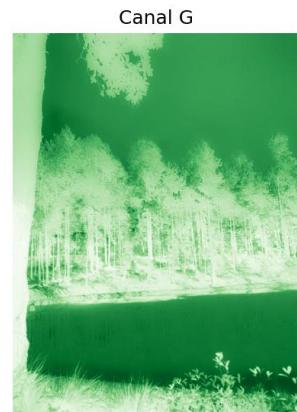
La imagen en su forma **RGB** mostró la composición básica del color mediante tres canales primarios (rojo, verde y azul).

Los **histogramas** de cada canal presentaron diferentes distribuciones de intensidad, lo que permitió identificar qué color dominaba en la escena. Por ejemplo, una concentración mayor en el canal rojo indica tonos cálidos predominantes.

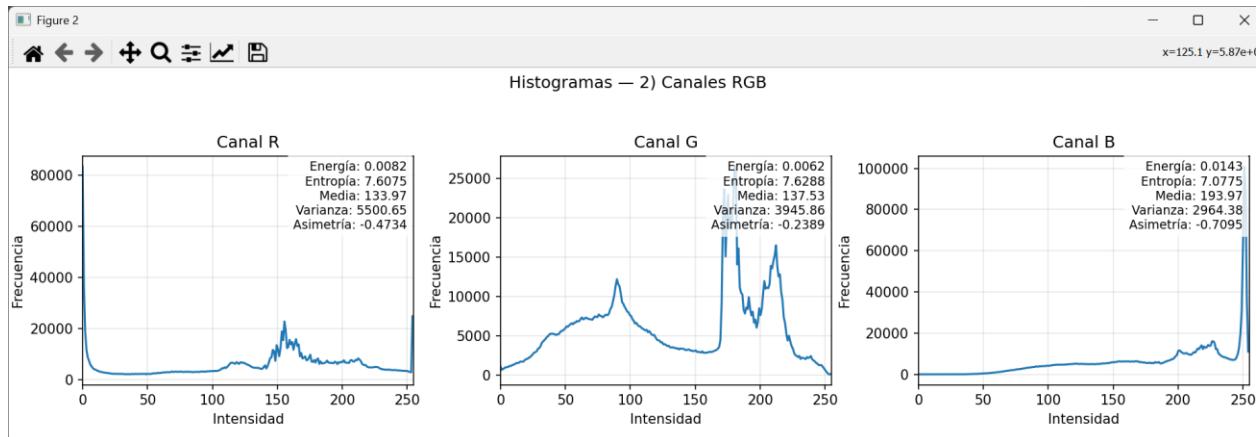
Las métricas de **energía** y **varianza** fueron relativamente altas, reflejando una imagen con buen contraste y distribución amplia de intensidades.



2) Canales RGB



Histogramas



## 2. Escala de grises (BT.601)

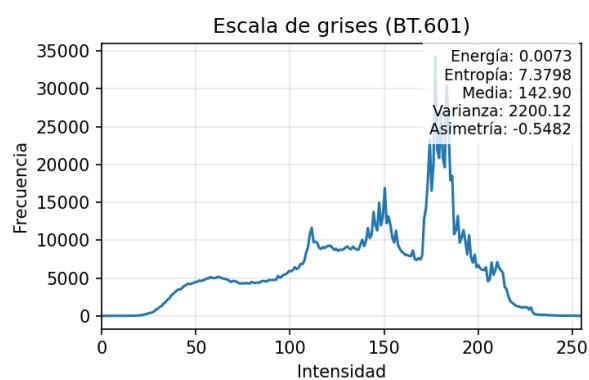
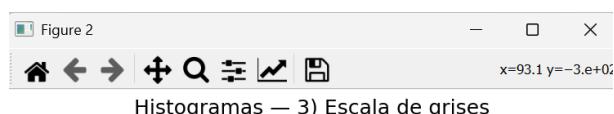
Al convertir la imagen a **escala de grises** mediante la norma **BT.601**, se eliminó la información cromática y solo permaneció la **luminancia**.

El histograma se redujo a una única curva que describe el brillo promedio de cada píxel.

En este punto se observó que la **entropía** disminuyó ligeramente, ya que se perdió la variabilidad de color, aunque la **energía** se mantuvo alta al conservar las diferencias de iluminación.

### 3) Escala de grises

Escala de grises (BT.601)



### 3. Binarización con slider interactivo

El **slider de umbral** permitió experimentar visualmente cómo cambia la segmentación de la imagen al variar el valor  $t$ .

Con valores bajos de umbral, la imagen tendía a verse oscura (muchos píxeles a 0), mientras que con umbrales altos predominaban las zonas blancas.

El histograma de la imagen binaria presentó dos picos bien definidos en los extremos (0 y 255), lo que provocó una **energía muy alta** (concentración extrema) y una **entropía baja**, indicando que la imagen se volvió más predecible y menos diversa en tonos.

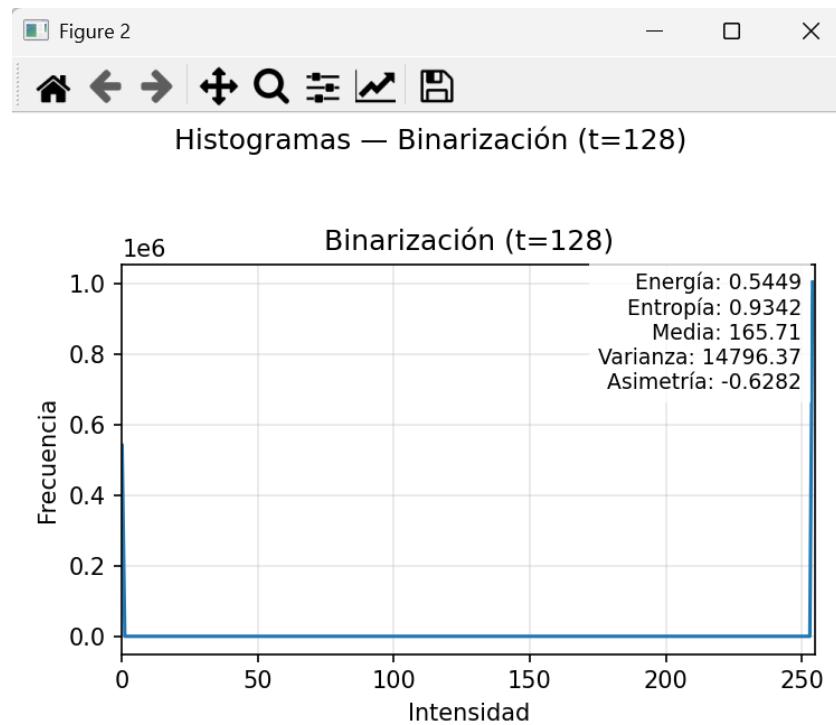
---

#### 4) Binarización con slider de umbral

Binarización ( $t=128$ )



Umbral t  128



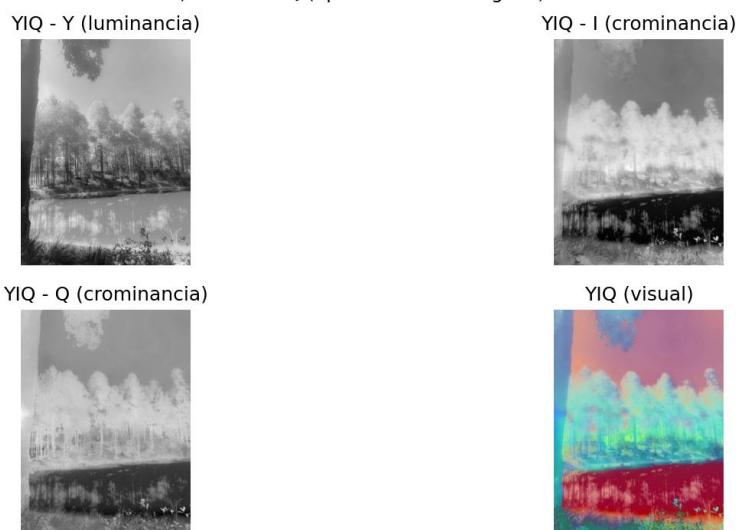
#### 4. Modelo YIQ

En la conversión a **YIQ**, la componente **Y (luminancia)** mantuvo la estructura original de la imagen, similar a la versión en grises, mientras que las componentes **I** y **Q** capturaron la información de color (crominancia) de manera separada.

El análisis de histogramas reveló que **Y** concentra la mayor variación de intensidades, mientras que **I** y **Q** mostraron curvas suaves con menor rango.

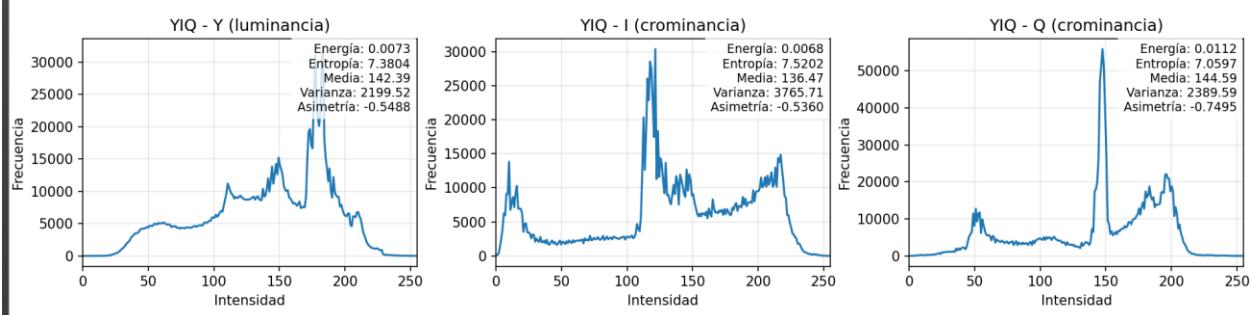
Este modelo resultó útil para entender cómo la televisión analógica y algunas técnicas de compresión priorizan la luminancia sobre el color, conservando los detalles perceptuales más importantes.

5) Modelo YIQ (aplicado a RGB original)



Histogramas

Histogramas — 5) Modelo YIQ (aplicado a RGB original)

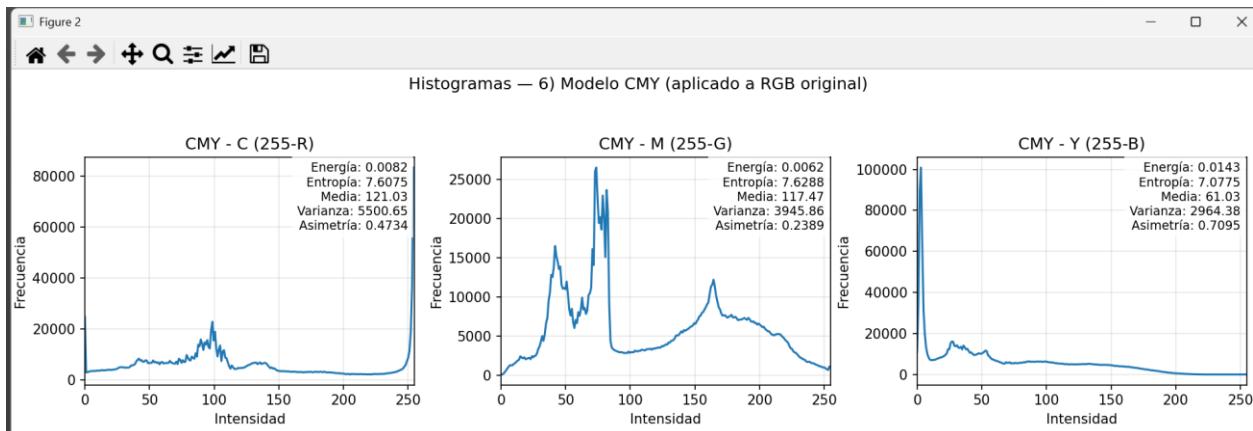
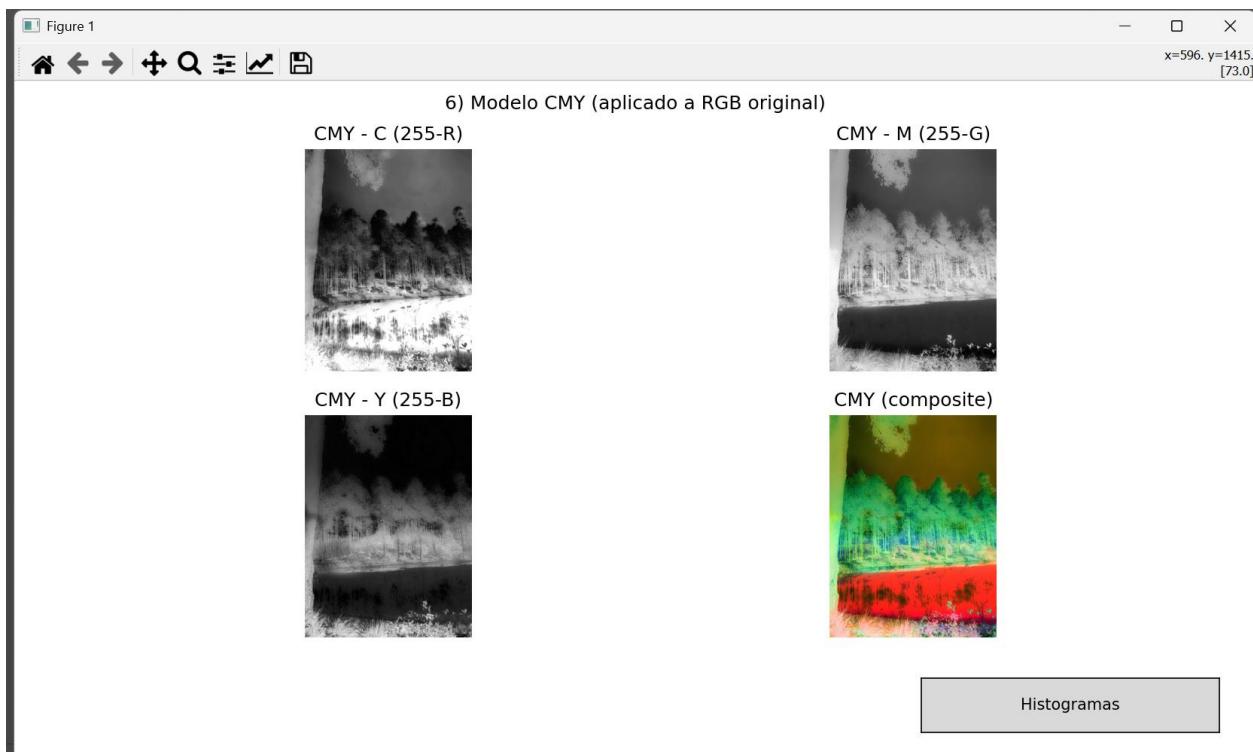


## 5. Modelo CMY

El modelo **CMY** invierte la lógica del RGB, mostrando cómo el color se genera de manera **sustractiva** (propia de impresoras).

Las imágenes **C**, **M** y **Y** mostraron un comportamiento inverso a los canales RGB: donde el rojo era fuerte, el canal C (255-R) se volvía débil, y así sucesivamente.

Los histogramas reflejaron esa inversión tonal. La **energía** y **varianza** se mantuvieron comparables a las del RGB, confirmando que este modelo representa la misma información, solo que invertida cromáticamente.



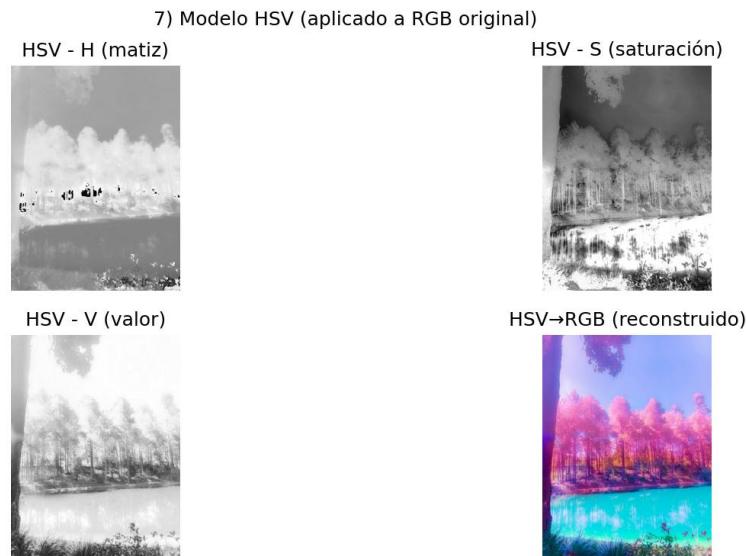
## 6. Modelo HSV

El modelo **HSV** separó la imagen en sus atributos perceptuales:

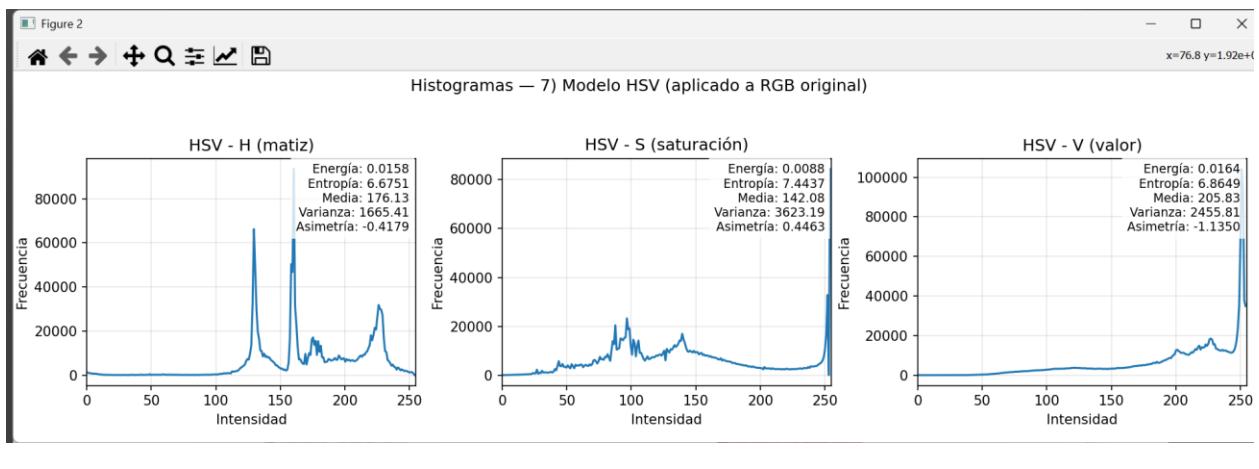
- **H (matiz)** mostró los tipos de color presentes.
- **S (saturación)** representó la intensidad del color (zonas grises o desaturadas).
- **V (valor)** capturó el brillo general de la imagen.

Los histogramas de **H** fueron más dispersos, reflejando diversidad de tonos; **S** tendió a agruparse hacia valores medios, mientras que **V** mantuvo una distribución similar al canal de luminancia.

Este modelo se percibió como **más intuitivo** para manipular el color y segmentar objetos por su tonalidad, lo que explica su uso en edición de imágenes y visión por computadora.



Histogramas



## 9) Comparación general y conclusiones parciales

Transformación	Efecto en la imagen	Entropía	Energía	Interpretación
<b>RGB original</b>	Información completa	Alta	Media	Imagen compleja y balanceada
<b>Escala de grises</b>	Pérdida de color, conservación de luminancia	Media	Alta	Menos variabilidad cromática
<b>Binarización</b>	Solo dos niveles (0,255)	Muy baja	Muy alta	Imagen simple y segmentada
<b>YIQ</b>	Separación luminancia–crominancia	Media	Media	Aísla la información relevante para compresión
<b>CMY</b>	Inversión cromática	Media	Media	Modelo sustractivo, útil en impresión
<b>HSV</b>	Descomposición perceptual (H,S,V)	Alta	Variable	Adecuado para segmentar por color o brillo

## 10) Cómo Reproducir

1. Clonar el repo o copiar la carpeta del proyecto.
  2. En VS Code, seleccionar intérprete: anaconda3/envs/python.exe.
  3. Instalar dependencias probadas: **Requisitos (mínimos)**
    1. **Python 3.9+** (tu entorno “Nirvana” usa 3.9.18)
    2. **NumPy**
    3. **Matplotlib** (con widgets Button y Slider)
    4. **PyQt5** (backend Qt5Agg para las ventanas)
    5. **Pillow (PIL)**
    6. **OpenCV-Python (cv2)**
    7. **scikit-image**
  4. Colocar imagen de su preferencia en sus archivos
  5. Ejecutar el comando de la sección Ejecución / Uso.
- 

## 11) Conclusión

La práctica ha permitido comprender la utilidad de diferentes modelos de color para representar y procesar imágenes. A través de la conversión de una imagen del espacio de color RGB a otros modelos como YIQ, CMY y HSV, se han observado cómo las imágenes pueden ser descritas y manipuladas de manera más eficiente dependiendo del contexto y del propósito de la tarea.

- **YIQ** es útil en **compresión y transmisión de video**, ya que separa la luminancia de la crominancia, permitiendo reducir el ancho de banda sin perder detalles importantes del brillo.
- **CMY** es esencial en la **impresión a color**, donde se utilizan tintas cian, magenta y amarilla para generar colores.
- **HSV** es particularmente útil en aplicaciones de **edición de imágenes y segmentación**, ya que su estructura es más intuitiva y alineada con la percepción humana de los colores.

El uso de **histogramas** como herramienta de análisis visual de los valores de intensidad en cada uno de los componentes de los modelos de color permite una comprensión más profunda de la distribución y el contraste de los colores. En general, los modelos de color permiten trabajar de

manera más eficiente en diversas aplicaciones, desde la compresión de video hasta la edición de imágenes y la impresión.

Esta práctica demuestra cómo los diferentes modelos de color se pueden aplicar dependiendo de las necesidades de procesamiento de la imagen, y cómo cada modelo tiene ventajas y limitaciones según el contexto en el que se utilice.

---

## 12) Reflexión en equipo

**Como equipo, uno de los mayores retos fue organizar el trabajo y comprender la estructura adecuada de la práctica para poder presentarla correctamente. Al inicio, tuvimos dificultades para coordinar las tareas, dividir el código y mantener un formato uniforme en el documento. Sin embargo, gracias a las retroalimentaciones de la profesora, fuimos identificando los puntos débiles de nuestro trabajo y aprendimos a mejorar tanto la parte técnica como la presentación escrita.**

**También aprendimos la importancia de trabajar de forma colaborativa y comunicativa, revisando juntos los resultados, verificando que el código funcionara en todos los equipos y que cada parte estuviera bien documentada. Esta práctica nos permitió comprender mejor el proceso de análisis de imágenes, pero también nos dejó una enseñanza sobre la organización, la responsabilidad compartida y la mejora continua en el trabajo en equipo.**

---

## 13) Extensión de la práctica. Modelos de color

Sesión 12. Reto grupal: Comparación de modelos de color

Modelo RGB (Red, Green, Blue)

**¿Qué canal parece tener más información visual?**

El canal verde (G) contiene más información visual, ya que el ojo humano es más sensible a ese rango de color y suele conservar mejor los detalles y contrastes.

### **¿Cómo se combinan los colores para formar otros?**

En este modelo los colores se generan por adición de luz. Al combinar los tres colores primarios (rojo, verde y azul) en distintas intensidades se obtienen todos los tonos del espectro visible. Por ejemplo, rojo + verde = amarillo, verde + azul = cian, y rojo + azul = magenta.

### **¿Por qué este modelo es el más usado en pantallas?**

Porque las pantallas, monitores y cámaras trabajan con luz emitida. El modelo RGB se adapta a la forma en que los dispositivos electrónicos producen color mediante la mezcla aditiva de esos tres componentes básicos.

Modelo CMY (Cyan, Magenta, Yellow)

### **¿Qué diferencias notan respecto al RGB?**

A diferencia del RGB, el modelo CMY es sustractivo, no aditivo. En lugar de sumar luz, parte del blanco y resta longitudes de onda para crear los colores, lo que invierte los valores de los canales.

### **¿Por qué este modelo se usa en impresión?**

Porque en la impresión los colores se obtienen mediante tintas que absorben luz. Al superponer las capas de cian, magenta y amarillo sobre un fondo blanco, se reflejan los colores deseados.

### **¿Cómo se genera el negro en este modelo?**

El negro se produce al combinar C, M y Y en altas intensidades, aunque en la práctica se agrega una tinta negra adicional (modelo CMYK) para lograr un negro más profundo y ahorrar tinta de color.

Modelo HSV (Hue, Saturation, Value)

### **¿Qué representa cada canal?**

- Hue (matiz): el tipo de color (rojo, verde, azul, etc.).
- Saturation (saturación): la intensidad o pureza del color.
- Value (valor): el nivel de brillo o luminosidad.

### **¿Por qué este modelo es útil para edición de color?**

Porque separa la tonalidad del brillo y la saturación, permitiendo modificar solo una propiedad sin alterar las demás. Esto facilita la edición precisa, la corrección de color y la segmentación en visión por computadora.

## **¿Cómo cambia la percepción del color al modificar la saturación o el valor?**

Al reducir la saturación, los colores se vuelven más grises y apagados; al cambiar el valor, se vuelven más claros u oscuros, alterando directamente su luminosidad y contraste visual.

## **Comparación final**

### **¿Cuál modelo les pareció más intuitivo?**

El modelo HSV fue el más intuitivo porque describe el color tal como lo percibimos: por su tono, intensidad y brillo.

---

## **13)Referencias**

### **Libros**

Gonzalez, R. C., & Woods, R. E. (2018). *Digital Image Processing* (4th ed.). Pearson.

Referencia base sobre procesamiento digital de imágenes, histogramas y modelos de color (RGB, CMY, YIQ, HSV).

Pratt, W. K. (2007). *Digital Image Processing: PIKS Scientific Inside* (4th ed.). John Wiley & Sons.

Consultado para comprender los fundamentos de luminancia, crominancia y métricas de energía y entropía.

---

### **librerías de Python**

Harris, C. R., Millman, K. J., van der Walt, S. J., et al. (2020). *Array programming with NumPy*. *Nature*, 585(7825), 357–362. <https://doi.org/10.1038/s41586-020-2649-2>

Hunter, J. D. (2007). *Matplotlib: A 2D graphics environment*. *Computing in Science & Engineering*, 9(3), 90–95. <https://doi.org/10.1109/MCSE.2007.55>

Bradski, G. (2000). *The OpenCV Library*. *Dr. Dobb's Journal of Software Tools*. <https://opencv.org/>

Van der Walt, S., Schönberger, J. L., Nunez-Iglesias, J., et al. (2014). *scikit-image: Image processing in Python*. *PeerJ*, 2, e453. <https://doi.org/10.7717/peerj.453>

Clark, A. (2024). *Pillow (PIL Fork) Documentation*. <https://pillow.readthedocs.io/>

Matplotlib Developers. (2024). *Matplotlib Widgets API (Button, Slider)*.  
[https://matplotlib.org/stable/api/widgets\\_api.html](https://matplotlib.org/stable/api/widgets_api.html)

---

### **Recursos complementarios y de apoyo**

Wikimedia Commons. (2024). *Color Models: RGB, CMY, HSV, YIQ*.  
[https://commons.wikimedia.org/wiki/Color\\_models](https://commons.wikimedia.org/wiki/Color_models)

Tutorialspoint. (2024). *Python Image Processing – Color Spaces*.  
[https://www.tutorialspoint.com/python\\_image\\_processing/python\\_color\\_spaces.htm](https://www.tutorialspoint.com/python_image_processing/python_color_spaces.htm)

OpenAI. (2025). *Asistencia técnica para desarrollo de prácticas de análisis de imágenes*.