MUESTREO e INTERPOLACIÓN

APELLIDOS:	**NOMBRE:**
APELLIDOS:	**NOMBRE:**

Fecha de entrega: hasta el 13 de diciembre

El problema que vamos a tratar consiste en reducir o ampliar una señal (secuencia en *Matlab*). Reducirla (diezmarla o submuestrearla) consiste en conservar sólo una muestra de cada S muestras originales. Ampliarla consiste en generar a partir de la original $x[N]$, otra con más muestras, típicamente un factor S más. El problema entonces será qué valores poner en las nuevas muestras creadas.

El interés de este tipo de funciones radica en poder cambiar de ritmo de muestreo. Si por ejemplo tenemos una secuencia de N datos correspondientes a una frecuencia de muestreo f_m y queremos archivarlos en un formato que exija otra frecuencia de muestreo f'_m podremos hacerlo facilmente con este tipo de operaciones.

La cosa es especialmente fácil si el cociente (f'_m/f_m) se reduce a una fracción sencilla. Por ejemplo, si tengo 2 segundos de audio a $f_m = 5000Hz$ (10000 muestras) y deseo pasar a $f'_m = 7500$, como el cociente es 3/2, aplicaría una ampliación por un factor 3 (obteniendo 30000 muestras a un ritmo de 15000 Hz) seguida por un diezmado a ritmo 2 (quedandome con 15000 muestras, correspondientes a 2 segundos a ritmo 7500 muestras/sec).

Esta es la razón por la cual la frecuencia de muestreo en un CD audio sea de 44100 Hz. Podemos entender que sea de ese orden ya que eso permite reproducir fielmente frecuencias hasta unos 22 Khz, un poco por encima del límite del oyente normal (unos 20Khz). ¿Pero por qué 44100 y no un número más redondo como 44000 o 45000 Hz? La respuesta la vemos si factorizamos dicho número:

$$44100 = 2 \times 2 \times 3 \times 3 \times 5 \times 5 \times 7 \times 7$$

Vemos que 44100 es factorizable en enteros pequeños, lo que facilita el "remuestreo" de dichos datos a otros ritmos de muestreo, que también sean producto de enteros pequeños.

Diezmado de una secuencia: ilustración del "aliasing" en una dimensión

Sabemos por el teorema del muestreo que si muestreamos una señal por debajo de su frecuencia de Nyquist (el doble de la máxima frecuencia presente) tendremos problemas. Técnicamente este problema se le denomina "aliasing" y consiste en que las frecuencias superiores a la mitad de la frecuencia de muestreo aparecen en la secuencia muestreada como frecuencias distintas (esto es, con otra identidad, en inglés a eso se le llama tener un "alias", de ahí el nombre de "aliasing").

Para ilustrar este fenómeno partiremos de una secuencia representando una sinusoide de 50 Hz con una duración de 1 segundo. La vamos a muestrear a una frecuencia suficientemente alta (5000 Hz) para que a efectos prácticos sea como tener un continuo.

```
>> fs=5000; t=[0:1/fs:1.0];        % Un segundo a intervalos de 1/5000
>> f=50; y=cos(2*pi*f*t);          % Secuencia de f=50 Hz muestreada finamente.
```

Vamos a diezmar la secuencia `y[]` tomando una muestra de cada S (en *Matlab* usar `y(1:S:end)`). La nueva secuencia tendrá una frecuencia de muestreo S veces menor:

$$f'_m = \frac{f_m(\text{ original })}{S} = \frac{5000}{S} Hz$$

Como la señal original tiene 50 Hz, la mínima frecuencia de muestreo admisible serían 100 Hz, lo que supone que podríamos admitir un diezmado máximo de S=50 (5000 Hz/50 = 100Hz). Lo que vamos a hacer es usar factores de diezmado superiores (por lo que las frecuencias de muestreo de las nuevas secuencias serán inferiores a la marcada por Nyquist), viendo como en la secuencia submuestreada aparecen frecuencias diferentes a la única existente en la señal real (50 Hz). Para ello, pintaremos en una gráfica la señal original y la secuencia diezmada. Para remarcar nuestra idea de que `y[ ]` es un continuo la pintaremos con `plot`, mientras que la secuencia diezmada la pintaremos con `stem`.

```
>> S=10; tt=t(1:S:end); yy=y(1:S:end); plot(t,y,'g'); hold on; stem(tt,yy,'r'); hold off
```

En el caso anterior, al ser $S < 50$, la nueva frecuencia de muestreo, f'_m es igual a $5000/S = 5000/10 = 500Hz$, y la señal submuestreada reproduce adecuadamente la original. Para verlo mejor podemos pedir a *Matlab* que solo muestre el fragmento de la señal correspondiente a una decima de segundo:

```
>> set(gca,'Xlim',[0 0.1])
```

Volver a mostrar la señal al completo (usando set(gca,'Xlim',[0 1])) y repetir las ordenes anteriores (usar ↑, no volvais a teclear todo) usando los valores de S que listamos a continuación, indicando la frecuencia "aparente" de la señal submuestreada. Para determinar dicha frecuencia tener en cuenta que al estar cubriendo un segundo de señal el número de crestas que nos aparecen en la ventana serán justamente los ciclos por segundo, esto es, la frecuencia aparente en Hz.

Factor S	75	91	98	100	101	109	196	200	204
Frec.muestreo (5000/S) (Hz)	67	55	51	50	49	46	25	25	24
Frec. aparente (Hz)									

Hay que darse cuenta de que una vez que muestreamos ya no podemos ver la señal original (en verde). Viendo sólo la señal muestreada pensaríamos que tenemos una frecuencia totalmente distinta. Este es el verdadero problema del "aliasing": las frecuencias presentes en la señal original por encima de la frecuencia de Nyquist no desaparecen, sino que aparecen como frecuencias fantasmas (o "alias") donde no se las espera.

Aliasing en audio

Podríais pensar que una señal de audio está hecha para oirla, no para verla, por lo que tal vez, a pesar de lo que vemos en la pantalla, dichas secuencias submuestreadas se oirían correctamente. Vamos a comprobarlo generando un pitido de frecuencia creciente y ver que sucede. Usaremos un muestreo con una frecuencia de f_s=8000 Hz:

```
>> fs=8000; t=[0 : 1/fs : 6];   % Seis segundos de tiempo a saltos de 1/8000 sec
>> f=150*t;  % Frecuencia proporcional a t. Barremos de f=0Hz (t=0) hasta f=900Hz (t=6)
>> pitido=0.5*cos(2*pi*f.*t);  % Pitido de amplitud 0.5 y frecuencia creciente 0 -> 900 Hz
>> sound(pitido,fs);
```

Describir el sonido escuchado.

Cambiar la línea anterior `f=150*t` por la siguiente: `f=2000*t`, incrementando la velocidad de cambio de la frecuencia, de forma que ahora varia entre 0 (t=0) y 12000 Hz (t=6). Volver a construir el "pitido" y hacerlo sonar. ¿Escucháis lo esperado? Describir el sonido y justificar lo que ois.

También podeis comprobar el efecto del submuestreo en los datos del fichero `vozam.wav`, que contiene voz muestreada a 44100 Hz. Con este ritmo de muestreo es casi seguro que no hemos perdido nada con respecto al continuo, por lo que al diezmarla es casi como si muestreasemos un continuo. Probad con S=2 (equivalente a muestrear a 22050 Hz), S=5 (8820 Hz), S=10 (4410 Hz) y S=20 (2205 Hz).

```
>> [x fs]= wavread('vozam.wav'); sound(x,fs);
>> S=2; sound(x(1:S:end),fs/S);
```

Notad que para que suene adecuadamente debemos informar a *Matlab* que la secuencia submuestreada debe ser enviada a la tarjeta de sonido con un ritmo adecuadamente reducido, `fs/S`.

Determinar escuchando los resultados a partir de qué factor S los resultados se distorsionan apreciablemente. ¿Está relacionada la frecuencia obtenida con el ancho de banda de la señal original que vimos en la práctica anterior?

Como conclusión, siempre que se muestree una señal analógica o se submuestree una secuencia digital, hemos de evitar el *aliasing*. Para ello debe aplicarse **antes** de muestrear un filtro *antialiasing* consistente en un filtro paso bajo que elimine aquellas frecuencias por encima de la mitad de la frecuencia de muestreo que vayamos a usar. Por ejemplo, las cámaras digitales tienen una lámina justo antes del sensor que actua como filtro "antialiasing". Más cerca de nosotros, el sistema óptico del ojo tiene una frecuencia de corte ajustada a la malla de muestreo (conos) de la retina.

Diezmado de imágenes: aparición de "aliasing" en dos dimensiones

Una imagen `img` (matriz 2D en*Matlab*) podemos también diezmarla por un factor T usando:

```
img_diezmada=img([1:T:end],[1:T:end]);  % Muestrea una imagen img[][] por un factor T
```

Cargemos la imagen `'mill512.bmp'` de 512×512 píxeles y visualizemosla con diferentes resoluciones:

```
>> img=imread('mill512.bmp'); colormap(gray(256)); image(img); axis off; truesize;
>> S=2; img_red=img(1:S:end,1:S:end);  % Una muestra de cada S en ambas dimensiones
>> figure; colormap(gray(256)); image(img_red); axis off; truesize;
```

Probad con S=2 y S=3. Fijaos en la zona de la valla (abajo izquierda) y en la pared de ladrillo (arriba a la derecha). Comparar la imagen a máxima resolución con sus versiones a menor resolución. ¿Qué efectos aparecen? ¿Cómo lo interpretariais en términos de aliasing? ¿Por qué aparecen esos efectos precisamente en esas zonas?

Repetir el diezmado (S=2) pero sobre una versión pasobajo de la imagen obtenida por convolución con una máscara:

```
>> mask=[1 2 1; 2 4 2; 1 2 1]/16;
>> img2=conv2(double(img),mask,'same');
```

¿Han disminuido los efectos del aliasing en la imagen reducida?

Aunque como acabamos de ver, los efectos del aliasing son apreciables en imágenes "reales", dichos efectos serán más evidentes conforme las frecuencias de nuestra imagen original sean más altas. Vamos a ver su efecto en imágenes "sintéticas". Se trata de crear imágenes con una frecuencia lo más alta que permita nuestro dispositivo de visualización (en este caso nuestra pantalla) y visualizarlas para ver posibles efectos de aliasing.

Para ello vamos a crear una imagen formada por la frecuencia máxima que podemos presentar en nuestra pantalla. Como la unidad de muestreo es el pixel la frecuencia máxima es aquella cuyo periodo es 2 pixels (2 puntos de muestreo por periodo). Eso corresponde a una imagen con una raya negra (ancho 1 pixel) , una blanca (ancho 1 pixel), negra, blanca, etc. Cada raya negra + blanca (2 pixeles) corresponden a 1 periodo de nuestro patrón. El siguiente código crea y muestra dicha imagen (de tamaño 512x512 pixeles). El parámetro `th` corresponde al giro de las franjas negro/blanco (en 2D aparece la noción de dirección de una frecuencia, inexistente en 1D). Por defecto, para `th=0` tenemos una serie de franjas horizontales, una en cada pixel.

```
>> N=512; x= ones(N,1) * [0:N-1]; y=[0:N-1]' * ones(1,512);
>> th=0.0; im=127*(1+cos(pi*(sin(th)*x+cos(th)*y)));   % Genero imagen rayas inclinadas angulo th
>> image(im); colormap(gray(256)); truesize;
```

Ejecutar el código anterior tal como está escrito. Observar la imagen resultante. Hacer un zoom y verificar que efectivamente tenemos un patrón de rayas horizontales, 1 por pixel. Fijaos que estáis en el límite del Teorema de Muestreo. Usando el ratón hacer un resize de la ventana, haciendola más pequeña ¿Qué sucede? ¿Por qué?

Correr el mismo código pero ahora haciendo `th=0.01`. Esto corresponde a un giro casi despreciable de las líneas, que no debería notarse apenas. ¿Qué véis ahora? Describir la imagen y hacer un diagrama a la derecha de la imagen que observáis.

Hacer un zoom en uno de los puntos "conflictivos" ¿Qué veis?

Los efectos que hemos visto en estos ejemplos nos indican que aunque el límite teórico del Teorema de Muestreo sean 2 puntos de muestreo (píxeles en nuestro caso) por ciclo de la frecuencia más alta, no es conveniente apurar dicho límite teórico.

Aliasing en el muestreo temporal de una secuencia de imágenes

En los ejemplos anteriores hemos visto como muestrear señales de audio o imágenes a frecuencias de muestreo demasiado bajas puede presentar problemas.

Vamos a estudiar el *aliasing* en otra situación, el cine. Partimos de una señal $I(x, y, t)$ consistente en imágenes (x,y) que cambian con el tiempo t. En el cine dicha señal es muestreada en el tiempo generandose una sucesión de imágenes $\{I(x, y, t_n)\}$. Dicho muestreo en la variable temporal provoca uno de los efectos más visibles y por todos conocidos de "aliasing": el efecto "rueda-de-carreta-moviendose-hacia-atrás-en-película-de-indios". Este es un caso de "aliasing" provocado por un muestreo "inadecuado" en la secuencia de imágenes. "Inadecuado" quiere decir demasiado lento para los rápidos cambios (altas frecuencias) que tienen lugar cuando los colonos huyen de los indios. Durante la mayor parte del tiempo, las cosas se mueven mucho más lentamente entre frame y frame (frecuencias temporales bajas) por lo que la frecuencia usada es perfectamente adecuada.

Hemos escrito una pequeña GUI en *Matlab* que simula este efecto, llamada `rueda_gui()` usando una rueda de 8 radios. Dentro de la aplicación podemos seleccionar la velocidad de la carreta (en Km/h) y el ritmo de muestreo de la pelicula (en frames por segundo, por defecto 12, `dt=1/12`).

Al pulsar el botón de `Play` el programa hace girar la rueda (en el sentido de las agujas del reloj) a la velocidad especificada, pero solo muestra su aspecto cada dt segundos (en principio 12 veces por segundo, `dt=1/12` segundos). Se trata de correr dicho programa con diferentes velocidades de la carreta (con 12 frames por segundo) e indicar en la siguiente tabla si la rueda *parece* moverse hacia adelante (en sentido del reloj) o hacia atrás (contrario a las agujas del reloj). Recordar que la rueda **siempre** está moviendose en una dirección, se trata de que anoteis su dirección **aparente**:

Velocidad (km/h)	2	4	8	12	16	20	25	26	50
Dirección									

Existe una velocidad entre 20-30 Km/h para la cual la rueda se ve estacionaria. ¿Cuál es la condición matématica que provoca dicha estabilidad? ¿Es un fenomeno único o se repetirá para diferentes velocidades? ¿Cuáles?

Correr el programa para una velocidad de 16 km/h para 12 y 24 fps. ¿Se aprecia alguna diferencia en las direcciones? ¿Cúal es la correcta?

Ampliación de una secuencia: interpolación

Estudiaremos ahora el proceso inverso al submuestreo. Se trata de pasar de una secuencia $x[]$ con N muestras a otra $y[]$ con F×N muestras, conteniendo más o menos la misma información. Si la primera secuencia tenía asociada una frecuencia de muestreo f_m, a la nueva $y[]$ le corresponderá una frecuencia $F \times f_m$. En la nueva secuencia conoceremos 1 de cada F muestras y deberemos "inventarnos" los valores intermedios, que dependerán de los valores vecinos. Esto es, estamos ante un caso de interpolación.

Como sabemos, hay muchos tipos de interpolaciones posibles. La más sencilla es la llamada de "vecino más próximo" (orden 0) y consiste en replicar cada muestra conocida en los (F-1) huecos. Un poco más elaborado sería una interpolación "lineal" (orden 1), donde las muestras intermedias se disponen en la recta que une a los puntos ya conocidos. Por ejemplo, para el caso $F = 2$, donde la nueva señal $y[]$ tiene el doble de muestras que la original, ambos casos se expresarían como:

$$\text{Más próximo} \begin{cases} y[2k] = x[k] \\ y[2k+1] = x[k] \end{cases} \qquad \text{Lineal} \begin{cases} y[2k] = x[k] \\ y[2k+1] = (x[k] + x[k+1])/2 \end{cases} \qquad \text{para } k = 0 \ldots N-1.$$

Sería fácil implementar las fórmulas anteriores en *Matlab* . Sin embargo, vamos a ver un enfoque más general, que nos va a permitir entender el proceso de la interpolación desde una nueva perspectiva más relacionada con el procesado de señales.

La interpolación como inserción de ceros + convolución con una máscara

El problema de la interpolación puede subdividirse en dos partes:

1. Inserción de $(F-1)$ ceros entre las N muestras originales de `x[]` obteniendo una secuencia `x0[]` con $F \times N$ datos, de los cuales sólo N están inicializados. Esto se implementa en la función `x0=inser0(x,F)` que coge la secuencia `x[]` e inserta `F-1` ceros entre sus muestras.

2. Llenar los "huecos", mediante la convolución de la secuencia con ceros `x0[]` con otra secuencia `b[]`. Diferentes "máscaras de interpolación" darán lugar a los diferentes tipos de interpolación: La función `rellena(x0,b)` recibe una secuencia `x0[]` (insertada con ceros) y la convoluciona con una segunda secuencia `b[]`, dando como resultado la secuencia interpolada.

Veamos el caso de una ampliación por un factor 2, usando como mascara de convolución `b=[1 1]`:

```
>> t=pi*[0:0.1:0.9]; x=sin(t);    % Vector original con 10 muestras de sin(x).
>> x0 = inser0(x,2);              % Duplico muestras insertando ceros.
>> y=rellena(x0,[1 1]);           % Convoluciono con b=[1 1]
>> subplot(211); stem(x0,'r'); subplot(212); stem(y,'b');   % Pintamos los resultados
```

¿Qué tipo de interpolación de las dos mencionadas antes estamos haciendo? Dibujad unas pocas muestras de la señal con ceros `x0[]` y ver como al convolucionar con el filtro `b=[1 1]` que se va moviendo se obtiene dicho resultado.

¿Cual sería la máscara `b[]` a usar para un factor $F = 3$?

Repetir de nuevo para F=2 pero usando ahora `b=[0.5 1 0.5]` y pintar de nuevo los resultados. ¿Qué interpolación tenemos ahora?

Notad que ambos `b[]` usados tienen algo en común, ambos son filtros pasobajo que promedian las muestras de la señal original. Esta es nuestra nueva interpretación de la interpolación desde el punto de vista del tratamiento de señal: **Interpolación = Inserción de ceros + Filtro Pasobajo**
En realidad para ser propiamente filtros paso-bajo deberían cumplir que la suma de sus muestras fuese 1. Lo que ocurre es que para que la energía media de la señal final sea similar a la de la original hay que hacer una magnificación por un factor F para compensar el "bajón" de energía media que supuso la inserción de ceros. Con este enfoque las mascaras de convolución usadas podrían escribirse como: `b = 2 * [0.5 0.5]` o `b = 2 * [0.25 0.5 0.25]`, separando de esta forma la magnificación ($\times 2$) del filtrado paso-bajo ($\sum b[i] = 1$) propiamente dicho.

Interpolación = Inserción de ceros + Magnificación factor F + Filtro Pasobajo

Si teneis audio, podeis oir y comparar las dos tipos de interpolaciones aplicadas a una señal de voz:

```
>> [x fs]=wavread('audio.wav');
>> b=[1 1]; x0=inser0(x,2) y=rellena(x0,b);  % Probar también con b=[0.5 1 0.5]
>> sound(x,fs);          % Original
>> sound(y,2*fs);        % Interpolada
```

Notad que al aumentar el número de muestras si queremos que siga sonando "igual" debemos aumentar tambien el ritmo con el que las mandamos a la tarjeta de sonido (`2*fs` en vez de `fs`).
¿Cuál de las versiones interpoladas se oye "mejor"?

Intentemos ahora entender por qué una inserción de ceros seguido de un paso bajo es equivalente a una interpolación de nuestra secuencia. Consideremos cual debería ser el comportamiento de un interpolador ideal. Nuestra secuencia original `x[]` está muestreada a una cierta frecuencia f_m y por lo tanto la frecuencia máxima de su espectro corresponde a la frecuencia de Nyquist, $f_m/2$. ¿Cómo debería ser el espectro de la señal ampliada con un factor F=2? La nueva señal tiene un ritmo de muestreo doble, $2f_m$, y la nueva frecuencia de Nyquist es $(2f_m)/2 = f_m$. Por lo tanto en el espectro de la nueva señal hay espacio para nuevas frecuencias no presentes en la señal original (desde $f_m/2$ hasta f_m). La pregunta es, ¿qué poner en esas nuevas frecuencias para las cuales no teníamos información en la secuencia original? Lo más sencillo sería dejarlas a 0. Por lo tanto la TF de una interpolación "ideal" deberia tener los mismo valores que la de la señal original en $[0, f_m/2]$ y valer cero en frecuencias superiores (desde $f_m/2$ hasta la nueva frecuencia de Nyquist $(F * f_m)/2$). ¿Donde encajamos en este esquema la inserción de ceros y el filtrado pasobajo? Veamos las TF de las diferentes etapas. Empecemos con la TF de la secuencia con ceros comparandola con la original.

```
>> [x fs]=wavread('gil2.wav',4000);   % Leemos 4000 muestras del fichero gil2.wav
>> F=2; x2=inser0(x,F);
>> ver_tf(x,fs,'r'); hold on; ver_tf(x2,F*fs,'b'); hold off
```

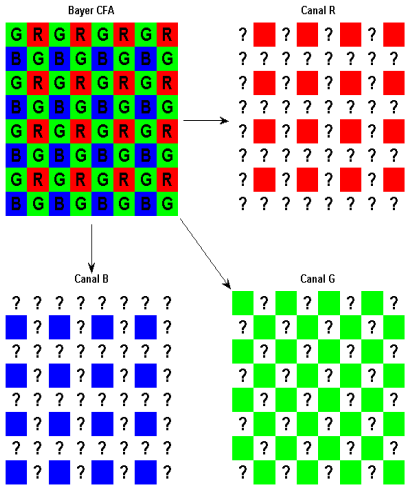
Dibujad la TF original (izda) y la de la secuencia con ceros insertados (derecha).

Repetir para F=3. ¿Cúal es el efecto de introducir ceros en una secuencia desde el punto de vista de su transformada de Fourier? ¿En qué se diferencia esa TF de la que debe ser la TF de una interpolación ideal? ¿Cuál es el objeto de aplicar un filtro pasobajo a la secuencia con ceros?

ENTREGA OPCIONAL: interpolación en cámaras digitales

Es bien sabido que para una imagen en color necesitamos especificar una tripleta de colores (RGB) para cada pixel (posición espacial). Sin embargo, por razones técnicas, la inmensa mayoría de los sensores de cámaras digitales son monocromos. Para capturar la información de color, delante del sensor hay un filtro que consiste en una reticula de filtros de colores (CFA, Color Filter Array) verde, rojo y azules, que hace que ciertos píxeles del sensor capten información sobre la componente verde de la imagen, otros sobre la roja, etc. Lo importante es darse cuenta de que dicho sensor solo nos da una componente (R,G ó B) por pixel. Las demás tendremos que inventarnoslas (interpolarlas).

Uno de las configuraciones más usadas de CFAs en las cámaras digitales es el llamado filtro de Bayer, donde la disposición espacial de los filtros de color se refleja en la imagen adjunta. Como se observa, la mitad de los pixeles del sensor dan información del canal verde (G), una cuarta parte del canal rojo (R), y la otra cuarta parte del canal azul(B). La mayor importancia asignada al verde se debe a que es la longitud de onda a la que el ojo humano es más sensible. Una vez que tratamos de reconstruir los tres planos (RGB) de la imagen en color nos encontramos con que dos terceras partes de los valores deben ser interpolados.



Para estudiar la reconstrucción de los datos de un sensor de Bayer, hemos escrito una rutina denominada `rgb2bayer(img)`, que recibe una imagen RGB (matriz de Alto×Ancho×3) y devuelve la misma imagen RGB, pero en la que se han puesto a `NaN` (Not a Number) los valores no captados por un filtro de Bayer (la mitad de los verdes y tres cuartas partes de los rojos y azules).

```
>> img=imread('flowers.tif'); bayer=rgb2bayer(img);
```

En una imagen RGB, podemos acceder al canal rojo como `img(:,:,1)`, al verde como `img(:,:,2)`, etc. Vamos a examinar 5×5 pixeles de los tres canales del sensor de Bayer, ilustrando las pérdidas que se producen.

```
>>  bayer([1:5],[1:5],1), bayer([1:5],[1:5],2), bayer([1:5],[1:5],3),
```

Comparemos ahora visualmente la imagen original y la muestreada:

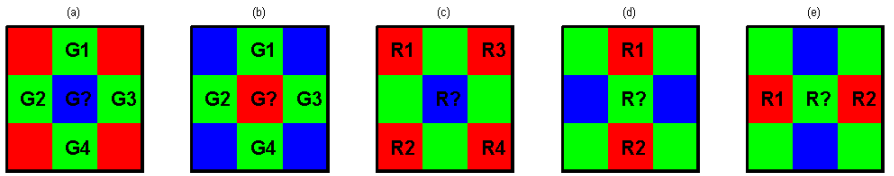
```
>> subplot(211); image(img); subplot(212); image(uint8(bayer)); truesize;
```

Hacer un zoom (seleccionar zoom en el menú) en una zona del sensor de Bayer que sea blanca en la imagen original. ¿Qué veis? ¿Por qué? ¿Y si haceis zoom sobre una flor de color amarillo?

Se trata ahora de interpolar los valores que faltan (NaN) de cada canal. Hemos visto en el apartado anterior, que la interpolación lineal es una técnica sencilla que da resultados aceptables. En el caso de imágenes recibe el nombre de **bilinear** (2D) y la aplicaremos independientemente a cada canal. Las fórmulas de la interpolación bilinear aplicadas al problema anterior se resumen en las siguientes reglas (distintas para el canal verde frente al rojo/azul, por sus diferentes densidades de muestreo):

∗ **Canal Verde:** en las dos situaciones posibles (etiquetadas (a) y (b) en la figura adjunta), tenemos cuatro valores adyacentes conocidos (arriba, abajo, derecha, izquierda). El valor del canal verde en el pixel central se calcula como la media de esos cuatro vecinos.

∗ **Canal Rojo/Azul:** podemos encontrarnos tres casos: dos vecinos arriba y abajo (d), dos vecinos a los lados (e) o cuatro vecinos en las esquinas (c). En los dos primeros el valor a usar es la media de los dos vecinos y en el tercero la media de los cuatro. El caso del canal azul es idéntico al rojo.



Como en el caso 1D una forma eficiente de aplicar la interpolación es a través de una convolución con una máscara pasobajo (aquí los ceros ya están insertados). En el caso del canal rojo y azul, la máscara es la representada a la derecha.

1/4	1/2	1/4
1/2	1	1/2
1/4	1/2	1/4

Justificar cómo la convolución con la máscara anterior da lugar a las reglas establecidas para la interpolación de los canales rojo y azul. Para ello posicionarla en las distintas posibilidades de los pixeles rojos en la figura anterior y comprobar como los resultados se ajustan a las reglas establecidas.

Usando el razonamiento anterior, deducir ahora los valores que debe tener la máscara de convolución a usar en la interpolación del canal verde, y rellenarlos en la tabla adyacente.

Vamos ahora a ver los resultados. Llamemos `img_out` a la imagen reconstituida.

```
>> img_out=bayer;     % Inicializamos imagen de salida con los datos del sensor de Bayer
>> img_out(isnan(bayer))=0.0;       % Ponemos a ceros los valores missing (NaN en bayer)
>> mask_rb = [0.25 0.5 0.25; 0.5 1 0.5; 0.25 0.5 0.25],    % Mascara 3x3 canal rojo/azul
>> mask_g  = [Valores deducidos en la pregunta anterior]   % Mascara 3x3 para canal verde
>> img_out(:,:,1)=conv2(img_out(:,:,1),mask_rb,'same');    % Convolucion canal rojo  (Rgb)
>> img_out(:,:,2)=conv2(img_out(:,:,2),mask_g,'same');     % Convolucion canal verde (rGb)
>> img_out(:,:,3)=conv2(img_out(:,:,3),mask_rb,'same');    % Convolucion canal azul  (rgB)
>> img_out(img_out>255)=255; img_out(img_out<0)=0;         % Hacemos clip a [0 255]
>> img_out=uint8(img_out);                                 % Convertimos a bytes (uint8)
```

Idealmente, la nueva imagen `img_out` debería ser lo más parecida posible a la original `img`. Para cuantificar la diferencia podemos restar ambas imágenes y calcular la desviación standard o la media de las discrepancias (en valor absoluto).


```
>> dif=double(img)-double(img_out);
>> fprintf('%.1f %.1f\n',std(dif(:)), mean(abs(dif(:)))); end
```

Desviación σ	Mean $\lvert E_{ij}\rvert$

Para comparar las diferencias visuales, mostremos las dos imágenes una al lado de la otra, mediate los comandos: `>> subplot(211); image(img); subplot(212); image(img_out);`
Hacer un zoom en ambos casos en la zona de la margarita que hay en la parte inferior de la imagen. ¿Qué efecto se observa en los bordes de la margarita en la imagen reconstruida? ¿A qué es debido?

Otro zona donde se aprecian claras diferencias visuales es en la línea blanca en la zona inferior de la imagen original. ¿Sigue siendo blanca en la reconsrucción?

Se observa que la simple interpolación lineal es demasiado simple para dar una buena imagen RGB a partir del mosaico de Bayer. De hecho, los problemas serían más evidentes con una imagen más "complicada". Repetir la recuperación con una imagen más dificil, `'faro.jpg'`. Comentar los resultados con la nueva imagen. ¿Por qué la nueva imagen es más complicada de recuperar?