

UNIVERSIDAD DE MURCIA

COMPRESIÓN MULTIMEDIA

PCEO

Entrega Opcional: Prácticas 3 y 4

Autor:

Jose Antonio Lorencio Abril
(joseantonio.lorencioa@um.es)

Profesor:

Jesús Damián Jiménez Re

Convocatoria Enero 2021/22

Índice

1. Práctica 3. Audio, imágenes y mapas de colores en Matlab	3
1.1. Ejercicio 1.	3
1.2. Ejercicio 2.	5
1.3. Ejercicio 3.	14
1.4. Ejercicio 4 . (Opcional)	16
2. Práctica 4. Compresión de imágenes.	18
2.1. Ejercicio 1.	18
2.2. Ejercicio 2.	25
2.3. Ejercicio 3.	32
3. Conclusión	36

1. Práctica 3. Audio, imágenes y mapas de colores en Matlab

1.1. Ejercicio 1.

Escribe una función *testdct(fname)* que:

1. Lea un archivo de imagen *fname* y genere una matriz truecolor RGB (el archivo puede estar en formato BMP o JPEG, y el tipo de color original puede ser indexado o truecolor)

```
function [mse,dmaxdifer] = testdct(fname)
% Entradas:
% fname: Un string con nombre de archivo, incluido sufijo
% Salidas:
% mse: Error cuadratico medio entre matrices original y reconstruida
% dmaxdifer: Maxima diferencia entre pixeles homologos

% Lee info de imagen
s=imfinfo(fname);
TO=s.FileSize; % Tamano original

% Carga archivo, convierte a RGB y calcula tamano
if strcmp(s.ColorType,'indexed') & s.BitDepth==8
    tipo=0;
    [Y,map]=imread(fname);
    X=uint8(round(ind2rgb(Y,map)*255));
elseif strcmp(s.ColorType,'truecolor') & s.BitDepth==24
    tipo=1;
    X=imread(fname);
else
    disp('Formato no reconocido. Terminado.');
    return;
end
```

Cuadro 1: Fragmento de *testdct(fname)*

2. Convierta la matriz RGB leída del espacio de color RGB al espacio YCbCr

```
% Pasamos de RGB a YCbCr
% Formato double para poder procesar con DCT
YCbCr = double(rgb2ycbcr(RGB));
```

Cuadro 2: Fragmento de *testdct(fname)*

3. Amplíe sus dimensiones a múltiplos de 8 píxeles

```
% Si tamano no es multiplo de 8, replica ultimas filas y/o columnas
[m n plan]=size(X); % tamano original
mpad = rem(m,8); if mpad>0, mpad = 8-mpad; end
npad = rem(n,8); if npad>0, npad = 8-npad; end
Xamp=cat(1, XYCbCr, repmat(XYCbCr(m,:,:),[mpad 1 1]));
Xamp=cat(2, Xamp, repmat(Xamp(:,n,:),[1 npad 1]));
mamp=m+mpad; % Alto ampliado de la imagen (num de filas)
namp=n+npad; % Ancho ampliado de la imagen (num de columnas)
```

Cuadro 3: Fragmento de *testdct(fname)***4. Calcule la transformada DCT de la matriz ampliada en bloques de 8x8 píxeles**

```
% Calculamos la transformada DCT de la matriz ampliada
% en bloques de 8x8
XampDCT = imdct(Xamp);
```

Cuadro 4: Fragmento de *testdct(fname)***5. Calcule la transformada inversa iDCT, también por bloques, a partir de la matriz de coeficientes obtenida en el paso anterior**

```
% Calculamos la transformada inversa
XampInvDCT = imidct(XampDCT);
```

Cuadro 5: Fragmento de *testdct(fname)***6. Convierta la matriz resultante al espacio RGB**

```
% Convierte a espacio de color RGB
% Para ycbcr2rgb: % Intervalo [0,255]->[0,1]->[0,255]
Xrecrd=round(ycbcr2rgb(XampInvDCT/255)*255);
Xrec=uint8(Xrecrd);
```

Cuadro 6: Fragmento de *testdct(fname)***7. Reduzca sus dimensiones al tamaño original**

```
% Repone el tamano original
Xrec=Xrec(1:m,1:n, 1:3);
```

Cuadro 7: Fragmento de *testdct(fname)***8. Almacene la matriz de la imagen en un nuevo archivo BMP**

```
% Genera nombre archivo descomprimido <nombre>_des.bmp
[pathstr,name,ext] = fileparts(fname);
nombrecomp=strcat(name,'_des','.bmp');
% Guarda archivo descomprimido
imwrite(Xrec,nombrecomp,'bmp')
```

Cuadro 8: Fragmento de *testdct(fname)*

9. Muestre el error cuadrático medio MSE producido por la doble conversión y la mayor diferencia entre píxeles homólogos

```
% Calculo de MSE
mse=(sum(sum((double(Xrec)-double(X)).^2)))/(m*n*3);
% Test de valor de diferencias double
ddifer=abs(double(Xrec)-double(X));
dmaxdifer=max(max(ddifer));
```

Cuadro 9: Fragmento de *testdct(fname)*

Para escribir la función, sigue las indicaciones dadas en la presentación de la práctica y usa las funciones auxiliares disponibles en la librería: imlee, imdct, imidct e imescribe. Apícalas al menos a cinco de los archivos de imagen (Img00.bmp, Img01.bmp, ... Img20.jpg) disponibles en la librería, e indica el máximo MSE obtenido y la máxima diferencia entre píxeles homólogos. ¿A qué se debe el error cometido?

Voy a aplicarlo a las imágenes 00, 03, 06, 11 y 15, y los resultados pueden verse en la tabla siguiente:

Imagen	MSE	MaxDiffer
Img00	0.3639	2
Img03	0.3959	2
Img06	0.3800	2
Img11	0.4452	2
Img15	0.1397	1

Cuadro 10: MSE y MaxDiffer obtenidos con *testdct* para las imágenes seleccionadas

Las imágenes de diferencias a simple vista se ven completamente negras todas ellas, por lo que no hay mucho que comentar, salvo que las imágenes originales y las recuperadas son prácticamente iguales.

El error cometido se debe al paso de redondeo que se aplica tanto al principio de la función (cuantización) como al final, al pasar al espacio RGB. Como se observa, no obstante, los errores cometidos son muy pequeños, y son inapreciables a simple vista.

1.2. Ejercicio 2.

Analice el script *testQuantDCT8x8.m* y haga lo siguiente:

- Explique qué es lo que hace

Primero calcula la transformada discreta del coseno (DCT) de la imagen p , que es una imagen 8x8. Tras esto, normaliza los coeficientes obtenidos y usa un valor umbral para descartar los coeficientes que se consideran (respecto a ese umbral) irrelevantes.

De los valores seleccionados, se toma el mínimo como un escalón mínimo, aunque este no es valor del escalón que utiliza, sino uno que introducimos a mano. Este escalón se utiliza para calcular los valores cuantizados.

Tras esto, se supone que los datos son codificados, enviados y recibidos, y pasa a su recuperación. Para ello, calcula la aproximación a la DCT multiplicando los valores cuantizados por el escalón, y a estos valores les aplica la Transformada Inversa del coseno. Ahora habremos recuperado la imagen, aunque con pérdida de información en el proceso, debido al proceso de cuantización/descuantización.

- **Asigne a *escalon* el valor 1. Compare visualmente los valores *originales* y *recuperados*. Explique por qué son diferentes. Explique la situación par valores de escalón 0.5,2,5,...**

La diferencia se debe, como hemos señalado en el punto anterior, al proceso de cuantización/descuantización, en el que se produce pérdida de información. La razón es que al cuantizar estamos dividiendo y redondeando al entero más cercano. Esto provocará pérdida de información siempre que los valores dejen de ser enteros al dividir.

Con 0.5, el resultado es mejor que con 1, y esto tiene mucho sentido, ya que estamos multiplicando por 2, lo cual, en cierto sentido, que los valores sean 'más enteros'. Al usar 2 como escalón, el resultado sigue siendo el esperado, es peor que usar 1, ya que dividir por 2 produce decimales adicionales. No obstante, con 5 el resultado parece ser mejor (visualmente) que con los valores anteriores. La razón es que al aumentar el escalón, acercamos los valores a 0 cuando dividimos, y al redondear es más probable que obtengamos 0's. Razonablemente, el resultado respecto a las diferencias numéricas (MSE), empeora, pues estamos haciendo una mayor cuantización.

Al probar con escMin , obtenemos un escalón de 239. Esto produce una imagen recuperada totalmente gris, y esto sigue estando acorde con lo que conocemos. Al aumentar el valor, más valores serán 0, y llegado el punto en el que todos o casi todos lo sean, habremos perdido demasiada información para poder recuperar la imagen.

- **Asigne a *escalon* varios valores, como 1,2,5,... Compare visualmente el valor de *dct* y *quant*. ¿Cuándo se hacen nulos los valores cuantizados?**

Un valor de $\text{dct } d_{i,j}$ se hace 0 cuando $\text{abs}(d_{i,j}) < 0,5$.

- **Descomente la línea que permite visualizar el contenido de *seleccionados*' y asigne a *escalon* el valor *escMin* para el resto de los apartados.**

- **Para *umbral=0.5*. ¿Cuál es el valor del escalón de cuantización? ¿Cuántos valores no nulos hay en la matriz *quant*? ¿Por qué? ¿Cuántos coeficientes descuantizados, *desquant*, quedan? Compare la imagen recuperada con los valores recuperados. ¿Cuántas matrices base se están utilizando para recuperar la imagen? ¿Por qué?**

Se obtiene un valor $\text{escMin}=238.9954$, y en la matriz quant hay un único valor no nulo, ya que estamos dividiendo por ~ 239 , lo que hace que casi todos los valores tengan un módulo inferior a 0.5, y esto, tal como hemos visto en el apartado anterior, hace que los valores cuantizados se vayan a 0.

Al descuantizar, se obtiene un único valor, el único que había cuantizado, multiplicado por el escalón.

Solo se está utilizando una matriz para recuperar la imagen, pues se utilizan tantas matrices como coeficientes no nulos (la imagen recuperada es una combinación lineal de las matrices base).

Esto hace que la imagen recuperada sea únicamente una imagen en gris, totalmente diferente a la imagen original.

- **Responda a las preguntas del apartado anterior para valores del umbral 0.3, 0.2, 0.1, 0.05 y 0.01.**

Obsérvese que basta un umbral de 0.3 para recuperar una imagen suficientemente parecida a la original.

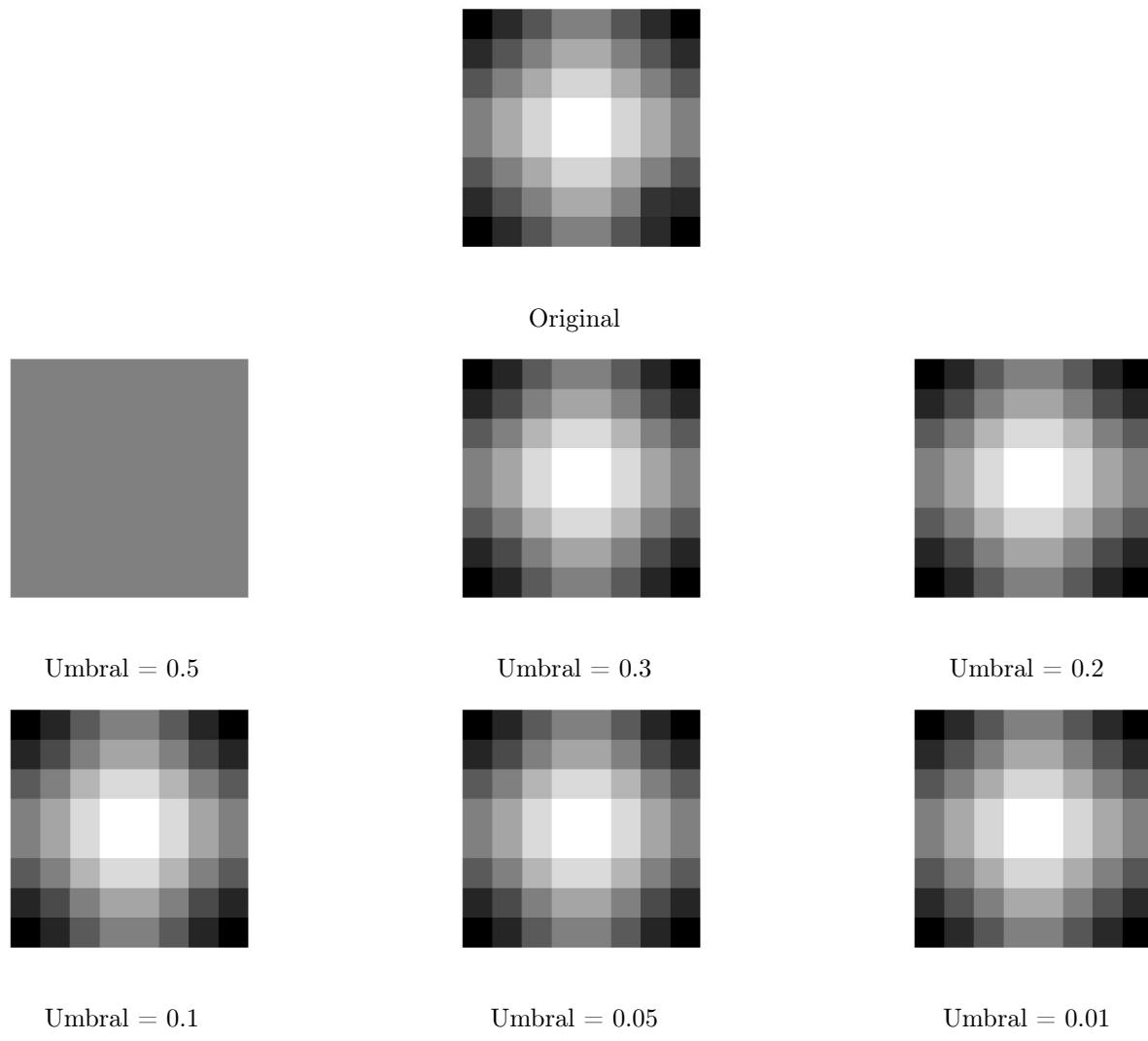


Figura 1: Imágenes recuperadas para diferentes valores de umbral

Umbral	escMin	Valores no nulos en quant	Coefficientes en desquant = Num matrices
0.5	238.9954	1	1
0.3	89.758	3	3
0.2	89.758	3	3
0.1	89.758	3	3
0.05	89.758	3	3
0.01	5.0339	5	5

Cuadro 11: Estudio del escalón para diferentes valores de umbral

- Como consecuencia de los dos apartados anteriores, ¿cuántos coeficientes transformados considera que son suficientes para codificar la imagen original sin perder demasiada calidad? Si sumara las 3 matrices base más relevantes, ¿qué figura obtendría? Tiene las 64 imágenes base de la DCT en las transparencias.

Parece que usar 3 coeficientes parece más que suficiente para codificar la imagen. También podríamos usar 5 y perder aún menos calidad, aunque la diferencia no parece ser excesivamente importante.

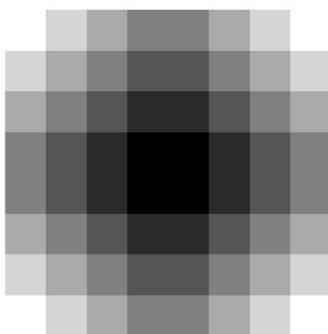
Las tres matrices más importantes son la (1,1), la (1,3) y la (3,1). Lo hacemos en Matlab:

```
%Aproximadamente, deben ser:
p1=[60,60,60,60,60,60,60; ...
    60,60,60,60,60,60,60; ...
    60,60,60,60,60,60,60; ...
    60,60,60,60,60,60,60; ...
    60,60,60,60,60,60,60; ...
    60,60,60,60,60,60,60; ...
    60,60,60,60,60,60,60; ...
    60,60,60,60,60,60,60];
p2=[60,40,20,00,00,20,40,60; ...
    60,40,20,00,00,20,40,60; ...
    60,40,20,00,00,20,40,60; ...
    60,40,20,00,00,20,40,60; ...
    60,40,20,00,00,20,40,60; ...
    60,40,20,00,00,20,40,60; ...
    60,40,20,00,00,20,40,60; ...
    60,40,20,00,00,20,40,60];
p3=p2';
p=p1+p2+p3
figure(1), imagesc(p), colormap(gray), axis square, axis off
pp=269*p1-90*p2-90*p3
figure(2), imagesc(pp), colormap(gray), axis square, axis off
```

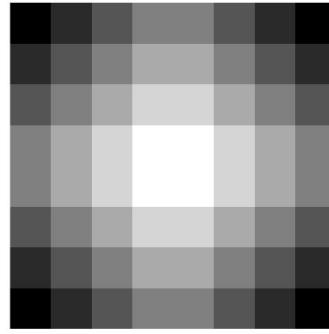
Cuadro 12: Código de las 3 matrices buscadas y su suma

He reconstruido a mano las 3 matrices que queríamos, y las he sumado directamente, así como usando los coeficientes obtenidos en el apartado anterior. Los resultados son:

Observamos como al sumar directamente, obtenemos algo muy similar a la imagen complementaria de la original. Al aplicar los coeficientes obtenidos en el proceso de cuantización y descuantización, como era de



Suma directamente



Suma con los coeficientes obtenidos anteriormente

Figura 2: Resultados de combinar las tres matrices base

esperar, recuperamos algo muy similar a la imagen original.

- Volviendo a $umbral=0.5$, ¿cuál es el valor de los píxeles de la imagen recuperada? Teniendo en cuenta la cantidad de coeficientes transformados que han sido retenidos, ¿podría predecir ese valor a partir de la imagen original?

El valor de todos los píxeles de la imagen recuperada es 29,8739. Dado que solo se retiene un coeficiente, parece sensato pensar que el valor con el que se recuperarán los píxeles será la media de los valores originales.

```
suma = 0;

for j=0:7
    for i=0:7
        suma = suma + p(i+1,j+1);
    end
end

media = suma/64

>> media = 29.8750
```

Cuadro 13: Cálculo de la media

Observamos que los valores son suficientemente cercanos para reafirmarnos en que se recupera, aproximadamente, el valor medio.

- Renombre la variable q y llámela p para los siguientes apartados.
- Asigne a $umbral$ los valores 0.5, 0.3, 0.2, 0.1, 0.05 y 0.01. Para cada valor, ¿cuál es el valor del escalón de cuantización? ¿Cuántos valores no nulos hay en la matriz $quant$? ¿Por qué? ¿Cuántos coeficientes descuantizados, $desquant$, quedan? Compare la imagen recuperada con los valores recuperados, ¿cuántas matrices base se están utilizando para recuperar la imagen? ¿Por qué?

La razón de que haya tan pocos (y cada vez menos) valores nulos es que los valores cambian muy repentinamente. Esto, para expresarlo como combinación lineal de elementos de la base, necesitará muchos coeficientes no nulos, para poder capturar la gran discontinuidad y la poca correlación que hay en los datos a lo largo de la discontinuidad, ya que sabemos que la DCT funciona mejor cuando los datos están correlacionados.

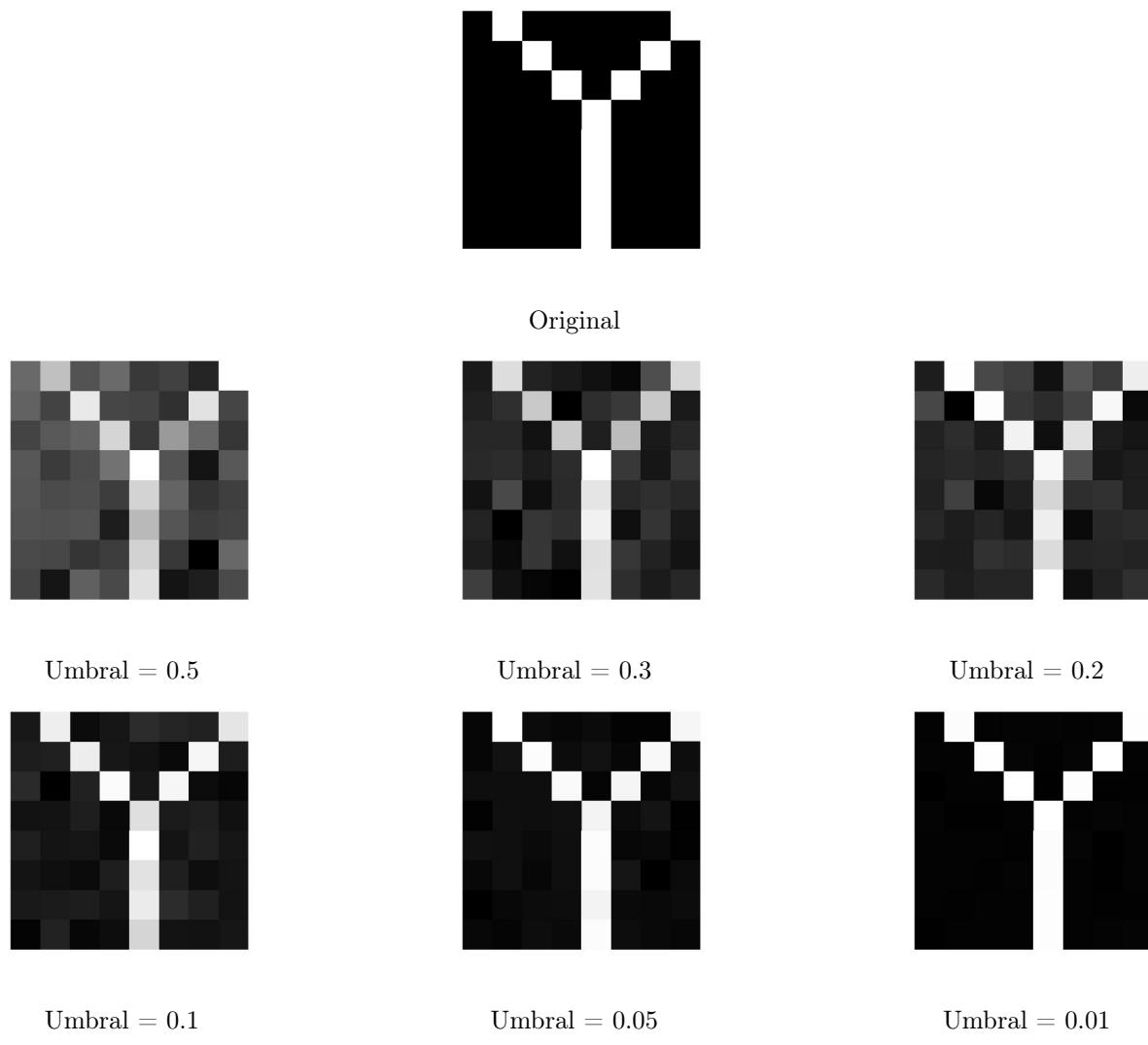


Figura 3: Imágenes recuperadas para diferentes valores de umbral

Umbral	escMin	Valores no nulos en quant	<i>Coeficientes en desquant</i> =	<i>Num matrices</i>
0.5	6.8484	22		22
0.3	4.1857	32		32
0.2	2.7815	35		35
0.1	1.6332	50		50
0.05	0.7079	57		57
0.01	0.1516	64		64

Cuadro 14: Estudio del escalón para diferentes valores de umbral

Como observamos en la figura, esta vez se requiere un umbral muy bajo para poder recuperar una imagen que no sea demasiado distinta a la original.

- **Como consecuencia del análisis anterior, ¿cuántos coeficientes transformados considera que son suficientes para codificar la imagen original sin perder demasiada calidad?**

Parece que hasta que no elegimos un umbral de 0.05, la imagen recuperada es demasiado diferente de la original. Por otro lado, al elegir el umbral de 0.01 recuperamos la imagen original, pero necesitamos 64 coeficientes, por lo que la codificación no consigue reducir el número de valores requeridos para representar la imagen. Así, es razonable elegir el umbral 0.05 y utilizar 57 coeficientes no nulos. (Aunque tampoco ahorraremos mucho espacio, pero las opciones de gran ahorro devuelven malas imágenes).

- **Contraste su percepción con el valor del error cuadrático medio entre la imagen original y la recuperada**

```
% Calculo de MSE
mse=(sum(sum((double(recuperados)-double(originales)).^2)))/(8*8)
```

Cuadro 15: Cálculo del MSE

Umbral	MSE
0.5	2.71
0.3	1.1954
0.2	0.6963
0.1	0.2468
0.05	0.045
0.01	0.0022

Cuadro 16: Valores de MSE para cada umbral

Vemos que andábamos en la dirección correcta, y que el umbral 0.05 parece una buena opción. Aunque, tal y como mencionábamos, si fuese crítico ahorrar espacio, podríamos seleccionar un umbral de 0.1, pero aumenta mucho el mse.

- **Justifique por qué los resultados para estas dos imágenes son tan diferentes**

Como he comentado, se debe a las discontinuidades que presenta la segunda imagen, mientras que la primera cambia de colores de una forma mucho más gradual, lo que hace que haya una gran correlación entre los valores. También es importante la asimetría, porque las imágenes base se adaptan mal a la falta de la misma.

En conclusión, se necesitarán más coeficientes al haber más cambios bruscos de tonalidad y al presentar un mayor grado de asimetría.

- Construya otras imágenes p de 8×8 y prediga cómo serán sus coeficientes transformados y su recuperación para distintos umbrales y escalores.

```
p1=[60,40,20,00,00,20,40,60; ...
60,40,20,00,00,20,40,60; ...
60,40,20,00,00,20,40,60; ...
60,40,20,00,00,20,40,60; ...
60,40,20,00,00,20,40,60; ...
60,40,20,00,00,20,40,60; ...
60,40,20,00,00,20,40,60; ...
60,40,20,00,00,20,40,60];
```

Primera imagen

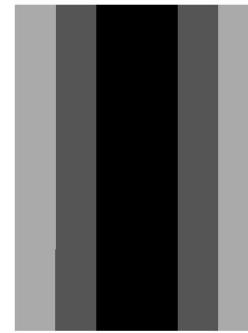


Imagen P1

```
p2=[60,00,00,00,00,00,00,60; ...
00,60,00,00,00,00,60,00; ...
00,00,60,00,00,60,00,00; ...
00,00,00,60,60,00,00,00; ...
00,00,00,60,60,00,00,00; ...
00,00,60,00,00,60,00,00; ...
00,60,00,00,00,00,60,00; ...
60,00,00,00,00,00,60];
```

Segunda imagen

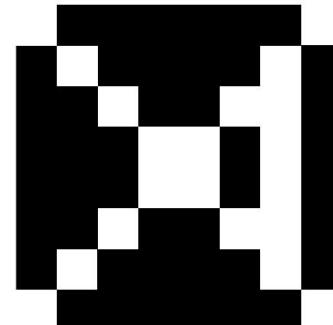


Imagen P2

```
p3=[60,40,20,00,00,20,40,60; ...
40,60,40,20,20,40,60,40; ...
20,40,60,40,40,60,40,20; ...
00,20,40,60,60,40,20,00; ...
00,20,40,60,60,40,20,00; ...
20,40,60,40,40,60,40,20; ...
40,60,40,20,20,40,60,40; ...
60,40,20,00,00,20,40,60];
```

Tercera imagen

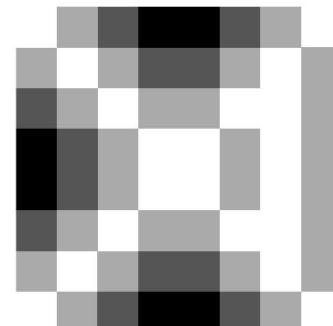


Imagen P3

Figura 4: Imágenes creadas

Pueden verse las imágenes en 4.

La primera de ellas es una imagen de la base, por lo que debería necesitar un único coeficiente para poder ser recuperada y, en principio, el valor del umbral y el escalon no deberían ser muy relevantes. No obstante, en la

práctica, esto no sucede así literalmente. El problema puede ser que yo no haya capturado de forma exacta cuáles son los valores de la matriz base (lo cual es altamente probable, pues lo he hecho a ojo). Pero tampoco se aleja tanto de mi previsión, usa entre 2 y 3 coeficientes únicamente, dependiendo del umbral.

La segunda he intentado que fuese similar a la segunda de las analizadas anteriormente, es decir, que presentase un cambio brusco en la tonalidad y que fuese asimétrica. Por la misma razón que antes, podemos predecir que necesitará un bajo umbral y una gran cantidad de coeficientes no nulos para no perder mucha calidad. Efectivamente, esto es lo que sucede, y necesitamos llegar hasta el umbral 0.01 para obtener un $\text{mse} < 1$.

Para la última, he tratado de suavizar la anterior, eliminando los cambios bruscos de tonalidad, pero dejando la asimetría. De esta forma vamos a averiguar cuál de estos factores es más decisivo en nuestro estudio. No estoy seguro de qué obtendremos, pero el resultado debería ser que se necesitan menos coeficientes no nulos que en el caso anterior para codificar la imagen sin una pérdida grande de calidad. Necesitamos 27 coeficientes para codificar la imagen con un $\text{mse} < 1$, frente a los 23 que necesitábamos para la anterior. La explicación deberá ser que al graduar la imagen, en realidad lo que he conseguido es que los valores estén menos correlacionados, y por tanto a DCT le cueste más codificarla.

1.3. Ejercicio 3.

Escribe una función *testquant* (*fname*, *caliQ*, basada en la función *testdct*(*fname*) que:

- Lea un archivo de imagen *fname* y genere una matriz truecolor RGB (el archivo puede estar en formato BMP o JPEG, y el tipo de color original puede ser indexado o truecolor)
- Convierta la matriz RGB leída del espacio de color RGB al espacio YCbCr
- Amplíe sus dimensiones a múltiplos de 8 píxeles
- Calcule la transformada DCT de la matriz ampliada en bloques de 8x8 píxeles
- Cuantice la transformada DCT aplicando la tabla de cuantización estándar JPEG, multiplicada por un factor de calidad *caliQ* (entero positivo mayor o igual que 1)

Los pasos anteriores son exactamente como en el ejercicio 1. Ahora, tras calcular la transformada DCT, hacemos lo siguiente:

```
% Aplicamos la función quantmat para cuantizarla
Xlab = quantmat(XampDCT, caliQ);
```

Cuadro 17: Cuantización de la transformada discreta

- Descuantice la transformada DCT cuantizada de la imagen, aplicando la misma tabla de cuantización y el mismo factor de calidad

```
%y descuantizamos
Xtransrec = desquantmat(Xlab, caliQ);
```

Cuadro 18: Descuantización de la transformada discreta cuantizada

A partir de aquí, todo vuelve a ser como en el ejercicio 1.

- Calcule la transformada inversa iDCT, también por bloques, a partir de la matriz de coeficientes obtenida en el paso anterior
- Convierte la matriz resultante al espacio RGB
- Reduzca las dimensiones al tamaño original
- Almacene la matriz de imagen en un nuevo archivo BMP
- Muestre el error cuadrático medio MSE producido y la mayor diferencia entre píxeles homólogos

Aplica la función al menos a cinco de los archivos de imagen disponibles en la librería, usando al menos tres factores de calidad para cada una (50, 100 y 500). ¿En qué proporción aumenta en promedio el MSE para un factor de calidad dado? ¿A qué se debe el aumento del MSE?

Imagen	Factor de Calidad	MSE	Factor de aumento	DMaxDifer
Img00	1	0.6553		4
	50	25.3115	36.63	45
	100	34.5450	54.24	62
	500	75.1721	114.71	159
Img04	1	0.6739		4
	50	30.2257	44.85	49
	100	52.2359	77.51	74
	500	164.2558	243.74	155
Img09	1	0.7173		4
	50	71.6119	99.84	52
	100	110.6733	154.29	96
	500	237.6656	331.33	126
Img13	1	0.7460		4
	50	9.6361	12.92	46
	100	16.4664	22.07	78
	500	83.5416	111.99	112
Img18	1	0.6911		3
	50	21.5836	31.23	36
	100	38.3716	55.52	45
	500	155.8288	225.48	111

Cuadro 19: testQuant aplicada a diversas imágenes, con diversos factores de calidad

El aumento se debe a que el factor 1 indica la máxima calidad, que es decreciente al aumentar el factor de calidad. Eso quiere decir que la imagen recuperada será menos fiel conforme aumentemos el factor de calidad, y por tanto aumenta el mse al aumentar el factor de calidad.

Más aún, si vamos a la función *quantmat*, vemos que el factor de calidad se usa como factor multiplicativo en la matriz de luminancia y de crominancia recomendados en el estándar. Esto hace que aumenten los valores de estas matrices y entonces la cuantización varíe. Cuanto más cerca queda el valor de la unidad (cuanto más bajo es el factor de calidad), menor es el escalón y, por tanto, menos hacen variar la matriz transformada DCT, por lo que se pierde menos información en el proceso de cuantización/descuantización.

1.4. Ejercicio 4 . (Opcional)

Escribe una función *wavtest (fname)* que:

- Lea un archivo WAV, cuyo nombre '*<file>.wav*' se da como argumento *fname*
- Lo reproduzca tres veces: a velocidad normal, más lenta y más rápida
- Visualiza gráficamente todos los canales del archivo, cada uno en una ventana distinta
- Lo convierta a formato monoaural, promediando los distintos canales
- Almacene los 5 primeros segundos de señal del sonido mono en otro archivo llamado '*<file>short.wav*'
- Como ejemplo de archivo, se puede usar *hcesto.wav*, proporcionado en la librería

```
function wavtest(fname, vel)
%Funcion que reproduce el archivo de audio fname a velocidad vel
% tambien muestra los distintos canales de audio
%Entradas
% fname: nombre del archivo .wav
% vel: velocidad de reproduccion
% Lee senal
[y,Fs] = audioread(fname);

[lon, canales]=size(y);
n=(0:lon-1)'; t=n/Fs;

%muestra los canales
tiledlayout(canales,1)
for i=1:canales
    nexttile;
    plot(t,y(:,i));
    xlabel('t (s)'); ylabel('y (V)');
    set(gca,'XGrid','on', 'YGrid','on','GridLineStyle',':');
    title('Senal Audio ' + fname + ' x(t) para el canal ' + int2str(i));
end

sound(y,Fs*vel); %reproduce la cancion a velocidad vel

%lo convierte a monoaural
ymono = zeros(lon, 1);
for i=1:lon
    ymono(i) = sum(y(i,:))/canales;
end

%tomamos 5 segundos y los guardamos en un nuevo archivo
numdatos=Fs*5;
y5=ymono(1:numdatos,:);
[~,nomb,~] = fileparts(fname);
audiowrite(nomb + "short.wav", y5,Fs);
```

Cuadro 20: Función *wavtest(fname, vel)*

Podemos ver que es una función sencilla, y que está basada en las últimas diapositivas de la práctica 3.

He decidido que la velocidad pueda ser pasada como parámetro porque así pueden hacerse distintas pruebas, además de que Matlab parece pasar los archivos de audio a un reproductor, sin usar una cola, por lo que si quería

reproducir los tres audios seguidos había que diseñar algún tipo de mecanismo que esperase a que terminase cada audio para seguir. No merece la pena y no creo que ese sea el objetivo de la práctica.

Para comprobar que la transformación a mono funciona, basta pasar un archivo de audio que tenga más de un canal, y luego pasar el archivo short generado, y ver que ahora solo tiene uno. Vamos a usar el proporcionado en la librería, en el que vemos, primero, los dos siguientes canales:

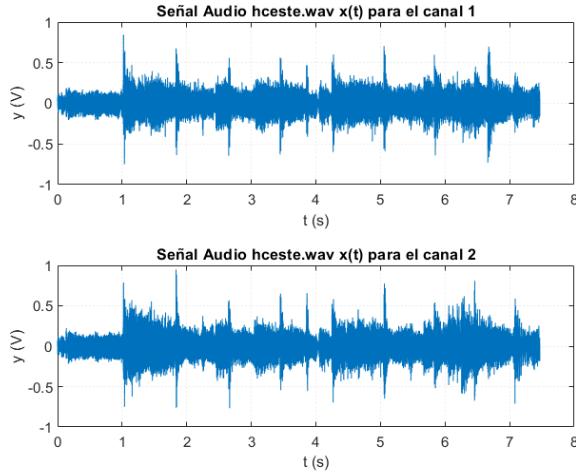


Figura 5: Los dos canales de *hcestewav*

Al aplicarle la función `y` obtener el archivo mono, y aplicarle a este, de nuevo, `wavtest`, obtenemos la siguiente gráfica:

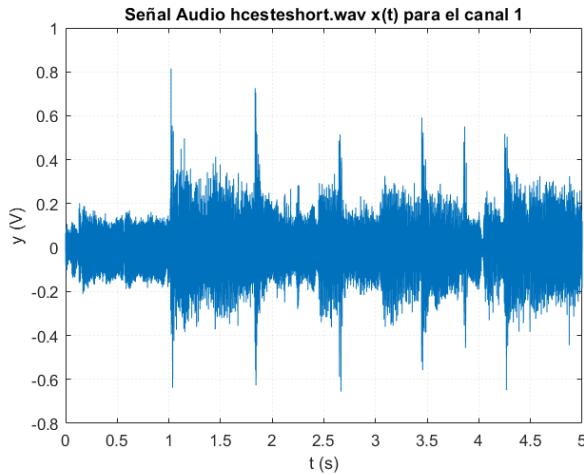


Figura 6: El único canal de *hcesteshort.wav*

Y vemos como, efectivamente, funciona.

2. Práctica 4. Compresión de imágenes.

2.1. Ejercicio 1.

Basándose en la función *jcomdes*, escribir una pareja de funciones para comprimir y descomprimir archivos de imagen bitmap, usando las tablas Huffman por defecto:

- Función de compresión: $RC = jcom_dflt (fname, caliQ)$
 - Se basa en la primera mitad de *jcomdes*
 - Aplica tablas Huffman por defecto
 - Genera y almacena un archivo comprimido **.hud*
 - Calcula y devuelve la relación de compresión *RC*

- Función de descompresión: $[MSE, RC] = jdes_dflt (fname)$
 - Se basa en la segunda mitad de la función *jcomdes*
 - Aplica tablas Huffman por defecto
 - Lee un archivo comprimido **.hud*
 - Genera y almacena un archivo descomprimido **_des_def.bmp*
 - Calcula y devuelve:
 - La relación de compresión *RC*
 - El error cuadrático medio *MSE*
 - Visualiza la imagen bitmap original y la descomprimida

Apícalas al menos a cinco de los archivos de imagen disponibles en la librería, y analiza y discute los resultados obtenidos para al menos tres factores de calidad (50, 100, 500).

Lo único que hay que hacer es tomar la función *jcomdes* y dividirla en dos partes, la de compresión y la de descompresión.

Por supuesto, entre ambas partes hay que añadir el proceso de creación del fichero comprimido y su lectura, lo cual no se hace en la función *jcomdes* original.

Para ello, hay que hacer algo similar a lo que hicimos en la práctica 2, y guardar todos los valores que necesitamos recuperar después para la descompresión, es decir, los tamaños originales y ampliados, el factor de calidad, y las matrices de codificación *Y*, *CB*, *CR*.

Además, calculamos el RC, mediante la fórmula

$$RC = 100 \cdot \frac{T0 - TC}{T0}$$

```
%Generamos el nombre del archivo comprimido
[pathstr,nomb,ext] = fileparts(fname);
nombrecomp = strcat(nomb, '.hud');

% Creamos el archivo
fid = fopen(nombrecomp,'w');
% Transformamos las tablas obtenidas a B
% (con bits2bytes) y las codificamos:
%CodedY
[SBYTES_Y, ULTL_Y]=bits2bytes(CodedY);
U_LENS_Y=uint32(length(SBYTES_Y));
U_ULTL_Y=uint32(ULTL_Y);
U_SBYTES_Y=uint32(SBYTES_Y);
%CodedCB
[SBYTES_CB, ULTL_CB]=bits2bytes(CodedCb);
U_LENS_CB=uint32(length(SBYTES_CB));
U_ULTL_CB=uint32(ULTL_CB);
U_SBYTES_CB=uint32(SBYTES_CB);
%CodedCR
[SBYTES_CR, ULTL_CR]=bits2bytes(CodedCr);
U_LENS_CR=uint32(length(SBYTES_CR));
U_ULTL_CR=uint32(ULTL_CR);
U_SBYTES_CR=uint32(SBYTES_CR);
%Codificamos m, n, mamp, namp y CaliQ
U_M=uint32(m);
U_MAMP=uint32(mamp);
U_N=uint32(n);
U_NAMP=uint32(namp);
U_CALQ=uint32(caliQ);

% Escribimos una a una las variables en el archivo
fwrite(fid,U_N,'uint32');
fwrite(fid,U_NAMP,'uint32');
fwrite(fid,U_M,'uint32');
fwrite(fid,U_MAMP,'uint32');
fwrite(fid,U_CALQ,'uint32');
%CodedY
fwrite(fid,U_LENS_Y,'uint32');
fwrite(fid,U_ULTL_Y,'uint32');
fwrite(fid,U_SBYTES_Y,'uint32');
%CodedCb
fwrite(fid,U_LENS_CB,'uint32');
fwrite(fid,U_ULTL_CB,'uint32');
fwrite(fid,U_SBYTES_CB,'uint32');
%CodedCr
fwrite(fid,U_LENS_CR,'uint32');
fwrite(fid,U_ULTL_CR,'uint32');
fwrite(fid,U_SBYTES_CR,'uint32');
%c'est fini
fclose(fid);

% Calculamos RC
TAM_CAB=length(U_CALQ)+ length(U_MAMP)+length(U_NAMP)+length(U_M)+length(U_N);
TAM_DAT=length(U_SBYTES_Y)+ length(U_SBYTES_CB)+length(U_SBYTES_CR);
TAM_TOT = TAM_CAB + TAM_DAT;
RC = 100* ((TO-TAM_TOT)/TO);
```

Cuadro 21: Código de *jcom_dflt* para guardar los datos en el nuevo archivo */name/.hud*

En *jdes_dflt*, entonces, lo que hacemos es recuperar estos valores, y lo demás se deja tal y como estaba en *jcomdes*. También creamos un nuevo archivo en el que guardaremos la imagen recuperada tras la descompresión. También calculamos el MSE y el RC.

```
% Abrimos el fichero comprimido,
% Generamos el nombre para el archivo descomprimido <nombre>_des_def.bmp y
% obtenemos los datos comprimidos
[pathstr,nomb,ext] = fileparts(fname);
nombrecomp=strcat(nomb,'_des_def','.bmp');
fid = fopen(fname,'r');
archivoOrig = strcat(nomb, '.bmp');
[XOR, ~, ~, ~, ~, ~, ~, TO] = imlee(archivoOrig);

% Leemos los parametros de la imagen original
n= double(fread(fid, 1, 'uint32'));
namp= double(fread(fid, 1, 'uint32'));
m= double(fread(fid, 1, 'uint32'));
mamp= double(fread(fid, 1, 'uint32'));
caliQ= double(fread(fid, 1, 'uint32'));

% CodedY
LENS_Y = double(fread(fid, 1, 'uint32'));
ULTL_Y = double(fread(fid, 1, 'uint32'));
U_SBYTES_Y = fread(fid, LENS_Y, 'uint32');
SBYTES_Y = double(U_SBYTES_Y);
% Obtenemos CodedY original
CodedY=bytes2bits(SBYTES_Y, ULTL_Y);

% CodedCb
LENS_CB = double(fread(fid, 1, 'uint32'));
ULTL_CB = double(fread(fid, 1, 'uint32'));
U_SBYTES_CB = fread(fid, LENS_CB, 'uint32');
SBYTES_CB = double(U_SBYTES_CB);
% Obtenemos CodedCb original
CodedCb=bytes2bits(SBYTES_CB, ULTL_CB);

% CodedCr
LENS_CR = double(fread(fid, 1, 'uint32'));
ULTL_CR = double(fread(fid, 1, 'uint32'));
U_SBYTES_CR = fread(fid, LENS_CR, 'uint32');
SBYTES_CR = double(U_SBYTES_CR);
% Obtenemos CodedCR original
CodedCr=bytes2bits(SBYTES_CR, ULTL_CR);

%C'est fini
fclose(fid);

...
%Calculamos el MSE
MSE=(sum(sum((double(Xrec)-double(XOR)).^2))) / (m*n*3);

%Calculamos RC
RC = 100*(TO-TC)/TO;
```

Cuadro 22: Código de *jdes_dflt* para obtener los datos comprimidos anteriormente

Y ahora pasamos a su aplicación a 5 imágenes, que van a ser las 0,3,6,11 y 15.



Original



caliQ = 50
RC = 95.7334
MSE = 25.3115



caliQ = 100
RC = 97.3896
MSE = 34.5450



caliQ = 500
RC = 98.9086
MSE = 75.1721

Figura 7: Aplicación a *Img00.bmp*



Original



caliQ = 50
RC = 90.1289
MSE = 57.3105



caliQ = 100
RC = 93.7016
MSE = 98.4059



caliQ = 500
RC = 97.8265
MSE = 282.2025

Figura 8: Aplicación a *Img03.bmp*



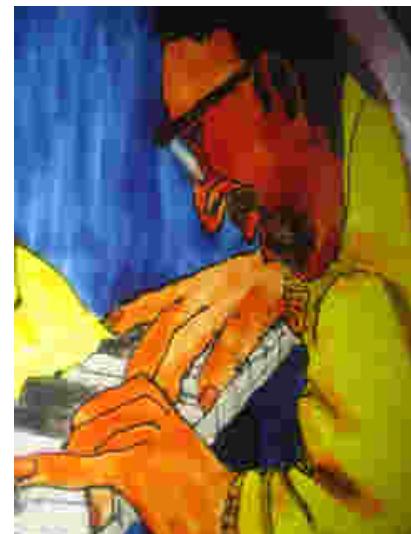
Original



caliQ = 50
RC = 91.3497
MSE = 54.5403



caliQ = 100
RC = 94.4479
MSE = 87.7848



caliQ = 500
RC = 98.0178
MSE = 239.4674

Figura 9: Aplicación a *Img06.bmp*



Original



caliQ = 50
RC = 80.1348
MSE = 36.7402



caliQ = 100
RC = 87.2252
MSE = 53.9567



caliQ = 500
RC = 95.2403
MSE = 160.2773

Figura 10: Aplicación a *Img11.bmp*



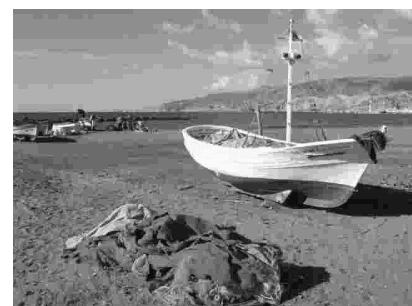
Original



caliQ = 50
RC = 84.4647
MSE = 19.9968



caliQ = 100
RC = 89.2396
MSE = 33.0000



caliQ = 500
RC = 95.4693
MSE = 108.0573

Figura 11: Aplicación a *Img15.bmp*

Podemos observar que usar caliQ = 100 supone una mejora significativa en la relación de compresión respecto a caliQ = 50, y no empeora demasiado el MSE, visualmente no se perciben diferencias entre las imágenes recuperadas y las originales.

Por otro lado, al aumentar hasta 500 el factor de calidad, Si bien se mejora la relación de compresión, la calidad de la imagen se ve muy resentida, aumentando considerablemente el MSE.

Por tanto, parece que usar un factor de calidad de 100 es una buena opción de compromiso entre compresión y calidad.

2.2. Ejercicio 2.

Escribe una pareja de funciones, similar a la anterior, para comprimir y descomprimir archivos de imagen bitmap, pero que, en lugar de utilizar las tablas de especificación Huffman por defecto, construya unas tablas a medida basadas en las frecuencias de los valores presentes en la imagen original:

- Función de compresión: $RC=jcom_custom (fname, caliQ)$
 - Se basa en la primera mitad de *jcomdes*
 - Aplica tablas Huffman a medida
 - Genera y almacena un archivo comprimido **.huc*

- Calcula y devuelve la relación de compresión RC
- Función de descompresión: $[MSE, RC] = jdes_custom (fname)$
 - Se basa en la segunda mitad de *jcomdes*
 - Aplica tablas Huffman a medida, almacenadas en el archivo
 - Lee un archivo comprimido **.huc*
 - Genera y almacena un archivo descomprimido **_des_cus.bmp*
 - Calcula y devuelve RC y MSE
 - Visualiza la imagen bitmap original y la descomprimida

Para hacer esto, solo hay que modificar un poco el código de las dos funciones del ejercicio 1. Basta añadir la obtención de las tablas Huffman personalizadas mediante la función *encodeScans_custom*, lo demás es igual, por tanto solo voy a mostrar esta función, que codifica en binario los scans recibidos como parámetro usando tablas Huffman adaptadas, como estudiamos en la práctica 2. Para la segunda función, debemos añadir el proceso de decodificación mediante tablas Huffman adaptativas, *decodeScans_custom*, que será la operación inversa de la función anterior, y que está basada en la práctica 2.

Como puede verse en el código, es muy similar a *encodeScans_dflt*, con la diferencia de que debemos seguir el proceso de codificación Huffman aprendido en clase paso a paso. El proceso es el siguiente, para cada uno de los scans:

1. Obtener las frecuencias de los valores, FREQ
2. Construir las tablas de especificación:
 - a) BITS: lista que contiene el número de palabras código que se generan de cada tamaño
 - b) HUFFVAL: lista con los símbolos de la fuente, ordenados por longitudes crecientes de la palabra código que los representa
3. Construimos las tablas del código Huffman:
 - a) HUFFCODE: contiene las palabras código, ordenadas por valor creciente de los códigos
 - b) HUFFSIZE: longitudes de las palabras código, ordenadas por longitudes crecientes
4. Construimos las tablas de codificación:
 - a) EHUFCO: contiene las palabras código, expresadas en decimal, e indexadas por los mensajes de la fuente, en orden creciente.
 - b) EHUFSI: longitudes de las palabras código, indexadas por los mensajes de la fuente, en orden creciente.
5. Codificamos la información original usando estas tablas

```
% Separa las matrices bidimensionales para procesar separadamente
YScan=XScan(:,:,1); CbScan=XScan(:,:,2); CrScan=XScan(:,:,3);

% Recolectar valores a codificar
[Y_DC_CP, Y_AC_ZCP]=CollectScan(YScan); [Cb_DC_CP, Cb_AC_ZCP]=CollectScan(CbScan); [Cr_DC_CP,
Cr_AC_ZCP]=CollectScan(CrScan);

% Construir tablas Huffman para Luminancia y Crominancia
FREQ_Y_DC=Freq256((Y_DC_CP(:,1)));
[BITS_Y_DC, HUFFVAL_Y_DC] = HSpecTables(FREQ_Y_DC);
% tablas del Código Huffman
[HUFFSIZE_Y_DC, HUFFCODE_Y_DC] = HCodeTables(BITS_Y_DC, HUFFVAL_Y_DC);
% Tablas de Codificación Huffman
[EHUFCO_Y_DC, EHUFSI_Y_DC] = HCodingTables(HUFFSIZE_Y_DC, HUFFCODE_Y_DC, HUFFVAL_Y_DC);
ehuf_Y_DC=[EHUFCO_Y_DC EHUFSI_Y_DC];
% Hacemos lo propio con los valores en zigzag
FREQ_Y_AC=Freq256((Y_AC_ZCP(:,1)));
[BITS_Y_AC, HUFFVAL_Y_AC] = HSpecTables(FREQ_Y_AC);
[HUFFSIZE_Y_AC, HUFFCODE_Y_AC] = HCodeTables(BITS_Y_AC, HUFFVAL_Y_AC);
[EHUFCO_Y_AC, EHUFSI_Y_AC] = HCodingTables(HUFFSIZE_Y_AC, HUFFCODE_Y_AC, HUFFVAL_Y_AC);
ehuf_Y_AC=[EHUFCO_Y_AC EHUFSI_Y_AC];
% Tablas de crominancia
C_DC_CP = [Cb_DC_CP; Cr_DC_CP];
% Y hacemos como antes
FREQ_C_DC=Freq256((C_DC_CP(:,1)));
[BITS_C_DC, HUFFVAL_C_DC] = HSpecTables(FREQ_C_DC);
[HUFFSIZE_C_DC, HUFFCODE_C_DC] = HCodeTables(BITS_C_DC, HUFFVAL_C_DC);
[EHUFCO_C_DC, EHUFSI_C_DC] = HCodingTables(HUFFSIZE_C_DC, HUFFCODE_C_DC, HUFFVAL_C_DC);
ehuf_C_DC=[EHUFCO_C_DC EHUFSI_C_DC];
C_AC_ZCP = [Cb_AC_ZCP; Cr_AC_ZCP];
FREQ_C_AC=Freq256((C_AC_ZCP(:,1)));
[BITS_C_AC, HUFFVAL_C_AC] = HSpecTables(FREQ_C_AC);
[HUFFSIZE_C_AC, HUFFCODE_C_AC] = HCodeTables(BITS_C_AC, HUFFVAL_C_AC);
[EHUFCO_C_AC, EHUFSI_C_AC] = HCodingTables(HUFFSIZE_C_AC, HUFFCODE_C_AC, HUFFVAL_C_AC);
ehuf_C_AC=[EHUFCO_C_AC EHUFSI_C_AC];
% Codifica en binario cada Scan
CodedY=EncodeSingleScan(YScan, Y_DC_CP, Y_AC_ZCP, ehuf_Y_DC, ehuf_Y_AC);
CodedCb=EncodeSingleScan(CbScan, Cb_DC_CP, Cb_AC_ZCP, ehuf_C_DC, ehuf_C_AC);
CodedCr=EncodeSingleScan(CrScan, Cr_DC_CP, Cr_AC_ZCP, ehuf_C_DC, ehuf_C_AC);
```

Cuadro 23: Código de *encodeScans_custom* para codificar los scans pasados como parámetro

Vamos a usar las mismas imágenes que antes para estudiar el comportamiento de estas funciones.



Original



caliQ = 50
RC = 96.0507
MSE = 25.3115



caliQ = 100
RC = 97.7694
MSE = 34.5450



caliQ = 500
RC = 99.3670
MSE = 75.1721

Figura 12: Aplicación a *Img00.bmp*



Original



caliQ = 50
RC = 90.1289
MSE = 57.3105



caliQ = 100
RC = 93.7905
MSE = 98.4059



caliQ = 500
RC = 98.1751
MSE = 282.2025

Figura 13: Aplicación a *Img03.bmp*



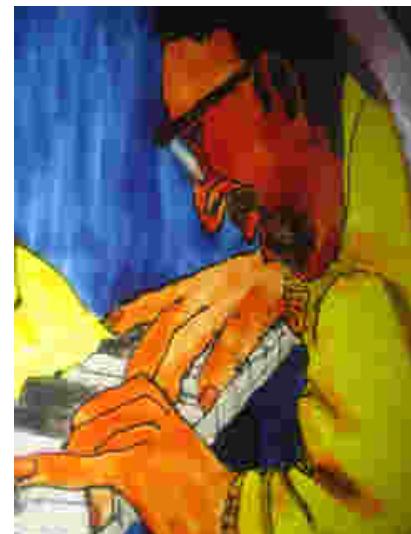
Original



caliQ = 50
RC = 91.2052
MSE = 54.5403



caliQ = 100
RC = 94.3451
MSE = 87.7848



caliQ = 500
RC = 98.2422
MSE = 239.4674

Figura 14: Aplicación a *Img06.bmp*



Original



caliQ = 50
RC = 80.2422
MSE = 36.7402



caliQ = 100
RC = 87.7585
MSE = 53.9567



caliQ = 500
RC = 96.4746
MSE = 160.2773

Figura 15: Aplicación a *Img11.bmp*



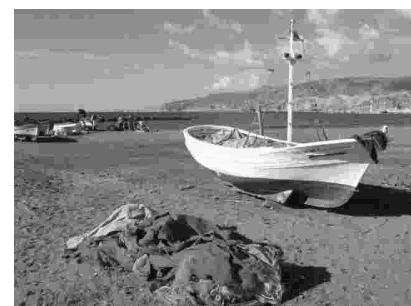
Original



caliQ = 50
RC = 85.1835
MSE = 19.9968



caliQ = 100
RC = 90.1365
MSE = 33.0000



caliQ = 500
RC = 96.7776
MSE = 108.0573

Figura 16: Aplicación a *Img15.bmp*

Podemos observar como mantiene los MSE muy similares o incluso idénticos a los obtenidos mediante tablas discretas, pero mejora en la mayoría de los casos la relación de compresión. Por tanto, en términos de compresión parece una mejor decisión usar tablas customizadas. No obstante, puede ser que el aumento de cómputo que implica el cálculo de las tablas customizadas haga que no merezca la pena efectuar este cambio. Esto es una cuestión compleja que habría de ser detenidamente estudiada, pero, como digo, en base a la compresión, elegiríamos las customizadas.

2.3. Ejercicio 3.

Aplica ambos compresores y descompresores sobre algunas imágenes artificiales construidas por ti y en formato BMP. En el ejercicio 2 del guión 3 estudiaste el efecto de la cuantización sobre un bloque 8x8 y debiste sacar tus conclusiones sobre cuándo obtendrás coeficientes nulos y cuáles son los que degradan más la imagen original. Ahora vamos a añadir la información que tenemos de cómo se comprime en JPG con tablas Huffman por defecto. Una de ellas es que a menor categoría menor es la longitud del código Huffman.

- Estudia qué te permitiría obtener la codificación más óptima en la codificación de los DC y de los AC

Los DC son el primer valor de las matrices de coeficientes, y los AC son el resto de valores. El primero, como sabemos, es el que es constante.

Su codificación se hace más ligera conforme aumentamos el valor de caliQ. Esto se debe a que al aumentar

este valor, usaremos menos coeficientes de la base y, por tanto, podremos usar codificaciones más cortas para ellos.

Por tanto, la codificación óptima debe obtenerse en conjugación con los valores de la relación de compresión y el MSE. El óptimo será el mayor valor posible de caliQ que no implique un empeoramiento inasumible del MSE.

Obviamente, esto dependerá de cada imagen, y es esperable que imágenes con alta correlación puedan tomar un valor mayor para caliQ que aquellas con baja correlación.

- **Aplica tus conclusiones a las que ya tienes sobre la cuantización de bloques 8x8 para diseñar unas imágenes que te permitan responder a las siguientes preguntas**

- **¿Cómo deberían ser tus imágenes para conseguir mayor compresión?**

La mejor compresión se obtendría con imágenes muy correlacionadas, obviamente una imagen lisa de un único color sería muy comprimida. Sin irnos tan al extremo, podríamos pensar en una imagen que use únicamente tonalidades de un mismo color, y también sería bastante comprimida.

- **¿Y cómo deberían de ser para que resulten muy difíciles de comprimir?**

Debemos ahora irnos al otro extremo, es decir, a una imagen con abundancia de colores y tonalidades, en la que no hubiese apenas correlación en las transiciones de color.

- **¿Con cuáles predices que perderás más calidad?**

Es esperable un resultado similar al de la compresión, se perderá más calidad cuanto menos correlación presente la imagen.

- **¿Con cuáles obtendrás un problema de contorneado?**

Los problemas de contorneado ocurrirán ante cambios bruscos en la imagen, de nuevo, con las imágenes con poca correlación es esperable obtener este tipo de problemas.

- **Construye todas tus imágenes del mismo tamaño y aplica siempre la misma calidad**

Observa que todos los ficheros BMP del mismo tamaño en píxeles siempre ocupan el mismo tamaño en bytes.

Tenemos las siguientes imágenes y resultados:

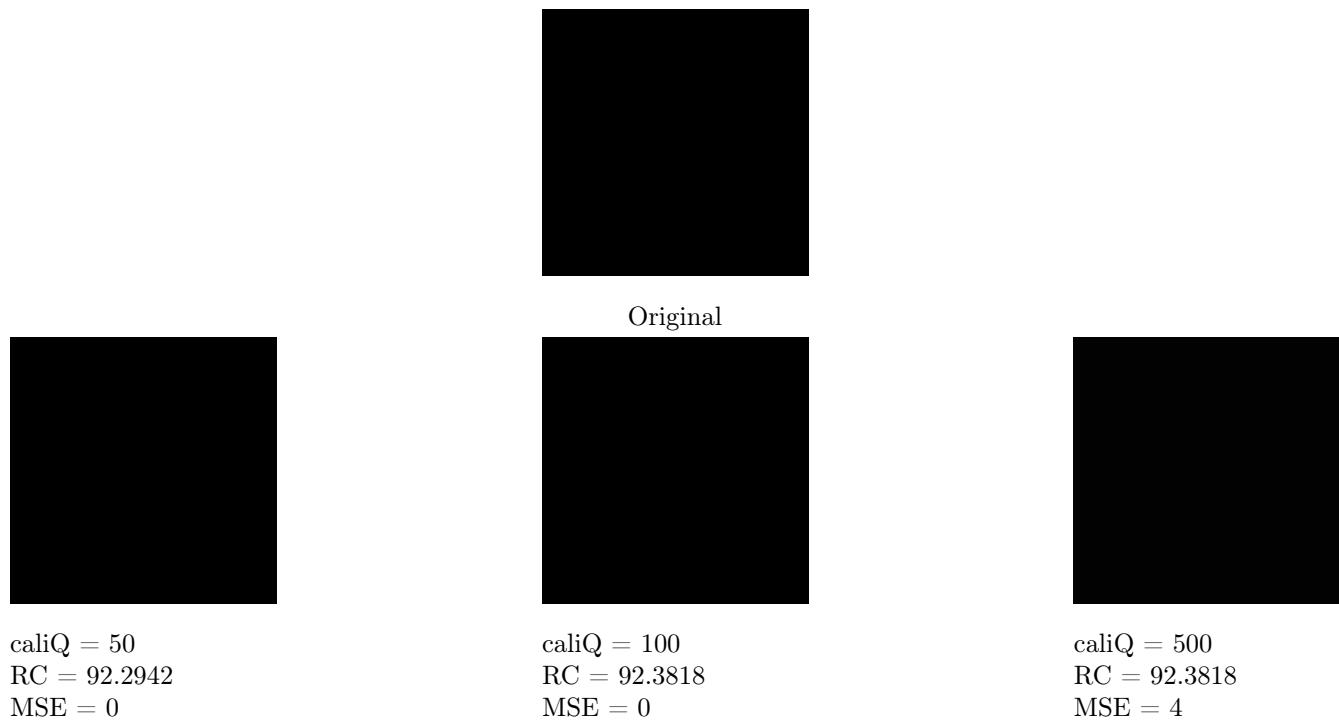


Figura 17: Aplicación a una imagen lisa

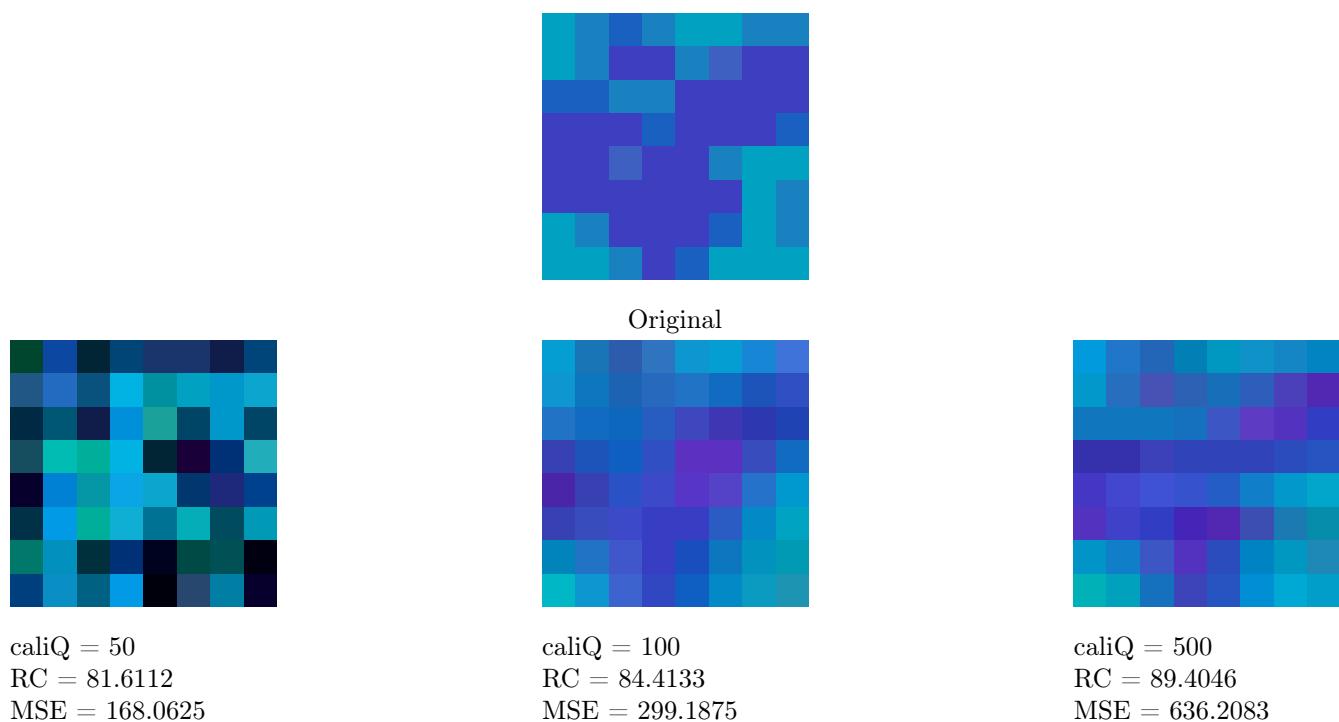


Figura 18: Aplicación a una imagen con varias tonalidades de un color

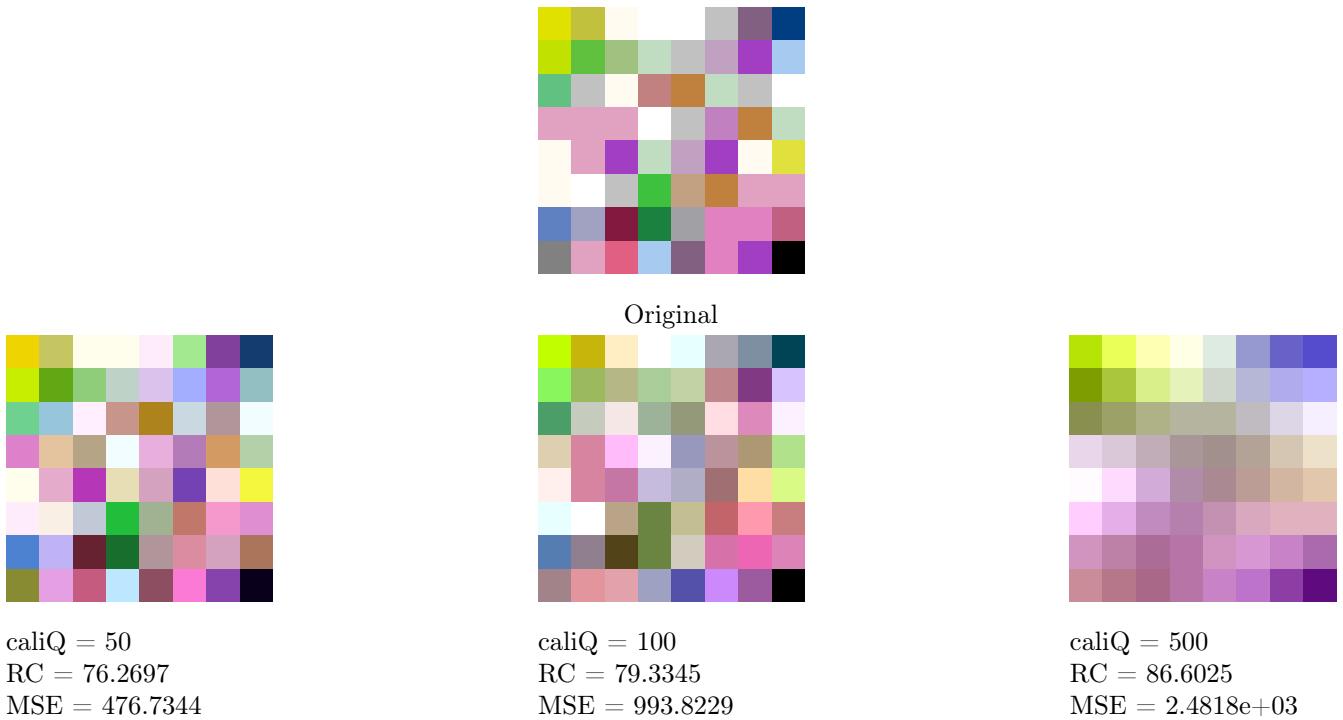


Figura 19: Aplicación a una imagen con varios colores sin relación

Podemos observar que nuestras predicciones eran correctas. La imagen lisa presenta la mayor relación de compresión y el menor MSE (prácticamente nulo), aunque observamos también que aumentar caliQ no tiene sentido a partir de 100, pues no se mejora la compresión.

La imagen con varias tonalidades de azul se encuentra en un punto intermedio. Observamos que con $\text{caliQ} = 50$ ocurre un fenómeno extraño, visualmente parece muy distinta de la original, incluso peor que con calidades superiores, pero el MSE es menor de lo que será posteriormente. Vemos de esta forma que no siempre desearemos un MSE pase lo que pase. Esto es una imagen muy pequeña, pero si tuviésemos una foto del cielo o del mar, probablemente prefiramos que cada bloque se parezca más a los de peor calidad que al de 50.

La última imagen pretendía que fuese muy mala, y parece que así es, pues vemos tasas bastante menores de compresión, junto con un MSE enorme desde el principio, pero que se dispara de una forma tremenda al aumentar caliQ.

3. Conclusión

Tras realizar todo este trabajo y estudiar el comportamiento de los programas desarrollados para diferentes entradas, observamos la potencia que tienen los métodos de compresión, pero también que las decisiones respecto a su funcionamiento que se deben tomar no son sencillas.

Debemos tener factores contrapuestos en cuenta, como son:

- **Relación de compresión:** a más relación de compresión, mejor
- **MSE:** a menor MSE, mejor
- **Capacidad de cómputo requerida:** a menor capacidad requerida, mejor

Estos factores son complejos de considerar en conjunto, pues una mayor relación de compresión implicará un peor MSE, y al revés. Y, a su vez, si queremos, por ejemplo, utilizar tablas customizadas para mejorar la RC sin comprometer el MSE, necesitaremos más tiempo de cómputo para llevar a cabo el proceso.

Así, un estudio mucho más exhaustivo debió realizarse para proponer un estándar funcional y eficiente, de forma que fuese lo mejor posible en el mayor rango de posibles situaciones.