PROYECTO de DITHERING

Para que sea más sencillo usaremos inicialmente una imagen en blanco y negro (512x512 con 8 bits/píxel, 256 niveles de gris) contenida en el fichero 'imagen.png'. Cargad la imagen con **`imread()`**, convertidla a double con **`im=im2double(im)`** para convertirla a valores entre 0 y 1 y visualizarla con **`imshow()`**. En la parte superior de la imagen se ha añadido una escala de grises para apreciar mejor cómo funcionan los distintos algoritmos sobre tonos "puros" de gris.

Como referencia aplicad a la imagen el algoritmo del ejercicio anterior que ponía a 1 los píxeles con valor igual o superior a 0.5 y a 0 los que estén por debajo. Recordad operar con la imagen como un todo, evitando hacer bucles. Adjuntad la imagen resultante. Para cuantificar el error cometido, calcular con std2() la desviación standard de la resta entre la imagen original y nuestro resultado.

Como sucedió con la imagen en color del ejercicio de clase, el resultado es bastante malo. En particular, en la zona de la escala de grises la salida se mantendrá a 0 hasta que el nivel de gris alcanza el valor 0.5 y luego pasa a valer 1, dando lugar a una salida "binaria". Podríamos pensar que esto es inevitable al usar en la salida únicamente los valores 0 (negro) y 1 (blanco). Sin embargo, como en el ejemplo en color del ejercicio del otro día, es posible obtener una imagen visualmente más atractiva si usamos la densidad de puntos negros en la imagen de salida para representar el nivel de gris del original. Como entonces, la idea es que a una cierta distancia el ojo "mezcle" los valores 0/1 en un tono medio de gris más o menos oscuro según la proporción entre píxeles negros y blancos.

1. Dithering por difusión del error

Una familia de técnicas de dithering se basa en el concepto de propagación/difusión del error. En estos algoritmos el error cometido en cada píxel se tiene en cuenta al procesar los siguientes píxeles.

En http://www.tannerhelland.com/4660/dithering-eleven-algorithms-source-code/ podéis encontrar una exposición de estas técnicas de difusión. La diferencia entre los distintos algoritmos está en cómo se recorre la imagen y en cómo se propaga el error y a cuántos píxeles posteriores afecta. En su versión más sencilla (propagación del error a un único píxel) el algoritmo sería:

1) Inicializamos el error cometido como e=0.

2) Con un bucle recorremos todos los píxeles de la imagen y en cada píxel:

   a) Sumamos e al valor del píxel a procesar, obteniendo el valor V
   b) Comparamos V con 0.5, asignando 0 o 1 al resultado según la comparación.
   c) Recalculamos el nuevo error cometido e=(V-valor_salida) para usarlo en el siguiente píxel.

Imaginad que el algoritmo se aplica a una serie de píxeles con valores 0.4, 0.5, 0.4. La primera salida sería 0 (ya que 0.4<0.5) y el error cometido es e=(0.4 - 0) = 0.4. Sumando este error al siguiente píxel obtenemos 0.9 (0.5+0.4). Al compararlo con 0.5 la salida sería 1 (0.9≥0.5) y el error e= (0.9 - 1) = -0.1. El siguiente píxel quedaría 0.3 (0.4 - 0.1), lo que daría lugar a una salida de 0, etc. El resultado es una serie de 0 y 1 alternos que representan adecuadamente un tono medio de gris. De esta forma nos aseguramos de que aunque cada píxel tenga un error alto, la media de los píxeles de la salida (0/1) coincida LOCALMENTE con los valores de los píxeles de la imagen original. Notad que con el algoritmo original la salida serían todos 0's, al ser los tres píxeles menores de 0.5.

Con estos métodos lo ideal es recorrer la imagen pasando por píxeles contiguos. Si recorremos la imagen por filas, al saltar de fila estaríamos pasando el error entre dos píxeles que están lejos entre sí. Otra opción es resetear el error a 0 al terminar cada fila, pero podemos obtener mejores resultados si se recorren las filas de forma alterna: tras recorrer la 1ª fila hacia la derecha, recorremos la segunda hacia la izquierda y así sucesivamente. Gráficamente el esquema sería:

```
----------------->  (X,1)  ---------------------------> (X
 X) <--------------------------- (1,X) <----------------  1)
(1  ----------------->
```

La flecha indica la dirección de procesado, la X es el píxel siendo procesado y el 1 nos indica la casilla donde se suma el error cometido, que en este caso se pasa al siguiente píxel procesado. Al terminar una fila el error pasa al píxel de abajo en la siguiente fila.

Escribid una función **`function im=dither(im)`** para implementar este método, que recorra la imagen por filas, alternando la dirección en cada fila. La función recibe una imagen im (niveles de gris con valores "continuos" entre 0 y 1) y la machaca con la imagen resultado (imagen binaria con solo 0 y 1). Adjuntad el código de vuestra función dither() y una captura de la imagen resultado. **Al copiar estas imágenes tratad de mantened su tamaño en el documento. Si se cambia pueden aparecer efectos raros al eliminar o recolocarse los puntos).** Extraer también de la imagen la zona del ojo (desde la fila 110 a la 220 y columnas 320 a 440. Adjuntad captura del resultado (debe ser similar a la figura adjunta).

¿Cuál es el principal problema que se aprecia visualmente en la imagen resultante? Este problema es especialmente notorio en la zona de la escala de grises. ¿Cuál es su causa?

Volver a calcular la desviación standard (std2) de la diferencia entre la imagen original y el resultado y volcad su resultado. ¿Mejora respecto al método anterior?

¿Es adecuado este criterio matemático para juzgar la calidad "visual" del resultado?

Para mejorar este resultado podemos recorrer la imagen en una trayectoria que cambie continuamente de dirección. Una posibilidad es inspirarnos en la idea de una curva de Hilbert, una línea 1D capaz de cubrir por completo una superficie 2D. Si hacéis **`>>load hilbert`** veréis dos vectores I y J con las sucesivas coordenadas (i,j) de una curva de Hilbert que recorre por completo los 512 x 512 píxeles de nuestra imagen 2D. Para apreciar la forma de este recorrido haced un plot de ambas coordenadas (I,J). Haced zoom sobre el gráfico hasta que veáis el trazado de la curva de Hilbert. Adjuntad captura del zoom.

Cread una nueva función **`function im=dither_hilbert(im)`** que difunda como antes el error al píxel siguiente, pero ahora siguiendo la curva dada por las coordenadas I,J. Basta hacer un único bucle barriendo secuencialmente los píxeles I(k) e J(k) de la imagen. Adjuntad el código de la nueva función y una captura de la imagen resultado junto con el detalle como antes del ojo derecho.

Si queremos seguir recorriendo la imagen por filas podemos mejorar el resultado si difundimos el error a más de un píxel, como se refleja por ejemplo en el siguiente esquema:

```
-------->   X   4/8  ------------------------------> X
          3/8 1/8      <-------- 4/8   X  <------   1
                                 1/8  3/8
```

Ahora el error del píxel procesado (X) se reparte a 3 de sus vecinos (derecha, abajo y diagonal) usando las proporciones indicadas. Dado un error e=0.4 al procesar X, este error se repartirá entre los tres vecinos, sumando 0.2 (0.4·4/8) al píxel de la derecha, 0.15 (0.4·3/8) al de abajo y 0.05 (0.4/8) al de abajo en la diagonal. Notad que los píxeles a los que se traslada el error todavía no han sido procesados. Al llegar al final de una fila no tenemos píxeles a la derecha, por lo que se pasa todo el error al píxel de abajo. Al igual que antes la siguiente fila se recorre en sentido contrario. Adjuntad código de la nueva función y una captura del resultado aplicado a la imagen test junto con el detalle del ojo.

Para mejorar estos resultados se puede seguir aumentando el número de vecinos a los que propagamos el error, con la desventaja de que el programa se irá haciendo más complicado. Uno de los algoritmos de dithering más usados por su compromiso entre simplicidad y buenos resultados es el de Floyd-Steinberg (1976) o alguna de sus variantes que usa el esquema:

```
→      X    7/16
3/16  5/16  1/16
```

Notad como los divisores del error (antes 8, ahora 16) son potencias de 2. Esto nos permite implementar divisiones como desplazamientos de bits, con la consiguiente mejora en el rendimiento.

En algunos de los algoritmos anteriores se ha calculado la desviación standard (σ) de la diferencia entre el original y el resultado, pero hemos visto que este criterio matemático no se corresponde con la imagen visualmente más atractiva. Rellenad la siguiente tabla con las σ's encontradas para las 4 implementaciones anteriores:

comparación simple, propagación del error a un píxel (por filas y siguiendo la curva de Hilbert) y propagación del error a varios píxeles:

Resul	Comp directa	Prop 1 filas	Prop 1 hilbert	Prop 3 pix
σ(res-I_org)				

La razón de que visualmente las imágenes con dithering sean más atractivas es que como hemos comentado el ojo tiende a promediar los píxeles (0/1) individuales y a interpretarlos como niveles de gris. Para tener en cuenta esto, antes de calcular el criterio de la desviación estándar haremos previamente una operación sobre las imágenes binarias que simule lo que sucede en el ojo. Para ello vamos a promediar en las imágenes binarias los píxeles vecinos usando una operación de filtrado.

Estudiaremos con más detalle este tipo de operaciones más adelante, pero aquí solo vamos a aplicarlas. Para hacer este tipo de filtrado en MATLAB haremos:

```
G=fspecial('gauss',7,1.5); im_filt=imfilter(im_binaria,G,'symm');
```

El resultado de esta operación aplicada a una imagen binaria es el promedio en cada punto de los 7x7 valores alrededor de ese píxel usando los coeficientes de la matriz G. Cuanto más píxeles blancos haya en la vecindad de un píxel dado, mayor será su valor. Filtrad las cuatro imágenes binarias obtenidas en este apartado de la forma indicada y volved a calcular la desviación standard de sus diferencias con la imagen original. Rellenad con los resultados obtenidos la tabla adjunta:

Resul	Comp directa	Prop 1 filas	Prop 1 hilbert	Prop 3 pix
σ(res_filt-I_org)				

Observaréis que con este nuevo criterio las imágenes visualmente más atractivas obtienen ahora un resultado más bajo, indicando un mayor parecido con la original.

2. "Ordered dithering"

Aunque estos algoritmos de difusión del error obtienen buenos resultados (y pueden mejorarse ampliando la difusión del error a más píxeles), son obviamente más complicados de programar que el método original donde comparábamos cada píxel con 0.5 para decidir si la salida iba a ser un 0 o un 1. Además, los métodos de difusión no son fácilmente paralelizables, al depender el resultado en un píxel del procesado de los píxeles anteriores. Por el contrario el algoritmo original es sencilo de paralelizar: al tratar cada píxel independiente de los demás siempre podremos dividir la imagen y procesar los trozos por separado. En este apartado trataremos de mantener la simplicidad del método original mientras mejoramos sus resultados.

Para evitar el aspecto binario de la imagen obtenida al comparar el original con 0.5, vamos a añadir una componente aleatoria en la comparación. Queremos que un píxel con valor 0.9 tenga una alta probabilidad de quedarse como un 1, pero no una total seguridad. Para ello, en vez de comparar los píxeles siempre con el mismo

valor 0.5, usaremos un valor aleatorio entre 0 y 1 (distinto para cada píxel). La idea es que si el valor de la imagen original es alto es muy probable (pero no del todo seguro) que "supere" al correspondiente aleatorio y se transforme en un 1. Lo mismo sucede si el píxel original está cerca del cero: lo normal es que pierda y se transforme en un 0's con una alta probabilidad. Por el contrario, si tenemos un área uniforme con un valor de gris ~ 0.75, aproximadamente el 75% de esos píxeles "ganarán" y se pondrán a 1 y un 25% "perderán" y se pondrán a 0. Con un 75% de píxeles blancos el resultado percibido desde lejos sería un gris claro, similar al gris (0.75) de la imagen original. Con el método original todos esos píxeles se quedarían como 1 (al ser 0.75≥0.5) y obtendríamos un blanco sólido.

La forma más sencilla de implementar este método es crear una imagen del mismo tamaño que la original pero ahora llena de números aleatorios: **`R = rand(size(im));`** Los píxeles de la imagen de salida valdrán 0 o 1 dependiendo de si el píxel de la imagen original es mayor/igual o menor que el correspondiente píxel de la imagen R de aleatorios. Adjuntad el código usado (se valorará no usar bucles) y la imagen resultado. Dad el valor de la desviación standard de la diferencia entre el original y el resultado filtrado como se hizo en el apartado anterior. Vemos que el resultado, aunque mejor que el obtenido al comparar directamente con 0.5, es bastante peor que los obtenidos con las técnicas de difusión del error.

La forma de mejorar estos resultados es siendo más cuidadosos a la hora de diseñar la matriz de ruido R usada en la comparación. Una opción es el llamado dithering "ordenado", donde la matriz R se crea replicando unas matrices más pequeñas de tamaño 2x2, 4x4, ... diseñadas específicamente con este fin. A continuación se detalla un algoritmo recursivo para construirlas: en el proceso se usa una matriz auxiliar M con un tamaño que se duplica en cada etapa. Inicialmente se parte de M=0 y δ=1 y se aplica la siguiente regla:

$$\delta=\delta/4. \quad M=\begin{pmatrix} M & M+2\delta \\ M+3\delta & M+\delta \end{pmatrix}$$

Al llegar al tamaño deseado, a partir de la matriz M se genera la correspondiente matriz de "ruido" $R_n=(M+\delta/2)$.

En el 1er nivel (1x1) tenemos M=0, δ=1 por lo que $R_0 = (M+1/2)=0.5$. Replicando este valor obtendríamos una matriz R=0.5 lo que correspondería a usar el algoritmo inicial en el que comparábamos la imagen siempre con el mismo valor 0.5. Para el siguiente tamaño (2x2):

$$\delta=0.25 \quad \text{y} \quad M=\begin{pmatrix} 0 & 0.5 \\ 0.75 & 0.25 \end{pmatrix} \rightarrow R_1=\begin{pmatrix} 0.125 & 0.625 \\ 0.875 & 0.375 \end{pmatrix}.$$

Volcad los valores de la matriz R_3 (8x8) obtenida con este algoritmo.

Vemos que en vez de ser aleatorios los valores de estas matrices cubren el intervalo [0,1], separados por un salto de 1/(nº de elementos) y están organizados de forma que se evitan los valores consecutivos en posiciones adjuntas en la matriz. De esta forma se obtiene un buen patrón de dithering. Una vez que disponemos de la matriz

R_n, para obtener la matriz R a usar en la comparación, basta replicar R_n hasta llegar al tamaño de nuestra imagen. Usad para ello el comando repmat():

B=repmat(A,M,N) or B=repmat(A,[M,N]) creates a larger matrix B consisting of an M-by-N tiling of copies of matrix A.

Tras obtener la matriz R el proceso es el mismo: comparar la imagen con R y dar valores 0 o 1 a la salida dependiendo del resultado de la comparación. Adjuntad el código para crear R a partir de R_3 (8x8), la imagen resultante final y el detalle alrededor del ojo derecho. Volcad también el valor de la desviación standard de la diferencia entre el original y el resultado filtrado. Vemos que con estas técnicas de "ordered dithering" se obtienen resultados muy similares a los de difusión del error siendo más sencillas y rápidas de implementar.

Dithering usando ruido azul:

Aunque el resultado anterior es bastante bueno, en las zonas homogéneas de la imagen tienden a aparecer patrones regulares, debidos a la regularidad intrínseca de las matrices R_n. Este efecto es más visible en la zona de la escala de grises que, dependiendo de la matriz R_n usada, puede llegar a verse como un conjunto discreto de "niveles" de gris en vez de la gradación continua original. Repetir el proceso de dithering usando la matriz R_2 (4x4) y adjuntad una captura de la zona de la escala de grises en la imagen resultante.

Para evitar la aparición de este tipo de patrones, podemos tratar de dar a la matriz R un carácter más aleatorio. Al usar la función rand() el ruido obtenido es lo que en procesado de señal se denomina "ruido blanco", ya que en él están presentes por igual todas las frecuencias. En estas aplicaciones de "dithering" es más adecuado un tipo de ruido denominado "blue noise", un ruido con un predominio de frecuencias altas (lo que en el caso de la luz daría lugar a un color azulado, de ahí el nombre).

Este tipo de ruido es más complicado de crear y Matlab no tiene una función para hacerlo, por lo que os daré yo una muestra. Cargad los datos del fichero blue.mat haciendo **>>load blue** y veréis una matriz **B** de tamaño 64x64 con una muestra de este tipo de ruido (con valores entre 0 y 1). Para ver cómo este ruido "azul" es preferible frente al ruido "blanco", recordad lo que dijimos de que el ojo tiende a hacer un promedio/filtrado de la imagen.

Crearemos dos imágenes de ruido R1 y B1 (ambas de tamaño 128x128 con valores entre 0 y 1). Para la primera (ruido blanco) usaremos la función rand(). Para la segunda replicaremos la muestra de ruido azul hasta conseguir el tamaño pedido. Mostrad ambos ruidos uno al lado del otro usando: **imshow([R1 B1])** y adjuntad una captura. Aplicad luego a ambos ruidos R1 y B1 el filtrado usado antes y adjuntad una nueva captura de ambos ruidos uno al lado del otro tras su filtrado.

La idea es que cuanto más parecido sea el ruido filtrado a un nivel uniforme de un gris medio más adecuado será ese ruido para usar en un algoritmo de dithering.

Replicad ahora la muestra de ruido azul hasta el tamaño de la imagen original y usarla para hacer el dithering de la imagen. Adjuntad la imagen resultado.

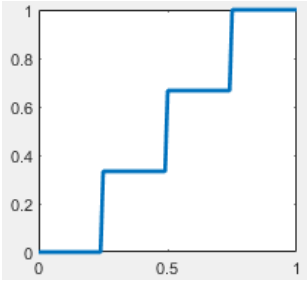
Para aplicar estos algoritmos de "dithering" a imágenes en color, lo más sencillo es aplicarlos independientemente a cada uno de los tres planos de color de la imagen. Cargad la imagen del fichero "color.jpg" y aplicadle un "dithering" (usando ruido azul) a sus planos de color. Adjuntad código usado y la imagen resultado. Si hacéis zoom sobre la imagen veréis que solo aparecen 8 colores (3 bits): negro (000), blanco (111), los 3 colores puros rojo (100), verde (010) y azul (001) y sus colores complementarios cyan (011), magenta (101), amarillo (110).

3. Dithering con más de 2 niveles

Sin llegar al caso extremo de tener que quedarnos con dos niveles la calidad de una fotografía siempre se degradará si reducimos el número de niveles de grises en la salida. En la figura adjunta podéis observar el resultado de usar directamente una imagen de 8 bits/píxel con 256 niveles como salvapantallas de mi kindle (derecha).

Como mi kindle solo soporta 4 bits/píxel, solo tengo 16 tonos de gris en la foto de la derecha. El fenómeno que se aparece se denomina “posterización” o aparición de falsos bordes. El problema es que con tan pocos niveles el ojo empieza a ser capaz de distinguir los cambios de uno a otro, dando lugar a la percepción de falsos bordes. Es fácil reducir estos problemas generalizando las técnicas que hemos aprendido en este proyecto. Es sencillo modificar los **algoritmos de difusión del error** que hemos visto para obtener más niveles en la salida. Basta cambiar la comparación binaria de antes (si V≥0.5, out=1 else out=0) por una asignación de los diferentes niveles de salida dependiendo del valor de V.

Por ejemplo, con 4 niveles usaríamos 4 valores (0, 1/3, 2/3, 1) equi-espaciados en el intervalo [0,1] como los **4 niveles de reconstrucción** de la salida. Luego se definen (con una separación de 1/4) los **5 niveles de decisión** (0, 1/4, 1/2, 3/4, 1) que marcan los intervalos que deciden la salida a asignar. La gráfica adjunta ilustra la relación entre la entrada (eje X) y la salida (eje Y) para este cuantificador: si el valor V es menor de 0.25 la salida sería un 0, si V está entre 0.25 y 0.5 sería 1/3, entre 0.50 y 0.75 tendríamos 2/3 y para un valor V>0.75 la salida sería 1.



Escribid la función **`function v=quant(v,L)`** que reciba un valor v en [0,1] y devuelva el correspondiente valor de salida usando L niveles. Adjuntad el código de la función y usadla para hacer una gráfica como la anterior donde el eje X muestra el valor de entrada y el eje Y el de salida, pero para el caso de L=8 niveles en la salida. Adjuntad los valores de reconstrucción y decisión correspondientes a L=8 niveles.

Una vez que tenemos la función **`quant()`** podemos usarla en vez de la comparración con 0.5 para asignar el valor de salida dentro de cualquier función de propagación del error. No hay que cambiar nada más ya que la parte de recorrer la imagen y propagar el error se mantiene como antes. Partid de la función de difusión de error siguiendo la curva de Hilbert y modificadla para crear una nueva función: **`function im=dither_hilbert_L(im,L),`** cuyo 2º argumento de entrada indica el número L de niveles deseados en la salida. Adjuntad código de la nueva función y el resultado de aplicarla a la imagen test para el caso de L=8 niveles. Usad **`hist(im8(:),100)`** para visualizar el histograma de la nueva imagen y comprobar que solo tiene 8 niveles. Adjuntad el histograma obtenido.

Como comparación, vamos a crear la imagen que resultaría si simplemente nos quedáramos con los 3 bits (2^3=8 niveles) más significativos de la imagen original. Para ello podéis partir de la imagen original en tipo uint8 (sin convertirla a valores entre 0 y 1) y usar la función bitand() para poner a 0 sus 5 bits menos significativos. Adjuntad el código usado y una captura de la imagen resultante.

Antiguamente manejar imágenes en lo que se denominaba color "verdadero" (los habituales 8 bit/pixel por canal = 256^3 colores) podía ser muy costoso debido a las limitaciones de almacenamiento/memoria. Una posibilidad era limitarse a imágenes en color con solo 8 bit/píxel en total (para los tres canales). Los 8 bits se repartían entre los canales Rojo (3), Verde(3) y Azul(2). Esto equivalía a codificar la imagen usando 2^3=8 niveles en sus planos R y G y 2^2=4 niveles en el plano B. Adjuntad el código usado para obtenerla y una captura de la imagen (3+3+2) obtenida junta a la original.

Notad que hemos pasado de una paleta de 256^3 = 16777216 colores posibles a disponer sólo de 2^8 = 256 colores. En este caso los resultados son muy similares, habiendo conseguido una imagen en color que ocupa lo mismo que una monocroma (8 bits/pixel) y una tercera parte de la original.

4. Recuperación de una imagen oculta

En la imagen ("imagen.png") que hemos usado en esta práctica había oculta una segunda imagen, guardada en su bit menos significativo. La imagen oculta es una imagen en color de tamaño 340 (filas) x 256 (columnas) almacenada con 1 bit/pixel por canal. En la imagen están guardados primero los bits del canal rojo, luego los del verde (1 bit/píxel) y finalmente los del azul (1 bit/píxel). Los planos de color están separados por una línea de 512 píxeles negros (0). Usando estas indicaciones extraer la imagen oculta. Adjuntad el código usado para que yo vea cómo lo habéis hecho y adjuntad una captura de la imagen obtenida.