Práctica 2. Modelado de la señal de voz como un proceso aleatorio y su aplicación en la transmisión sobre un canal con pérdidas (modelo 1)

Dada una señal de voz digitalizada, con una frecuencia de muestreo de 8kHz representada por una profundidad 8 bits, esta se transmite sobre un canal de transmisión con pérdidas.

El objetivo de la práctica es modelar la señal de forma que podamos estimar el valor de las muestras de voz que no han llegado al receptor, y hacer uso de la estimación para reconstruir una señal.

Nos dan 8 señales sobre las que trabajar.

```
frequency = 8000; % 8kHz
quasicero = 1e-15
```

```
quasicero =
1.0000e-15
```

Añado el parámetro quasicero, un valor muy pequeño con el que voy a sustuir las probabilidades cero.

PRIMER MODELO

DESCRIPCIÓN

Lo primero que tenemos que hacer sí o sí es construir una base de datos que contenga la información de todas las señales que nos han ofrecido.

Las señales nos la dan en un archivo .wav, así que haremos uso de la función **audioread()** para acceder a su información.

```
[s1] = audioread('df1_cln.wav');
[s2] = audioread('df2_cln.wav');
[s3] = audioread('df3_cln.wav');
[s4] = audioread('dm1_cln.wav');
[s5] = audioread('dm1_cln.wav');
```

Más información sobre la sintaxis en https://es.mathworks.com/help/matlab/ref/audioread.html.

La función *audioread* nos ha devuelto un vector conteniendo todos los que contenía el archivo .wav sin tener en cuenta la profundidad (véase la nota siguiente). Sabiendo que la señal está codificada a 8 bits, el conjunto de posibles valores que puede tomar cada muestra de la señal se encuentra en el intervalo [-128, 127] (son 256 números, el máximo representable con 8 bits). Entonces, multiplicando cada muestra por 128, logramos decodificar los datos.

Nota: *si ha comprobado la documentación, si especifica el dataType a native dentro de la función audioread, la información ya estaría interpretada teniendo en cuenta la profundidad real que incluye el archivo. Sin embargo, en nuestro caso no nos resulta relevante, pues la señal ha sido guardada con una profundidad de 16 bits, aunque sabemos de antemano que su codificación real es de 8 bits. Por tanto, no especificamos el parametro dataType porque queremos que audioread nos devuelva los datos sin decodificar.*

```
s1 % se coloca para que en Live Editor se muestre la información que ha devuelto audioread()
```

```
s1 = 16998×1
    0.0013
    0.0019
    0.0004
    0.0006
    0.0013
         0
   -0.0014
   -0.0013
   -0.0006
   -0.0013
   -0.0018
   -0.0009
   -0.0003
   -0.0009
   -0.0021
     ⋮
     ⋮
```

```
s1=double(int8(s1*256));
s2=double(int8(s2*256));
s3=double(int8(s3*256));
s4=double(int8(s4*256));
s5=double(int8(s5*256));
```

Ahora, construyamos la base de datos con todos los datos obtenidos de las señales que nos han ofrecido. Para ello, concatenaremos las matrices de datos de las señales.

Para concatenar los vectores, recurrimos al operador punto y coma, podemos disponer las matrices una debajo de otra.

Nota: *Véase https://es.mathworks.com/help/matlab/math/creating-and-concatenating-matrices.html#OverviewCreatingAndConcatenatingExample-3 para obtener más información sobre la concatenación de matrices.*

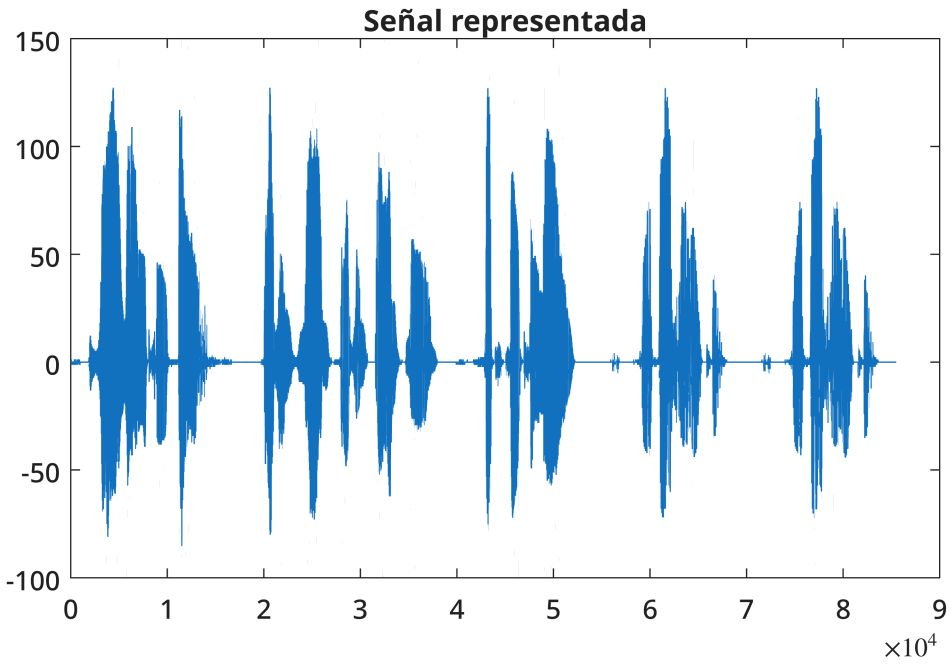
```
S = [s1;s2;s3;s4;s5]
```

```
S = 85502×1
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
    -1
```

⋮

Con fines ilustrativos, representaremos la información en una gráfica:

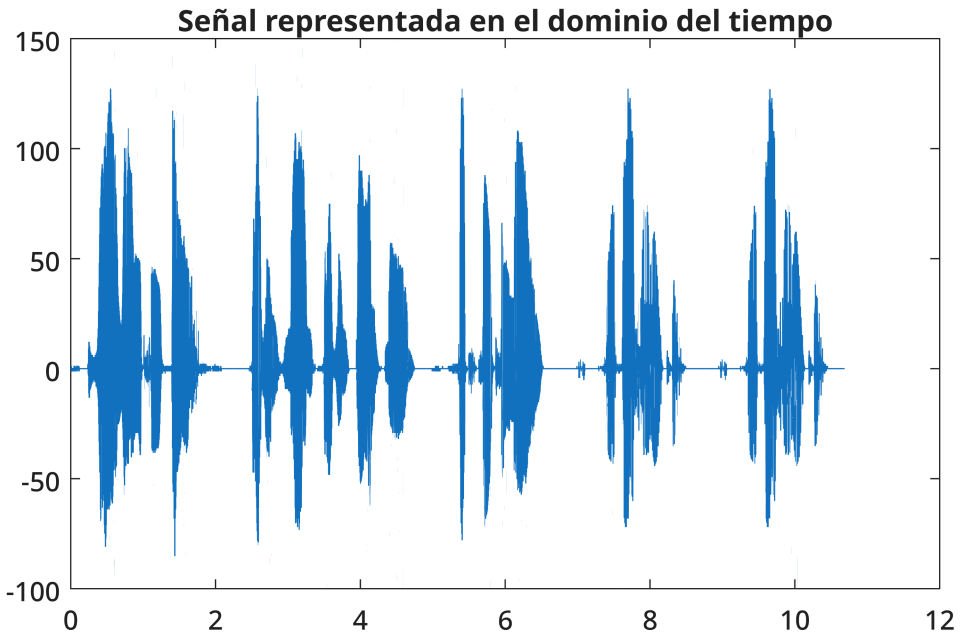
```
figure;
plot(S)
title("Señal representada")
```



Sin embargo, para que la gráfica esté realmente correcta, necesitamos graficar sobre el dominio del tiempo.

Con la función **length()** voy a obtener el número de elementos que tiene el vector, el número de muestras totales, y sabiendo que un segundo tiene 8.000 muestras (frecuencia de muestreo a 8kHz), divido el número de muestras entre la frecuencia, y graficaré correctamente los datos en el dominio del tiempo:

```
figure;
plot([1:length(S)]/frequency, S)
title("Señal representada en el dominio del tiempo")
```



Con nuestro supervector que contiene la información de todas las señales que nos han dado, ahora necesitamos saber la probabilidad de que S tome un valor dentro del espacio muestral. Recordemos que podemos representar con 8 bits, 256 números, intervalo [-128, 127]. Así que, vamos a construir una tabla de *puntuación* que contabilice el número de veces que ha aparecido un posible valor del espacio muestral.

Creamos dicha tabla de puntuación:

```
howManyTimesDoesItAppear = [-128:127; zeros(1,256)]
```

```
howManyTimesDoesItAppear = 2×256
  -128  -127  -126  -125  -124  -123  -122  -121  -120  -119  -118  -117  -116 ···
     0     0     0     0     0     0     0     0     0     0     0     0     0
```

Nota: *La fila superior la uso como encabezado para que sea más intuitivo.*

Y ahora, recurro a un bucle **for** para contar el número de veces que aparece un valor del espacio muestral en S.

```
for row_index = 1:length(S)
    howManyTimesDoesItAppear(2, S(row_index, 1) + 128) =
(howManyTimesDoesItAppear(2, S(row_index, 1) + 128) + 1);
    % Observa que "S(row_index, 1) + 128" es el índice que referencia la
    % columna.
end

howManyTimesDoesItAppear
```

```
howManyTimesDoesItAppear = 2×256
  -128  -127  -126  -125  -124  -123  -122  -121  -120  -119  -118  -117  -116 ···
     0     0     0     0     0     0     0     0     0     0     0     0     0
```

A partir de la tabla de puntuación, yo ya puedo obtener la probabilidad de que S tome un valor dentro del espacio muestral. Simplemente tengo que dividir la tabla de *puntuación* entre el número de muestras que tiene el vector S.

```
probabilityMatrix = howManyTimesDoesItAppear(2, :) / length(S)
```

```
probabilityMatrix = 1×256
         0         0         0         0         0         0         0         0 ···
```

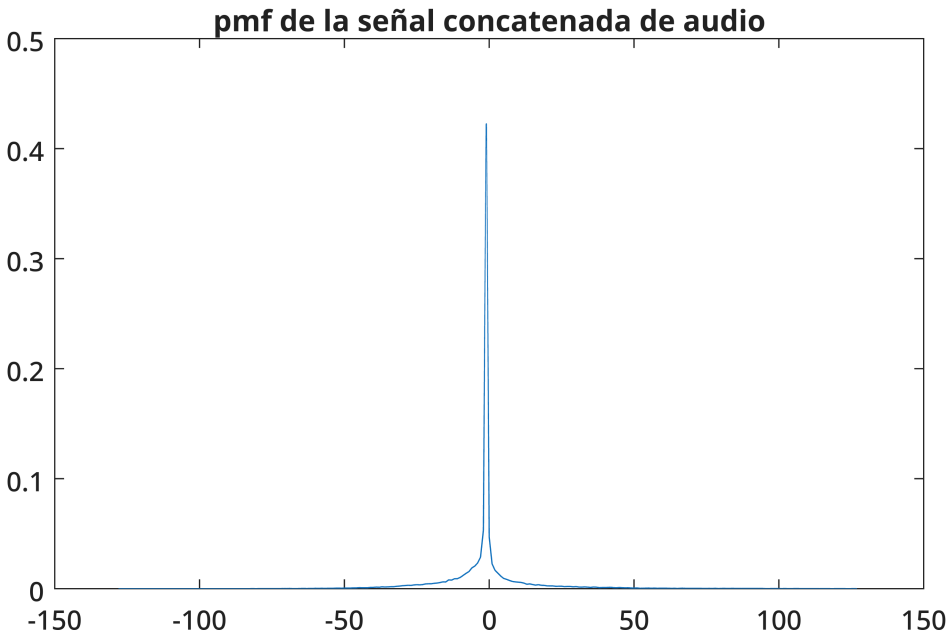
Antes de seguir, reemplazo las probabilidades 0 por valores muy pequeños, puesto que una probabilidad 0 implica suceso imposible, y eso no es del todo cierto en esta situación. Nosotros hemos obtenido 0 por la aproximaciones de las operaciones.

```
probabilityMatrix(probabilityMatrix==0) = quasicero
```

```
probabilityMatrix = 1×256
    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000 ···
```

Vamos a representar gráficamente la matriz de probabilidad que acabamos de crear:

```
figure;
plot(([-128:127]), probabilityMatrix)
title('pmf de la señal concatenada de audio')
```



RECONSTRUCCIÓN

Ya hemos descrito nuestra muestra. Sabemos que nuestra señales afrontarán medios sin pérdidas. Ahora tenemos que abordar cómo actuaremos en los momentos en los que perdemos una muestra.

Para determinar el valor de las muestras perdidas, recurriremos a dos estimaciones:

LA MODA

La primera estimación consistirá en reemplazar la muestra perdida con aquella con mayor probabilidad (moda). Para ello, necesitaré saber la posición en la que se encuentra la moda en mi matriz de probabilidad. Usaré la función **max()**.

```
[trendProbability, trendPosition]= max(probabilityMatrix)
```

```
trendProbability =
0.4225
trendPosition =
128
```

Nota: *Más información sobre la sintaxis en* https://es.mathworks.com/help/matlab/ref/double.max.html.

Finalmente, tenemos que ajustar el parámetro de posición para que se encuentre en el intervalo [-128,127]:

```
trendPosition = trendPosition - 129
```

```
trendPosition =
-1
```

PROBABILIDAD PONDERADA

La segunda estimación consistirá en reemplazar la muestra perdida con una probabilidad ponderada. El valor que viene dado por la siguiente expresión:

$$\sum_{i=1}^{256} m_i P[S = m_i]$$

```
weightProbabilityPosition = [-127:128]*probabilityMatrix'
```

```
weightProbabilityPosition =
-0.0019
```

EVALUACIÓN

Entramos en la recta final, supondremos que enviaremos una señal de voz por un canal con pérdidas. Esta perderá una muestra de voz cada 10 muestras de voz transmitidas. Usaremos los métodos de estimación que hemos preparado para tratar de reconstruir la señal. Para evaluar la calidad de la reconstrución, haremos uso de un parámetro llamado SNR, del inglés, *signal to noise relation*.

Empecemos primero por cargar el fichero test.wav, que contiene la señal que se enfrentará al médio con pérdidas:

```
[realSignal] = audioread('test.wav');
```

Decodificamos la información:

```
realSignal=double(int8(realSignal*256));
```

Para simular la pérdida, cada 10 muestras reemplazaremos el valor de la señal real con uno aleatorio entre 1 y 10 con la función **randi()**.

```
lossySignal = realSignal;
```

```
for row_index = 1:10:length(lossySignal)
    lossySignal(row_index, 1) = randi([1,10]);
end
```

Observación: *La clave del reemplazamiento cada 10 muestras se encuentra en la expresión "1:10:length(lossySignal)", donde indicamos el salto entre valores.*

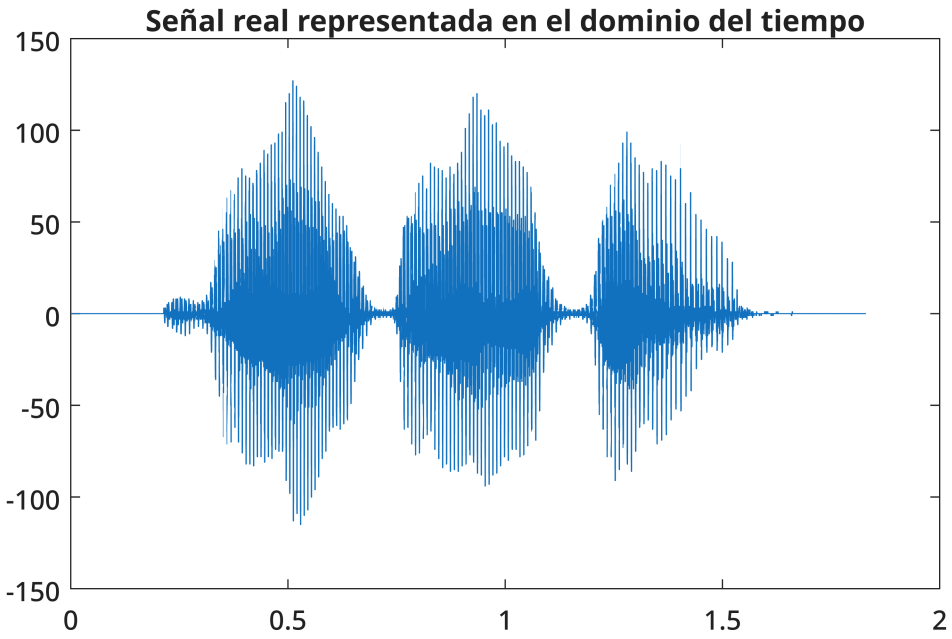
Ahora, vamos a tratar de representar el resultado de nuestra señal afectada por las deplorables condiciones de nuestro medio con pérdidas, para compararlo con la señal original.

Para la escucha de las, usaremos la función **sound()**, que convierte matrices de datos de señal en sonido:

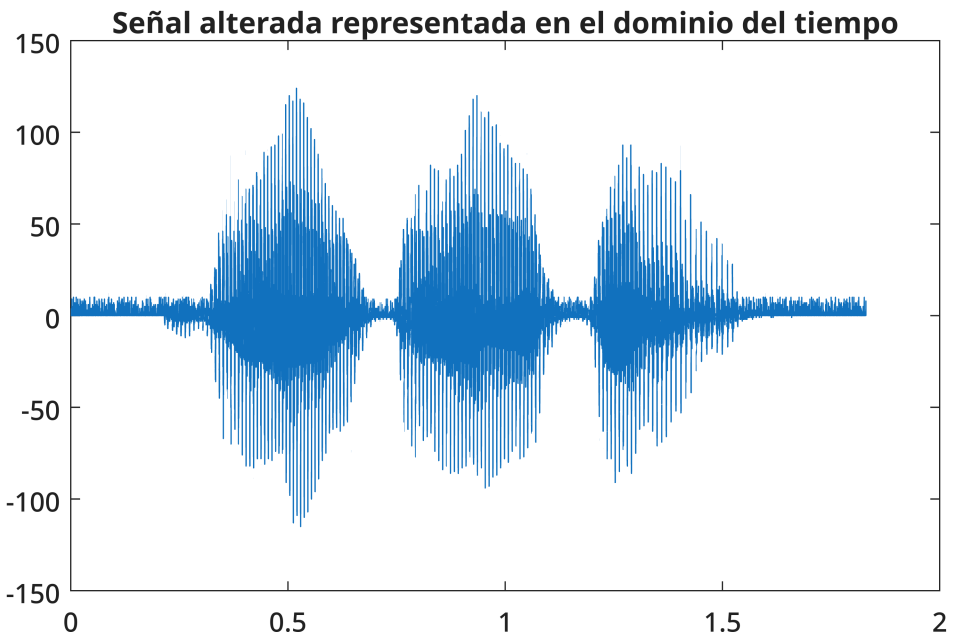
```
% sound(realSignal, frequency, 8);
% sound(lossySignal, frequency, 8);
```

Gráficamente, estos serían los resultados:

```
figure;
plot([1:length(realSignal)]/frequency, realSignal)
title("Señal real representada en el dominio del tiempo")
```




```
figure;
plot([1:length(lossySignal)]/frequency, lossySignal)
title("Señal alterada representada en el dominio del tiempo")
```



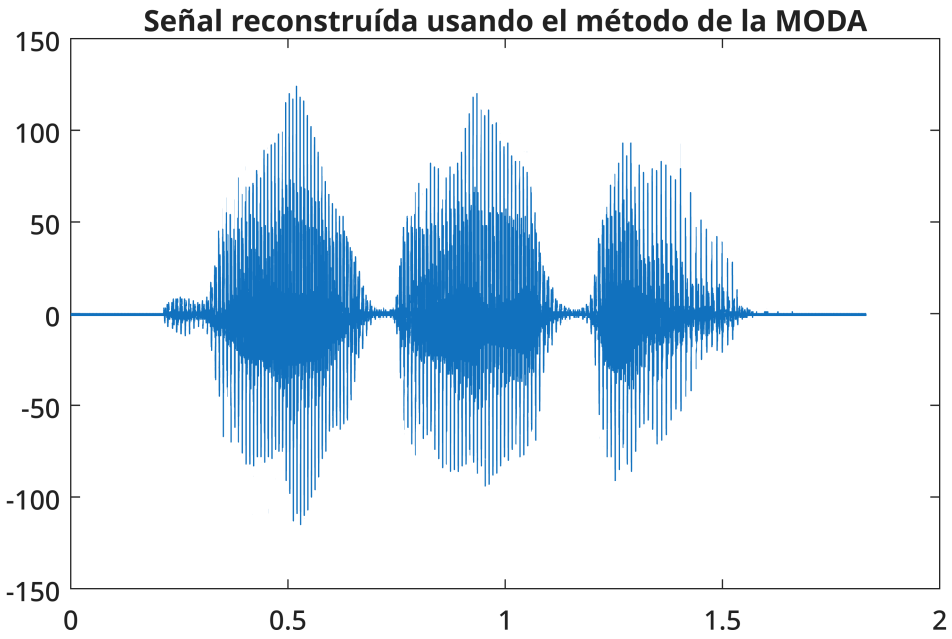
El trabajo no ha terminado aquí, vamos a reconstruir nuestra señal original.

Como sabemos que perdemos información cada 10 muestras, significa que cada 10 muestras tendremos que aplicar nuestros métodos de estimación. Al tener dos métodos, haremos dos reconstrucciones y las compararemos usando la SNR:

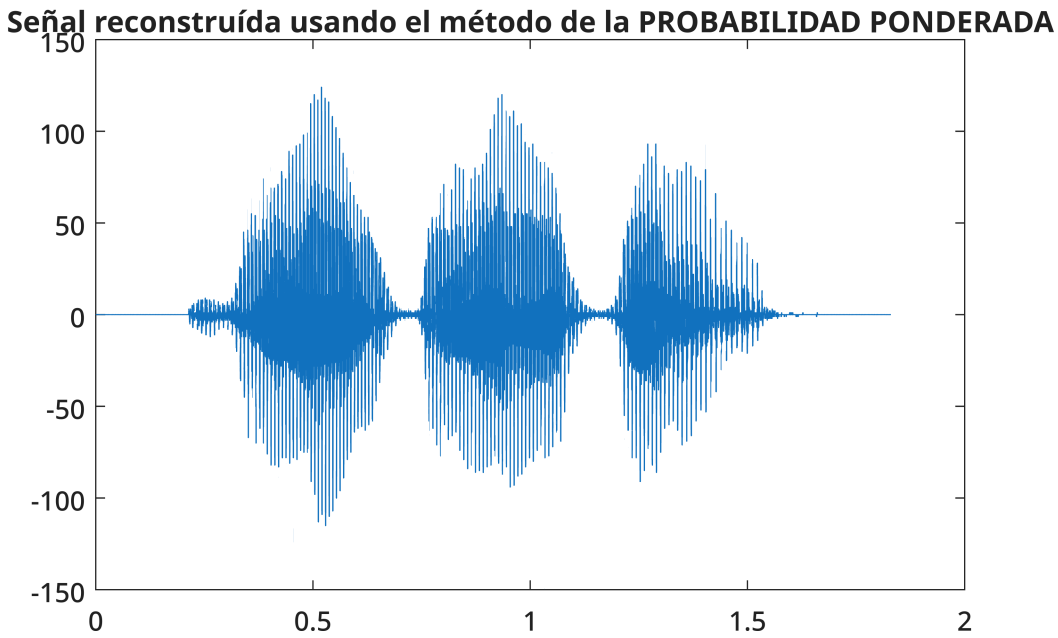
```
trend_reconstructedSignal = lossySignal;
weightProbability_reconstructedSignal = lossySignal;

for row_index = 1:10:length(lossySignal)
    trend_reconstructedSignal(row_index, 1) = trendPosition;
    weightProbability_reconstructedSignal(row_index, 1) =
weightProbabilityPosition;
end
```

Representando gráficamente, veamos como quedan nuestras señales reconstruídas:

```
figure;
plot([1:length(trend_reconstructedSignal)]/frequency,
trend_reconstructedSignal)
title("Señal reconstruída usando el método de la MODA")
```




```
figure;
plot([1:length(weightProbability_reconstructedSignal)]/frequency,
weightProbability_reconstructedSignal)
title("Señal reconstruída usando el método de la PROBABILIDAD PONDERADA")
```



Para escucharlo, usaríamos la función **sound()**.

Ahora, vamos a calcular la SNR de ambas señales. La SNR viene dada por la siguiente expresión:

$$\mathrm{SNR}_{k(}(\mathrm{dB}) = 10\log_{10}\frac{\sum_{n=1}^{N} x[n]^{2}}{\sum_{n=1}^{N}(x[n]-r_k[n])^2}$$

donde $x[n]$ es la señal enviada y $r_k[n]$correspondería a las reconstrucciones, $k=1$ siendo la reconstruída con el método de la moda y $k=2$, con la probabilidad ponderada. Ncorresponde a la suma total total de muestras.

```
SNR1 = 10*log10(((realSignal'*realSignal))/((realSignal-
trend_reconstructedSignal)'*(realSignal-trend_reconstructedSignal)))
```

```
SNR1 =
9.8253
```

```
SNR2 = 10*log10(((realSignal'*realSignal))/
((realSignal-weightProbability_reconstructedSignal)'*(realSignal-
weightProbability_reconstructedSignal)))
```

```
SNR2 =
9.8361
```

Obtenemos dos SNR positivas. ¿Qué implica? Pues que la señal es casi 10 veces más fuerte que el ruido. Además observamos que, la estimación usando el método de la probabilidad ponderada es ligeramente mejor.

```
% sound(trend_reconstructedSignal)
% sound(weightProbability_reconstructedSignal)
% Redactado por Alexandru Theodor Muntenaş.
```