Universidad
Rey Juan Carlos

Práctica 1. Espacio de color YCbCr y codificación entrópica de imágenes

Estándares de Comunicación de Audio y Video

David Santa Cruz Del Moral

48147936N

Curso 2023/2024

ÍNDICE

Ejercicio 1. Conversión del espacio de color RGB a YCbCr.

1.1. Realice un script que cargue la imagen arbol_640x426.jpg utilizando la función imread y represéntela mediante la función imshow. ¿Qué representa cada una de las dimensiones de la matriz que almacena la imagen? Nota: la función imread solo trabaja con valores enteros sin signo de 8 bits.

Ejercicio 1

1.1

```
arbol = "arbol_640x426.jpg";
imread(arbol)
imshow("arbol_640x426.jpg")
```

Ilustración 1: Código usado para el apartado 1.1

```
ans = 426×640×3 uint8 array
ans(:,:,1) =

    15    14    15    16    15    13    17    23    14    14    14    14    15
    15    14    15    16    16    15    18    23    11    14    16    17    16
    16    14    14    15    14    13    13    16    13    15    16    15    13
    16    15    14    15    14    12    11    12    15    16    15    13    12
    15    14    14    16    17    16    15    15    14    15    16    16    15
    15    14    15    15    16    16    15    15    16    15    15    15    15
    15    16    16    15    14    14    13    13    17    15    14    13    14
    14    15    16    16    15    15    16    16    14    15    16    16    16
    14    14    15    15    15    15    15    15    15    15    16    15    14
    14    14    15    15    15    15    15    15    14    15    15    15    15
    15    15    15    15    15    15    15    15    14    15    15    15    16
    15    15    15    15    15    15    15    15    15    15    15    15    15
    16    15    15    15    15    15    15    16    16    16    15    14    14
    16    16    15    15    15    15    15    16    16    15    15    15    15
    16    16    15    15    15    15    15    16    16    15    15    15    15
     ⋮
```

Se ha cargado y enseñado una imagen que previamente había sido guardada en una variable. Como resultado se muestra la matriz correspondiente a la imagen y la propia fotografía.

Ilustración 2: Solución de la ejecución del apartado 1.1

Las dimensiones de la matriz son las indicadas en "*ans*". Estas dimensiones de la matriz representan:

1. **Ancho (width):** Cantidad de píxeles en una fila de la imagen.
2. **Altura (height):** Cantidad de píxeles en una columna de la imagen.
3. **Canales de color (channels):** Número de componentes de color por píxel (3 por ser una imagen en formato RGB).

Esta información es útil para entender cómo se organiza la representación digital de la imagen en la matriz.

1.2. A continuación, genere una función (en un script separado del principal) que permita convertir una imagen almacenada en RGB al espacio de color YCbCr. La función estará formada por la siguiente estructura:

[YCbCr] = RGBTOYCbCr(RGB)

Para convertir del espacio de color RGB al espacio YCbCr se utilizarán las expresiones de conversión BT.709:

$Y = 0.2126 * R + 0.7152 * G + 0.0722 * B + 16$

$C_B = 0.5389 * (B - Y) + 128$

$C_R = 0.6350 * (R - Y) + 128$

Genere en una única figura las imágenes de las 3 componentes YCbCr en un vector de imágenes de 1x3.

```
function [YCbCr] = RGBtoYCbCr(RGB)
    % Leer la imagen
    img = imread(RGB);
    RGB = double(RGB);
    R = img(:,:,1);
    G = img(:,:,2);
    B = img(:,:,3);

    % Aplicar las expresiones de conversión
    Y = 0.2126 * R + 0.7152 * G + 0.0722 * B + 16;
    Cb = 0.5389 * (B - Y) + 128;
    Cr = 0.6350 * (R - Y) + 128;

    % Combinar las tres componentes en un solo vector de imágenes 1x3
    YCbCr = uint8(cat(3, Y, Cb, Cr));

    % Mostrar la imagen original y las componentes Y, Cb, Cr en una única figur
    figure;
    subplot(2, 2, 1), imshow(img), title('Imagen Original');
    subplot(2, 2, 2), imshow(Y), title('Y Component');
    subplot(2, 2, 3), imshow(uint8(Cb)), title('Cb Component');
    subplot(2, 2, 4), imshow(uint8(Cr)), title('Cr Component');
end
```

Ilustración 3: Función para pasar de RGB al espacio YCbCr

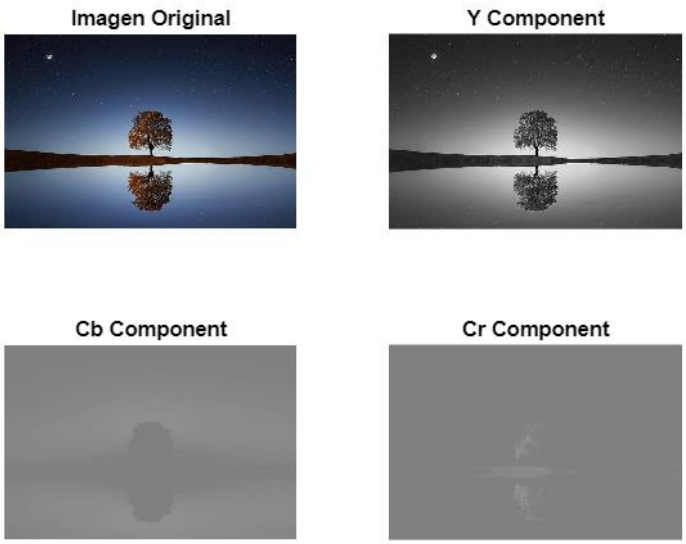

Ilustración 4: Resultado del apartado 1.2

Esta función mostrará una figura con la imagen original y las componentes Y, C_b y C_r. De la imagen original se sacan las componentes RGB para luego usarlas como medio para sacar las componentes YC_bC_r y así poder observar las diferentes componentes de forma separada.

Ejercicio 2. Cálculo de la entropía y representación con histograma.

2.1. En este ejercicio se realizará un script para calcular la entropía (bits/pixel) de las imágenes "caballos_640x427.jpg", "arbol_640x426.jpg" y "niebla_640x456.jpg". Para ello, utilice la función imread para cargar las imágenes en RGB y después utilice la función rgb2gray para convertir las imágenes RGB a escala de grises (8 bits/pixel, valores entre 0 y 255). Posteriormente, realice el algoritmo en una función que se llame CalculaEntropia para calcular la entropía (bits/pixel) de cada una de las imágenes (no está permitido utilizar la función de Matlab entropy). ¿Qué imagen presenta una mayor entropía? ¿Qué significado tiene?

```
caballos = "caballos_640x427.jpg";
arbol = "arbol_640x426.jpg";
niebla = "niebla_640x456.jpg";

% Carga y conversión de las imagenes a escala de grises
caballos_gris = rgb2gray(imread(caballos));
arbol_gris = rgb2gray(imread(arbol));
niebla_gris = rgb2gray(imread(niebla));


% Calculo de la entropía de las imagenes
entropia_caballos = CalculaEntropia(caballos_gris);
entropia_arbol = CalculaEntropia(arbol_gris);
entropia_niebla = CalculaEntropia(niebla_gris);

% Resultado de las entropías de las imagenes
disp(['Entropía de la imagen de caballos: ' num2str(entropia_caballos) ' bits/pixel']);
disp(['Entropía de la imagen de arbol: ' num2str(entropia_arbol) ' bits/pixel']);
disp(['Entropía de la imagen de niebla: ' num2str(entropia_niebla) ' bits/pixel']);
```

Ilustración 5: Código usado para resolver el apartado 2.1

```
% Función para calcular la entropía de una imagen en escala de grises
function entropia = CalculaEntropia(imagen)
    % Obtener histograma de la imagen
    histograma = imhist(imagen);

    % Calcular la probabilidad de cada nivel de intensidad
    probabilidad = histograma / sum(histograma);

    % Eliminar entradas con probabilidad cero
    probabilidad = probabilidad(probabilidad > 0);

    % Calcular la entropía (Fórumla tema 2)
    entropia = -sum(probabilidad .* log2(probabilidad));
end
```

Ilustración 6: Función para calcular la entropía

```
Entropía de la imagen de caballos: 6.4665 bits/pixel
Entropía de la imagen de arbol: 7.3369 bits/pixel
Entropía de la imagen de niebla: 7.4851 bits/pixel
```

Ilustración 7: Resultado del apartado 2.1

La entropía indica el número mínimo medio de bits necesarios para codificar los símbolos de una fuente de información y mide la variabilidad de la fuente, por lo que cuanto más baja sea esta, menor número medio de bits necesarios por símbolo.

Como se puede observar, la imagen con mayor entropía es "*niebla_640x456.jpg*". Esto significa que es la imagen que más número medio de bits necesita para codificar cada símbolo.

2.2. Obtenga el valor de entropía conjunta de las imágenes del apartado 2.1.

```
% Concatenación las imágenes en un solo vector
vector_conjunto = [caballos_gris(:); arbol_gris(:); niebla_gris(:)];

% Calcular la entropía conjunta
entropia_conjunta = CalculaEntropia(vector_conjunto);

% Mostrar la entropía conjunta
disp(['Entropía conjunta de las imágenes: ' num2str(entropia_conjunta) ' bits/pixel']);
```

Ilustración 8: Código usado para el apartado 2.2

Se han juntado todas las imágenes en escala de grises usadas en el apartado (*Ilustración 5*) anterior en un solo vector. Este vector se ha introducido en la función mostrada en *Ilustración 6* para calcular la entropía conjunta de las imágenes. El resultado que da en MATLAB es:

Entropía conjunta de las imágenes: 7.5777 bits/pixel.

Ejercicio 3. Codificación Huffman.

En este ejercicio se realizará la codificación Huffman de una imagen de 8x8 píxeles.

Resuelva los siguientes apartados en un script:

3.1. Cargue la imagen "icono_8x8.jpg", conviértala en escala de grises y posteriormente en blanco y negro.

```
% Imagen en escala de grises
icono = "icono_8x8.jpg";
icono_gris = rgb2gray(imread(icono));

% Conversión de la imagen a blanco y negro
imagen_bw = imbinarize(icono_gris);

% Mostrar las imágenes
figure;
subplot(3, 1, 1);
imshow(icono);
title('Imagen Original')

subplot(3, 1, 2);
imshow(icono_gris);
title('Imagen en Escala de Grises');

subplot(3, 1, 3);
imshow(imagen_bw);
title('Imagen en Blanco y Negro');
```

Ilustración 9: Código usado para realizar el apartado 3.1

En primer lugar, se ha cargado la imagen *"icono_8x8.jpg"* y se ha pasado a escala de grises mediante la función *rgb2gray*. El resultado se ha guardado en la variable "*icono_gris*" que se usará más tarde para mostrar la imagen monocromo.

Posteriormente, para obtener la imagen en blanco y negro se usa la función "*imbinarize*".

La última parte del código es la puesta en pantalla de las imágenes obtenidas junto con la original. Nótese que la imagen original y la monocromo son iguales.

Ilustración 10: Resultado del apartado 3.1

3.2. Obtenga las probabilidades de ocurrencia de cada símbolo de la imagen en blanco y negro.

```
% Histograma de la imagen blano y negro
histograma = imhist(imagen_bw);

% Calcular la probabilidad de ocurrencia de cada símbolo
total_pixeles = numel(imagen_bw);
probabilidades = histograma / total_pixeles;

% Mostrar las probabilidades
disp('Probabilidad de ocurrencia del color blanco (símbolo 1):');
disp(probabilidades(1));
disp('Probabilidad de ocurrencia del color negro (símbolo 2):');
disp(probabilidades(2));

% Plot
figure;
subplot(2, 1, 1);
imhist(imagen_bw);
title('Histograma 1');

subplot(2, 1, 2);
histogram(imagen_bw);
title('Histogrma 2')
```

Ilustración 11: Código usado para realizar el apartado 3.2

Para calcular las probabilidades de ocurrencia de cada símbolo de la imagen en blanco y negro es necesario obtener su histograma, donde se muestra el símbolo y el número de pixeles asociado a cada pixel. Con estos datos se puede obtener el número total de pixeles que hay en la imagen con la función "*numel*", que devuelve el número de elementos que hay un array; con el valor obtenido (64 pixeles) se obtienen las probabilidades de ocurrencia dividiendo dicho valor al histograma, es decir, se consigue un histograma relativo que mide la probabilidad de ocurrencia de un nivel.

```
Probabilidad de ocurrencia del color negro (símbolo 0):
    0.5312
Probabilidad de ocurrencia del color blanco (símbolo 1):
    0.4688
```

Ilustración 12: Solución del apartado 3.2

Las probabilidades de ocurrencia de cada símbolo son las mostradas en la *Ilustración 12*. La operación que ha hecho el programa ha sido:

$$P(n_i) = \frac{h(n_i)}{\sum_{\text{para todo i}} h(n_i)}$$

siendo $h(n_i)$ el número de pixeles del símbolo i.

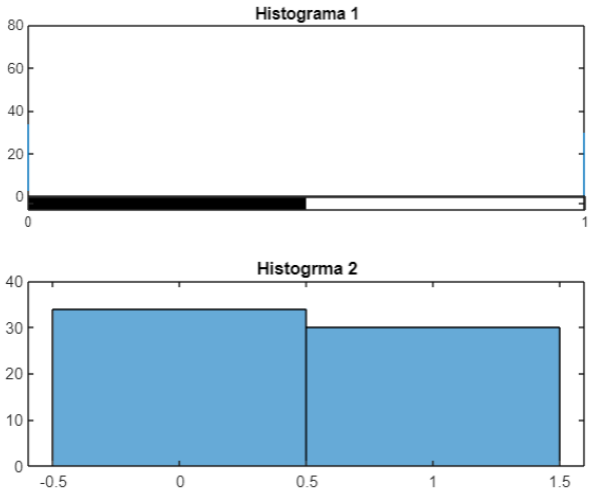

Ilustración 13: Histogramas del apartado 3.2

Si se hacen las operaciones a mano, se puede observar que los resultados son correctos:

Símbolo 0: 34/64 = 0.53125

Símbolo 1: 30/64 = 0.46875

Por otro lado, se muestra el histograma de la imagen en blanco y negro de dos formas diferentes: con la función "*imhist*" y con "*histogram*".

3.3. Obtenga el valor en código binario de la imagen en blanco y negro siguiendo la lógica de la codificación Huffman (sin utilizar función de alto nivel que realice la codificación).

```
function codHuff = CalcularCodHuffman(prob)
    if prob(1) > prob(2)
        codHuff = ["1", "0"];
    else
        codHuff = ["0", "1"];
    end
end
```

Ilustración 14: Función para obtener los códigos Huffman

```
% Calculo de los códigos Huffman
simbolos = [0 1];
prob = transpose(probabilidades);
codHuff = CalcularCodHuffman(prob);

% Mostrar códigos
disp('Ej. 3.3')
disp('Código Huffman del color negro (símbolo 0):');
disp(codHuff(1));
disp('Código Huffman  del color blanco (símbolo 1):');
disp(codHuff(2));
```

Ilustración 15: Código usado para resolver el apartado 3.3

La codificación de Huffman consiste en asignar códigos cortos a los símbolos más probables, y códigos más largos para los símbolos menos probables. Como en este caso solo tenemos dos símbolos por tratarse de una imagen binaria (blanco y negro), solo es necesario un bit por código. Por tanto, la función que sigue la lógica para la codificación Huffman es muy sencilla, se trata de un bloque combinacional que asigna el código 0 al menos probable y el 1 al más probable.

```
Código Huffman del color negro (símbolo 0):
     1
Código Huffman  del color blanco (símbolo 1):
     0
```

Ilustración 16: Resultado del apartado 3.3

3.4. Convierta a decimal el código binario generado en el apartado anterior y obtenga el rendimiento del codificador.

```
% Función para convertir un código binario a decimal
function decimalValues = binary2decimal(codigosBinarios)
    % Convierte códigos binarios a valores decimales

    numCodigos = length(codigosBinarios);
    decimalValues = zeros(1, numCodigos);

    for i = 1:numCodigos
        decimalValues(i) = bin2dec(codigosBinarios{i});
    end
end
```

Ilustración 17: Función para pasar de binario a decimal

En la *Ilustración 17* se muestra la función que se usa para pasar de binario a decimal los códigos de Huffman. Simplemente se recorre el array que contiene los códigos y se van convirtiendo uno a uno con la función "*bin2dec*".

En la *Ilustración 18* se observa la función que sirve para calcular la eficiencia del codificador. Esta función simplemente aplica la fórmula:

$$\eta = \frac{H}{L} \qquad \text{Donde} \qquad \mathrm{L} = \sum_{i=1}^{M} p_i l_i \ \ bits/simbolo$$

Por tanto, se introducen la entropía de la imagen (calculada con la función mostrada en *Ilustración 6*), las probabilidades y los códigos Huffman. Con estos parámetros se pueden calcular L y, así, la eficiencia.

```
% Función para calcular el rendimiento del codificador
function eficiencia = calcular_eficiencia(entropia, listaProbabilidades, codigosHuffman)
    % Calcular la longitud media de símbolo (L)
    numCodigos = length(codigosHuffman);
    longitudesCodigos = zeros(1, numCodigos);

    for i = 1:numCodigos
        longitudesCodigos(i) = length(codigosHuffman{i});
    end

    longitudMedia = sum(longitudesCodigos .* listaProbabilidades);

    % Calcular la eficiencia mediante la relación entropía/L
    eficiencia = entropia / longitudMedia;
end
```

Ilustración 18: Función para obtener la eficiencia del codificador

```
% Conversión a decimal el código binario y obtención el rendimiento del codificador
entropia_img = CalculaEntropia(imagen_bw);
decodedDecimal = binary2decimal(codHuff);
efficiency = calcular_eficiencia(entropia_img, prob, codHuff);

% Resultados
disp('Ej. 3.4')
disp(['Números decimales: ' num2str(decodedDecimal)]);
disp(['Eficiencia: ' num2str(efficiency)]);
```

Ilustración 19: Código usado para solucionar el apartado 3.4

```
Ej. 3.4
Números decimales: 1  0
Eficiencia: 0.99718
```

Ilustración 20: Solución apartado 3.4