Imprimir
Note: You can choose to save the module's content as a PDF document, instead of printing.
To do so, click on the 'Print' link above. In the window that appears, select the PDF
printer as the printer of choice, then click 'Print' and enter any additional information needed.

Tratamiento de señales

Introducción

Instructions: Seguir los enlaces y realizar las actividades propuestas.

Introducción

En este capítulo nos centramos en algunos aspectos de tratamiento de señales unidimensionales. Empezamos con cuestiones básicas de generación y representación de señales. A continuación vemos un método sencillo para detección de patrones (subseñales) en una determinada señal. En las restantes secciones utilizaremos la transformada wavelet discreta. En primer lugar damos la más sencilla de ellas, que es la transformada de Haar. Las dos últimas secciones muestran métodos eficientes para la compresión de señales y eliminación del ruido, respectivamente, basados en la transformada wavelet.

Detección de patrones, transformada wavelet, compresión y reducción del ruido

---

Representación de señales

Todas las señales que analicemos estarán representadas como vectores ∟fila o columna, pero se recomienda que estén siempre en columna. Estas

señales pueden proceder de aparatos de medida, representación de funciones, etc. Naturalmente antes de analizar la señal hay que preparar los datos y cambiarlos a un formato que sea apropiado para MATLAB. A continuación veremos algunos casos concretos que suelen ser habituales.

Datos en un fi└chero ASCII

Supongamos que nos llegan los valores numéricos en ASCII de una señal que deseamos tratar en un fi└chero llamado signal.txt, es decir, el fi└chero signal.txt tiene 1024 datos ordenados en una columna y cada └la corresponde a un dato. Prácticamente lo mismo que digamos para este tipo de datos vale también para └ficheros de sonido en formato *.wav. Eeste tipo de señales se pueden cargar directamente, con el botón derecho del ratón aparece la opción ┼"Import Data". Si inspeccionamos ahora el espacio de trabajo de MATLAB veremos que ha creado una variable llamada signal con los valores del fi└chero. Además es un vector columna y el formato de cada valor es de doble precisión. Si lo deseamos ahora podemos renombrar esa variable. Por ejemplo para duplicarla con nombre s escribiremos

```
>> s = signal;
```

Recordamos que, una vez los datos son un vector, representarlos grá└ficamente es sencillo con la orden plot.

```
>> plot(s);
```

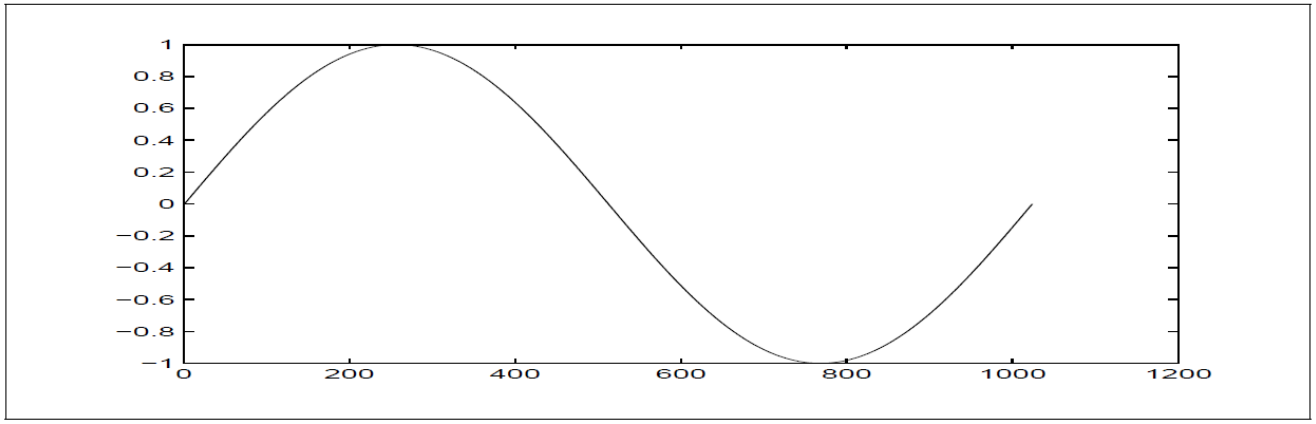

Figura 1: Un ejemplo de `plot()`.

Datos creados a partir de una función

Algunas veces puede ser conveniente ┼fabricar◄ la señal a partir de una función dada (para docencia, pruebas, ejemplos, simulaciones, etc). Para ello evaluaremos la función en ciertos puntos y con el resultado construiremos la señal. Por ejemplo, para generar una señal con 1024 datos evaluados con la función

$$g(x) = 20x^2(1-x)^4\cos(12\pi x)$$

en el intervalo $[0,1]$ utilizaremos un método conocido como "vectorization":

```
>> x = 0:1/1023:1;
signal = (20*x.^2).*((1-x).^4).*cos(12*pi*x);
```

Recordemos aquí la diferencia entre el operador multiplicación * y el ┼punto multiplicación◄ .* es importante. Cuando MATLAB lee * hace una multiplicación usual de matrices (incluido el caso de escalar por matriz); cuando utilizamos .* el programa realiza un producto de matrices o vectores componente a componente. Esta misma regla se aplica a las potencias ^ y .^. Al representarla, la porción del plano XY que queremos representar se ┗fija con la orden axis().

```
>> plot(signal);  axis([0 1024 -0.5 0.5]);
```

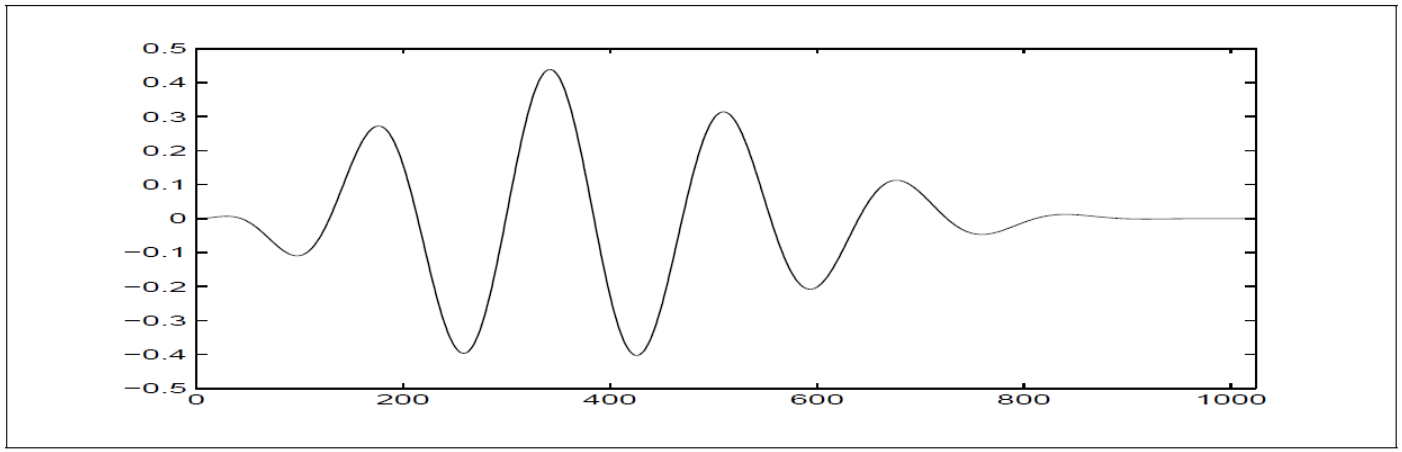

Figura 2: Señal creada a partir de la función $g(x)$.

Detección de patrones en señales

Si pretendemos detectar un patrón o subseñal x en una señal y podemos utilizar la convolución de señales, en el modo adecuado, y siempre que se cumpla que la energía de x (i.e., el cuadrado de su norma) es no inferior a la energía de cualquier subseñal de y de la misma longitud que x. Es preciso observar que en la convolución las entradas de las señales †corren◀ en sentido contrario, por ello es preciso invertir el orden de las entradas en x. A modo de ejemplo, tomamos como y una señal con 4096 datos evaluados con la función

$$f(t) = 20t^2(2-t)^5(4-t)^2\cos(12\pi t) + 30t^2(2-t)^2(4-t)^5\cos(24\pi t)$$

en el intervalo $[0;4]$, y como x la subseñal comprendida entre las entradas 500 y 999, inclusive, ambas en columna:

```
>> t=0:4/4095:4;
y=(20*t.^2).*((2-t).^5).*((4-t).^2).*cos(12*pi*t)+(30*t.^2).*((2-t).^2).*((4-t).^5).*cos(24*pi*t);
y=y';
x=y(500:999);
```

Para invertir el orden de las entradas utilizamos el comando flipud, al ser un vector columna:

```
>> x = flipud(x);
```

Ahora podemos realizar la convolución de ambas señales, a pesar de tener distinta longitud, cuyo resultado es una señal de longitud $L+l-1$, donde l es la longitud de x y L es la longitud de y. El bloque que nos interesa es el comprendido entre las entradas l y L, que expresa el producto escalar de la subseñal elegida de y con cada bloque de la misma longitud, desde el principio, desplazándose una posición en cada paso, hasta que la última entrada de la subseñal llega a multiplicar a la última entrada de y.

```
>> z = conv(y,x);
l = length(x);
z = z(l:end-l+1);
```

Por último queda normalizar la señal generada, de manera que si se cumple nuestra hipótesis obtengamos una señal cuyo máximo es 1, y que se alcanza justo en la posición donde empieza el patrón de la subseñal:

```
>> u = z/norm(x)^2;
find(u>0.999)
```

Donde obtenemos como resultado la entrada 500, que es donde empezaba el patrón. De forma alternativa también se puede utilizar la convolución circular con el comando cconv, obteniendo el mismo resultado si se realiza adecuadamente. A continuación se indica una función que realiza lo anterior (llamémosle al fichero detpat.m):

```
function [p] = detpat(x,y)
% Indica la posición o posiciones (entrada inicial) donde se
```

```
% encuentra el patrón o subseñal x en y.
% Precisa que la energía de x sea no inferior a la energía
% de cualquier otra subseñal de y de la misma longitud.
% Ambas señales deben estar en columna.
x = flipud(x);
z = conv(y,x);
l = length(x);
z = z(l:end-l+1);
u = z/norm(x)^2;
p = find(u>0.999)
```

La transformada de Haar

La transformada de Haar es una herramienta para el tratamiento de señales. Tiene asignado un nombre (de familia wavelet) en MATLAB: Wavelet de Haar: `haar'. Podemos obtener información concreta con el comando waveinfo en la línea de comandos:

```
>> waveinfo('haar');
```

Es lo que se conoce como una transformada ortogonal, transformada lineal que preserva el producto escalar, y por tanto la norma o, equivalentemente, la energía de las señales. Por otra parte, una de las características que la hacen muy interesante es el hecho de que su transformada inversa consiste, matricialmente, en multiplicar por su traspuesta, ya que la inversa de una matriz ortogonal es su traspuesta. Otro aspecto a destacar es el conocido como compactación de la energía: Por ejemplo, a nivel 1, tras realizar la transformada hemos "redistribuido" la enrgía de manera que está la mayor parte en la primera mitad de la transformada. Esta primera mitad viene a ser una representación, reescalada, de la señal original. Los detalles teóricos, que solo precisan de un conocimiento básico de álgebra lineal, los podeis encontrar en el enlace al material, que forma parte de un libro de los profesores del curso, haciendo clic aquí. Como ejemplo básico, supongamos que nuestra señal es

$$soriginal = (3, 5, 4, 7, 1, 9, 3, 2)$$

En un primer nivel, la transformada de Haar actuaría de manera que en la primera mitad aparecerían unos promedios ponderados (subseñal de aproximación o tendencia), y en la segunda mitad unas diferencias ponderadas (subseñal de detalles o fluctuaciones):

$$stransformada = \frac{1}{\sqrt{2}}(8, 11, 10, 5, -2, -3, -8, 1)$$

Es decir, por pares, la primera mitad son los promedios aritméticos, multiplicados por $\sqrt{2}$, y la segunda las diferencias.

Vamos a utilizar los comandos básicos de la transformada de Haar, calculando las subseñales de aproximación y de detalle, tanto a nivel 1 como a niveles superiores, su representación gráfica y la reconstrucción de la señal a partir de los coeficientes. Los comandos básicos para el análisis con transformada son:

- **dwt / idwt:** Descomposición/ reconstrucción de un único nivel;
- **wavedec / waverec**: Descomposición/ reconstrucción de cualquier nivel;
- **detcoef / appcoef**: Extracción de los coeficientes de detalles y aproximación;
- **wmaxlev:** Descomposición del máximo nivel;
- **wrcoef**: reconstrucción selectiva.

Haremos un ejemplo paso a paso. Tomaremos la función anterior $g(x)$ muestreada con 2048 puntos en el intervalo $[0, 1]$. Para obtener los coecientes wavelet a nivel uno se utiliza el comando:

```
>> [C,L] = wavedec(signal,1,'haar');
```

La función wavedec calcula los coeficientes wavelet realizando los productos escalares de la señal con las wavelets y las scaling de nivel 1 y almacena los valores de la aproximación y detalles en un solo vector llamado C; el vector L contiene las longitudes. La orden necesita tres argumentos, el primero es la señal, el segundo los niveles de descomposición y el tercero es la wavelet a utilizar (en nuestro caso, Haar).

```
>> a1 = appcoef(C,L,'haar',1);
d1 = detcoef(C,L,1);
```

Para representar gráficamente los valores se usa la orden plot. Por ejemplo, para representar las subseñales de aproximación y detalles escribiremos

```
>> plot(a1); axis([1 length(a1) -1 1]);
plot(d1); axis([1 length(d1) -0.25 0.25]);
```

Como ya conocemos, para que ambas gráficas sean representadas en una misma ventana se usa subplot como se muestra a continuación.

```
>> subplot(1,2,1);
plot(a1);
title('Aproximación');
axis([1 length(a1) -1 1]);
subplot(1,2,2);
plot(d1);
title('Detalles');
axis([1 length(d1) -0.25 0.25]);
```

Para construir la primera señal promedio y la primera señal de detalle del análisis de multirresolución debemos usar estos valores como coeficientes en las bases de wavelets y de scaling. Para esta tarea se dispone del comando wrcoef. Acepta cinco argumentos: el primero indica si queremos aproximaciones o detalles, el segundo y tercero son los coeficientes y las longitudes de las subseñales que produjo la orden wavedec, el cuarto es la

wavelet a utilizar y por último el nivel de aproximación o detalle que deseamos calcular. Finalmente representaremos las dos señales en una única ventana.

```
>> ls = length(signal);
A1 = wrcoef('a',C,L,'haar',1);
D1 = wrcoef('d',C,L,'haar',1); subplot(1,2,1);
plot(A1);
title('Aproximación');
axis([1 ls -1 1]);
subplot(1,2,2);
plot(D1);
title('Detalle');
axis([1 ls -0.1 0.1]);
```

Todo lo anterior sirve para otras familias de wavelets ortogonales. En el enlace anterior a los apuntes de teoría se puede consultar información acerca de otras familias de wavelets ortogonales. También se puede utilizar el comando waveinfo en MATLAB. Por ejemplo, para las ortogonales que utilizaremos, waveinfo('haar'), waveinfo('db') y waveinfo('coif').

Compresión de señales

El material que fundamenta las dos últimas secciones de este tema se basa en un capítulo del libro de los profesores del curso, cuyo enlace está aqui.

Dos de los comandos básicos de MATLAB para compresion de señales unidimensionales son:

- wthresh Aplica un umbral (’hard’ o ’soft’) a la señal que aparezca en el argumento.
- wthcoef Aplica un umbral (’hard’ o ’soft’) a los coeficientes de la transformada wavelet.

Se pretende comprimir una señal, pero sólo desde el punto de la transformada wavelet, es decir, calculando el número de coeficientes de la transformada que necesitamos conservar para mantener un porcentaje de la energía de la señal alto, de manera que la señal comprimida no pierda demasiada calidad respecto de la señal original. El esquema básico a seguir es:

1. Cálculo de la transformada wavelet de la señal (hay que decidir qué tipo de wavelet y qué nivel de transformada son los más convenientes).

2. Cálculo del valor umbral que vamos a aplicar a la transformada. Éste se calcula teniendo en cuenta el porcentaje de energía de la señal que queremos conservar.

3. Umbralizado de la transformada. Se anulan los valores por debajo del umbral. Esta transformada wavelet modificada es la que tomaremos como señal comprimida.

4. Cálculo del ratio de compresión teniendo en cuenta el número de valores no nulos de la transformada después de aplicarle el umbral.

5. Cálculo de la transformada wavelet inversa de la transformada umbralizada. La señal resultante es la versión "descomprimida" de la señal original.

6. Cálculo del error cometido, como el error cuadrático medio entre la señal original y la reconstruída. También se pueden comparar las señales visualmente, acústicamente....

Para ver el proceso, tomamos una señal de 2^{14} datos muestreando la siguiente función en el intervalo $[0,1]$.

$$f(x) = 20x^2(1-x)^2\cos(64\pi x) + 30x^2(1-x)^4\sin(30\pi x)$$

```
>> x=0:1/(2^14-1):1;
s=(20*x.^2).*((1-x).^2).*cos(64*pi*x)+(30*x.^2).*((1-x).^4).*sin(30*pi*x);
```

A continuación realizamos la descomposición wavelet a nivel 10, utilizando la wavelet 'coif5':

```
>> [C,L] = wavedec(s,10,'coif5');
```

En el siguiente paso reordenamos los valores absolutos de la transformada de forma decreciente:

```
>> C_dec = abs(sort(-abs(C)));
```

Pretendemos conservar el 99.99% de la energía de la señal. Para ello necesitaremos disponer del perfil de energía acumulada. Para facilitar las cosas, dado que es un aspecto que utilizaremos más veces, ponemos a continuación el fichero cumenergy.m, que se deberá poner en el directorio de trabajo:

```
function [y] = cumenergy(x)
%Calcula el perfil de energía acumulada normalizado

y = cumsum(x.^2)/(norm(x)^2);
```

Ahora buscamos el indice hasta acumular un porcentaje de energía especificado. Una forma sencilla es calcular el número de índices que sobran:

```
>> ind_sobran = find(cumenergy(C_dec)>=0.9999);
```

Por tanto, el valor correspondiente al primer indice que sobra es el umbral:

```
>> umbral = C_dec(ind_sobran(1));
```

Anulamos valores de la transformada que no pasan el umbral. Utilizaremos el método hard ('h'):

```
>> C_sig = wthresh(C,'h',umbral);
```

Recomponemos la señal:

```
>> s_rec = waverec(C_sig,L,'coif5');
```

Para una comparación visual, podemos realizar la gráfica de las dos señales:

```
>> figure;
subplot(2,1,1); plot(s);
axis([1 2^14 min(s) max(s)]); title('original');
subplot(2,1,2); plot(s_rec);
axis([1 2^14 min(s_rec) max(s_rec)]); title('reconstruccion');
```

Lo lógico es conocer algunos datos sobre la compresión, como por ejemplo cual es el factor de compresión. Para ello necesitamos saber cuantos coeficientes sobrevivieron tras realizar la transformada y el umbralizado. Eso lo podemos saber conociendo el número de unos en el mapa de significancia:

```
>> map = C_sig~=0;
val_sig = sum(map);
```

El ratio de compresión, y su redondeo, se calculan ahora:

```
>> [comp,err] = sprintf('%d:%d',2^14,val_sig);
[comp_aprox,err] = sprintf('%d:%d',round(2^14/val_sig),1);
```

Anteriormente hemos podido comparar visualmente la señal original y la reconstruida tras la compresión. Para dar un valor cuantitativo, podemos utilizar el error RMS entre la señal original y la reconstruida. Necesitaremos el fichero rms.m, que también pondremos en el directorio de trabajo:

```
function [e] = rms(x,y)
%Calcula el error RMS entre dos señales.

e = sqrt(norm(x-y)^2/length(x));
```

Utilizamos por tanto la función anterior para el cálculo de error RMS:

```
>> error = rms(s,s_rec);
```

Para una visualización completa del resumen de resultados (que, de hecho, están en el espacio de trabajo), podemos utilizar lo siguiente:

```
>> [l_long_orig,err] = sprintf('Longitud original: %d \n',2^14);
[l_val_sig,err] = sprintf('Valores significativos: %d \n',val_sig);
[l_comp,err] = sprintf('Factor compresion de %s \n',comp);
[l_comp_aprox,err] = sprintf('\t (aproximadamente de %s) \n',comp_aprox);
[l_rms,err] = sprintf('Error RMS: %d \n',error);
sprintf('%s%s%s%s%s',l_long_orig,l_val_sig,l_comp,l_comp_aprox,l_rms)
```

Reducción del ruido en señales

En esta sección vamos a utilizar los mismos comandos básicos de MATLAB que ya utilizamos en la sección anterior. El proceso básico para reducir el ruido de la señal consta de las siguientes etapas:

1. Descomposición: Se elige un tipo de wavelet, un determinado nivel de descomposición y se realiza la transformada wavelet de la de la señal.

2. Umbralizado de los coeficientes de fluctuación: Se elige un umbral y se aplica a la transformada. Hay diferentes posibilidades: umbralizado estricto (hard thresholding) o umbralizado suave (soft thresholding), global o dependiente del nivel, etc. También existen multiples criterios para seleccionar el umbral.

3. Reconstrucción: Se calcula la transformada wavelet inversa de la transformada umbralizada.

Este proceso también lo ilustraremos con un ejemplo. Construimos una señal de 2^{11} puntos a partir de la función

$$f(x) = 4x^2(1-x)^3(2-x)^2\cos(18x(1+x))$$

en el intervalo $[0,2]$:

```
>> x = 0:2/(2^11-1):2;
s = 4*x.^2.*(1-x).^3.*(2-x).^2.*cos(18*x.*(1+x));
```

Ahora le añadimos un ruido blanco gaussiano con desviación típica 0.1:

```
>> s = s + 0.1*randn(1,2^11);
```

Podemos dibujar la gráfica de esta señal para comprobar el resultado visual tras añadirle el ruido blanco. Sobre esta señal aplicamos la transformada wavelet. Como en la sección anterior, utilizamos la wavelet 'coif5'. Fijamos el nivel en 9:

```
>> [C,L] = wavedec(s,9,'coif5');
```

A diferencia de la sección anterior, la selección del umbral tiene ahora el objetivo de eliminar el ruido, y este se basará en la fórmula de Donoho y Johnstone, que precisa estimar la desviación típica del ruido, la cual podemos obtener aproximadamente a través de la subseñal de detalles de primer nivel:

```
>> d1 = detcoef(C,L,1);
des_tip = std(d1);
umbral = sqrt(2*log(length(s)))*des_tip;
```

Ahora podemos aplicar el umbralizado a la transformada. Seleccionamos, por ejemplo, el método hard. A continuación reconstruimos la señal:

```
>> C_sig = wthresh(C,'h',umbral);

s_den = waverec(C_sig,L,'coif5');
```

Finalmente podemos obtener los resultados y una representación gráfica:

```
>> figure; hold on; axis([1 length(s) min(s) max(s)]); plot(s,'y');
plot(s_den,'k'); hold off;
error = rms(s,s_den);
[error_men,err] = sprintf('Error: %d \n',error);
ene_ret = 100*energy(s_den)/energy(s);
[ene_ret_men,err] = sprintf('Energía retenida: %2.3f %% \n',ene_ret);
sprintf('%s%s',error_men,ene_ret_men)
```

Como complemento a los comandos de las dos últimas secciones, se pueden consultar en MATLAB los siguientes, más avanzados y que ofrecen un proceso más automatizado:

- wdencmp Función básica de compresión y reducción de ruido con wavelets.
- ddencmp Proporciona los valores por defecto para compresión y reducción de ruido con wavelets.
- wden Reducción de ruido de forma automática.
- thselect Calcula diferentes tipos de umbral (ver también wthrmngr).

Actividades propuestas

Actividad 1:

Consideremos la señal y de 2^{12} puntos generada a partir de la función

$$f(x) = 4x^2(1-x)^3(2-x)^2\cos(18x(1+x))$$

en el intervalo $[0, 2]$. Tomamos la subseñal x comprendida entre las entradas 3200 y 3700. Utilizad el proceso de detección de patrones para ubicar que, efectivamente, encontramos dicha subseñal como patrón en la posición 3200.

Actividad 2:

Construid una señal de 2^{12} puntos a partir de la función

$$f(t) = 20t^2(2-t)^5(4-t)^2\cos(12\pi t) + 30t^2(2-t)^2(4-t)^5\cos(24\pi t)$$

en el intervalo $[0, 4]$. Aplicad el proceso completo de la sección de compresión de señales, incluyendo representación gráfica y obtención de resultados.

Actividad 3:

Consideremos la señal y de 2^{11} puntos generada a partir de la función

$$f(x) = 20x^2(1-x)^2\cos(64\pi x) + 30x^2(1-x)^4\sin(30\pi x)$$

en el intervalo $[0, 1]$. A dicha señal añadid un ruido blanco gaussiano con desviación típica 0.15, y posteriormente aplicad el proceso completo de eliminación de ruido descrito en la sección correspondiente.

Bibliografía

F. Martínez Giménez, A. Peris y F. Rodenas, Tratamiento de señales digitales mediante wavelets y su uso con Matlab, Editorial ECU.