PRÁCTICA de CUANTIFICACIÓN

APELLIDOS:	NOMBRE:
APELLIDOS:	NOMBRE:

Fecha de entrega: hasta el 16 de enero

En el tema de muestreo vimos como pasar de una señal analógica, x(t), a una muestreada, x[n], discretizando el soporte de la función. Sin embargo, para que nuestro ordenador pueda manejar dicha señal es preciso pasar al discreto **también** en la otra dimensión, esto es, el valor de la función. En principio, las muestras $x[n]$ pueden tomar cualquier valor (precisión infinita). Nuestro ordenador por el contrario sólo maneja números de precisión finita. El necesario paso de una x[n] tomando cualquier valor a una $\hat{x}$ $[n]$ con precisión finita es llamado **cuantificación**.

Estudiar la cuantificación con *Matlab* tiene el problema de que ya partimos de algo cuantizado (si está en nuestro ordenador está cuantizado). Sin embargo, si la cuantificación inicial era muy fina (numeros reales o un muestreo a 16 bits) y nosotros vamos a hacer una cuantificación más basta (empleando p.e. de 1 a 8 bits) a efectos prácticos tendremos cualitativa y cuantitativamente los mismos fenómenos.

Empezaremos trabajando con un segmento de señal de audio, muestreada a 44100 Hz, de unos 2 segundos de duración:

```
>> [v fs]=wavread('audio.wav'); sound(v,fs);
```

Cuantificador uniforme

En primer lugar estudiaremos la **cuantificación uniforme**, implementada con la función `q_unif()`. En su versión más sencilla se le llama con un vector `x[]` a cuantizar indicando el número de niveles L a la salida y devuelve la versión cuantizada, `vq=q_unif(v,8)`. El cuantificador divide en L partes iguales el intervalo entre el mínimo y el máximo valores de la entrada.

Opcionalmente `q_unif` devuelve los niveles de decisión y reconstrucción $\{x_k, r_k\}$ del cuantificador resultante. Dichos niveles pueden darse como argumento a la función `q_plot` para que pinte una representación gráfica de la función cuantificadora:

```
>> L=4; [vq x r]=q_unif(v,L); figure; q_plot(x,r);
>> L=8; [vq x r]=q_unif(v,L); figure; q_plot(x,r);
```

Escuchar la nueva señal (para p.e. L=8) y usar `show2en1` para apreciar gráficamente las diferencias entre la original y la versión cuantizada:

```
>> sound(vq,fs); figure; plot(v-vq)
>> figure; show2en1(v,q_unif(v,L),fs);
```

Comentad las diferencias (visuales y/o acústicas) en el aspecto de ambas señales.

Para comparar los resultados cuantitativamente, disponemos de la función `snr(v,vq)` que devuelve la relación señal ruido (SNR) entre la señal original `v` y el "ruido" o error de cuantificación `(v-vq)`:

$$SNR = 10 \log_{10} \frac{\sum_k v(k)^2}{\sum_k (v(k) - vq(k))^2}$$

El siguiente bucle calcula la SNR o SQNR (Signal to Quantization Noise Ratio) para distintos L's:

```
>> for bits=1:5, L=2^bits; fprintf('Niveles %2d -> SQNR %.1f\n',L,snr(v,q_unif(v,L))); end
```

Rellenar con los resultados la siguiente tabla:

Bits/muestra	1	2	3	4	5
Niveles	2	4	8	16	32
SQNR Uniforme (dB)					

¿Qué ganancia aproximada en dB se obtiene por cada bit adicional en una cuantificación uniforme?

Cuantificación uniforme sobre imágenes

Vamos a repetir los resultados anteriores trabajando con imágenes en vez de con audio, para darnos cuenta de cómo cambia la situación al utilizar un tipo de señal distinta.

```
im=imread('GIRL.BMP');                    % cargamos una imagen
imd = double(im);                         % Pasamos a float para hacer algunas operaciones
figure; imagesc(im); colormap(gray(256)); % La visualizamos.
```

Ahora, usando el bucle: `for k=1:6, L=2^k; imq=q_unif(imd,L); snr(imd,imq), end;` similar al anterior, rellenaremos el siguiente cuadro:

Niveles	2	4	8	16	32	64
SQNR Uniforme						

Vemos que los resultados (en dB) son bastante mejores que los de audio. Las imágenes tienen un rango dinámico (variación entre posibles valores) menor que el audio, lo que les permite mejores resultados para un número menor de niveles. Para imágenes se suelen usar 8 bits (12 en algunas aplicaciones) mientras que con audio es usual trabajar hasta con 16 bits.

Modificando un poco el bucle anterior podemos visualizar la imagenes cuantificadas:

```
figure; colormap(gray(256));
for bits=1:6,
  L=2^bits; imq=q_unif(imd,L);               % Cuantificamos uniformenente L niveles
  subplot(2,3,bits); imagesc(imq); axis off  % Visionamos imagen
end
```

¿Que efecto se aprecia en las imágenes cuantizadas con pocos niveles? ¿En qué zonas de la imágen se aprecia mejor?

Para que os déis cuenta de cómo las medidas matemáticas no son una buena medida de los aspectos perceptuales de una cuantificación comparar la cuantificación a dos niveles obtenida antes:

```
figure; colormap(gray(256)); imq=q_unif(imd,2); imagesc(imq); axis off
```

con la siguiente imagen (también binaria), obtenida con el siguiente comando MATLAB:

```
figure; colormap(gray(256)); imq2=dither(im); imagesc(imq2); axis off
```

¿Qué imagen es preferible perceptualmente? ¿Cuál sería vuestra apuesta sobre qué imagen tiene una mayor SNR?

Cuantificador óptimo

Un cuantificador uniforme sólo es óptimo si la variable a cuantificar tiene una función de densidad de probabilidad (pdf) $p_x(x)$ uniforme. Veremos cual es la pdf de nuestro vector de entrada.

Al estar trabajando en un entorno discreto (y ser nuestra variable inicial discreta) no podremos calcular una pdf continua $p_x(x)$. Estimaremos una aproximación discreta a la pdf, lo que se conoce como un **histograma**. En la práctica las diferencias son pocas. La función `hist()` de *Matlab* calcula el histograma del vector de entrada:

```
[h vv] = hist(v,64);  % Histograma (aprox de la pdf en 64 puntos)
sum(h), h=h/sum(h); sum(h),
bar(vv,h);            % Pintamos histograma como gráfico de barras
```

¿Por qué hacemos lo de `h=h/sum(h);`?

Se observa claramente como el histograma resultante no es uniforme, sino bastante picudo alrededor del cero (la señal de voz pasa mucho más tiempo en los alrededores del cero que en los extremos). Sin embargo el cuantificador uniforme cubría todo el intervalo, [-1,1] lo que era poco eficiente al dedicar demasiada atención a las colas de la distribución (valores que aparecen muy raramente).

Lo óptimo sería hacer variar gradualmente el tamaño de los intervalos, desde muy grandes en las colas hasta muy pequeños cerca del cero. Dada una pdf $p_x(x)$ es posible diseñar un cuantificador óptimo o de Max-Lloid usando algoritmos iterativos. Usando una versión aproximada de la pdf como es el histograma podemos implementar una versión discreta de dichos algoritmos (donde básicamente cambiamos integrales sobre la pdf por sumas del histograma). Lo primero es calcularnos una versión más fina (más parecida a la pdf subyacente) del histograma que la que teniamos:

```
[h vv]=hist(v,2048);  h=h/sum(h);  bar(vv,h);     % Hist de 2048 niveles.
```

La función que implementa los algoritmos iterativos es `opthisto()`:

```
>> help opthisto
[x r db]=ophisto(h,vv,L);
Halla la cuantificación óptima en L niveles de una variable discreta a partir de su histograma h[]
en los valores vv[], devolviendo niveles de decision/reconst x[], r[] y error resultante en dB.
```

Recibe la información de probabilidades de la señal v (en forma de histograma `h[], vv[]`) así como el número de niveles deseados L, y la función nos devuelve los L+1 niveles de decisión x_k y los L niveles de reconstrucción r_k "óptimos":

```
L=8; [x r]=opthisto(h,vv,L);
```

Vemos como interactivamente nos va dando la información de la SNR conseguida en las sucesivas iteraciones. Cuando la variación en una iteración (delta) es muy pequeña (menor que 10^{-4}) el algoritmo termina. El algoritmo parte de un cuantificador uniforme y mostramos gráficamente en pantalla su evolución. ¿Cúal es el apecto más destacado que observais en la evolución hacia el óptimo?

Para visualizar mejor el comportamiento de un cuantificador óptimo podemos visualizar los anchos de los intervalos de decisión en función de su posición:

```
anchos = x(2:end) - x(1:end-1); plot(r, anchos,'ro');
```

Para el caso anterior ¿de qué orden es el intervalo máximo usado? ¿y el mínimo? ¿Dónde suceden cada uno de ellos? Comparar dichos anchos con el intervalo de un cuantificador uniforme con los mismos 8 niveles.

Observamos que el algoritmo de optimización nos indica la relación señal-ruido (SQNR) para la secuencia `x[]` que le sirvio de "entrenamiento". Podemos por lo tanto correr `opthisto(h,vv,L)` para varios valores de L y rellenar la tabla siguiente:

Niveles	2	4	8	16	32
SQNR Óptimo (dB)					

Comparar estos resultados con los del cuantificador uniforme (audio) de antes. ¿Es siempre mejor el cuantificador óptimo? ¿De que orden de dB son las ganancias obtenidas? ¿Dónde se notan más las ventajas de un diseño óptimo?

Introducción a las técnicas DPCM

Los métodos anteriores estaban basados en cuantificar cada muestra por separado, sin prestar atención a su entorno. Sin embargo cualquier señal tiene un alto grado de redundancia, lo que hace que muestras vecinas tiendan a ser similares. Esta es la base de las técnicas DPCM (Diferencial Pulse Code Modulation), donde la cantidad a cuantificar no es el valor de una muestra en si, sino la diferencia entre nuestra muestra y una predicción basada en muestras vecinas (anteriores):

$$d[n] = x[n] - \hat{x}[n]$$

En su versión más simple, nuestra predicción de la muestra n, $\hat{x}[n]$ puede ser una combinación lineal de los valores de las muestras anteriores:

$$\hat{x}[n] = \sum_{k=1}^{P} c_k x[n-k] \quad \text{y el valor a transmitir queda} \quad d[n] = x[n] - \sum_{k=1}^{P} c_k x[n-k]$$

Las llamadas técnicas de *predicción lineal* permiten, dada una señal, `x[n]` y un orden de predicción P (número de muestras anteriores a usar en la predicción) calcular los coeficientes de predicción c_k óptimos (en el sentido de minimizar las discrepancias entre los valores predichos y los verdaderos). Dicho calculo está implementado en la función `predlpc`:

```
>> [x fs]=wavread('audio'); coefs=predlpc(x,1), coefs=predlpc(x,2),
```

¿Cuál sería la fórmula óptima para predecir `x[n]` en función de `x[n-1]` y `x[n-2]`?.

Veamos la aplicación a una codificación predictiva. La función `d=dpcm(v,coefs)` recibe una señal y unos coeficientes de predicción y devuelve los errores de predicción en la variable de salida (en los P primeros valores de `d[n]` se guardan los P primeros valores de la señal original, necesarios para iniciar la decodificación). La función `dec(d,coefs)` recibe los P primeros valores, los errores de predicción y los coeficientes a usar y recupera la señal original. Veamos como funciona para P=3:

```
P=3; coefs=predlpc(x,P);
d=dpcm(x,coefs); xr=dec(d,coefs); plot(x-xr)
```

¿Son idénticas ambas señales? ¿A qué pueden ser debidas las diferencias observadas? ¿Cuál es la SNR entre ambas?

Para entender las posibles ventajas de trabajar con `d[]` en vez de la `x[]` original, basta comparar su aspecto y sus respectivos histogramas:

```
>> subplot(311); plot(x,'b'); hold on; plot(d,'r'); hold off
>> [h,v]=hist(x,128); subplot(312); bar(v,h); set(gca,'Xlim',[-1 1]);
>> [h v]=hist(d,128); subplot(313); bar(v,h); set(gca,'Xlim',[-1 1]);
```

¿Qué histograma es más "estrecho"? Calcular (usando la función `std`) la desviación standard de `x[]` y `d[]`. La desviación standard está directamente relacionada con el "ancho" del histograma.
Con nuestros conocimientos actuales, ¿cuál de las dos señales (`x[]` o `d[]`) podrá ser cuantificada con un menor error para el mismo número de niveles? ¿Por qué?

Sin embargo, para que nuestra simulación sea del todo realista, antes de que el decodificador intente recuperar la señal hay que cuantificar los valores del error de predicción. Recordar que al ser los coeficientes numeros reales, los valores predichos también lo son, y por lo tanto los errores, por lo que si no los cuantificamos no ganamos nada (de hecho, "engordamos" la señal).
Vamos a ver los resultados para una cuantificación uniforme de 16 niveles de los errores de predicción:

```
>> dq=q_unif(d,16); xr=dec(dq,coefs);
```

Escuchar a la señal recuperada. ¿Suena bien? ¿Cuál es la SNR entre `x` y `xr`? ¿Es este resultado aceptable? (recordad el resultado de un cuantificador uniforme sobre la señal original para 16 niveles).

Obviamente el problema se origina en que, debido a la cuantificación, el decodificador recibe una copia imperfecta de los errores de predicción. La medida de dicho error puede estimarse por la desviación estandard de `(d-dq)`. ¿De qué orden es? Uno por supuesto espera que la recuperación venga afectada por errores similares. Sin embargo si calculamos la desviación standard para `(x-xr)` vemos que es mucho mayor. ¿De qué orden es? ¿Cuánto mayor que las alteraciones de los errores?

Esta situación nos indica que nuestro sistema es inestable. Alteraciones en su entrada (los errores de predicción) no se mantienen constantes, sino que se magnifican en su salida (la señal decodificada).
El problema es que el codificador está usando la señal original para predecir, mientras que el decodificador usa la señal recuperada (obviamente el decodificador no tiene acceso a la original). Si los errores de predicción se transmiten de forma exacta (como antes) la recuperada es idéntica a la original y todo funciona. Sin embargo, en cuanto cuantizamos las diferencias, la señal recuperada deja de ser igual a la original y el codificador y decodificador dejan de estar en sincronía. Uno predice basado en la original, el otro en la recuperada. Si ambas empiezan a divergir, cada vez tienen menos en comun, y se aumentan las diferencias.
Para que entendais el problema vamos a ver lo que haría nuestro codificador DPCM ante la siguiente entrada: `100, 102, 120, 120, 117, 116, ...` (correspondiente a una señal en la que hay un salto brusco de 102 a 120). Usaremos un predictor muy sencillo, donde nuestra predicción es simplemente el valor anterior $\hat{x}[n] = x[n-1]$, y al error de predicción se le aplica un cuantificador de cuatro niveles:

error	$(-\infty,-2]$	$(-2,0]$	$(0,2]$	$(2,\infty)$
Q(e)	-5	-1	1	5

Rellenar las columnas vacias de la tabla siguiente, de acuerdo al comportamiento del codificador/ decodificador DPCM que hemos presentado:

`x[n]`	Pred `(x[n-1])`	`d[n]`	`dq[n]`	`r[n]=r[n-1]+ dq[n]`	`e_r[n]=x[n]-r[n]`
102	100	2	1	101	1
120	102				
120	120				
117	120				
116	117				

Trás el error cometido en el paso de 102 a 120 (por el fallo de la "predicción") ¿Se aprecia una reducción del error de recuperación en los paso sucesivos? ¿Por qué las correcciones calculadas por el codificador no son útiles para el decodificador? ¿Cuál es el problema del diseño de este esquema?

Repetir el ejercicio con un nuevo esquema en el que la predicción se basa no en la señal original, sino en la recuperada $\hat{x}[n] = r[n-1]$. Notad que mientras que es imposible que el decodificador tenga acceso a la señal original, si que es posible que el codificador tenga acceso a la recuperada (basta que el codificador haga lo que haría el decodificador con los datos disponibles).

`x[n]`	Pred `(r[n-1])`	`d[n]`	`dq[n]`	`r[n]=r[n-1]+ dq[n]`	`e_r[n]=x[n]-r[n]`
102	100	2	1	101	1
120	101				
120					
117					
116					

Se ve que, aunque hay errores grandes ante un cambio brusco (al fallar la predicción), dichos errores son puntuales, puesto que tienden a corregirse, al contrario que en el otro caso, donde tendían a incrementarse, dado que el codificador no era "consciente" de dichos errores.

La función `dq=dpcm_q(x,coefs,L)` implementa la versión correcta de un codificador DPCM. Notad que al contrario que antes hay que darle el número de niveles para cuantificar el error, ya que la cuantificación está integrada en el proceso de predicción. La salida son las diferencias **ya** cuantizadasr. El proceso completo (codificar/decodificar) sería:

```
>> L=16; dq=dpcm_q(x,coefs,L); xr=dec(dq,coefs);
```

Estimar la desviación standard de `(x-xr)` ¿Qué valor tiene? Compararla con la desviación de `(d-dq)` obtenida antes para 16 niveles. ¿Qué supone esto sobre estabilidad del nuevo esquema?

Calcular ahora la SQNR del nuevo esquema para `L=2,4,8,16` niveles. Rellenar la siguiente tabla:

Niveles	2	4	8	16
SQNR DPCM (dB)				

Comparar dichos resultados con los del cuantificador óptimo. ¿Cómo es posible que nuestro DPCM supere al cuantificador óptimo? Pensad en la diferencia entre un codificador de texto que codifique letras por separado y otro cuya unidad de codificación sean "grupos de letras".

Como vemos, un codificador DPCM puede dar resultador bastante mejores que un PCM (codificación de muestras por separado). En la práctica, para mejorar aún más sus resultados se usan codificadores DPCM adaptativos (ADPCM), cuyas carácteristicas (sus coeficientes de predicción, los parámetros del cuantificador, etc) se van actualizando, para adaptarse a cambios en la señal de entrada. En el apartado siguiente (opcional) usaremos el caso más sencillo posible de codificador DPCM (un sólo coeficiente de predicción, cuantificador del error con un sólo bit) para ilustrar las ventajas de las técnicas adpatativas.

Modulación Delta (Entrega opcional)

Vamos a estudiar un caso especial de cuantificación diferencial (DPCM) llamada modulación delta. La modulación delta es el caso más sencillo de cuantificación DPCM. Nuestro predictor va a ser de orden 1, esto es, solo nos apoyaremos en la muestra inmediatamente anterior. La predicción será $\hat{x}[n] = c\,x[n]$, con c un valor similar a 1, típicamente un poco menor, p.e. $c = 0{,}9$. La razón de esto es asegurarnos estabilidad, impidiendo que a nuestra predicción le de por crecer desmesuradamente. Por otra parte, la cuantificación de los errores de predicción también es muy sencilla. Se usa un cuantificador de dos niveles (1 bit), mandando 0/1 según el error de predicción sea negativo (nos

hemos pasado en la predicción) o positivo (se ha quedado corta la estimación). El receptor recibe dicho datos y resta/suma un incremento (delta) al valor anterior. Los intervalos de decisión del cuantificador son por lo tanto $(-\infty, 0] \cup (0, \infty)$

Se trata de que escribáis una funcion en *Matlab* , `[bits rec]=dm(v,delta)` que implemente la modulación delta (de hecho, que relleneis el *cascarón* que os suministro, `dm.m`). El primer argumento de entrada es la señal a codificar, `v[ ]`. El segundo argumento debe especificar el incremento/decremento que aplica el demodulador al recibir un 0/1 del modulador. Si el error de predicción es mayor que cero, el codificador emite un 0 y el demodulador sumará `delta` al último valor recuperado. Si por el contrario el error es menor o igual que cero se emite un 1 y el decodificador debe restar `delta` al último valor decodificado. La función anterior debe devolver en `bits` los 0/1 que mandaría por la línea.

Como todo buen codificador DPCM debe hacer también el trabajo del decodificador para predecir sobre la señal decodificada y asegurar la sincronía con el decodificador, nuestra función debe ser capaz de devolvernos la señal recuperada, `rec`. Dicha señal se obtiene a base de sumar/restar (según los valores de `bits`) delta al valor anterior. Usar un coeficiente de predicción $c = 0{,}95$ y recordad que para que sea estable vuestro codificador debe predecir sobre la señal recuperada y no sobre la original.

Usar ahora la función que acabais de escribir con distintos valores de delta y rellenar las tres primeras filas de la primera columna en la tabla siguiente:

Delta	SNR rec (dB)	SNR post (dB)
0.01		
0.05		
0.1		
Variable		

Usando la función `show2en1(x,rec,fs)` comparar el original con la salida de la demodulación delta para delta=0.01. ¿Qué se observa en las zonas donde la señal original oscila con rápidez? ¿Por qué no puede seguir la modulación delta a la señal original? ¿Podríamos mejorar este comportamiento variando el valor de delta? ¿En qué sentido?

Repetir para delta=0.1, comparando de nuevo la original y recuperada. ¿Es ahora capaz de seguir nuestro modulador a la señal? ¿Qué problemas presenta ahora la recuperación de la señal?

Obviamente la aplicación directa del demodulador (debido a los contínuos incrementos/decrementos, introduce un claro ruido en la salida. Podemos mejorar notablemente los resultados postprocesando la señal de salida del modulador, aplicando un filtro pasobajo.

```
>> delta=0.01; [bits rec]=dm(x,delta); post=low_pass(rec,fs); fprintf('%.1f db\n',snr(x,post));
```

La función `low_pass` aplica un filtro FIR de tipo pasobajo con una frecuencia de corte de unos 5000 Hz (suficiente para dejar pasar lo fundamental de la voz). Rellenar las tres primeras filas de la segunda columna de la tabla anterior para los distintos valores de delta. ¿Cuál es el mejor valor de SNR y con qué valor de delta lo obtenemos?

Para terminar de arreglar la situación anterior vamos a introducir una mejora en nuestro método. El problema fundamental es que, como se ha visto, un delta fijo presenta problemas. Si es grande mete ruido en las zonas tranquilas y si es bajo no es capaz de seguir a la señal cuando cambia. La solución es tener un delta variable, obteniendo así una modulación delta adaptativa, que es el ejemplo mas sencillo de la familia de cuantificadores ADPCM (Adaptative Diferential). En las implementaciones reales son fundamentales los algoritmos adaptativos, ya que las características de cualquier señal cambian con el tiempo. La modulación delta adaptativa es el caso más simple que podemos estudiar.

Se trata simplemente de establecer una regla (que pueda ser seguida por codificador y decodificador) que adapte la delta a las circustancias: grande cuando la señal varia, pequeña cuando está tranquila. Una regla muy sencilla y que funciona bastante bien es la siguiente:

Si los dos bits últimos comparten signo, aumentar delta. Si alternan signo, disminuir delta

Y eso es todo. Basta arrancar con un delta dado, definir un factor `F>1` y incrementar `delta=delta*F` o decrementar `delta=delta/F` según estemos en un caso u otro.

¿Por qué la regla anterior cumple con nuestra idea de cómo se debe comportar `delta`?

La función `[bits rec]=dm_var(x)` implementa la regla anterior. No hay que darle un delta puesto que lo inicializa a un valor medio y lo modifica dependiendo de como vayan las cosas.

¿Cual es la nueva relación señal ruido entre `rec` y el original `x`?. Visionarlas con `show2en1`. ¿Cómo se comporta el incremento delta? ¿Apreciais algún problema?

Nuestra nueva modulación delta adaptativa también puede beneficiarse de un filtrado pasobajo final. Aplicar a nuestro resultado actual `rec` el mismo filtro pasobajo de antes. Comparar (usando `show2en1`) la señal original y la obtenida trás el pasobajo final. ¿Cual es la relación señal ruido del resultado intermedio? ¿Y del resultado final trás el pasobajo? Rellenar la última fila de la tabla anterior con vuestros resultados.

Con la modulación delta adaptativa hemos conseguido unos resultados bastante aceptables usando 1 bit por muestra (diferencia). Recopilar los datos obtenidos en distintas parttes de la práctica y

comparar la relación señal/ruido obtenida para nuestra señal con un cuantificador uniforme, óptimo y modulador delta, todos ellos trabajando a un bit por muestra:

MÉTODO	Uniforme	Óptimo	Modulación delta
SNR (1 bit=2 niveles)			

Antes de echar las campanas al vuelo y alabar sin mesura a la modulación delta, he de comentaros que os he estado engañando todo este rato. El valor de fs usado es de 44100 Hz, cuando siendo una señal de voz (ancho de banda 4/5 KHz) bastaría con 8000 o 10000 Hz. Esto es, la señal con la que estamos trabajando estaba **sobremuestreada**. Si intentaseis aplicar la la modulación delta a la misma señal pero submuestreada en un factor 4 o 5 veríais que los resultados son mucho peores. ¿Por qué creéis que la modulación delta se beneficia de un sobre muestreo de la señal?

Si todo esto es cierto ¿cuál es la ventaja de la modulación delta?. La respuesta es simplicidad en una implementación real. Acostumbrados como estamos a trabajar simulandolo todo en un ordenador no nos damos cuenta de que hay aspectos tan (o más importantes) que simplemente los resultados de un algoritmo. En un caso tenemos un cuantificador óptimo (con todas sus complejidades) o uno uniforme, con un conversor A/D de p.e. 16 o 32 niveles. En el otro un modulador delta. Ambas opciones pueden tener un rendimiento similar en cuanto a relación señal/ruido. Sin enbargo el modulador delta es mucho más simple. En hardware sería simplemente retener el último valor (retardo) y compararlo con el actual (comparador). Esto es lo que hace que se siga trabajando con moduladores delta y no tanto el que ganes o pierdas un dB más o menos en la relación señal/ruido.