Grado en Ingeniería de Tecnologías de Telecomunicación

SISTEMAS DE CODIFICACIÓN Y ALMACENAMIENTO

Codificación de imágenes mediante DCT

PRÁCTICA 3

Universidad de GRANADA

1. Introducción

Las técnicas de codificación mediante transformada se basan en la aplicación de transformaciones que llevan los datos a un dominio distinto (generalmente el dominio de la frecuencia) en donde resulta más sencillo determinar qué componentes de los datos son menos importantes (contienen menos energía o son irrelevantes para nuestros sentidos) y por tanto nos ayudan a seleccionar de qué componentes de los datos prescindir para representarlos con una menor cantidad de bits (comprimirlos). Este enfoque se ha usado satisfactoriamente tanto para la codificación de señales de audio como imágenes y vídeos.

Entre las transformadas más utilizadas se encuentran la transformada discreta de Fourier, la transformada de coseno discreta (DCT), la transformada de wavelet discreta (DWT) y la transformada de Karhunen-Loève (KLT), cada una con sus propias aplicaciones y ventajas particulares.

En esta práctica vamos a implementar y analizar un codificador/descodificador basado en la DCT, que resulta especialmente útil para codificar imágenes. Para evaluar su rendimiento, compararemos los resultados con los obtenidos mediante la modulación por impulsos codificados (PCM), uno de los esquemas más sencillos de codificación de señales. Por tanto, de forma previa a la implementación del sistema de codificación basado en la DCT implementaremos un sistema PCM. En ambos casos haremos uso de los cuantificadores incluidos en `scalib` y que ya hemos usado en prácticas anteriores.

2. Modulación por impulsos codificados

Desarrollada en 1937 por el ingeniero británico Alec Reeves, la modulación por impulsos codificados (PCM) desempeña un papel importante en la conversión de señales analógicas a digitales. Estrictamente hablando, PCM comprende el proceso completo de conversión de una señal analógica en digital, esto es, una etapa de muestreo, otra de cuantificación y otra etapa de codificación. No obstante, también puede ser usado para recodificar señales digitales con una menor tasa de bits. En ese caso, la etapa de muestreo no suele llevarse a cabo (salvo que se quiera modificar también la frecuencia de muestreo) y el proceso se centra en la cuantificación y codificación.

El proceso de cuantificación de una señal se puede describir como la combinación de dos procesos de mapeo: un mapeo del codificador y un mapeo del descodificador. El primero divide el rango de

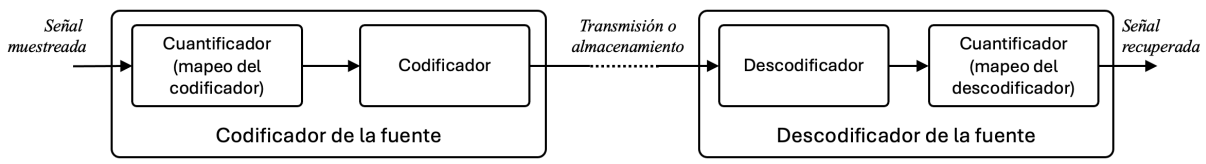

Figura 1: Procesos llevados a cabo por un sistema de codificación de la fuente para la transmisión o almacenamiento digital de una señal.

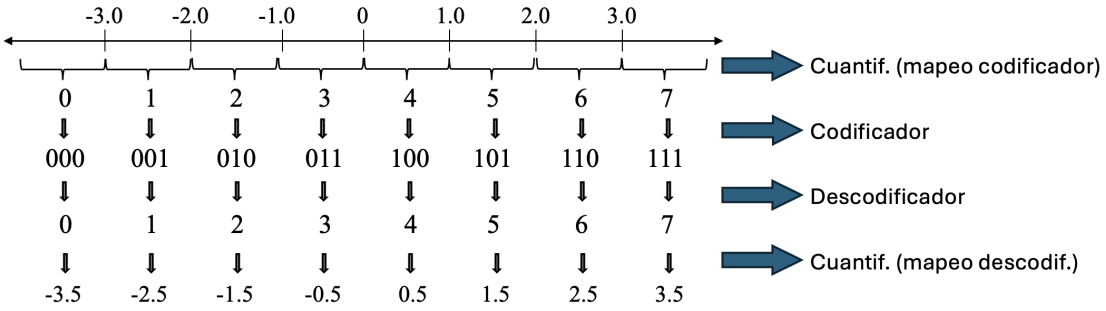

Figura 2: Procesos de cuantificación y codificación/descodificación de una señal.

valores en el que varía la señal en un número finito de intervalos y le asigna un número (o identificador) a cada uno. El mapeo del descodificador, por su parte, asigna un valor de reconstrucción a cada intervalo en los que se había dividido el rango de valores de la señal original. Entre un mapeo y otro se realiza la codificación, que transforma los identificadores asignados por el mapeo del codificador del cuantificador en secuencias de bits, la transmisión por el canal (o almacenamiento) de la secuencia binaria y la descodificación, que deshace el proceso llevado a cabo por la codificación (véanse Figuras 1 y 2). Cuantificación y codificación están muy relacionados entre sí y a menudo se llevan a cabo de forma conjunta, lo que permite combinar el descodificador y la parte del cuantificador que lleva a cabo el mapeo de descodificador en un solo proceso. No obstante, son procesos conceptualmente distintos y nosotros los vamos a implementar por separado.

Las clases `UniformSQ`, `OptimalSQ` y `OptimalVQ` implementadas en `scalib` cuentan con dos funciones, `encode` y `decode`, que llevan a cabo las dos partes del proceso de cuantificación (el mapeo del codificador y el mapeo del descodificador) por separado. Es posible, por tanto, cuantificar una señal llamando a estas funciones de manera consecutiva (en lugar de llamar a `quantize`), como se muestra en el siguiente ejemplo:

```
import scipy.io.wavfile as wf
from scalib import UniformSQ, signalRange

fs, x = wf.read('altura.wav')                    # Lee la señal de voz
qtz = UniformSQ(3, signalRange(x))               # Crea un cuantificador de 3 bits
ids = qtz.encode(x)                              # Mapeo del codificador
xq = qtz.decode(ids)                             # Mapeo del descodificador
wf.write('altura3b.wav', fs, xq.astype(x.dtype)) # Guarda la señal cuantificada
```

Por otro lado, para la conversión de la salida del cuantificador en una secuencia de bits podemos usar la clase `FixedLengthCoder`, implementada también en `scalib`, y que permite crear codificadores de palabras de longitud fija con un determinado número de bits. A continuación se muestra un ejemplo de su uso:

```
from scalib import FixedLengthCoder

coder = FixedLengthCoder(8)  # Crea un codificador de 8 bits
code = coder.encode(ids)     # Codifica una secuencia de identificadores (enteros positivos)
...                          # Procesa el código (transmisión, almacenamiento, etc.)
data = coder.decode(code)    # Descodifica un secuencia de bits
```

Ejercicio 1

Implemente un codificador y un descodificador PCM para imágenes basado en un cuantificador uniforme y códigos de longitud fija. Para ello, cree dos funciones con la siguiente interfaz:

```
def encoderPCM(data, dataRange, b):
    ...
    ids = ...  # Mapeo del codificador
    code = ... # Codificador
    return code
```

```
def decoderPCM(code, dataRange, b):
    ...
    ids = ...  # Descodificador
    data = ... # Mapeo del descodificador
    return data
```

donde:

- `data` $\rightarrow$ Array bidimensional que contiene la imagen que vamos a codificar.
- `code` $\rightarrow$ Array unidimensional con la imagen codificada.
- `dataRage` $\rightarrow$ Tupla de 2 elementos con el mínimo y el máximo valor posible de las muestras en `data`.
- `b` $\rightarrow$ Número de bits usado para codificar cada muestra en `data`.

Use estas funciones para codificar y descodificar la imagen de Lena con diferentes tasas bits. En cada caso, calcule la SNR de la imagen descodificada y represéntela junto a la imagen original para evaluar la calidad percibida. Tenga en cuenta que `decoderPCM` devuelve un array unidimiensional al que deberá dar forma de imagen (array bidimensional) posteriormente.

3. Transformada de coseno discreto

Desarrollada inicialmente como una técnica matemática para analizar señales en el dominio de la frecuencia, la transformada de coseno discreta (DCT) ha demostrado ser una herramienta fundamental en el ámbito de la codificación de imágenes. Es ampliamente utilizada para transformar bloques de píxeles de una imagen en el dominio del espacio a un dominio de frecuencia, permitiendo una representación más concentrada de la energía de la imagen.

Dado un bloque de $N \times N$ píxeles, $\mathbf{X}$, podemos obtener la matriz de coeficientes DCT, $\mathbf{\Theta}$, con la representación del bloque en el dominio de la frecuencia, usando la siguiente expresión (trasformada DCT directa):

$$\Theta = \mathbf{C}\mathbf{X}\mathbf{C}^T \tag{1}$$

donde $\mathbf{C}$ es la matriz de transformación, de tamaño $N \times N$, cuyo elemento en la fila n y columna k se calcula como:

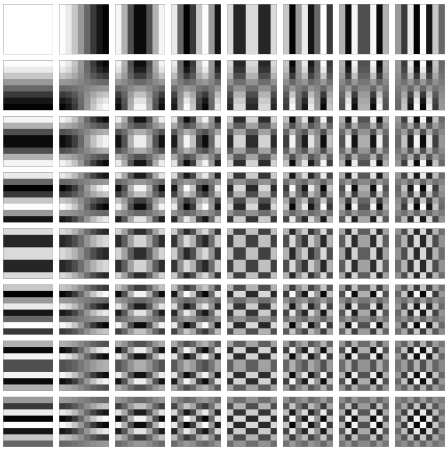

Figura 3: Representación del producto exterior de las filas de la matriz de transformación, $\mathbf{C}$, para $N = 8$.

$$C[n,k] = \sqrt{\frac{a[n]}{N}} \cos \frac{(2k+1)n\pi}{2N} \qquad n,k = 0,1,...,N-1, \qquad a[n] = \begin{cases} 1 & n = 0 \\ 2 & n > 0 \end{cases} \tag{2}$$

De formada similar, dada la representación en el dominio de la frecuencia de un bloque de píxeles de tamaño $N \times N$, $\mathbf{\Theta}$, podemos recuperar el bloque original, $\mathbf{X}$, mediante la siguiente expresión (trasformada DCT inversa):

$$\mathbf{X} = \mathbf{C}^T \mathbf{\Theta} \mathbf{C} \tag{3}$$

La matriz de coeficientes, $\mathbf{\Theta}$, contiene la información del bloque en el dominio de la frecuencia, siendo el coeficiente de la posición $0,0$ el que representa a la componente continua (frecuencia 0) y a medida que crecen las filas o las columnas va aumentando la frecuencia hasta llegar a la posición N,N, que representa la mayor frecuencia posible. La Figura 3 muestra el patrón de frecuencias espaciales horizontales y verticales para una DCT bidimensional aplicada a bloques de 8×8 píxeles.

Ejercicio 2

Implemente una función que calcule la matriz de transformación de la DCT a partir del parámetro N usando la ecuación 2:

```
def dct(N):
    C = ...
    return C
```

A continuación cree una función que aplique la transformada DCT directa (ecuación 1) a un bloque de datos bidimensional[1] y otra función similar que deshaga la transformación aplicando la transformada DCT inversa (ecuación 3):

```
def dDCT(data, C):
    coef = ...
    return coef
```

```
def iDCT(coef, C):
    data = ...
    return data
```

Puede comprobar que el funcionamiento de las funciones es el correcto comparando el resultado que producen con el de las funciones del mismo nombre incluidas `scalib`.

4. Codificación basada en la DCT

La codificación de imágenes basada en la DCT ha sido implementada exitosamente en diversos estándares de compresión, como JPEG (Joint Photographic Experts Group) y MPEG (Moving Picture Experts Group), siendo ampliamente utilizada en la industria multimedia. La DCT permite codificar imágenes con una cantidad reducida de bits, eliminando la información redundante, y sin sacrificar de manera significativa la calidad percibida. A grandes rasgos, la codificación basada en la DCT de una imagen comprende:

1. La división de la imagen en bloques de $N \times N$ píxeles.
2. La aplicación de la DCT directa a cada bloque, obteniendo una matriz de $N \times N$ coeficientes.
3. La selección de aquellos coeficientes que concentran la mayor parte de la información del bloque.
4. La cuantificación y codificación de los coeficientes seleccionados.

El nivel de compresión conseguido (tasa de bits resultante) depende de la cantidad de coeficientes que seleccionemos y del número de bits usado para cuantificar los coeficientes seleccionados. Una forma sencilla de seleccionar los coeficientes que concentran la mayor parte de la información es el método conocido como muestreo zonal. En este método, el número de bits usado para cuantificar cada coeficiente se determina a priori en base a la varianza del coeficiente, asumiendo que a mayor varianza mayor es la información que recoge un coeficiente dado. Aquellos coeficientes con varianzas muy bajas serán descartados (se asignan 0 bits para cuantificarlos). En la práctica, los coeficientes de menor varianza y generalmente descartados se corresponde con aquellos que representan las frecuencias altas. Por tanto una forma simplificada de llevar a cabo la asignación de bits y que evita el cálculo

[1] Recuerde que puede calcular el producto de matrices (arrays bidimensionales de `numpy`) usando el operador `@`.

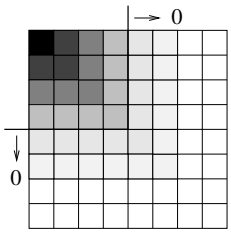

Figura 4: Representación de muestreo zonal $4:1$ para una matriz de coeficientes DCT de 8×8.

de la varianza de los coeficientes consiste en asignar un número fijo de bits para la cuantificación y codificación de los coeficientes que se corresponden con las frecuencias bajas (aquellos localizados en la esquina superior izquierda de la matriz en el caso bidimensional) y descartar los demás. En la Figura 4 se muestra la DCT de un bloque 8×8 píxeles de una imagen. Conforme nos alejamos de la esquina superior izquierda, los coeficientes de la DCT tienden a ser nulos, por lo que podemos suponer que sólo los coeficientes de la submatriz 4×4 de la esquina superior izquierda (componentes en frecuencias más bajas) portan la información relevante, mientras que el resto pueden considerarse nulos sin cometer mucho error. De esta forma se reduce el número de coeficientes necesario para representar el bloque de imagen en un factor 4. El efecto sobre la imagen será equivalente a un filtrado paso-baja.

Ejercicio 3

Construya un codificador/descodificador de imágenes basado en la DCT y en muestreo zonal. Para ello, implemente dos funciones con la siguiente interfaz:

```
def encoderDCT(data, dataRange, N, Nsel, b):
    code = ...
    return code
```

```
def decoderDCT(code, dataRange, N, Nsel, imageShape, b):
    data = ...
    return data
```

donde:

- `data` $\rightarrow$ Array bidimensional que contiene la imagen que vamos a codificar.
- `code` $\rightarrow$ Array unidimensional con la imagen codificada.
- `dataRage` $\rightarrow$ Tupla de 2 elementos con el mínimo y el máximo valor posible de las muestras en `data`.
- `N` y `Nsel` $\rightarrow$ Tamaño de los bloques y de los subbloques seleccionados durante el muestreo zonal (por ejemplo, `N=8` y `Nsel=4` representa bloques de 8×8 píxeles y muestreo zonal $4:1$.).
- `b` $\rightarrow$ Número de bits del cuantificador usado para cuantificar los coeficientes seleccionados.
- `imageShape` $\rightarrow$ Tupla con las dimensiones de `data` (tamaño de la imagen).

Si `data` no es divisible entre `N`, `encoderDCT` debe añadirle filas/columnas hasta que lo sea[2]. A su vez, `decoderDCT` debe tener en cuenta que `code` puede contener más píxeles de los esperados según `imageShape` por este motivo.

[2] Puede usar la función pad de numpy para ello.

Las imágenes suelen representarse con matrices cuyos valores son siempre positivos. Por ejemplo, el valor de un píxel codificado con 8 bits suele ser un número entero entre 0 y 255 o un número real (*float*) entre 0 y 1. Sin embargo, los codificadores basados en la transformada DCT son más eficientes (permiten una mayor compresión de los datos) cuando la entrada está centrada en 0. Por este motivo, es habitual escalar las imágenes antes de codificarlas mediante DCT restándole la mitad del rango (para que cada píxel tenga un valor entre -128 y 127 o entre -0.5 y 0.5, por ejemplo).

Ejercicio 4

Codifique la imagen de Lena (`lena.png`) en escala de grises y a continuación recupere la imagen a partir del código. Use las funciones implementadas en el Ejercicio 3. Puede usar la siguiente estructura:

```
im = io.imread(dataPath + 'lena.png', as_gray=True)     # im está en el rango [0, 1]
imZM = im - 0.5                                         # imZM está en el rango [-0.5, 0.5]
code = encoderDCT(imZM, ...)
imrZM = decoderDCT(code, ...)                           # imrZM está en el rango [-0.5, 0.5]
imr = imrZM + 0.5                                       # imr está en el rango [0, 1]
```

Pruebe con diferentes tamaños de bloque y tasas de bits de cuantificación. En cada caso, calcule la SNR de la imagen tras el proceso de codificación/descodificación, el tamaño de la imagen codificada (en kB) y valores la calidad percibida (representando las imágenes). Comente los resultados.

Por último, compare los resultados obtenidos con los que se obtendrían con un codificador PCM (use las funciones implementadas en el Ejercicio 1). A igual tamaño de la imagen codificada, ¿qué codificador obtiene una mejor SNR?