

Práctica 1. Señales digitales

Resumen

En esta práctica se estudia la lectura y la reproducción de señales digitales de audio e imagen utilizando MATLAB. También se estudian las clases numéricas disponibles en MATLAB y las funciones que permiten pasar de una clase a otra.

1. Audio

Las secuencias de audio digital usadas en las prácticas de TDSC están almacenadas en formato WAV/WAVE. Además de la secuencia de bits que resulta de la codificación, los ficheros WAV incluyen una cabecera con información sobre la secuencia de audio y el codificador utilizado. Así, la cabecera contiene el valor de la frecuencia de muestreo, el número de canales y el número de bits por muestra, entre otros parámetros. Puede ver la información almacenada en la cabecera de un fichero de audio utilizando la función `audioinfo`.

MATLAB permite leer un fichero WAV mediante la función `audioread`. El fichero `v1.wav` contiene una secuencia de voz de banda ancha. Lee `v1.wav` ejecutando

```
[x fs] = audioread('v1.wav')
```

donde `x` es la secuencia leída (decodificada) y `fs` es la frecuencia de muestreo asociada a `x`.

 Obtén los siguientes parámetros del audio del fichero `v1.wav`:

Número de canales:

Frecuencia de muestreo: muestras/segundo

Tasa binaria: bits/muestra kbps

Comprueba que la clase (o tipo de datos) de los elementos del vector `x` es *double* (coma flotante de 64 bits).

Por defecto `audioread` usa la clase *double* para representar las muestras de la secuencia decodificada y normaliza los valores de amplitud para que estén entre -1 y 1 (verifica que todos los valores de `x` están en ese intervalo). Sin embargo, los datos almacenados en un fichero `wav` no están generalmente codificados con 64 bits (como es el caso del fichero `v1.wav`). Si queremos que MATLAB represente los valores de la secuencia decodificada con la misma clase con la que están representados en la secuencia de bits, la instrucción `audioread` debe tener `'native'` como segundo argumento.


Comprueba que la verdadera clase de los datos almacenados en `v1.wav` es *int16* (entero con signo de 16 bits).

MATLAB dispone de dos funciones para reproducir un vector de audio: **sound** y **soundsc**. Conoce la diferencia entre estas dos funciones usando la ayuda de MATLAB. Aparte del vector a reproducir, estas funciones requieren el valor de la frecuencia de muestreo a la que se debe realizar la conversión D/C. Si dicha frecuencia de muestreo es distinta a la frecuencia de muestreo asociada a la secuencia de audio, la reproducción estará distorsionada.

El fichero **vt1.wav** contiene una secuencia que es la versión de banda estrecha de la secuencia de voz de **v1.wav**.

 ¿Cuál es la frecuencia de muestreo de **vt1.wav**? muestras/segundo

Escucha las secuencias de voz que contienen los ficheros **v1.wav** y **vt1.wav**.

 ¿En qué secuencia aprecias una menor distorsión?

Al hablar, la forma de la señal que generamos varía dependiendo del fonema que pronunciamos. Podemos clasificar los sonidos de la voz en:


- *Sonidos sonoros*. Corresponden a señales cuasiperiódicas. Tienen componentes espectrales en la *frecuencia de tono* del sonido (f_0) y en sus armónicos (kf_0). Por ejemplo, todas las vocales son sonidos sonoros.
- *Sonidos sordos*. Corresponden a señales con variaciones más erráticas, más rápidas y de menor amplitud que las de los sonidos sonoros. Ejemplos de sonidos sordos: /f/, /s/ y /z/.

Visualice **x** (secuencia del fichero **v1.wav**) ejecutando **plot(x)**. Para ver con mayor claridad la secuencia, amplía la gráfica horizontalmente (en el menú de **Tools-Options** marca las opciones **Horizontal Zoom** y **Horizontal Pan**). Ve al principio de la señal y haz zoom para ver con claridad las variaciones de amplitud. Desplázate horizontalmente e identifique segmentos de sonidos sonoros y sonidos sordos.

Los ficheros **v_e.wav** y **v_s.wav** contienen segmentos de los fonemas /e/ (sonido sonoro) y /s/ (sonido sordo) de una secuencia de voz. Visualiza, utilizando **subplot** y **plot**, las secuencias que contienen estos dos ficheros y compruebe que cada secuencia tiene las características que corresponde al tipo de sonido (sonoro o sordo) al que pertenece.

Vamos a estimar la *frecuencia de tono* del sonido de **v_e.wav**. Para ello:

1. Visualiza la secuencia de **v_e.wav** con **plot** (con el eje temporal en muestras).
2. Obtén la duración de un periodo: $T =$ muestras
3. Obtén la frecuencia de tono: $f_0 =$ Hz

 **SUGERENCIA.** ¿Cuál es tu frecuencia de tono? Pronuncia y graba una vocal durante un par de segundos utilizando el tono con el que hablas habitualmente. Lee el fichero generado en MATLAB y sigue los pasos descritos anteriormente.

El fichero `clarinete.wav` contiene un segmento del sonido de una nota musical interpretada con un clarinete.

 Lee `clarinete.wav` y obtén el valor de los siguientes parámetros:

Número de canales:

Frecuencia de muestreo: muestras/segundo

Tasa binaria: kbps

Observa que el audio es estéreo y que la frecuencia de muestreo es superior a la de los ficheros de voz. Visualiza las 1000 primeras muestras de los dos canales de `clarinete.wav` en una misma gráfica.

Observe que las dos secuencias son cuasiperiódicas (similares a las de los sonidos sonoros de la voz). En este caso f_0 es la frecuencia de la nota musical interpretada. Comprueba que hay un retardo entre los dos canales. Este retardo, junto con otras diferencias entre los dos canales, permite que nuestro sistema auditivo pueda localizar la fuente del sonido.

2. Imágenes

Las imágenes digitales usadas en las prácticas de TDSC son imágenes de 8 bits. En estas imágenes, cada componente está representada con un entero sin signo de 8 bits (0 es el valor mínimo y 255 el máximo). En MATLAB, esta clase se llama `uint8`.

Para representar en MATLAB una imagen RGB de N_1 filas y N_2 columnas utilizaremos un vector 3D de $N_1 \times N_2 \times 3$ elementos `uint8` (figura 1(a)). Las dos primeras variables del vector indexan el píxel de la imagen, mientras que la tercera indexa la componente de color. También podemos representar las componentes RGB separadamente usando tres matrices de $N_1 \times N_2$ elementos (figura 1(b)).

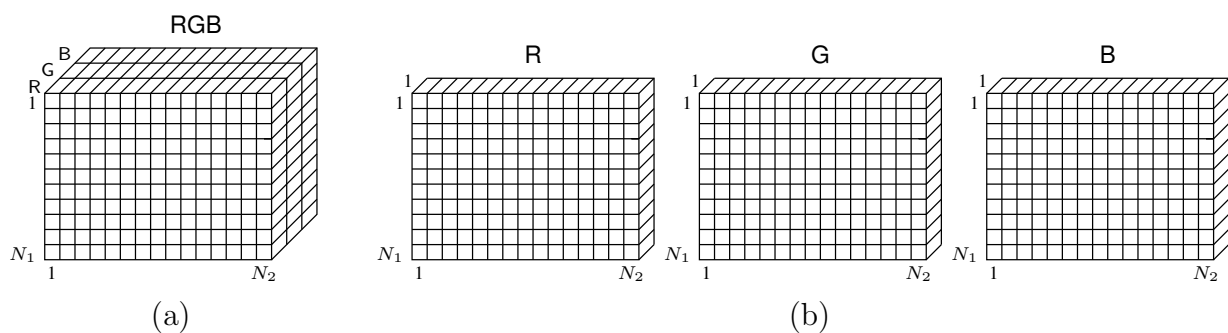




Figura 1. Representación de una imagen RGB: (a) mediante un vector $N_1 \times N_2 \times 3$. (b) mediante tres matrices $N_1 \times N_2$ (imágenes monocromas).

Las imágenes digitales utilizadas en las prácticas de TDSC están en formato PNG (*Portable Network Graphics*). Lee la imagen RGB del fichero `i1.png` con:

```
x = imread('i1.png');
```

 A partir de la variable `x`, obtén la siguiente información de la imagen leída:

Número de filas: Número de columnas: Número de componentes:

 Teniendo en cuenta el número de bits del fichero `i1.png` y el número de píxeles de la imagen, obtén la tasa binaria expresada bits/píxel:

Visualiza `x` ejecutando `imshow(x)`. Selecciona **Tools-Data Cursor** en el menú de la figura, apunta con el ratón a un píxel y pulsa el botón izquierdo: obtendrás del valor de las coordenadas del píxel y de sus componentes (R, G y B). Señala píxeles de distintos colores y compruebe que los valores de las componentes de cada píxel está en consonancia con su color.

Para visualizar la luminancia de la imagen `x` en *escala de grises* debemos:

- 1) Obtener la luminancia de cada píxel de `x` ($Y[n_1, n_2]$)
- 2) Visualizar una imagen RGB cuyas componentes sean

$$R[n_1, n_2] = G[n_1, n_2] = B[n_1, n_2] = Y[n_1, n_2].$$

Visualiza la luminancia de `x` en escala de grises ejecutando:

```
y = rgb2gray(x);
imshow(y);
```

En este código:

- La función `rgb2gray` devuelve la matriz `y` (clase `uint8`) que es la luminancia de `x`.
- Hemos aprovechado el hecho de que si `y` es una *matriz* de clase `uint8` (imagen monocroma) y ejecutamos `imshow(y)`, MATLAB visualiza la imagen con componentes:

$$R[n_1, n_2] = G[n_1, n_2] = B[n_1, n_2] = y[n_1, n_2]$$

esto es, visualiza `y` en escala de grises.

Teniendo en cuenta todo lo anterior, visualiza `x` y sus componentes R, G y B (estas tres últimas en escala de grises). Utiliza `subplot` e `imshow` para visualizar las cuatro imágenes simultáneamente.

Utilizando la función `rgb2ycbcr`, obtén la imagen `x` en el espacio de color YCbCr. Visualiza `x` y sus componentes Y, Cb y Cr (estas tres últimas en escala de grises).

3. Clases numéricas en MATLAB

En las secciones anteriores, hemos trabajado con secuencias cuyas clases eran de clases *double* y *uint8*. En esta sección, revisaremos las clases numéricas definidas en MATLAB y las funciones que permiten pasar de una clase a otra. Prestaremos especial atención a las clases de enteros puesto que la codificación de enteros es fundamental en codificación de fuente.

La siguiente lista describe las clases numéricas disponibles en MATLAB:

- *double* y *single* representan números reales con 64 bits y 32 bits, respectivamente. Ambas clases codifican los números reales con coma flotante.
- *int8*, *int16*, *int32* e *int64* representan enteros con 8, 16, 32 y 64 bits, respectivamente. La clase *intb* ($b \in \{8, 16, 32, 64\}$) codifica los enteros entre -2^{b-1} y $2^{b-1} - 1$ en complemento a dos (por ejemplo, la clase *int8* codifica -3 como 11111101).
- *uint8*, *uint16*, *uint32* y *uint64* representan enteros sin signo con 8, 16, 32 y 64 bits, respectivamente. La clase *uintb* ($b \in \{8, 16, 32, 64\}$) codifica los enteros entre con 0 y $2^b - 1$ en binario directo (por ejemplo, la clase *uint16* codifica 2048 como 0000100000000000).

Podemos conocer la clase de una variable con la función `class`. Por defecto, MATLAB utiliza la clase *double*. Por ejemplo, si ejecutamos

```
>> a = 3; class(a)
```

obtenemos 'double'.

MATLAB dispone de las funciones de conversión de clase: `double`, `single`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32` y `uint64`. Estas funciones permiten cambiar la clase de cualquier variable numérica (el nombre de la función indica la clase destino). Puede que sea imposible representar el argumento de la función de conversión en la clase destino. Así, al ejecutar

```
>> a = -3.7; b = uint8(a)
```

el valor de `b` no es 3.7 porque la clase *uint8* solo permite representar enteros entre 0 y 255. Cuando esto ocurre, la función de conversión devuelve el valor más próximo al argumento que puede representarse en la clase destino. Así, el valor de `b` es 0 en el ejemplo anterior.

También podemos realizar la conversión con la función `cast`. Por ejemplo, al ejecutar

```
>> a = 37.5; b = cast(a, 'uint32')
```

la variable `b` es de la clase *uint32* y su valor es 38.

Algunas funciones de MATLAB para generar vectores, como `ones` o `zeros`, permiten especificar la clase del vector generado. Así, al ejecutar

```
>> a = zeros(1,2,'int8');
>> a(1,1) = -140
```

el vector `a` es de la clase `int18` y su contenido es `[-128 0]`.

Al realizar operaciones aritméticas con algún operando de clase entera hay que tener en cuenta que MATLAB solo permite:

- aritmética entre operandos enteros de la misma clase entera (clase en la que también se proporciona el resultado). Así al ejecutar

```
>> a = int16(150); b = uint8(2); c = a*b
```

MATLAB devuelve un error y no genera `c`. En cambio, si ejecutamos

```
>> a = uint8(150); b = uint8(2); c = a*b
```

no hay error, `c` es `uint8` y su valor es 255.

- aritmética entre una clase entera y la clase `double` (el resultado se proporciona en la clase entera). Por ejemplo, al ejecutar

```
>> a = double(150); b = uint8(2); c = a*b
```

la variable `c` es `uint8` y su valor es 255.

En estos dos últimos casos el producto es 255 y no 300 debido a que no podemos representar enteros mayores que 255 en `uint8`. Otro inconveniente de la aritmética entera es que no podemos realizar multiplicaciones de vectores de clases enteras (al menos uno de los operandos debe ser escalar).

Los problemas descritos anteriormente pueden solucionarse convirtiendo los operandos enteros a `double`. Tras realizar la operación, siempre podemos convertir el resultado a una clase entera si es necesario. Veámoslo con un ejemplo. Considera los vectores `x = [0 3 6]` e `y=[3 6 9]`, ambos de clase `uint8`. Supongamos que `y` es una aproximación de `x` y que queremos calcular el error (de aproximación) cuadrático medio. Podríamos ejecutar

```
>> e = x-y
>> MSE = (e*e')/3
```

Pero la ejecución de la primera línea proporciona el vector `e = [0 0 0]` (que es erróneo) y la segunda línea no se ejecuta porque involucra la multiplicación de dos vectores de clase entera. En cambio, si ejecutamos

```
>> e = double(x)-double(y)
>> MSE = (e*e')/3
```

obtenemos `e = [-3 -3 -3]` y `MSE = 9` (ambos correctos).