Trabajo Fin de Grado

Librería de reconocimiento de voz en coma fija para sistemas empotrados

Fixed-point speech recognition library for embedded systems

Autor

Héctor Bescós Giménez

Director

Isidro Urriza Parroqué

Escuela de Ingeniería y Arquitectura, Universidad de Zaragoza.

2018

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

D./Dª. Héctor Bescós Giménez,

con nº de DNI 18169587R en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) grado, (Título del Trabajo) Librería de reconocimiento de voz en coma fija para sistemas empotrados,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 25 de junio de 2018

H. Bescós

Fdo: Héctor Bescós Giménez

Resumen

El trabajo consiste en la elaboración de una librería de funciones en lenguaje C con aritmética en coma fija para el desarrollo de un sistema de reconocimiento de comandos de voz en el *ezdsp5535* de Texas Instruments.

El diseño sigue una metodología basada en capas con el fin de que pueda ser portada fácilmente a otras plataformas. La librería utiliza algoritmos basados en modelos ocultos de Markov, y para el entrenamiento de los modelos se utiliza el IDE de Matlab con funciones diseñadas ex profeso. Se desarrolla una aplicación Matlab con GUI para facilitar las tareas de entrenamiento y se comprueban las prestaciones y portabilidad de la librería mediante su utilización en la implementación de demostradores en dos tipos de plataformas.

Summary

The project consists in the elaboration of a library of functions for speech recognition. It is developed in C language with fixed-point arithmetic and implemented in the TI's ezdsp5535.

The design follows a layer-based methodology, so it can work on many other platforms. The library uses algorithms based on hidden Markov models, and for the training of the models it is used Matlab IDE with specifically designed functions. In addition, a GUI application is developed with Matlab to facilitate training tasks. The performance and portability of the library are tested with the implementation of demonstrators in two types of platforms.

Índice

Lista de figuras

Lista de tablas

Lista de acrónimos

MFCC	Mel-Frequency Cepstral Coefficient
HMM	Hidden Markov Model
GUI	Graphical User Interface
WAV	Waveform Audio File Format (WAVE)
TI	Texas Instruments
CCS	Code Composer Studio
FFT	Fast Fourier Transform
DFT	Discrete Fourier Transform
DCT	Discrete Cosine Transform
DSP	Digital Signal Processor
HWAFFT	FFT Hardware Accelerator
FPU	Floating-Point Unit
ROM	Read-Only Memory

1. Introducción

1.1. Descripción del proyecto

El objetivo del proyecto es desarrollar una librería de funciones en lenguaje C que permita implementar en un microprocesador en coma fija aplicaciones que utilicen eventos basados en comandos de voz.

El diseño de la librería deberá seguir una metodología basada en capas, de tal forma que sea fácilmente portable entre distintas plataformas.

La librería utilizará algoritmos basados en modelos ocultos de Markov y será necesario desarrollar una aplicación en Matlab para el entrenamiento off-line de los modelos.

1.2. Metodología

La librería se programará en C utilizando metodología basada en capas y aritmética en coma fija. Para el desarrollo de los algoritmos de reconocimiento de voz en coma fija se utilizará Matlab y una vez validados se implementarán en C. Como sistema IDE se utilizará *Code Composer Studio* (CCS) de Texas Instruments (TI), la librería se validará con un demostrador que se ejecutará en un sistema de desarrollo *ezdsp5535* de Texas Instruments. Del mismo modo, se comprobará su portabilidad mediante el desarrollo de un demostrador en PC mediante el IDE *Dev-C++.*

Por otro lado se diseñará en Matlab una aplicación con entorno gráfico para el entrenamiento de los modelos ocultos de Markov.

El cronograma con las fases del trabajo que se seguirán es el siguiente:

- Revisión bibliográfica sobre la implementación de algoritmos de reconocimiento de voz en sistemas empotrados. (30h)
- Implementación del algoritmo basado en cadenas ocultas de Markov en coma flotante y la aplicación para el entrenamiento del modelo (*Matlab*). (60h)
- Rediseño de los algoritmos en coma fija (*Matlab*). (40h)
- Implementación en C de la librería y desarrollo de dos demostradores (*CCS*, *ezdsp5535,* y *Dev-C++*, PC). (70h)
- Evaluación de las prestaciones de funcionamiento de la librería. (10h)
- Diseño de un manual de usuario de la librería. (6h)
- Redacción de la memoria. (90h)

1.3. Estado del arte y alcance

El reconocimiento de voz es una disciplina estudiada en lingüística computacional y tiene el propósito de desarrollar métodos de comunicación hablada entre humanos y máquinas. Utiliza conocimientos interdisciplinares de lingüística, computación y electrónica.

Los modelos ocultos de Markov (HMM) comienzan a utilizarse en el reconocimiento de voz a comienzos de los años 70 y posteriormente en numerosas aplicaciones relacionadas con el aprendizaje por computador. En la actualidad se emplean en combinación con redes neuronales, aunque están surgiendo nuevos métodos como Deep learning o Long short-term memory, basados en redes neuronales y utilizados comercialmente con mejores resultados [1].

En este trabajo se pretende desarrollar una herramienta de reconocimiento de voz basada en HMM que no requiera de gran potencia de procesamiento y pueda ser incluida en proyectos sencillos para una amplia gama de sistemas empotrados.

1.4. Estructura de la memoria

Además del presente apartado de introducción, la memoria consta de cinco apartados y tres anexos:

Apartado 2 - *Funciones en lenguaje Matlab para la implementación del sistema de reconocimiento de voz*. Se desarrollan de manera justificada las funciones en lenguaje Matlab necesarias para implementar los procesos de entrenamiento y reconocimiento.

Apartado 3 - *Librería en lenguaje C para el reconocimiento de voz en sistemas de coma fija*. A partir de las funciones elaboradas en el apartado anterior se desarrolla una librería de funciones en lenguaje C con aritmética de coma fija para la implementación de sistemas de reconocimiento de voz.

Apartado 4 *Demostradores desarrollados para el reconocimiento de voz y evaluación de sus prestaciones*. Se elabora la aplicación en Matlab con GUI para el entrenamiento de los modelos y se desarrollan dos demostradores, uno en PC y otro en el ezdsp5535. Se evalúan las prestaciones de ambos mediante pruebas experimentales.

Apartado 5 - *Conclusiones.*

Apartado 6 - *Referencias*.

Anexos. Se introducen los conceptos básicos sobre la señal de voz, las grabadoras de audio digital y los modelos ocultos de Markov.

2. Funciones en lenguaje Matlab para la implementación de sistemas de reconocimiento de voz

Una señal de audio puede utilizarse para entrenar el codebook y los HMM, o para ser reconocida mediante éstos. La figura 2.1 muestra el esquema con los bloques principales del sistema de reconocimiento de voz. En este apartado se explican los detalles de cada uno de estos bloques y se desarrollan las funciones en lenguaje Matlab necesarias para implementarlos [3].

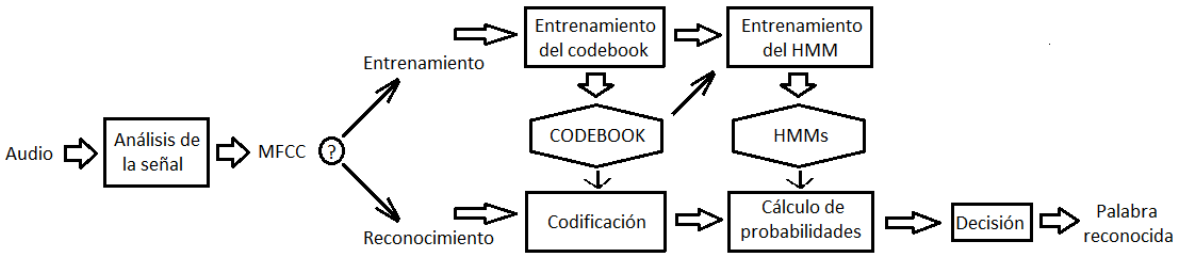

Figura 2.1: Esquema del sistema de reconocimiento de voz.

Análisis: La señal de audio con la palabra pronunciada se digitaliza, se recorta el segmento que contiene la voz y se obtienen los coeficientes cepstrales de las frecuencias de Mel (MFCCs). Éstos pueden ser utilizados para entrenar el sistema o para reconocer la palabra pronunciada.

Entrenamiento: Conocemos a priori cual es la palabra pronunciada y ésta forma parte del vocabulario. Los MFCCs se utilizan en la construcción del codebook y, una vez construido, se codifican y se emplean para entrenar el HMM correspondiente a esa palabra.

Reconocimiento: Desconocemos cual es la palabra pronunciada, y el codebook y los HMMs de todas las palabras del vocabulario están previamente entrenados. Se obtiene la palabra codificada y se aplica el reconocimiento con los HMMs de todo el vocabulario. El que obtenga la probabilidad más alta en la evaluación de los HMMs se decide como la palabra resultado del reconocimiento.

El listado de funciones desarrolladas es el siguiente, se dividen en tres grupos:

- Análisis de la señal
  - ✓ grabar_voz.m
  - ✓ filtro_preEnfasis.m
  - ✓ analisis_coper.m
  - ✓ recortar.m
  - ✓ matriz_filtros_mel.m
  - ✓ coeficientes_mel.m
  - ✓ cepstrum.m
  - ✓ analisis_palabra.m

- Codificación
  - ✓ algoritmo_Kmeans.m
  - ✓ error_por_distorsion.m
  - ✓ codebook_inicial.m
  - ✓ codificar_palabra.m
- Modelos ocultos de Markov
  - ✓ algoritmo_forward.m
  - ✓ algoritmo_backward.m
  - ✓ variable_Epsilon.m
  - ✓ variable_Gamma.m
  - ✓ algoritmo_BaumWelch.m
  - ✓ calculo_HMM_inicio.m
  - ✓ iteracion_HMM.m
  - ✓ algoritmo_forward_reconocimiento.m

Algunas de estas funciones serán trasladadas al lenguaje C en el apartado 3; las que intervienen en el reconocimiento de voz.

2.1. Análisis de la señal de audio

En este subapartado voy a explicar los pasos necesarios para obtener el análisis cepstral de la señal de audio, los Mel Frequency Cepstral Coefficients (MFCCs). La figura 2.2 muestra el esquema con los bloques que se van a desarrollar.

Figura 2.2: Esquema del análisis de la señal para la extracción de los MFCC.

En el anexo I se hace una introducción a las características de la señal de voz que se van a analizar.

2.1.1. Conversión analógico-digital de la señal de audio

El primer paso en la labor de analizar la señal de audio es el muestreo. Este consiste en tomar medidas de la señal analógica generada por el micrófono a intervalos constantes de tiempo. El teorema de Nyquist-Shannon demuestra que, para una señal analógica en banda base, la frecuencia mínima a la que se debe muestrear para obtener la señal digitalizada sin pérdida de información es

el doble del ancho de banda de ésta. El tracto vocal se comporta como un filtro paso bajo, y las tres primeras formantes se encuentran entre los 200 y 3.200 Hz, donde se acumula la mayor parte de la energía de la señal. El resto de frecuencias superiores son atenuadas, por lo que conviene filtrarlas para eliminar el ruido presente en esas partes del espectro. Utilizaré una frecuencia de muestreo de 8.000 Hz que es el estándar para transmisiones de voz en telefonía analógica [4].

Después de muestrear la señal de audio, el siguiente paso es la cuantificación de las muestras. Se utiliza una cuantificación de 16 bits, que es la más habitual para la codificación de voz. En el procesador digital de señales (DSP) TMS320C5535, así como en muchos otros sistemas empotrados, los enteros *int* son de 16 bits y sus instrucciones están optimizadas para la aritmética en este formato. La función Matlab diseñada **grabar_voz** se muestra en la figura 2.3.

```
function [senial]=grabar_voz(segundos,Fs)
% Devuelve "segundos" segundos de audio capturado en un canal (mono) de
% 16 bits y muestreado a "Fs" muestras por segundo.
recorder=audiorecorder(Fs,16,1);%16 bits, mono
record(recorder,segundos);
pause(segundos);
senial=getaudiodata(recorder);
end
```

Figura 2.3: Función Matlab *grabar_voz*.

En el anexo II se comenta el ruido de cuantificación.

Si tras llamar a la función pronunciamos una palabra ante el micrófono y representamos gráficamente el resultado, se obtiene la figura 2.4.

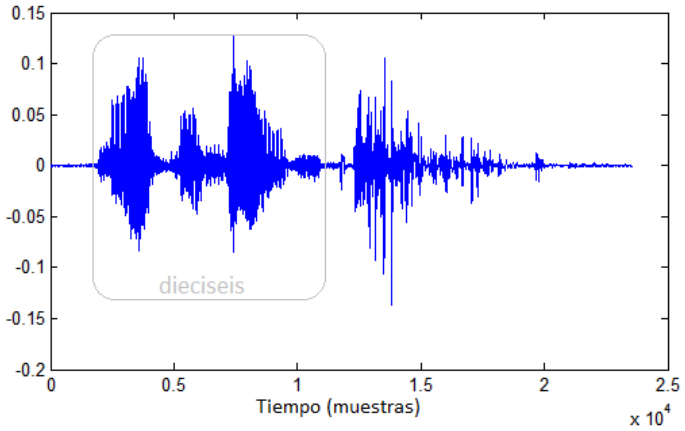

Figura 2.4: Representación temporal de una grabación en la que se pronuncia la palabra "dieciséis".

Este resultado lo utilizaré durante el resto de este subapartado como ejemplo para ilustrar las demás funciones.

2.1.2. Filtrado preénfasis

Debido a que la cavidad oral actúa como filtro paso bajo de primer orden, la señal sonora a su salida sufre una atenuación con una caída de 6 dB por octava. Con el filtro preénfasis se pretende corregir este efecto e igualar en lo posible el nivel de amplitudes en el espectro, atenuando las bajas frecuencias y enfatizando las altas. Utilizaré un filtro FIR de primer orden, de la forma (2.1) [5]. La representación de su respuesta en fase y magnitud se muestra en la figura 2.5.

$$H(z) = 1 - 0.95 \cdot z^{-1} \qquad (2.1)$$

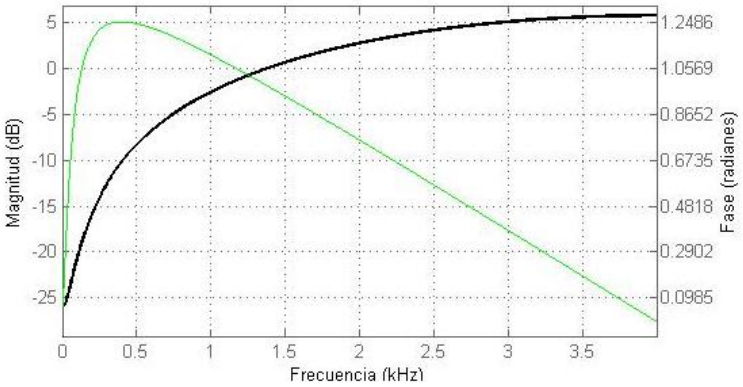

Figura 2.5: Respuesta en magnitud (negro) y fase (verde) del filtro preénfasis.

La función Matlab diseñada **filtro_preEnfasis** se muestra en la figura 2.6.

```
function [salida]=filtro_preEnfasis(x)
% Filtra pre-énfasis el vector "x".
[tamano, ~]=size(x);
salida=zeros(tamano-1,1);
for i=2:1:tamano
    salida(i)=x(i)-0.95*x(i-1);
end
```

Figura 2.6: función Matlab *filtro_preEnfasis*.

2.1.3. Detección de los extremos de la palabra

Con la grabación de audio digitalizada y filtrada, el siguiente paso es recortar el segmento que importa, el que contiene voz, y descartar el resto. Para ello se divide el segmento de audio en ventanas, y de cada ventana se evalúan dos características; la energía y los cruces por cero. El tamaño elegido de ventana debe ser lo suficientemente pequeño para obtener una buena resolución temporal, pero con un número de muestras suficiente para la frecuencia mínima que queremos analizar (200Hz). 25 milisegundos es el estándar recomendado [4]. Utilizaré un tamaño de 256 muestras, que es la potencia de dos más cercana al estándar; 32 milisegundos de audio.

La energía de una señal de voz es útil para distinguir los segmentos sonoros de los sordos y distinguir los tramos en los que solo hay ruido de los que también hay voz, pues el ruido tiene un nivel muy bajo de energía. En el caso de señales discretas se define como la suma al cuadrado del conjunto de las muestras.

$$E = \sum_{m=0}^{N-1} x(m)^2$$

Donde N es el número de muestras en la ventana.

El número de cruces por cero en un tramo de señal es el número de veces que ésta cambia de signo en ese tramo. Un número alto de éste valor indica que en la señal predominan las componentes de alta frecuencia, que son los segmentos que contienen voz. Los tramos donde solo hay ruido también presentan un gran número de cruces por cero, pero tienen muy baja energía.

$$z = \sum_{m=0}^{N-1} |sign[x(m)] - sign[x(m-1)]|$$

Combinando estas dos fórmulas se obtiene el llamado algoritmo coper, que nos da un valor con el que decidir si el tramo de ventana contiene o no señal de voz [6].

$$C = \sum_{m=0}^{N} |x(n) \cdot |x(n)| - x(n-1) \cdot |x(n-1)||$$

La figura 2.7 muestra la función **analisis_coper**.

```
function [resultado, ventanas]=analisis_coper(x, tamano_ventana)
%Divide el vector x en ventanas de tamaño tamano_ventana y obtiene un
%valor en proporción a la energía y cruces por cero para cada una.
%Devuelve el resultado como un vector de tamaño x y el valor en la
%posición de inicio de cada ventana. Devuelve el número de ventanas.
[~,tamano]=size(x);
ventanas=floor(tamano/tamano_ventana)-1;
resultado=zeros(1,tamano_ventana*ventanas);
for j=1:1:ventanas
    c=0;
    for i=1:1:tamano_ventana
        m=(j-1)*tamano_ventana+i+1;
        c=c + abs(x(m)*abs(x(m))-x(m-1)*abs(x(m-1)));
    end
    resultado((j-1)*tamano_ventana+1)=c;
end
```

Figura 2.7: función Matlab *analisis_coper.*

Si filtramos con el filtro preénfasis la señal de la figura 2.4 y la introducimos como vector de entrada a la función obtenemos, figura 2.8.

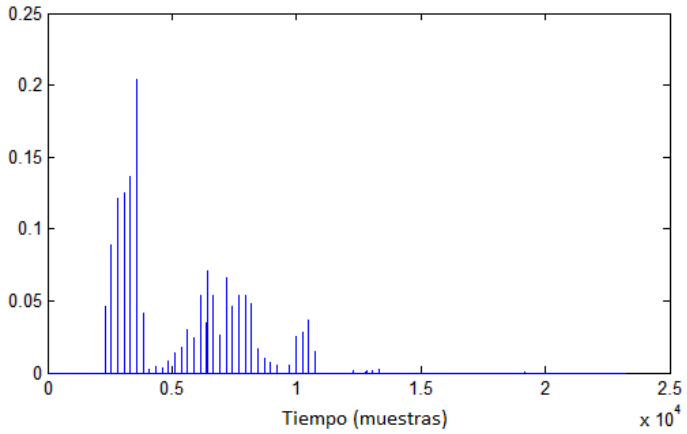

Figura 2.8: Análisis de energía y cruces por cero de un segmento de audio.

Se necesita establecer un umbral a partir del cual las ventanas que lo superen serán las sonoras, las que contengan voz. Este umbral se calcula experimentalmente mediante pruebas de voz y depende del tamaño elegido de ventana y de la ganancia del conversor A/D entre otros factores. Se puede obtener fácilmente con la representación gráfica en Matlab.

Una vez clasificadas las ventanas como sonoras o silenciosas podemos obtener los puntos de inicio y final de palabra. El punto de inicio se elige como la primera muestra de la primera ventana sonora. Encontrar el punto final es mas complicado, pues sabemos que una palabra puede contener breves silencios y fonemas no sonoros que estarán por debajo del umbral. Por ello se necesita establecer otro límite que determine con cuantas ventanas silenciosas consecutivas se considera que la palabra ha terminado. Este límite de ventanas se obtiene experimentalmente, mediante representación gráfica como en el caso anterior, y depende también del tamaño de ventana y de otros factores.

La figura 2.9 muestra el diagrama de flujo del algoritmo para la detección de los extremos del segmento de voz. Se comienza en el estado de reposo.

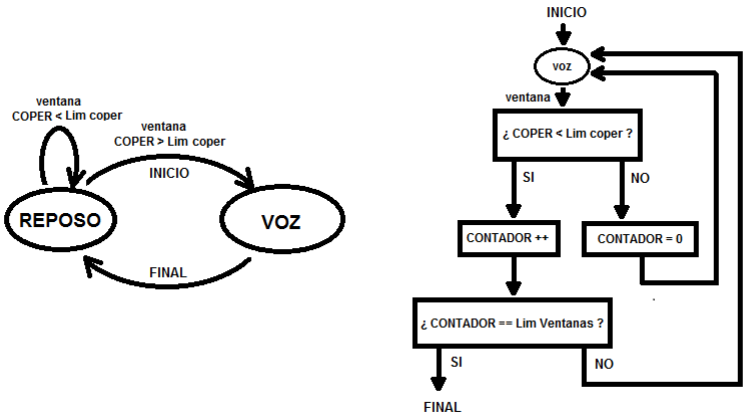

Figura 2.9: Algoritmo para la detección de los extremos del segmento de voz.

La función **recortar** se muestra en la figura 2.10. Devuelve el segmento de voz recortado a partir de los valores umbral y límite de ventanas silenciosas de entrada. También devuelve el tamaño en ventanas del segmento y el índice de la ventana de inicio. En caso de no detectar ninguna ventana

por encima del umbral (el audio no contiene voz) devuelve el valor cero en todas las variables de salida.

```
function [palabra,tamano,inicio]=...
                              recortar(x,tamano_ventana,limite,transiciones)
% Extrae del vector de audio de entrada "x" el tramo que contiene el
% segmento de voz, dividiéndolo en segmentos de tamaño "tamano_ventana".
% Las entradas "limite" y "transiciónes" son los umbrales de
% energía-cruces por cero y ventanas en silencio respectivamente.
% Devuelve el tamaño, y posición de inicio del segmento de voz.
[coper,ventanas]=analisis_coper(x', tamano_ventana);
inicio=0;%indice de la ventana de inicio
final=0;%indice de la ventana final
contador=0;%cuenta las ventanas bajo umbral dentro de la palabra
contador_cortes=0;%1 se ha detectado inicio, 2 se ha detectado final
for i=1:1:ventanas
    if coper((i-1)*tamano_ventana+1)>limite
        if contador_cortes==0
            inicio=i-1;
            contador_cortes=1;
        end
        contador=0;
    else
        if contador_cortes==1
            contador=contador+1;
            if contador==transiciones
                final=i-transiciones;
                contador_cortes=2;
            end
        end
    end
end
tamano=final-inicio;
if tamano==0
    palabra=0;
else
    palabra=x(inicio*tamano_ventana+1:final*tamano_ventana);
end
```

Figura 2.10: función Matlab *recortar*.

La representación del tramo de voz recortado que se obtiene a partir de la señal de la figura 2.4 se muestra en la figura 2.11.

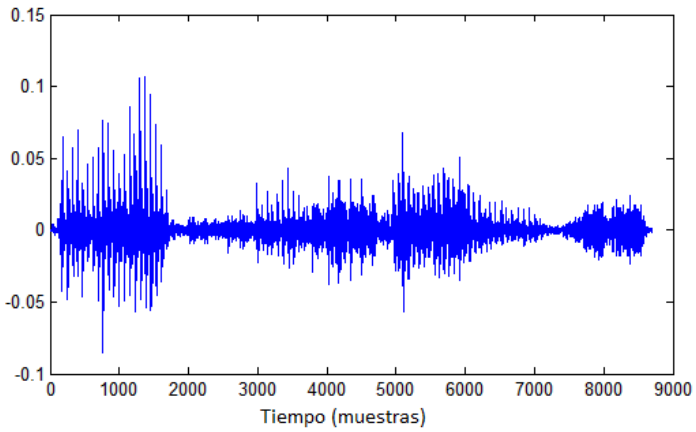

Figura 2.11: Segmento de voz filtrado y recortado.

2.1.4. Fragmentación en ventanas y multiplicación por la ventana de Hamming

La palabra filtrada y recortada ya está dispuesta para el análisis. El siguiente paso es dividirla en ventanas temporales de 256 muestras; 32 milisegundos cada ventana.

Extraer un segmento finito de muestras equivale a multiplicar la señal por un filtro rectangular, lo que produce una distorsión en el espectro resultante debido al efecto de discontinuidad en los extremos. Para reducir este efecto se multiplica el segmento por la ventana de Hamming [3]. En Matlab se utiliza el comando *hamming* para obtener un vector con los valores. La figura 2.12 muestra una ventana de Hamming de 256 muestras.

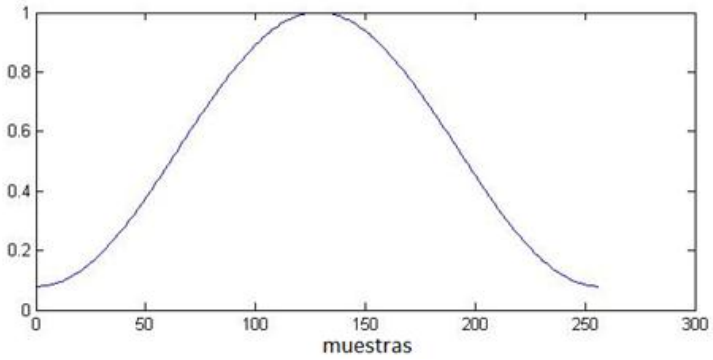

Figura 2.12: Ventana de Hamming de 256 muestras.

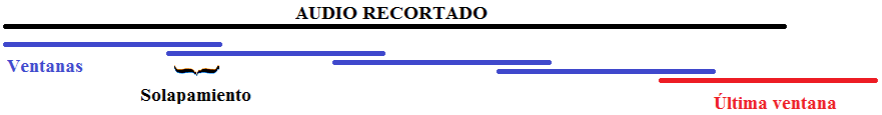

Figura 2.13: Enventanado del segmento de voz.

Conviene solapar las ventanas para obtener una mayor cantidad de información [3], como muestra la figura 2.13. Para simplificar el algoritmo, la última ventana se desprecia.

2.1.5. Diseño del banco de filtros de Mel

Para analizar la distribución espectral de la señal contenida en cada ventana, se divide el espectro en bandas de frecuencia y se miden los valores de energía en cada banda. Un banco de filtros de Mel es un conjunto de filtros triangulares solapados entre sí y distribuidos uniformemente en la escala de Mel [3].

La escala de Mel es una escala psicoacústica, construida en base a la percepción del oído humano de las diferencias entre distintos tonos. Es muy común en los algoritmos de reconocimiento del habla, donde se ha demostrado que su utilización aumenta la tasa de reconocimiento [17]. Una determinada distancia entre dos tonos se percibe más claramente si éstos son de baja que de alta frecuencia [4]. Por ello, el uso de ésta escala permite comprimir la información y extraer los datos más relevantes. Como referencia, 1.000 mels equivalen a 1.000 Hz, y el resto de frecuencias se calculan con la fórmula [3]:

$$m = 2595 \cdot log_{10}\left(1 + \frac{f}{700}\right)$$

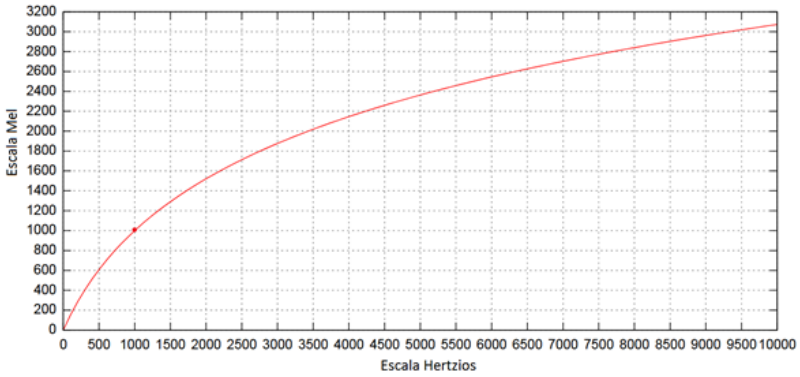

Figura 2.14: Escala de frecuencias de Mel frente a escala en Hertzios.

Para calcular los filtros se convierten las coordenadas abscisas de las aristas de los triángulos de mels a Hertzios, y la altura se divide por el tamaño de la base para igualar las áreas.

La función **matriz_filtros_mel** se muestra en la figura 2.15. El resultado es una matriz con los filtros triangulares almacenados en cada fila. Como entrada se introduce la frecuencia de muestreo, el tamaño de ventana, el número de filtros y la frecuencia máxima del último filtro. El tamaño en muestras de cada filtro será la mitad del tamaño de la ventana. Debido a que la función densidad espectral es simétrica, los filtros abarcan hasta la mitad de la frecuencia de muestreo. Se puede especificar la frecuencia máxima como parámetro de entrada a la función.

```
function [matriz]=matriz_filtros_mel(Fs,N,num_filtros,frec_max)
% Genera una matriz de banco de filtros Mel con cada filtro en una fila.
% Las entradas son la frecuencia de muestreo, tamaño de ventana, nùmero
% de filtros y la frecuencia máxima del último filtro.
matriz=zeros(num_filtros,N/2);
f_mel_max=2595*log10(1+frec_max/700);
intervalo=f_mel_max/(num_filtros+1);
inicio=zeros(num_filtros);
centro=zeros(num_filtros);
final=zeros(num_filtros);
for i=1:1:num_filtros  %Cálculo de la posición de los vértices
    inicio_mel=intervalo*(i-1)+1;
    inicio(i)=(700*(10^(inicio_mel/2595)-1))*N/Fs;
    centro_mel=inicio_mel+intervalo+1;
    centro(i)=700*(10^(centro_mel/2595)-1)*N/Fs;
    final_mel=centro_mel+intervalo+1;
    final(i)=700*(10^(final_mel/2595)-1)*N/Fs;
end
for i=1:1:num_filtros %Cálculo del resto de puntos
    for j=1:1:N/2
        if j>inicio(i)
            if j>centro(i)
                if j>final(i)
                else
                     matriz(i,j)=2*(j-final(i))/(centro(i)-final(i))/...
                                                    (final(i)-inicio(i));
                end
            else
                matriz(i,j)=2*(j-inicio(i))/(centro(i)-inicio(i))/...
                                                    (final(i)-inicio(i));
            end
        end
    end
end
```

Figura 2.15: función Matlab *matriz_filtros_mel*.

La figura 2.16 muestra un banco de 20 filtros con frecuencia máxima de 4.000 Hz y 128 muestras por filtro. El número de filtros es un parámetro que se obtiene mediante pruebas experimentales. Un valor grande aumenta la precisión en la estimación de la distribución de energía, pero también el número de cálculos y el tamaño de los datos a almacenar.

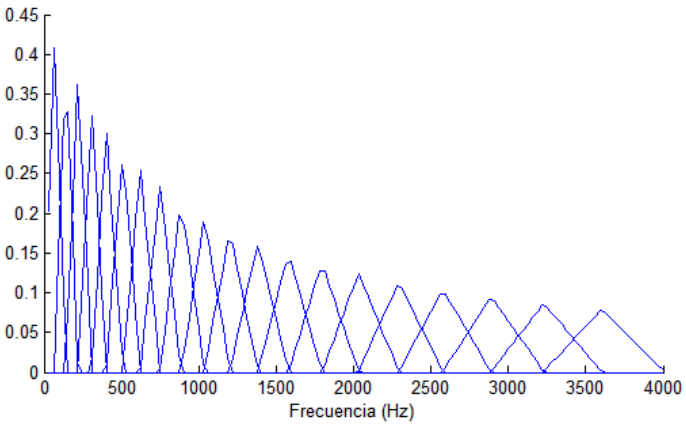

Figura 2.16: Banco de filtros de Mel.

2.1.6. Coeficientes Mel de la ventana

Los coeficientes de Mel de una ventana se obtienen multiplicando escalarmente las muestras de su función densidad espectral de energía por cada uno de los filtros del banco de filtros de Mel. Se obtiene un vector de coeficientes con información sobre la distribución de energía en el espectro. El número de coeficientes es igual al número de filtros en el banco y el valor de cada coeficiente es proporcional a la energía de la ventana en la banda de frecuencia abarcada por el correspondiente filtro [3]. La figura 2.17 muestra el diagrama explicativo del algoritmo.

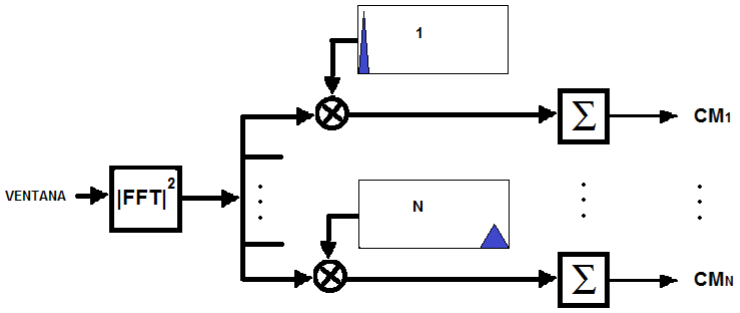

Figura 2.17: Obtención de los coeficientes de Mel con un banco de N filtros.

La función **coeficientes_mel** se muestra en la figura 2.18. Calcula los coeficientes de Mel de la ventana de entrada utilizando la matriz del banco de filtros de Mel y los devuelve en un vector.

```
function [coeficientes]=coeficientes_mel(x, matriz)
% Calcula los coeficientes de Mel de la ventana de entrada "x" utilizando
% la matriz de filtros Mel "matriz" y los devuelve en "coeficientes".
[~, tamano]=size(x);
[num_filtros, ~]= size(matriz);
X=abs(fft(x)).^2; %Densidad espectral de energia
coeficientes=zeros(1,num_filtros);
for i=1:1:num_filtros
    m=matriz(i,:);
    xx=X(1:tamano/2);
    coeficientes(i)=(m*xx');
end
```

Figura 2.18: función Matlab *coeficientes_mel*.

2.1.7. Coeficientes cepstrales de las frecuencias de Mel

El cepstrum de una señal se obtiene aplicando la transformada de Fourier a su espectro en escala logarítmica. La parte real contiene información del ritmo de cambio de las diferentes bandas de energía. El análisis cepstral es útil en el procesado de voz debido a que los impulsos periódicos de baja frecuencia producidos en las cuerdas vocales están convolucionados en el dominio del tiempo con el filtro del tracto vocal (modelo Source-Filter, anexo I). En el dominio frecuencial esta convolución es una multiplicación, que al aplicar la escala logarítmica se convierte en una suma de logaritmos. Esto permite la separabilidad lineal de las dos componentes [4].

La transformada del coseno es la parte real de la transformada de Fourier y se utiliza en procesado de señales de audio e imagen debido a su gran capacidad de compactación; es capaz de ofrecer en pocos coeficientes la información relativa a la distribución de energía del espectro de la señal. En reconocimiento de voz se utiliza un reducido número de los coeficientes resultantes, los que contienen la información de las formantes, y es contraproducente utilizar más de los necesarios. Descartamos el primer coeficiente que corresponde a la energía de la señal, y los últimos correspondientes a variaciones rápidas del espectro. Nos quedamos solo con la información relativa a su envolvente. Es necesario hacer pruebas experimentales para determinar el número que maximiza la tasa de reconocimiento [3]. La figura 2.19 muestra el diagrama explicativo del algoritmo.

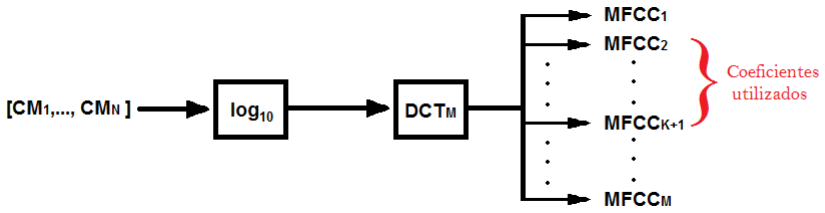

Figura 2.19: Obtención de k MFCCs a partir de N coeficientes Mel de la ventana.

La función **cepstrum** se muestra en la figura 2.20. Calcula los coeficientes cepstrales a partir del vector con los coeficientes de Mel como variable de entrada. El número de coeficientes calculado se introduce también como parámetro.

```
function [coef_cepstrales]=cepstrum(C, num_coeficientes)
% Calcula "num_coeficientes" coeficientes cepstrales del vector "C".
coef_cepstrales=zeros(1, num_coeficientes);
[~, NF]=size(C);
for i=1:1:num_coeficientes
    temp=0;
    for k=1:1:NF
        temp=temp+(log10(C(k))+2)*cos(i*(k-0.5)*pi()/NF);
    end
    coef_cepstrales(i)=temp;
end
```

Figura 2.20: función Matlab *cepstrum*.

Para reducir la influencia de las componentes de baja energía se aumentan los valores de la señal en escala logarítmica 2 unidades antes de aplicar la transformada del coseno [7].

La matriz MFCC contiene el conjunto de resultados que se obtienen tras el análisis cepstral de todas las ventanas que componen el fragmento de voz. Estos se ordenan cronológicamente, de forma que cada fila corresponde con una ventana (figura 2.21).

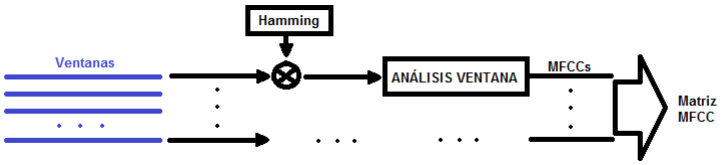

Figura 2.21: Obtención de la matriz MFCC.

La función **analisis_palabra** se muestra en la figura 2.22. Obtiene la matriz MFCC correspondiente al fragmento de voz de entrada. Como parámetros se introducen el tamaño de ventana, el porcentaje de solapamiento, el número de coeficientes cepstrales por ventana y la matriz de filtros de Mel. En el cálculo del número de ventanas a partir del tamaño total del fragmento se desprecia la última. Para enventanar se utiliza el comando Matlab *hamming* y se obtiene un vector del tamaño de la ventana con los valores del filtro de Hamming. Éste se multiplica por el vector con las muestras de la ventana.

```
function [MFCC]=analisis_palabra(x, tamano_ventana, solapamiento,...
                                                num_coef_cepstrales, matriz_mel)
% Obtiene la matriz MFCC del segmento de voz "x".
[tamano, ~]=size(x);
N_ventanas=floor(tamano/(tamano_ventana*(1-solapamiento))-1);
v_hamm=hamming(tamano_ventana); %Ventana de Hamming
MFCC=zeros(N_ventanas, num_coef_cepstrales);
for i=1:1:N_ventanas
    inicio=(i-1)*tamano_ventana*(1-solapamiento);
    tramo_filtrado=v_hamm.*x(inicio+1 : inicio + tamano_ventana);
    C=coeficientes_mel(tramo_filtrado', matriz_mel);
    K=cepstrum(C, num_coef_cepstrales);
    MFCC(i,:)=K;
end
```

Figura 2.22: función Matlab *analisis_palabra*.

Si se representa la matriz MFCC con 5 coeficientes cepstrales mediante el comando Matlab *imagesc* se obtiene la figura 2.23. Como entrada se ha utilizado el segmento de voz de la figura 2.11 y el banco de filtros de la figura 2.16.

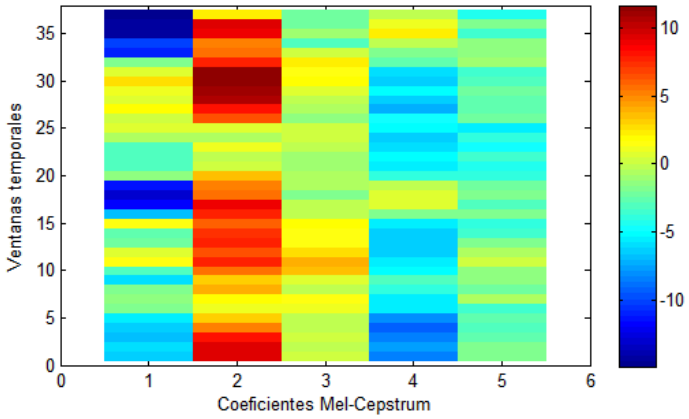

Figura 2.23: MFCCs de la palabra “dieciséis”.

Se observa que el fragmento se ha dividido en 37 ventanas temporales representadas en el eje Y, y los 5 coeficientes de cada ventana en el X.

Por último, la figura 2.24 muestra un ejemplo de utilización de todas las funciones diseñadas en este subapartado. Se obtiene la matriz MFCC a partir de una grabación de voz.

```
clear all
tiempo_grabacion=3;
Fs=8000;
muestras_ventana=256;
num_filtros=20;
limite_ECx0=0.002;
ventanas_bajo_limite=5;
solapamiento=0.25;
num_coef_cepstrales=5;
% Calculo matriz Mel
matriz_Mel=matriz_filtros_mel(Fs, muestras_ventana, num_filtros, Fs/2);
% Grabar audio
audio=grabar_voz(tiempo_grabacion, Fs);
% Obtener segmento de voz
audio_filtrado=filtro_preEnfasis(audio);
[palabra, ~, ~]=recortar(audio_filtrado, muestras_ventana,...
                                        limite_ECx0, ventanas_bajo_limite);
% Obtener matriz MFCC
MFCC=analisis_palabra(palabra, muestras_ventana, solapamiento,...
                                          num_coef_cepstrales, matriz_Mel);
```

Figura 2.24: procedimiento Matlab para obtener la matriz MFCC.

2.2. Codificación de los MFCCs

La codificación se define como el proceso de conversión de un sistema de datos origen a un sistema de datos destino, de forma que la información en destino sea equivalente a la de origen.

Un ejemplo es la codificación analógico-digital, en la que la señal analógica es convertida a un lenguaje binario y de este modo se hace posible su procesamiento, almacenamiento… por el sistema digital. Además de introducir un ruido de cuantificación, se produce una pérdida de información que depende del número de bits y de la frecuencia de muestreo de la conversión, pero el objetivo es que la señal digital represente lo mas fielmente posible a su equivalente analógica.

La codificación en el análisis de voz consiste en agrupar las ventanas con características fonéticas similares y asignarles un mismo código con el que trabajar en etapas posteriores. Es una forma mas de comprimir la información y traducirla a un formato manejable.

Los coeficientes MFCCs obtenidos de cada ventana temporal se pueden representar como vectores en un espacio de coordenadas n-dimensional, que parten del origen de coordenadas y terminan en un punto del espacio. La distancia euclídea entre dos puntos en un espacio n-dimensional se define como la longitud de la línea recta que une ambos puntos. Para los vectores $\mathbf{p} = (p_1, p_2, \ldots, p_n)$ y $\mathbf{q} = (q_1, q_2, \ldots, q_n)$:

$$d(\boldsymbol{p}, \boldsymbol{q}) = d(\boldsymbol{q}, \boldsymbol{p}) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2}$$

El objetivo es situar un determinado número de puntos, llamados codewords, en las posiciones del espacio que hacen que la suma total de distancias de cada vector a su codeword mas cercano sea mínima. Este conjunto de codewords se denomina codebook. El grupo de vectores cuyo elemento del codebook más cercano es el mismo se codifican con el mismo código. Entonces, cuantificar un vector consiste en encontrar su codeword mas cercano y asignarle el código correspondiente a ese codeword.

La distancia entre un vector y su codeword más cercano se denomina error por distorsión (o error de cuantificación), y la media del error por distorsión de todos los vectores codificados es el error de cuantificación medio [3].

2.2.1. Construcción del codebook mediante el algoritmo K-means

Para construir el codebook se necesita disponer de un gran número de vectores, cuantos más mejor, que representen lo mejor posible los diferentes fonemas. El proceso de obtención de estos vectores se llama entrenamiento del codebook. Consiste en repetir muchas veces las palabras que queremos que el sistema reconozca, obtener los coeficientes cepstrales de sus ventanas y almacenarlos en una lista. A partir de esta lista se construye el codebook. Llamaremos vector de observación a los coeficientes cepstrales obtenidos de una ventana [3].

Algoritmo K-means

El algoritmo K-means es un método de agrupamiento utilizado en el análisis de conjuntos. Consigue repartir de forma eficiente el conjunto de observaciones en K grupos. Para un conjunto de vectores, cada grupo está contenido en un subespacio del espacio euclídeo llamado clúster, y se caracteriza por un centroide que es el punto medio del clúster. Los grupos están distribuidos de tal manera que cada observación pertenece al grupo de su centroide más cercano.

El algoritmo trata de encontrar las posiciones de los centroides que minimizan el error de cuantificación de los vectores de observación obtenidos en el entrenamiento. Estos centroides serán los codewords del codebook.

Los pasos con los que se obtiene un codebook de M elementos a partir de una lista de N vectores de observación son los siguientes [3]:

1. Inicialización: Se eligen aleatoriamente M de los N vectores y se colocan los centroides iniciales en sus posiciones.
2. Búsqueda del codeword mas cercano: Para cada vector de observación se calcula la distancia a cada centroide y se agrupa con el mas cercano.
3. Actualización del centroide*:* Se recalculan las posiciones de cada centroide colocándolo en el centro de masas de su respectivo grupo.

4. Iteración: Se repiten los pasos 2 y 3 hasta llegar a un estado en el que el codebook no cambia, o el error medio de cuantificación está por debajo de un determinado umbral.

La función **algoritmo_Kmeans** se muestra en la figura 2.25. Efectúa una iteración del algoritmo K-means; los pasos 2 y 3. A partir de la matriz con los vectores de observación y el codebook inicial, agrupa las observaciones y recalcula la posición de cada elemento del codebook, colocándolo en el centro de masas de su agrupación.

```
function [codebook, error_total]=algoritmo_Kmeans(vectores, codebook)
% Obtiene el codebook y el error de cuantificación a partir de un
% grupo de vectores de observación y el codebook inicial.
[tam_v, coordenadas]=size(vectores);
[tam_c, ~]=size(codebook);
matriz=zeros(tam_c*tam_v, coordenadas);
elementos=zeros(tam_c, 1);
error_total=0;
for j=1:1:tam_v %Obtiene la matriz con los clusters
    [error, indice]=error_por_distorsion(vectores(j,:), codebook);
    error_total=error_total+error;
    elementos(indice)=elementos(indice)+1;
    matriz((indice-1)*tam_v+elementos(indice),:)=vectores(j, :);
end
for i=1:1:tam_c %Mueve cada codeword al centro de masas del cluster
    codebook(i, :)=...
                mean(matriz((i-1)*tam_v+1:(i-1)*tam_v+elementos(i), :),1);
end
```

Figura 2.25: función Matlab algoritmo_Kmeans.

La función **error_por_distorsion** se muestra en la figura 2.26. Obtiene la distancia del codeword mas cercano al vector de observación de entrada y el índice de la fila que contiene el codeword en la matriz del codebook.

```
function [distancia, indice]=error_por_distorsion(vector, codebook)
% Calcula la distancia euclídea de un vector a su codeword mas cercano.
% Obtiene el índice del codeword.
[tam_c, dim]=size(codebook);
D=zeros(1, tam_c);
for i=1:1:tam_c
    temp=0;
    for j=1:1:dim
        temp=temp+(vector(j)-codebook(i, j)).^2;
    end
    D(i)=sqrt(temp);
end
[distancia, indice]=min(D);
```

Figura 2.26: función Matlab *error_por_distorsion*

Para obtener las posiciones de los codewords iniciales utilizo la función **codebook_inicial** que se muestra en la figura 2.27. Con el comando Matlab *randperm(N)* se construye un vector con números del 1 al N ordenados aleatoriamente, siendo N el número de vectores de observación, y se toman los M primeros como índices de los vectores donde se sitúan las posiciones de los centroides iniciales.

```
function [codebook]=codebook_inicial(vectores, N_codewords)
% Selecciona "N_codewords" vectores de observación aleatorios
% como codewords iniciales.
[tam_v, coordenadas]=size(vectores);
p=randperm(tam_v);
codebook=zeros(N_codewords, coordenadas);
for i=1:1:N_codewords
   codebook(i, :)=vectores(p(i), :);
end
```

Figura 2.27: función Matlab *codebook_inicial*

Como ejemplo para mostrar el funcionamiento del algoritmo en un espacio bidimensional se utiliza el procedimiento de la figura 2.28. El resultado es un codebook de 16 elementos a partir de 1.000 vectores de observación. El ejemplo puede extenderse a cualquier número de dimensiones y el procedimiento sería el mismo.

```
clear all
dimensiones=2;
N_codewords=16;
N_iteraciones=50;
vectores=rand(1000,dimensiones);
% 1. Inicialización
codebook=codebook_inicial(vectores, N_codewords);
% 2. Búsqueda del codeword mas cercano
% 3. Actualización del centroide
[codebook, error]=algoritmo_Kmeans(vectores, codebook);
error_total=error;
% 4. Iteración
for i=1:1:N_iteraciones
    [codebook, error]=algoritmo_Kmeans(vectores, codebook);
    error_total=[error_total error];
end
```

Figura 2.28: procedimiento Matlab para el algoritmo K-means.

La representación gráfica del resultado se muestra en la figura 2.29. Los vectores de observación son los puntos verdes y el codebook los azules. La primera gráfica muestra la distribución del codebook inicial tras el paso 1 y la segunda tras 50 iteraciones del algoritmo K-means.

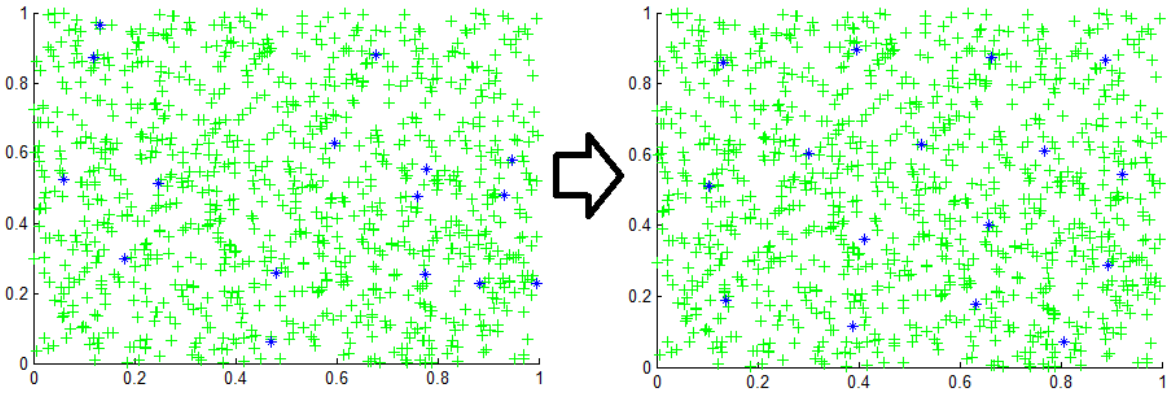

Figura 2.29: Codebook inicial y final tras 50 iteraciones.

El error medio de cuantificación se muestra en la figura 2.30. Su valor disminuye con cada iteración hasta llegar a un valor estable donde el codebook ya no cambia.

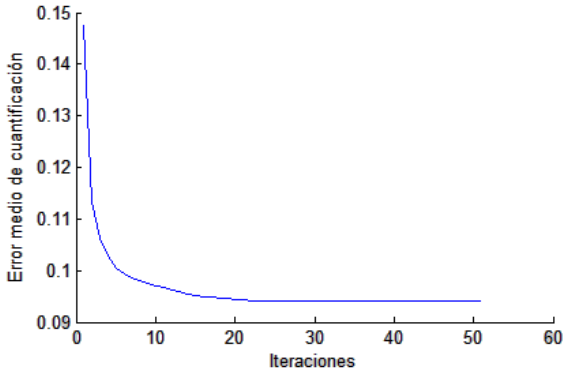

Figura 2.30: Evolución del error medio de cuantificación.

2.2.2. Codificación de ventanas con el codebook

Una vez está construido el codebook, el proceso de codificación de los MFCC de una nueva señal de voz consiste en calcular la distancia de cada vector de observación a cada codeword, y asignarle el valor numérico del índice del codeword cuya distancia sea menor. Éste será el valor de las ventanas codificadas. La palabra codificada será un vector con estos valores ordenados cronológicamente ($\mathbf{O}=O_1, O_2 \ldots O_T$, figura 2.31).

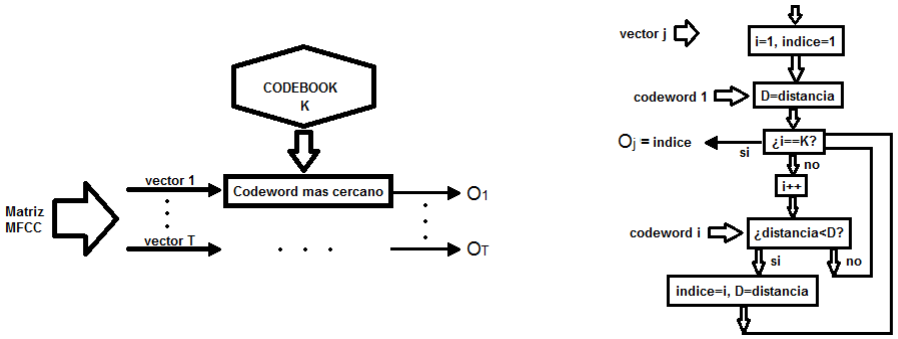

Figura 2.31: Algoritmo para la codificación de la matriz MFCC con un codebook de tamaño K.

La función **codificar_palabra** se muestra en la figura 2.32. Codifica la matriz MFCC de la palabra de entrada con el codebook, utilizando la función *error_por_distorsion*.

```
function [palabra_codificada]=codificar_palabra(vectores_palabra, codebook)
% Codifica la matriz "vectores_palabra" con el codebook.
[tam_v, ~]=size(vectores_palabra);
palabra_codificada=zeros(1, tam_v);
for i=1:1:tam_v
    [~, palabra_codificada(i)]=...
                    error_por_distorsion(vectores_palabra(i,:), codebook);
end
```

Figura 2.32: función Matlab *codificar_palabra*.

2.3. Aplicación de los modelos ocultos de Markov

Un segmento de voz dividido en T ventanas codificadas es un conjunto de T símbolos $\mathbf{V}=v_1, v_2,...v_k$ obtenidos durante las observaciones $\mathbf{O}=O_1, O_2...O_T$, que puede ser utilizado para entrenar el modelo oculto de Markov correspondiente a la palabra pronunciada, o para reconocer esa palabra por los modelos ya entrenados. Para el entrenamiento y reconocimiento de los HMM se emplean los algoritmos forward-backward y Baum-Welch [2].

En el anexo III se hace una introducción a los HMM.

2.3.1. Algoritmo forward-backward

El algoritmo Forward tiene como objetivo calcular $P(\mathbf{O}|\lambda)$, la probabilidad de obtener la secuencia de observación $\mathbf{O}$ por el modelo $\lambda=(A, B, \pi)$, de manera eficiente. Para ello utiliza un método de cálculo inductivo introduciendo una nueva variable; la variable forward $\alpha_t(i)$. Se define como la probabilidad de que se haya obtenido la secuencia de observaciones $O_1, O_2...O_t$ hasta el instante t y que el sistema se encuentre en el estado S_i en ese último instante, dado el modelo λ:

$$\alpha_t(i) = P(O_1 O_2 \dots O_t, q_t = S_i|\lambda)$$

El resultado es una matriz NxT; para un modelo de N estados con T observaciones. Cada columna se calcula a partir del resultado obtenido para la columna anterior, excepto la primera que se calcula mediante la observación inicial:

$$\alpha_1(i) = \pi_i \cdot b_i(O_1), \qquad 1 \leq i \leq N$$

El resto se calculan de forma inductiva, desde la segunda columna hasta la última, según la fórmula:

$$\alpha_{t+1}(j) = \left[\sum_{i=1}^{N} \alpha_t(i) a_{ij}\right] b_j(O_{t+1}), \qquad 1 \leq t \leq T-1, \;\; 1 \leq j \leq N$$

Una vez calculada la última columna, la probabilidad buscada se obtiene mediante la suma de sus elementos:

$$P(\boldsymbol{O}|\lambda) = \sum_{i=1}^{N} \alpha_T(i)$$

Este método se utiliza para resolver el problema de reconocimiento [2]. La matriz forward también es necesaria en el proceso de entrenamiento. La figura 2.33 muestra el código Matlab de la función **algoritmo_forward** diseñada.

```
function [Alfa, prob]=algoritmo_forward(Pi, A, B, O)
% Calcula la matriz forward y P(O|?). Las entradas son las matrices Pi, A
% y B del modelo y el vector de observaciones.
[N_estados, ~]=size(A);
[~, N_observaciones]=size(O);
Alfa=zeros(N_estados, N_observaciones);
Alfa(:, 1)=Pi.*B(O(1), :)'; %Primera columna
for t=2:1:N_observaciones %Inducción
    temp=Alfa(:, t-1)'*A;
    Alfa(:, t)=(temp.*B(O(t), :))';
end
prob=sum(Alfa(:, N_observaciones));
```

Figura 2.33: función Matlab *algoritmo_forward*.

La variable Backward $\beta_t(i)$ se define como la probabilidad de haber obtenido las observaciones O_{t+1}, O_{t+2},…,O_T habiendo partido del estado S_i en el instante t, dado el modelo λ:

$$\beta_t(i) = P(O_{t+1}O_{t+2} \ldots O_T | q_t = S_i, \lambda)$$

Las variables forman una matriz, del mismo tamaño que la matriz forward, con los instantes de tiempo ordenados en columnas. El cálculo de las columnas es similar pero en orden inverso, partiendo de la última hasta la primera.

La última columna se inicia dando valor uno a todos los elementos.

$$\beta_T(i) = 1, \qquad 1 \leq i \leq N$$

El resto de columnas se calculan de forma inductiva.

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij} b_j(O_{t+1}) \beta_{t+1}(j), \qquad t = T-1,\ T-2, \ldots, 1$$

El resultado es la matriz Backward, utilizada en el posterior proceso de entrenamiento [2]. La figura 2.34 muestra la función Matlab **algoritmo_backward** diseñada.

```
function [Beta]=algoritmo_backward(A, B, O)
% Calcula la matriz backward. Las entradas son las matrices A y B del
% modelo y el vector de observaciones.
[N_estados, ~]=size(A);
[~, N_observaciones]=size(O);
Beta=zeros(N_estados, N_observaciones);
Beta(:, N_observaciones)=ones(N_estados, 1);
for t=N_observaciones-1:-1:1
    temp=B(O(t+1), :).*Beta(:, t+1)';
    Beta(:, t)=temp*A';
end
```

Figura 2.34: función Matlab *algoritmo_backward*.

2.3.2. Algoritmo Baum-Welch

El algoritmo Baum-Welch es el método utilizado para resolver el problema del entrenamiento del HMM. El objetivo del algoritmo es ajustar los parámetros del modelo (A, B y π) para maximizar $P(\mathbf{O}|\lambda)$.

El método es iterativo; a partir de unos parámetros iniciales se obtienen unos parámetros finales ajustados a la secuencia de observación. Éstos vuelven a iterarse con la misma o con una nueva secuencia generada por el mismo modelo. Dado un número finito de observaciones no existe un método analítico que garantice la convergencia de los parámetros, el algoritmo Baum-Welch se limita a encontrar un máximo local en la probabilidad $P(\mathbf{O}|\lambda)$ [2]. La forma de proceder entonces es observar la tendencia de los valores de la probabilidad con cada iteración y detenerse al llegar a un valor máximo estable.

Se define una nueva variable, la variable épsilon $\xi_t(i, j)$, como la probabilidad de estar en el estado S_i en el instante t y en el S_j en el instante t+1, dada una secuencia de observación $\mathbf{O}$ y un modelo λ.

$$\xi_t(i,j) = P(q_t = S_i, q_{t+1} = S_j | \boldsymbol{O}, \lambda)$$

Su valor se obtiene mediante las variables forward y backward [2].

$$\xi_t(i,j) = \frac{\alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)}{P(\boldsymbol{O}|\lambda)}$$

El conjunto de valores se ordena como una matriz tridimensional de tamaño Nx(T-1)x(T-1). La figura 2.35 muestra el código Matlab de la función **variable_Epsilon** diseñada. Utiliza las funciones *algoritmo_forward* y *algoritmo_backward*.

```
function [Epsilon]=variable_Epsilon(Pi, A, B, O)
% Calcula la matriz tridimensional Epsilon. Las entradas son las
% matrices Pi, A y B del modelo y el vector de observaciones.
[N_estados, ~]=size(A);
[~, N_observaciones]=size(O);
[Alfa, prob]=algoritmo_forward(Pi, A, B, O);
[Beta]=algoritmo_backward(A, B, O);
Epsilon=zeros(N_observaciones-1, N_estados, N_estados);
for t=1:1:N_observaciones-1
    temp=B(O(t+1), :).*Beta(:, t+1)';
    temp=Alfa(:, t)*temp;
    Epsilon(t, :, :)=temp.*A;
end
Epsilon=Epsilon/prob;
```

Figura 2.35: función Matlab *variable_Epsilon*.

También es necesario definir una última variable; la variable gamma $\Upsilon_t(i)$. Se define como la probabilidad de estar en el estado S_i en el instante t, dada una secuencia de observación **O** y un modelo λ.

$$\gamma_t(i) = P(q_t = S_i|\boldsymbol{O},\lambda)$$

El valor se obtiene también mediante las variables forward y backward [2].

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{P(\boldsymbol{O}|\lambda)}$$

Los valores se ordenan como una matriz de tamaño NxT. La figura 2.36 muestra el código Matlab de la función **variable_Gamma** diseñada. Utiliza las funciones *algoritmo_forward* y *algoritmo_backward*.

```
function [Gamma]=variable_Gamma(Pi, A, B, O)
% Calcula la matriz Gamma. Las entradas son las matrices Pi, A y B
% del modelo y el vector de observaciones.
[Alfa, prob]=algoritmo_forward(Pi, A, B, O);
[Beta]=algoritmo_backward(A, B, O);
Gamma=Alfa.*Beta;
Gamma=Gamma/prob;
```

Figura 2.36: función Matlab *variable_Gamma*.

Mediante las matrices épsilon y gamma, calculadas a partir del modelo $\lambda=(A, B, \pi)$ y la secuencia de observaciones $\mathbf{O}$, se calcula un nuevo modelo $\bar{\lambda}=(\bar{A}, \bar{B}, \bar{\pi})$ tal que $P(\mathbf{O}|\bar{\lambda}) > P(\mathbf{O}|\lambda)$. Para ello se utilizan las fórmulas [2]:

$$\bar{\pi}_i = \gamma_1(i), \qquad \bar{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)}, \qquad \bar{b}_j(k) = \frac{\sum_{t=1,con\ O_t=v_k}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)}$$

Conviene aclarar que el numerador de la tercera fórmula se calcula como la suma, solamente, de los elementos de la fila j de la matriz gamma correspondientes a los instantes en que se ha observado el símbolo v_k. Es decir, el número esperado de veces en el estado j y observando el símbolo v_k.

La figura 2.37 muestra el código Matlab de la función **algoritmo_BaumWelch** diseñada. Utiliza las funciones *variable_Epsilon* y *variable_Gamma*, y el comando *sum* de Matlab para sumar elementos de una matriz.

```
function [Pi, A, B]=algoritmo_BaumWelch(Pi, A, B, O)
% Recalcula las matrices Pi, A y B. Las entradas son las matrices Pi, A y B
% del modelo y el vector de observaciones.
[~, N_observaciones]=size(O);
[N_estados, ~]=size(A);
[N_simbolos, ~]=size(B);
Epsilon=variable_Epsilon(Pi, A, B, O);
Gamma=variable_Gamma(Pi, A, B, O);
%Cálculo de matriz Pi
Pi=Gamma(:,1);
%Cálculo de matriz A
for i=1:1:N_estados-1
    temp=Gamma(i, :);
    den=sum(temp(1: end-1));
    if den>0
        for j=1:1:N_estados
            num=sum(Epsilon(:, i, j));
            A(i, j)=num/den;
        end
    end
end
%Cálculo de matriz B
for j=1:1:N_estados
    den=sum(Gamma(j, :));
    if den>0
        for k=1:1:N_simbolos
            num=0;
            for t=1:1:N_observaciones
                if O(t)==k
                    num=num+Gamma(j, t);
                end
            end
            B(k, j)=num/den;
        end
    end
end
```

Figura 2.37: función Matlab *algoritmo_BaumWelch*.

2.3.3. Entrenamiento de los HMM

Para el entrenamiento del HMM correspondiente a una palabra del vocabulario es necesario obtener un gran número de matrices MFCC codificadas de esa palabra, pronunciada por uno o varios hablantes. Estas matrices codificadas se almacenan en una lista, y cada una de ellas se utiliza como secuencia de observación $\mathbf{O}_i$ para recalcular los parámetros del modelo, hasta obtener el modelo final con $P(\mathbf{O}_i|\lambda)$ media máxima.

Modelo inicial

Para comenzar el cálculo es necesario establecer unos valores iniciales del modelo. Debido a las restricciones impuestas por ser un proceso temporal, explicadas en el anexo III, las matrices iniciales tienen la forma que se muestra en la figura 2.38.

$$\Pi = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \qquad A = \begin{bmatrix} 0.\overline{3} & 0.\overline{3} & 0.\overline{3} & 0 & \dots & 0 \\ 0 & 0.\overline{3} & 0.\overline{3} & 0.\overline{3} & 0 & \vdots \\ \vdots & 0 & 0.\overline{3} & 0.\overline{3} & 0.\overline{3} & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \dots & 0 & 0.\overline{3} & 0.\overline{3} & 0.\overline{3} \\ \vdots & \dots & \dots & 0 & 0.5 & 0.5 \\ 0 & \dots & \dots & \dots & 0 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 1/k & \dots & 1/k \\ \vdots & \ddots & \vdots \\ 1/k & \dots & 1/k \end{bmatrix}$$

Figura 2.38: Matrices iniciales de un HMM de k símbolos y 3 saltos de estado permitidos.

Los valores de la matriz Π son debidos a que se comienza siempre en el primer estado.

Los elementos de cada fila de la matriz A suman siempre uno. Sus valores se deben a la restricción impuesta al número de saltos de estado, normalmente se limita a 3. Damos la misma probabilidad a todos los saltos desde cada estado.

En la matriz B damos a todos los símbolos la misma probabilidad de observación en cada estado, y sus filas también suman siempre uno [2].

La figura 2.39 muestra el código Matlab **calculo_HMM_inicio** de la función diseñada para calcular las matrices iniciales.

```
function [Pi, A, B]=calculo_HMM_inicio(N_estados, N_simbolos, saltos_max)
% Calcula las matrices Pi, A y B iniciales. Las entradas son el número de
% estados, el número de símbolos y los saltos de estado máximos permitidos.
%Cálculo de matriz Pi inicial
Pi=zeros(N_estados, 1);
Pi(1)=1;
%Cálculo de matriz A inicial
A=zeros(N_estados);
fila=zeros(1, N_estados);
fila(1:saltos_max)=ones(1, saltos_max)/saltos_max;
for i=1:1:N_estados
    A(i, :)=fila;
    fila=[0 fila(1:N_estados-1)];
end
for i=N_estados-saltos_max+2:1:N_estados
    for j=i:1:N_estados
        A(i, j)=1/(N_estados-i+1);
    end
end
%Cálculo de matriz B inicial
B=ones(N_simbolos, N_estados)/N_simbolos;
```

Figura 2.39: función Matlab *calculo_HMM_inicio*.

Iteración de las secuencias de observación

Una iteración en el entrenamiento del HMM consiste en calcular un modelo final $\bar{\lambda}$, a partir de un modelo inicial λ y de la lista de secuencias de observaciones $\mathbf{O_1}$, $\mathbf{O_2}$... Para cada secuencia $\mathbf{O_i}$ se aplica el algoritmo Baum-Welch al modelo de entrada λ y se calcula un nuevo modelo $\bar{\lambda}_i$. Los elementos de las matrices del modelo final $\bar{\lambda}$ son la media aritmética de los correspondientes elementos de las matrices de los modelos $\bar{\lambda}_i$.

La figura 2.40 muestra el código Matlab **iteración_HMM** de la función diseñada. Calcula el modelo con cada secuencia de observación de la lista a partir del modelo inicial de entrada. Debido a que las secuencias no tienen un tamaño determinado, la lista de secuencias de observación de entrada está almacenada en formato *cell*, con un vector de observaciones por célula. Mediante el comando Matlab *cell2mat* se convierte cada célula a formato vector. Utiliza la función *algoritmo _BaumWelch* para obtener un nuevo modelo a partir de cada secuencia, y la función *algoritmo _forward* para calcular las probabilidades $P(\mathbf{O}_i|\lambda)$. Como salida devuelve el nuevo modelo y la probabilidad $P(\mathbf{O}_i|\lambda)$ media.

```
function [Pi, A, B, prob]=iteracion_HMM(Pi_i, A_i, B_i, O_cells)
% Recalcula las matrices Pi, A y B y obtiene la P(O|modelo) media.
% Las entradas son las matrices Pi, A y B del modelo inicial y los vectores
% de observaciones de la palabra almacenados en cells.
[repeticiones, ~]=size(O_cells);
Pi=zeros(size(Pi_i));
A=zeros(size(A_i));
B=zeros(size(B_i));
prob=0;
for n=1:1:repeticiones
    O=cell2mat(O_cells(n));
    [Pi_temp, A_temp, B_temp]=algoritmo_BaumWelch(Pi_i, A_i, B_i, O);
    [~, prob_temp]=algoritmo_forward(Pi_temp, A_temp, B_temp, O);
    Pi=Pi+Pi_temp;
    A=A+A_temp;
    B=B+B_temp;
    prob=prob+prob_temp;
end
Pi=Pi/repeticiones;
A=A/repeticiones;
B=B/repeticiones;
prob=prob/repeticiones;
```

Figura 2.40: función Matlab *iteracion_HMM*.

El modelo final obtenido vuelve a iterarse con la misma lista de secuencias de observación, utilizándolo esta vez como modelo inicial. Este proceso se repite indefinidamente, hasta obtener un buen modelo λ que utilizaremos para la palabra del vocabulario.

Como ejemplo de utilización de las funciones diseñadas, y para comprobar su correcto funcionamiento, se ha diseñado el procedimiento de la figura 2.41. En él se utiliza el comando Matlab *hmmgenerate* que genera secuencias de observación aleatorias a partir de las matrices A y B de un modelo real determinado. El objetivo es entrenar un HMM a partir de estas observaciones que resulte una estimación del modelo real que las ha generado. La función *hmmgenerate* comienza por defecto en el instante $t=0$ y el estado S_0, y genera la primera observación después del salto, por ello ha sido necesario añadir una fila y columna iniciales a la matriz A con $a_{01}=1$, y una fila a la matriz B como se muestra en la figura 2.41. Esto hace que en el instante $t=0$ se salte necesariamente al estado S_1.

```
clear all
N_observaciones=10; %Tamaño de cada serie de observaciones
N_repeticiones=100; %Número de series de observaciones
N_iteraciones=100;  %Iteraciones del algoritmo Baum-Welch

A_real=[0,1,0,0,0;  %Matriz transiciones Real
        0,          0.5, 0.5,  0.0,  0  ;
        0,          0,   0.5,  0.5,  0  ;
        0,          0,   0,    0.5,  0.5;
        0,          0,   0,    0,    1  ];
B_real=[0,0,0,1;    %Matriz observaciones Real
                    1,   0,    0,    0  ;
                    0,   1,    0,    0  ;
                    0,   0,    1,    0  ;
                    0,   0,    0,    1  ];
O={};
for i=1:1:N_repeticiones %Produce las sec. de observaciones
    [SEQ, STATES]=hmmgenerate(N_observaciones, A_real, B_real);
    O=[O; {SEQ}]; %#ok<AGROW>
end

[Pi, A, B]=calculo_HMM_inicio(4, 4, 3);%Genera matrices iniciales

vector_prob=zeros(N_iteraciones,1);
for i=1:1:N_iteraciones %Obtiene la estimación del modelo
    [Pi, A, B, vector_prob(i)]=iteracion_HMM(Pi, A, B, O);
end
```

Figura 2.41: procedimiento Matlab para el entrenamiento de HMM.

El resultado son las matrices del HMM entrenadas que se muestran en la figura 2.42, junto con las matrices iniciales a su izquierda. Estas matrices se han generado con 100 secuencias de observaciones iteradas 100 veces. Si comparamos los elementos de las matrices obtenidas con las matrices reales que se han definido en el código de la figura 2.41, podemos comprobar como los valores se aproximan. Se obtiene una mejor estimación cuanto mayor sea el número de secuencias de observación utilizadas.

	1	2	3	4
1	0.3333	0.3333	0.3333	0
2	0	0.3333	0.3333	0.3333
3	0	0	0.5000	0.5000
4	0	0	0	1

A

	1	2	3	4
1	0.2668	0.7332	1.4670e-218	0
2	0	0.3043	0.6957	1.4082e-89
3	0	0	0.3033	0.6967
4	0	0	0	1

	1	2	3	4
1	0.2500	0.2500	0.2500	0.2500
2	0.2500	0.2500	0.2500	0.2500
3	0.2500	0.2500	0.2500	0.2500
4	0.2500	0.2500	0.2500	0.2500

B

	1	2	3	4
1	1	6.5579e-42	0	0
2	1.1924e-49	1	0.0170	0.0104
3	0	5.0569e-57	0.9830	0.1305
4	0	0	7.8014e-68	0.8591

Figura 2.42: Matrices del HMM iniciales y entrenadas.

Por último queda mostrar cómo evoluciona la probabilidad $P(\mathbf{O}_i|\lambda)$ media con las iteraciones. El resultado obtenido del ejemplo anterior se muestra en la figura 2.43. Esta curva alcanza un máximo en el que el modelo ya no cambia al iterarlo. La representación gráfica de la variación de este valor con cada iteración permite estimar el momento en que el modelo ya está entrenado; en este caso con 10 iteraciones.

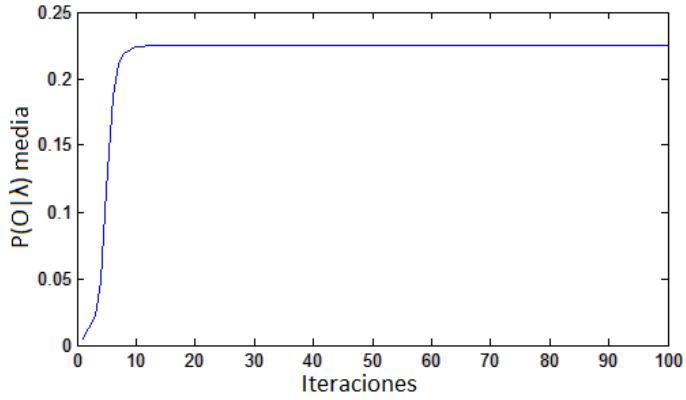

Figura 2.43: Variación de $P(\mathbf{O}_i|\lambda)$ media con las iteraciones.

2.3.4. Aplicación de los HMM para el reconocimiento

Una vez entrenados los HMM de todas las palabras del vocabulario puede hacerse el reconocimiento de una nueva secuencia de observaciones. Dada una secuencia de observaciones $\mathbf{O}$, el reconocimiento consiste en calcular $P(\mathbf{O}|\lambda_i)$ para todos los modelos, y el modelo que genere el valor mas alto será el de la palabra elegida como resultado del reconocimiento [2]. Es decir, se va a elegir cuál es el modelo con el que es más probable haber generado esa secuencia de observación.

El método para calcular estas probabilidades utiliza la matriz forward. El cálculo de esta matriz genera valores numéricos muy pequeños, más pequeños cuanto mayor sea el número de

observaciones en la secuencia, y en sistemas en coma fija pueden fácilmente producir underflow. Por ello es necesario escalar (o normalizar) los coeficientes a medida que se van calculando.

El método empleado es, después de calcular una columna de la matriz forward, recalcular sus elementos multiplicándolos por un coeficiente de escala c_t antes de calcular la siguiente columna.

$$c_t = \frac{1}{\sum_{i=1}^{N} \alpha_t(i)} \qquad (2.2)$$

Los coeficientes de la matriz forward escalados se definen como

$$\hat{\alpha}_t(i) = c_t \alpha_t(i) = \left[\prod_{s=1}^{t} c_s\right] \alpha_t(i) \qquad (2.3)$$

Se utiliza un método de cálculo inductivo. Para un determinado t, los coeficientes se obtienen mediante los coeficientes calculados y escalados en el instante anterior t-1.

$$\alpha_t(i) = \sum_{j=1}^{N} \hat{\alpha}_{t-1}(j) a_{ij} b_{ij}(O_t)$$

Manipulando las expresiones (2.2) y (2.3) obtenemos lo siguiente.

$$\prod_{t=1}^{T} c_t \sum_{i=1}^{N} \alpha_T(i) = c_T \sum_{i=1}^{N} \alpha_T(i) = 1$$

Equivalente a

$$\prod_{t=1}^{T} c_t \cdot P(\boldsymbol{O}|\lambda) = 1$$

Finalmente, en escala logarítmica

$$\log[P(\boldsymbol{O}|\lambda)] = -\sum_{t=1}^{T} \log(c_t)$$

Es decir, la probabilidad $\mathrm{P}(\mathbf{O}|\lambda)$ en escala logarítmica será la suma con signo negativo del logaritmo de los coeficientes de escala calculados para cada columna [2].

La figura 2.44 muestra el código Matlab **algoritmo_forward_Reconocimiento** de la función diseñada para el reconocimiento.

```
function [Log_prob]=algoritmo_forward_Reconocimiento(Pi, A, B, O)
% Calcula P(O|modelo) en escala logaritmica en base 10. Las entradas son
% las matrices Pi, A y B del modelo y el vector de observaciones.
[N_estados, ~]=size(A);
[~, N_observaciones]=size(O);
Alfa=zeros(N_estados, N_observaciones);
Log_prob=0;
for t=1:1:N_observaciones
    if t==1 % Primera columna
        Alfa(:, 1)=Pi.*B(O(1), :)';
    else % Resto de columnas
        temp=Alfa(:, t-1)'*A;
        Alfa(:, t)=(temp.*B(O(t), :))';
    end
    norm=sum(Alfa(:, t)); % Coeficiente de normalización
    if norm==0
        Log_prob=-Inf;
        return % Sale de la función
    else
        Alfa(:, t)=Alfa(:, t)/norm; % Normaliza la columna
    end
    Log_prob=Log_prob+log10(norm);
end
```

Figura 2.44: función Matlab *algoritmo_forward_Reconocimiento*.

En la figura 2.45 se muestra un diagrama explicativo de la utilización del algoritmo en el reconocimiento de una secuencia.

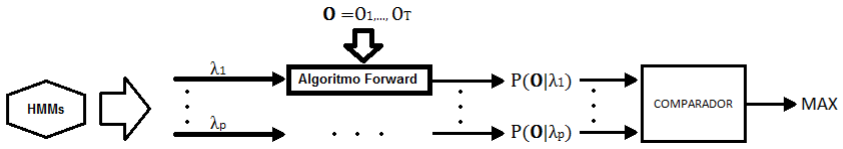

Figura 2.45: Reconocimiento de la secuencia **O** mediante los HMMs de p palabras entrenadas.

Es conveniente establecer un límite inferior para la probabilidad logarítmica obtenida, de modo que si ningún modelo lo supera no se obtenga ningún resultado en el reconocimiento. Esto reduce la posibilidad de que se den falsos positivos.

3. Librería en lenguaje C para el reconocimiento de voz en sistemas de coma fija

En este apartado se desarrolla una librería de funciones en lenguaje C para el reconocimiento de voz en coma fija a partir de las funciones en Matlab diseñadas en el apartado 2. El factor de escala se especifica a modo de comentario junto a la declaración de cada variable. El formato en coma fija se expresa de este modo: <W, Q>, con W la longitud de palabra y Q los bits de la parte decimal en complemento a2. En el caso de variables de entrada y salida de funciones se indica en el comentario de la cabecera de la función. Solamente se desarrollan las funciones necesarias para el reconocimiento, ya que el entrenamiento se hace con las funciones del apartado 2 en el IDE de Matlab.

El desarrollo de la librería sigue una metodología basada en capas, como se muestra en la figura 3.1.

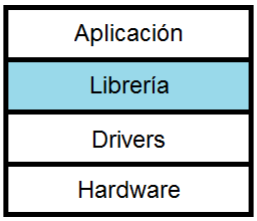

Figura 3.1: Modelo de programación de la librería en capas.

El código de la librería es independiente del resto de las capas. De este modo se pretende que la librería sea portable entre distintas plataformas y utilizable en diversas aplicaciones. Se emplean las recomendaciones del estándar del EECS 381 para la programación y comentado en C [14, 15, 16].

Los ficheros que componen la librería son los siguientes:

- recortar_audio.h
- analisis_cepstral.h
- codificar_palabra.h
- reconocer_palabra.h
- abs_transformada_Fourier.h
- logaritmo_10.h
- constantes.h

3.1. Fichero con los resultados del entrenamiento

Es necesario especificar una serie de constantes globales utilizadas en las funciones de la librería. Para ello se incluye un fichero llamado **constantes.h**. Estas constantes son los parámetros iniciales del programa y los resultados del entrenamiento; el codebook y las matrices de los HMM. La lista completa de constantes es la siguiente:

- Constantes del preprocesador. Precedidas del comando *#define*:
  - **TAM_VENTANA**: Número de muestras de cada ventana. Debe ser una potencia natural de dos.
  - **TAM_BUFF_REC**: Tamaño del buffer circular en el que se almacenan las muestras de audio recibidas. Debe ser una potencia natural de dos y con capacidad suficiente para almacenar el tamaño en muestras de cualquier palabra del vocabulario.
  - **TAM_BUFF**: Tamaño en ventanas del buffer circular de recepción.
  - **DESPLAZAMIENTO_VENTANA**: En ventanas solapadas, el número de muestras compartidas por dos ventanas adyacentes.
  - **DIV_DESPLAZAMIENTO_VENTANA**: Se utiliza para evitar el uso de divisiones al calcular el número de ventanas en el segmento de voz. Su valor es la inversa del producto entre el desplazamiento de ventana y 2^{16}.
  - **DESP_FFT**: Desplazamiento de bits a la izquierda antes de calcular la FFT. Mejora el resultado al reducir el ruido de redondeo. Su valor se determina experimentalmente.
  - **UMBRAL_RECONOCIMIENTO**: Valor que debe superar la variable coper para la detección del inicio de palabra. Con escala <32,30>.
  - **VENTANAS_BAJO_UMBRAL**: Número de ventanas consecutivas por debajo del umbral de reconocimiento para la detección del final de palabra.
  - **MIN_PROB**: Probabilidad umbral mínima del modelo con máxima probabilidad. Reduce la posibilidad de que se den falsos positivos. Con escala <32,15>.
  - **PALABRAS_ENTRENADAS**: Número de palabras del vocabulario entrenadas.
  - **TAMANO_CODEWORD**: Dimensión de los vectores de codewords.
  - **PALABRAS_CODEBOOK**: Número de codewords en el codebook.
  - **FILTROS_MEL**: Número de filtros en el banco de filtros de Mel.
  - **ESTADOS**: Número de estados en los HMM. Aproximadamente del número de ventanas solapadas que caben en el buffer circular.

- Constantes globales. Precedidas del identificador *static const*.
  - *uint16_t* **v_hamming[**N**]**: Vector del tamaño de ventana *N* con los valores del filtro de Hamming. Con escala <16,16>.
  - *uint16_t* **matriz_filtros_mel[**M**][**N/2**]**: Matriz bidimensional con los *M* filtros de Mel ordenados por filas. Con escala <16,16>.
  - *int16_t* **tabla_coseno[**K**][**M**]**: Matriz bidimensional con los valores de la función coseno, utilizados en el cálculo de la DCT para los *K* coeficientes de Mel. Con escala <16,11>.

- *int16_t* **codebook[**C**][**K**]**: Matriz con los *C* codewords ordenados en sus filas. Con escala <16,10>.
- *uint16_t* **A[**P**][**E**][**S**]**: Matriz tridimensional con las matrices de transición entrenadas de las *P* palabras del vocabulario. *E* es el número de estados y *S* los saltos de estado permitidos. Con escala <16,16>.
- *uint16_t* **B[**P**][**C**][**E**]**: Matriz tridimensional con las matrices de observación entrenadas de las palabras del vocabulario. Con escala <16,16>.

En el subapartado 4.1 se desarrolla una aplicación en Matlab para calcular los valores de estas constantes. Como ejemplo se muestra el fichero *constantes.h* de la figura 3.2.

```
#ifndef CONSTANTES_H_
#define CONSTANTES_H_

#define TAM_VENTANA 256
//Tamaño en muestras de las ventanas de enventanado. 2^Int
#define TAM_BUFF_REC 8192
//Tamaño del buffer circular de recepción de audio. 2^Int
#define MIN_tam_palabra 2000
//Límite mínimo del tamaño de palabra
#define DESPLAZAMIENTO_VENTANA 192
// =TAM_VENTANA*(1-solapamiento)
#define DIV_DESPLAZAMIENTO_VENTANA 341L
// =1/DESPLAZAMIENTO_VENTANA*65536
#define DESP_FFT 2
//Bits desplazamiento << entrada a FFT
#define TAM_BUFF 43
// =TAM_BUFF_REC/DESPLAZAMIENTO_VENTANA.
//Máximo número de ventanas solapadas que caben en TAM_BUFF_REC
#define UMBRAL_RECONOCIMIENTO 2147484L //<32,30>
//Umbral para detecciónde inicio y final de palabra con energía y cruces por cero
#define VENTANAS_BAJO_UMBRAL 5
//Ventanas mínimas bajo umbral para detección del final de palabra
#define MIN_PROB -3276800//<32,15>
//Valor mínimo de probabilidad para el reconocimiento
#define PALABRAS_ENTRENADAS 4
//Número de modelos de palabra
#define TAMANO_CODEWORD 5
//Tamaño de los codewords del codebook
#define FILTROS_MEL 20
//Número de filas de la matriz Mel
#define GANANCIA 30
//Ganancia en la recepción de audio
static const uint16_t v_hamming[256]={ 5243 , 5252 , 5279 , 5325 , 5389 ,...   };//<16,16>
static const uint16_t matriz_filtros_mel[20][128]={{ 13262 , 26794 , 18838,...}};//<16,16>
static const int16_t tabla_coseno[5][20]={{ 2042 , 1991 , 1892 , 1746 ,    ...}};//<16,11>
#define PALABRAS_CODEBOOK 8
//Número de símbolos en el codebook
static const int16_t codebook[8][5]={{ 11046 , -2376 , -10075 , -1644 ,...    }};//<16,10>
static const char nombre[4][16]={{"      uno       "},{"      dos       "},...         }};
#define ESTADOS 20
//Número de estados del HMM
static const uint16_t A[4][20][3]={{{ 3 , 63954 , 1579 },{ 51 , 62084 ,...   }}};//<16,16>
static const uint16_t B[4][8][20]={{{ 40381 , 48321 , 47725 , 52137 ,...     }}};//<16,16>

#endif /* CONSTANTES_H_ */
```

Figura 3.2: Ejemplo del archivo C *constantes.h*.

3.2. Funciones para la detección de los extremos de la palabra

Este fichero se encarga de dos procesos; el filtrado con el filtro preénfasis y la detección de extremos. La figura 3.3 muestra el fichero **recortar_audio.h** con la cabecera de las funciones.

```
#include <stdint.h>
#include "constantes.h"

#ifndef RECORTAR_AUDIO_H_
#define RECORTAR_AUDIO_H_

/*-------------------------------- recortar_audio ------------------------------------
 * Recibe una muestra de la señal de audio en rx, con escala <16,15>, se filtra con el
 * filtro preénfasis y se almacena en el buffer de salida palabra_a_reconocer.
 * Devuelve el valor de inicio de palabra y el tamaño cuando detecta el final
 * tamano_palabra_a_reconocer de la palabra recortada en palabra_a_reconocer.
 *-----------------------------------------------------------------------------------*/
void recortar_audio(int16_t rx_L, int16_t *palabra_a_reconocer,
                    int *inicio_palabra_a_reconocer, int *tamano_palabra_a_reconocer);

/*-------------------------------- analisis_ventana ----------------------------------
 * Devuelve los valores de inicio inicio_palabra_a_reconocer y tamaño
 * tamano_palabra_a_reconocer de la palabra recortada.La entrada energia_crucesPorCero,
 * con escala <32,30>, es la medida de energia y cruces por cero de la ventana completa e
 * indica si la ventana contiene voz en caso de superar UMBRAL_RECONOCIMIENTO. La entrada
 * indice_buffer indica el indice de la última muestra de la ventana en el buffer donde se
 * almacena el audio.
 *-----------------------------------------------------------------------------------*/
void analisis_ventana(uint32_t energia_crucesPorCero, int *indice_buffer,
                    int *inicio_palabra_a_reconocer, int *tamano_palabra_a_reconocer);

#endif /* RECORTAR_AUDIO_H_ */
```

Figura 3.3: fichero C *recortar_audio.h*.

La función **recortar_audio** se muestra en la figura 3.4. Recibe una a una las muestras de entrada de audio en formato <16,15>, las filtra con el filtro preénfasis y las almacena en el buffer circular *palabra_a_reconocer,* cuya dirección se pasa por valor a la función. Calcula la variable coper a medida que llegan las muestras, hasta alcanzar el final de la ventana. Entonces ejecuta la función *analisis_ventana*.

Repite este proceso con cada ventana hasta detectar el final de palabra. Una vez detectado lo indica en las variables de salida de la función; *inicio_palabra_a_reconocer* y *tamano_palabra _a_reconocer* que, excepto en ese momento, ambas valen cero.

Se utilizan las funciones *abs* y *labs* de la librería estándar *math.h* para calcular el valor absoluto de variables *int* y *long* respectivamente.

```
void recortar_audio(int16_t rx, int16_t *palabra_a_reconocer,
                        int *inicio_palabra_a_reconocer, int *tamano_palabra_a_reconocer)
{
    static int16_t rx_ant=0;//<16,15>
    // Guarda la muestra de audio recibida en el instante anterior
    const int32_t a=62259;//<32,16>
    //  =0.95  Constante del filtro preénfasis
    static int16_t anterior=0;//<16,15>
    // Guarda la muestra de audio filtrada en el instante anterior
    static int indice_buffer=0;
    // Indice del buffer de escritura palabra_a_reconocer
    static uint32_t energia_crucesPorCero=0;//<32,30>
    // Proporcional a la energía y cruces por cero en la ventana.
    palabra_a_reconocer[indice_buffer]=(((long)rx<<16)-(long)rx_ant*a)>>16;//<16,15>
    // filtro preénfasis
    rx_ant=rx;
    energia_crucesPorCero+=labs((long)palabra_a_reconocer[indice_buffer]*
                    abs(palabra_a_reconocer[indice_buffer])-(long)anterior*abs(anterior));
    anterior=palabra_a_reconocer[indice_buffer];
    if ((indice_buffer&(TAM_VENTANA-1))==(TAM_VENTANA-1))//Final de la ventana
    {
        analisis_ventana(energia_crucesPorCero, &indice_buffer,
                                inicio_palabra_a_reconocer, tamano_palabra_a_reconocer);
        energia_crucesPorCero=0;
    }
    indice_buffer++;
    if (indice_buffer==TAM_BUFF_REC)
    {
        indice_buffer=0;
    }
}
```

Figura 3.4: función C *recortar_audio*.

La función **analisis_ventana** se muestra en la figura 3.5. Recibe el valor de la variable coper de cada ventana, y lo compara con las constantes umbrales para decidir el comienzo y final de palabra.

Como salida indica, respecto al buffer circular de recepción, el índice de las muestras de inicio y el tamaño en muestras de la palabra.

```
void analisis_ventana(uint32_t energia_crucesPorCero, int *indice_buffer,
                      int *inicio_palabra_a_reconocer, int *tamano_palabra_a_reconocer)
{
    typedef enum { REPOSO,
                   RECEPCION
    }estadosRx_t;
    static estadosRx_t estado=REPOSO;
    //Estado antes o después de detectar el inicio de la palabra
    static uint16_t inicio=0;
    //Indice del buffer donde se detecta el inicio de la palabra
    uint16_t final;
    //Indice del buffer donde se detecta el final de la palabra
    static int contador_ventanas=0;
    //Contador de las ventanas bajo el umbral para detectar final palabra
    switch (estado)
    {
        case RECEPCION:
        {
            if (energia_crucesPorCero>UMBRAL_RECONOCIMIENTO)
            {//No se detecta el final de palabra
                contador_ventanas=0;
            }
            else
            {//Se detecta un posible final de palabra
                contador_ventanas++;
                if (contador_ventanas==VENTANAS_BAJO_UMBRAL)
                {//Se detecta el final de palabra
                    final=(*indice_buffer-VENTANAS_BAJO_UMBRAL*TAM_VENTANA)&
                                                                    (TAM_BUFF_REC-1);
                    energia_crucesPorCero=0;
                    *indice_buffer=0;
                    *inicio_palabra_a_reconocer=inicio;
                    if (final>inicio)
                    {//Palabra continua en el buffer circular
                        *tamano_palabra_a_reconocer= final-inicio;
                    }
                    else
                    {//Palabra cortada en el buffer circular
                        *tamano_palabra_a_reconocer=TAM_BUFF_REC+final-inicio;
                    }
                    estado=REPOSO;
                }
            }
        }
        break;
        default:
        {
            if (energia_crucesPorCero>UMBRAL_RECONOCIMIENTO)
            {//Se detecta el inicio de palabra
                estado=RECEPCION;
                inicio=(*indice_buffer-TAM_VENTANA)&(TAM_BUFF_REC-1);
            }
        }
    }
}
```

Figura 3.5: función C *analisis_ventana*.

3.3. Funciones para la obtención de los MFCC

Una vez tenemos el segmento de audio almacenado en el buffer circular y con los extremos de la palabra delimitados, podemos comenzar con el análisis. El fichero **análisis_cepstral.h** contiene las funciones necesarias para obtener la matriz de coeficientes cepstrales de la palabra. La figura 3.6 muestra el fichero con las cabeceras de las funciones.

```
#include <stdint.h>
#include "constantes.h"
#include "abs_transformada_Fourier.h"
#include "logaritmo_10.h"

#ifndef ANALISIS_CEPSTRAL_H_
#define ANALISIS_CEPSTRAL_H_

/*---------------------------------- analisis_cepstral ----------------------------------
 * Devuelve la matriz de coeficientes MFCC, con escala <16,10> y tamaño TAM_BUFF filas y
 * TAMANO_CODEWORD columnas, y el número de filas utilizadas de ésta, a partir del buffer
 * con el audio x, con escala <16,15>, y los valores de inicio, inicio_x, y tamaño de la
 * palabra recortada en el buffer, tamano_x. El tamaño de las ventanas es de TAM_VENTANA
 * muestras solapadas entre sí comenzando cada ventana DESPLAZAMIENTO_VENTANA muestras
 * despues del comienzo de la anterior.
 *-------------------------------------------------------------------------------------*/
int analisis_cepstral(int16_t *x, int inicio_x, int tamano_x,
                                                int16_t MFCC[TAM_BUFF][TAMANO_CODEWORD]);

/*------------------------------ filtrado_ventana_Hamming -------------------------------
 * Multiplica cada muestra de la ventana de audio por cada muestra de la ventana de
 * Hamming. La entrada x, con escala <16,15>,es el buffer con el audio. inicio_x es el
 * indice donde comienza el audio en el buffer. inicio es el indice donde empieza la
 * ventana respacto al inicio de palabra. La ventana tiene 256 muestras. La salida ventana
 * es un buffer de TAM_VENTANA muestras de 32 bits escalada y en formato complejo para la
 * fft. Los 16 bits mas significativos corresponden a la parte real con escala <16,17>, el
 * resto a la parte imaginaria.
 *-------------------------------------------------------------------------------------*/
void filtrado_ventana_Hamming(int16_t *x, int inicio_x, int inicio, int32_t *ventana);

/*------------------------------ filtrado_banco_filtros_Mel -----------------------------
 * Multiplica cada muestra del valor absoluto al cuadrado de la transformada de Fourier de
 * la ventana de audio por cada muestra de cada uno de los FILTROS_MEL filtros del banco
 * de filtros de mel, sumando los resultados de cada filtro. La entrada abs2_fft, con
 * escala <32,18>, son las muestras del valor absoluto al cuadrado de la transformada de
 * Fourier de la ventana de audio.
 * La salida mel , con escala <16,14>, es un vector de tamaño el número de filtros de Mel.
 *-------------------------------------------------------------------------------------*/
void filtrado_banco_filtros_Mel(uint32_t *abs2_fft, int16_t *mel);

/*------------------------------ transformada_inversa_coseno ----------------------------
 * Obtiene las muestras 2->TAMANO_CODEWORD de la transformada inversa del coseno del
 * vector de entrada log_mel, con escala <32,15>, y tamaño FILTROS_MEL. El vector
 * resultado lo devuelve en la fila indice_ventana de la matriz MFCC, con escala <16,10>.
 *-------------------------------------------------------------------------------------*/
void transformada_inversa_coseno(int32_t *log_mel, int indice_ventana,
                                                int16_t MFCC[TAM_BUFF][TAMANO_CODEWORD]);

#endif /* ANALISIS_CEPSTRAL_H_ */
```

Figura 3.6: fichero C *analisis_cepstral.h*.

La función **analisis_cepstral** se muestra en la figura 3.7. Calcula la matriz de coeficientes cepstrales a partir del buffer con el audio, la posición de inicio y el tamaño en muestras de la palabra. Divide el segmento en ventanas solapadas y devuelve el número de ventanas calculado.

Para cada ventana, aplica las funciones *filtrado_ventana_Hamming*, *abs_transformada_Fourier*, *filtrado_banco_filtros_Mel, logaritmo_10* y *transformada_inversa_coseno*. Obtiene los coeficientes cepstrales que almacena en cada fila de la matriz MFCC.

```
int analisis_cepstral(int16_t *x, int inicio_x, int tamano_x,
                                                    int16_t MFCC[TAM_BUFF][TAMANO_CODEWORD])
{
    int N_ventanas;
    // Número de ventanas solapadas enteras que tiene la palabra
    int inicio;
    // Indice de la primera muestra de la ventana
    int32_t ventana[TAM_VENTANA];//<16,15+DESP_FFT>  Formato Real_Imag
    // Muestras enventanadas y filtradas. Los 16 bits mas significativos son la parte real
    // y el resto la imaginaria (=0)
    uint32_t abs2_fft[TAM_VENTANA/2];//<32,14+2*DESP_FFT>
    // Valor absoluto al cuadrado de la transformada de Fourier
    int16_t mel[FILTROS_MEL];//<16,14>
    // Vector de coeficientes Mel de la ventana
    int32_t log_mel[FILTROS_MEL];//<32,15>
    // Logaritmo en base 10 de los coeficientes Mel
    N_ventanas=(((uint32_t)tamano_x*DIV_DESPLAZAMIENTO_VENTANA)>>16)-1;
    int i;
    for(i=0;i<N_ventanas;i++)
    {
            inicio=i*DESPLAZAMIENTO_VENTANA;
            filtrado_ventana_Hamming(x, inicio_x, inicio, ventana);
            abs_transformada_Fourier(ventana, abs2_fft);
            filtrado_banco_filtros_Mel(abs2_fft, mel);
            logaritmo_10(mel, log_mel, FILTROS_MEL);
            transformada_inversa_coseno(log_mel, i, MFCC);
    }
    return N_ventanas;
}
```

Figura 3.7: función C *analisis_cepstral*.

La función **filtrado_ventana_Hamming** se muestra en la figura 3.8. Multiplica las muestras de la ventana por las correspondientes del vector con los valores de la ventana de Hamming. El vector resultado lo convierte a formato real-imaginario de 32 bits. Los 16 bits más significativos son la parte real y el resto la imaginaria, con valor cero.

```
void filtrado_ventana_Hamming(int16_t *x, int inicio_x, int inicio, int32_t *ventana)
{
    int32_t x_h;//<32,31>
    // Muestra filtrada con el filtro Hamming
    int j;
    for (j=0;j<TAM_VENTANA;j++)
    {
        x_h=(int32_t)x[(inicio_x+inicio+j)%TAM_BUFF_REC]*(int32_t)v_hamming[j];
        ventana[j]=(x_h<<DESP_FFT)&0xFFFF0000;//<16,15+DESP_FFT>  Formato Real_Imag
    }
}
```

Figura 3.8: Función C *filtrado_ventana_Hamming*.

La función **filtrado_banco_filtros_Mel** se muestra en la figura 3.9. Multiplica escalarmente el vector con la densidad espectral de la ventana por cada fila de la matriz de filtros de Mel, y devuelve los resultados en un vector.

```
void filtrado_banco_filtros_Mel(uint32_t *abs2_fft, int16_t *mel)
{
    int32_t accum;//<32,30>
    // Guarda la suma acumulada en el calculo de las coeficientes Mel
    int j;
    for (j=0;j<FILTROS_MEL;j++)
    {//Multiplica la ventana por cada fila de la matriz de filtros Mel
        accum=0;
        int t;
        for (t=0;t<TAM_VENTANA/2;t++)
        {
            accum+=abs2_fft[t]*(matriz_filtros_mel[j][t]>>(2*DESP_FFT));
        }
        mel[j]=(accum>>16)+1;//0->2. Se suma 1 para log_10!=-inf
    }
}
```

Figura 3.9: función C *filtrado_banco_filtros_Mel*.

La función **transformada_inversa_coseno** se muestra en la figura 3.10. Calcula la transformada del coseno del vector con los coeficientes de Mel en escala logarítmica. Antes de aplicar la DCT se aumenta el valor de los coeficientes, como se explica en el subapartado 2.1.7.

```
void transformada_inversa_coseno(int32_t *log_mel, int indice_ventana,
                                                    int16_t MFCC[TAM_BUFF][TAMANO_CODEWORD])
{
    int32_t accum_c;//<32,26>
    // Guarda la suma acumulada en el calculo de la transformada inversa del coseno
    int j;
    for (j=0;j<TAMANO_CODEWORD;j++)
    {
        accum_c=0;
        int l;
        for(l=0;l<FILTROS_MEL;l++)
        {
            accum_c+=(int32_t)(tabla_coseno[j][l])*(log_mel[l]+9864L+65536L);
        }                                                          // -9864=log10(2^-1)*2^15
        MFCC[indice_ventana][j]=accum_c>>16;
    }
}
```

Figura 3.10: función C *transformada_inversa_coseno*.

3.4. Funciones para la codificación de la palabra

La matriz de coeficientes cepstrales MFCC contiene los resultados del análisis cepstral de las ventanas almacenados en sus filas. A continuación se codifica cada fila mediante el codebook. La figura 3.11 muestra el fichero **codificar_palabra.h** con la cabecera de la función.

```c
#include <stdint.h>
#include "constantes.h"

#ifndef CODIFICAR_PALABRA_H_
#define CODIFICAR_PALABRA_H_

/*-------------------------------- codificar_palabra ----------------------------------
 * Devuelve un vector de tamaño filas_MFCC, el número de filas de la matriz MFCC, con
 * escala <16,10>, de entrada con el valor codificado de cada fila. El valor de la fila
 * codificada es el índice del elemento con mínima destancia del codebook a esa fila.
 *-------------------------------------------------------------------------------------*/
void codificar_palabra(int16_t MFCC[TAM_BUFF][TAMANO_CODEWORD], int filas_MFCC,
                                                                int *palabra_codificada);

#endif /* CODIFICAR_PALABRA_H_ */
```

Figura 3.11: fichero C *codificar_palabra.h*.

La función **codificar_palabra** se muestra en la figura 3.12. Obtiene el vector con los códigos de cada ventana. Para cada fila de la matriz MFCC se calcula la distancia euclídea al cuadrado a todos los codewords y se comparan para decidir el índice cuya distancia es menor. Los resultados se almacenan en un vector.

```c
void codificar_palabra(int16_t MFCC[TAM_BUFF][TAMANO_CODEWORD], int filas_MFCC,
                                                                int *palabra_codificada)
{
    int32_t distancia_min;
    //Valor de distancia del vector MFCC al codeword mas cercano
    int16_t diferencia;//<16,10>
    //Diferencia entre las coordenadas de los vectores
    int32_t temp;//<32,20>
    //Guarda la suma acumulada para el cálculo de las distancias
    int min;
    //Indice del codeword con la mínima distancia
    int i;
    for (i=0;i<filas_MFCC;i++)
    {
        min=0;
        int j;
        for (j=0;j<PALABRAS_CODEBOOK;j++)
        {//Calculo del índice del codeword mas cercano al vector MFCC
            temp=0;
            int k;
            for (k=0;k<TAMANO_CODEWORD;k++)
            {
                diferencia=MFCC[i][k]-codebook[j][k];
                temp+=(int32_t)diferencia*diferencia;
            }
            if (j==0)
            {//Inicia el valor distancia_min
                distancia_min=temp;
            }
            if(temp<distancia_min)
            {
                min=j;
                distancia_min=temp;
            }
        }
        palabra_codificada[i]=min;
    }
}
```

Figura 3.12: función C *codificar_palabra*.

3.5. Funciones para el reconocimiento con los HMM

El vector con la palabra codificada contiene los símbolos de la secuencia de observación que se utilizará en el reconocimiento con los modelos ocultos de Markov. La figura 3.13 muestra el fichero **reconocer_palabra.h** con las cabeceras de las funciones.

```
#include <stdint.h>
#include "constantes.h"
#include "logaritmo_10.h"

#ifndef RECONOCER_PALABRA_H_
#define RECONOCER_PALABRA_H_

/*---------------------------------- reconocer_palabra ----------------------------------
 * Devuelve el valor del modelo de palabra con la probabilidad de generación de ocurrencia
 * máxima, max P(O|Modelo_i)->i. En caso de que ningún modelo supere el mínimo de
 * probabilidad establecido, PROB_MIN, devuelve -1. La entrada palabra_codificada es un
 * vector de tamaño tamano_palabra_codificada con los valores de las ventanas codificadas
 * con el codebook.
 *-------------------------------------------------------------------------------------*/
int reconocer_palabra( int *palabra_codificada , int tamano_palabra_codificada);

/*---------------------- probabilidad_ocurrencia_sea_generada_por_modelo -----------------
 * Devuelve el logaritmo decimal de la probabilidad de que la palabra codificada haya sido
 * generada por el modelo de palabra, en escala <32,15>. Utiliza el algoritmo forward y
 * las matrices A y B de transiciones y ocurrencias. La entrada palabra_codificada es un
 * vector de tamaño tamano_palabra_codificada con los valores de las ventanas codificadas
 * con el codebook. La entrada modelo_palabra es el índice del modelo de palabra.
 *-------------------------------------------------------------------------------------*/
int32_t probabilidad_ocurrencia_sea_generada_por_modelo(int *palabra_codificada,
                                    int tamano_palabra_codificada, int modelo_palabra);

/*------------------------------ indice_maxima_probabilidad -----------------------------
 * Compara los elementos del vector de entrada log_probabilidad, con escala <32,15> y
 * devuelve el indice del valor máximo o -1 si este no supera MIN_PROB.
 *-------------------------------------------------------------------------------------*/
int indice_maxima_probabilidad(int32_t *log_probabilidad);

#endif /* RECONOCER_PALABRA_H_ */
```

Figura 3.13: fichero C *reconocer_palabra.h*.

La función **reconocer_palabra** se muestra en la figura 3.14. Obtiene el índice del HMM de la palabra reconocida.

Para cada HMM del vocabulario entrenado, ejecuta la función *probabilidad_ocurrencia_sea_generada_por_el_modelo,* y se calcula un vector con las probabilidades almacenadas en escala logarítmica. Compara los valores del vector mediante la función *indice_maxima_probabilidad* y devuelve el índice del modelo con el valor máximo.

```
int reconocer_palabra(int *palabra_codificada, int tamano_palabra_codificada)
{
    int32_t log_probabilidad[PALABRAS_ENTRENADAS];//<32,15>
    // Logaritmo decimal de la probabilidad para cada modelo de palabra
    int modelo;
    // Indice del HMM evaluado
    for (modelo=0; modelo<PALABRAS_ENTRENADAS; modelo++)
    {
        log_probabilidad[modelo]=probabilidad_ocurrencia_sea_generada_por_modelo(
                                    palabra_codificada, tamano_palabra_codificada, modelo);
    }
    return indice_maxima_probabilidad(log_probabilidad);
}
```

Figura 3.14: función C *reconocer_palabra*.

La función **probabilidad_ocurrencia_sea_generada_por_modelo** se muestra en la figura 3.15. Calcula el logaritmo decimal de la probabilidad de que el vector de observaciones haya sido generado por el modelo; mediante el algoritmo forward y los coeficientes de escala de la matriz alfa. Utiliza la función *logaritmo_10*.

Debido a que la matriz de transiciones tiene un número determinado de elementos no nulos en cada fila se emplea una versión reducida. Esto consigue disminuir el espacio ocupado en memoria por los modelos ([1-(E+C)/(S+C)]·100%), y reducir el número de multiplicaciones en el cálculo de la matriz alfa ([1-(E+1)/(S+1)]·100%).

```
int32_t probabilidad_ocurrencia_sea_generada_por_modelo(int *palabra_codificada,
                                        int tamano_palabra_codificada, int modelo_palabra)
{
    uint16_t alfa[ESTADOS];//<16,15>
    // Vector con los valores de la columna t de la matriz alfa
    uint16_t alfa_t[ESTADOS];//<16,15>
    // Vector con los valores de la columna t de la matriz alfa de la iteración anterior
    int16_t norma;//<16,15>
    // Suma de los elementos del vector alfa
    uint32_t suma;//<32,31>
    // Guarda la suma acumulada para el cálculo de alfa
    int32_t log;//<32,15>
    // Logaritmo decimal de norma
    int32_t log_acum;//<32,15>
    // Acumulador del sumatorio de log
    int j;
    for (j=0; j<ESTADOS; j++)//Valor inicial del vector alfa {1,0,...,0}
    {
        alfa[j]=0;
    }
    alfa[0]=32768;// =1
    norma=(uint32_t)B[modelo_palabra][palabra_codificada[0]][0];
    if (norma==0)
    {//Si norma es igual a 0 se considera improbable y se le da el valor mínimo
        return MIN_PROB;
    }
    logaritmo_10(&norma, &log_acum, 1);
    for (j=1; j<tamano_palabra_codificada; j++)//Calculo inductivo matriz alfa
    {
        int n;
        for (n=0; n<ESTADOS; n++)
        {
            suma=0;
            int m;
            for (m=0; m<ESTADOS; m++)
            {
                if (m==n-2) suma+=(uint32_t)(alfa[m])*(uint32_t)(A[modelo_palabra][m][2]);
                else if (m==n-1) suma+=(uint32_t)(alfa[m])*
                                                    (uint32_t)(A[modelo_palabra][m][1]);
                else if (m==n)
                {
                    suma+=(uint32_t)(alfa[m])*(uint32_t)(A[modelo_palabra][m][0]);
                    break;
                }
            }
            alfa_t[n]=suma>>16;
        }
        norma=0;
        for (n=0; n<ESTADOS; n++)
        {
            alfa[n]=((uint32_t)(alfa_t[n])*
                            (uint32_t)(B[modelo_palabra][palabra_codificada[j]][n]))>>16;
            norma+=alfa[n];
        }
        if (norma==0)
        {
            return MIN_PROB;
        }
        else
        {
            for (n=0;n<ESTADOS;n++)//Se normaliza para evitar underflow
            {
                alfa[n]=((uint32_t)alfa[n]<<15)/(uint32_t)norma;
            }
            logaritmo_10 (&norma, &log, 1);
            log_acum+=log;
        }
    }
    return log_acum;
}
```

Figura 3.15: función C *probabilidad_ocurrencia_sea_generada_por_modelo*.

La función **indice_maxima_probabilidad** se muestra en la figura 3.16. Obtiene, a partir del vector con los logaritmos de las probabilidades de los modelos, el índice del elemento cuyo valor es máximo. Si éste no supera el valor umbral mínimo, la función devuelve el valor -1.

```
int indice_maxima_probabilidad(int32_t *log_probabilidad)
{
    int indice=0;
    // Indice del vector de valor máximo
    int i;
    for (i=1; i<PALABRAS_ENTRENADAS; i++)
    {// Obtiene el índice del vector de valor máximo
        if (log_probabilidad[i]>log_probabilidad[indice])
        {
            indice=i;
        }
    }
    if (log_probabilidad[indice]<=MIN_PROB)
    {// -1 si ninguno supera MIN_PROB
        return -1;
    }
    else
    {
        return indice;
    }
}
```

Figura 3.16: función C *indice_maxima_probabilidad*.

3.6. Especificaciones para la transformada de Fourier y el logaritmo decimal

Existen librerías optimizadas para estas dos funciones, en coma fija y flotante. También algunos DSP disponen de unidades hardware especializado, HWAFFT. Por ello, es necesario programar estas funciones específicamente para el sistema en el que se van a ejecutar para obtener los mejores resultados. Las cabeceras con las especificaciones para programar estas funciones se muestran en las figuras 3.17 y 3.18.

```
#ifndef ABS_TRANSFORMADA_FOURIER_H_
#define ABS_TRANSFORMADA_FOURIER_H_

/*------------------------------ abs_transformada_Fourier ------------------------------
 * Devuelve la densidad espectral de energía de la señal de entrada.
 * La entrada ventana de tamaño TAM_VENTANA y escala <16,15+DESP_FFT>. La salida abs2_fft
 * de tamaño TAM_VENTANA/2 y escala <32,14+2*DESP_FFT> es un vector con sus elementos el
 * módulo al cuadrado de la DFT de ventana de la muestra 0 a TAM_VENTANA/2-1.
 *-------------------------------------------------------------------------------------*/
void abs_transformada_Fourier(int32_t *ventana, uint32_t *abs2_fft);

#endif /* ABS_TRANSFORMADA_FOURIER_H_ */
```

Figura 3.17: fichero C abs_transformada_Fourier.h.

```c
#ifndef LOGARITMO_10_H_
#define LOGARITMO_10_H_

/*------------------------------------ logaritmo_10 ------------------------------------
 * Devuelve un vector con los logaritmos decimales de los elementos del vector de entrada.
 * La entrada x es un vector de tamaño tamano_x y escala <16,15>.
 * La salida log10_x es un vector de tamaño tamano_x y escala <32,15>.
 *--------------------------------------------------------------------------------------*/
void logaritmo_10(int16_t *x, int32_t *log10_x, int tamano_x);

#endif /* LOGARITMO_10_H_ */
```

Figura 3.18: fichero C logaritmo_10.h.

En los apartados 4.2 y 4.3 se muestran dos ejemplos de codificación de estas funciones; para *PC* y el *ezdsp5535*.

4. Demostradores desarrollados para el reconocimiento de voz y evaluación de sus prestaciones

4.1. Aplicación en Matlab con GUI para el entrenamiento del sistema de reconocimiento de voz

En este subapartado se explica el funcionamiento de la aplicación desarrollada para el entrenamiento de los demostradores. El programa utiliza las funciones Matlab del apartado 2. El objetivo es, mediante un interfaz gráfico creado con la herramienta de edición de GUI de Matlab, facilitar la introducción de los datos con los que entrenar el codebook y los HMM y guardar los resultados. Permite además comprobar el funcionamiento del sistema de reconocimiento de voz, modificar los parámetros iniciales y visualizar gráficamente los resultados de cada etapa.

El interfaz se compone de las secciones:

1. Introducción de los valores iniciales.
2. Grabación de voz.
3. Entrenamiento del codebook.
4. Entrenamiento de los HMM.
5. Reconocimiento de voz.
6. Guardar los resultados.

Introducción de los valores iniciales

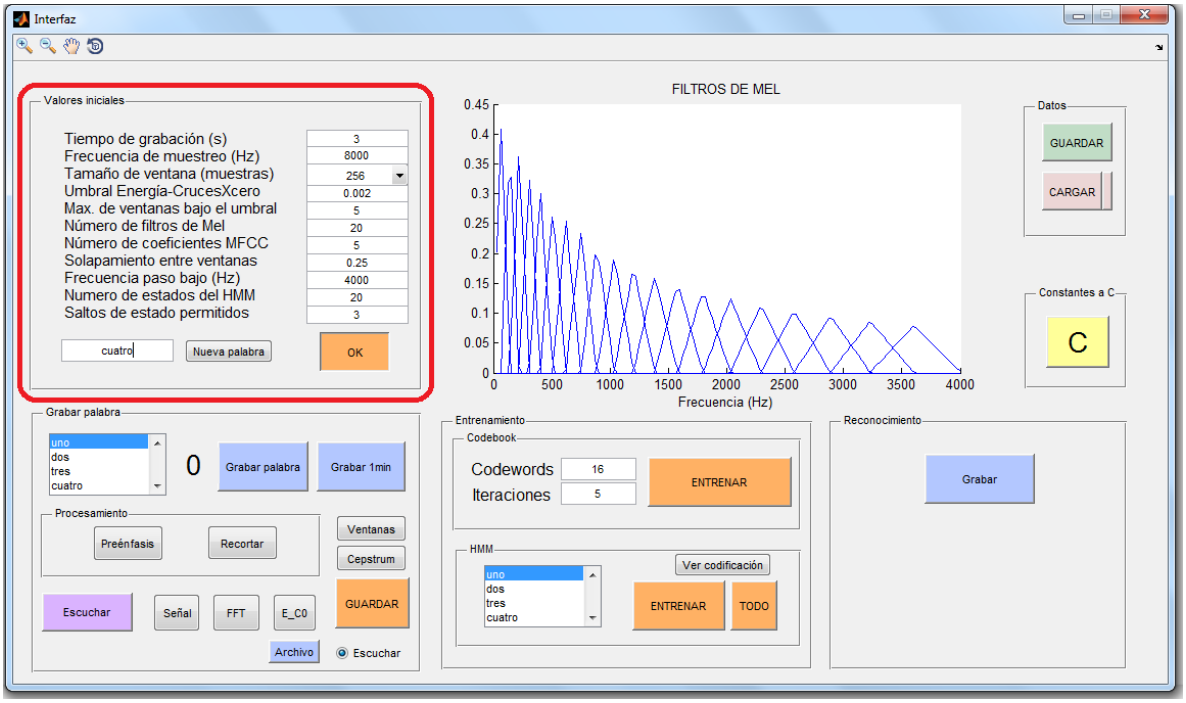

Figura 4.1: Interfaz para la introducción de los valores iniciales.

La figura 4.1 muestra en rojo el panel de introducción de los datos iniciales. En la parte superior hay una lista con las variables iniciales y un cuadro de texto (Edit Text) a la derecha de cada una, donde es posible modificar su valor. En la figura 4.1 se muestran los valores por defecto. En la parte inferior izquierda hay otro cuadro de texto. En él se introducen las palabras del vocabulario que queremos que el sistema reconozca, las que vamos a entrenar. Para introducir una palabra la escribimos en el cuadro y pulsamos el botón *Nueva palabra*. Aparecerá en los Listbox del resto de paneles y el cuadro se pondrá en blanco para la introducción de una nueva palabra. Una vez hemos introducido todos los datos pulsamos el botón *OK*.

Utiliza la función:

- ✓ matriz_filtros_mel.m

Grabación de voz

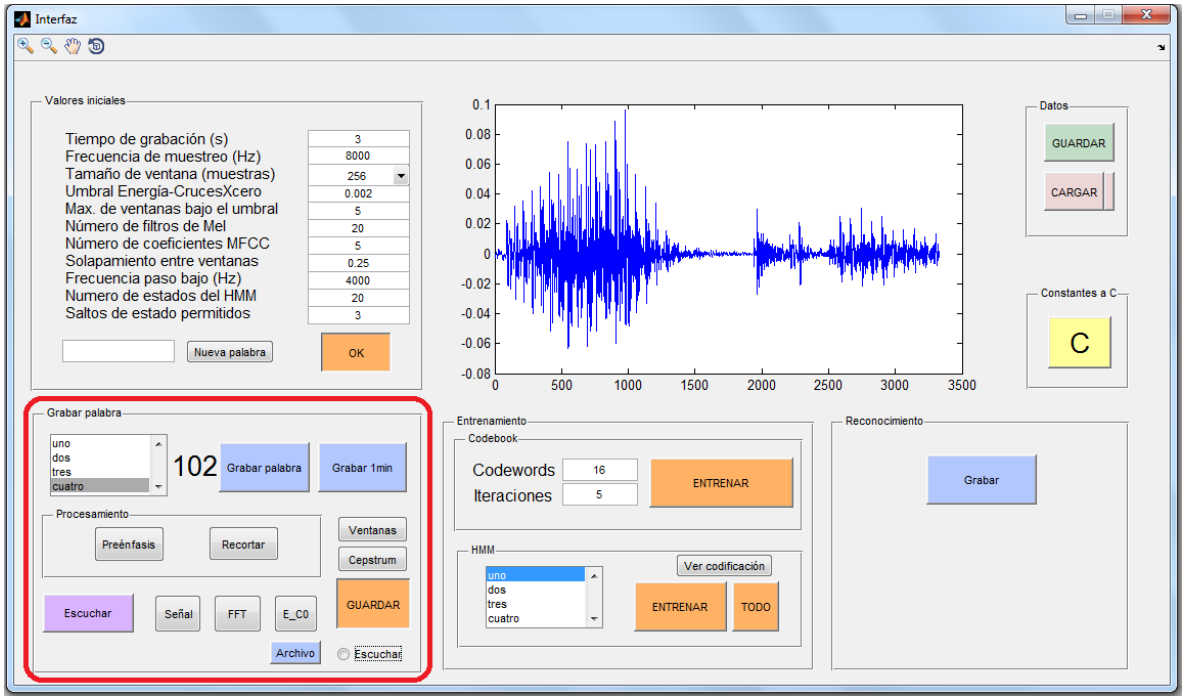

Figura 4.2: Interfaz para la grabación de voz.

La figura 4.2 muestra en rojo el panel de grabación de voz. En la parte superior izquierda se encuentra la lista (Listbox) con las palabras del vocabulario a entrenar. Seleccionamos una palabra y pulsamos *Grabar palabra*. A continuación pronunciamos la palabra en el micrófono hasta que aparezca la representación de la señal en el gráfico. Para analizarla, pulsamos los botones *Preénfasis* y *Recortar*. Se mostrarán los resultados de cada función en el gráfico. Con el botón *Escuchar* podemos reproducir el audio con las señales generadas durante el análisis, y mostrar su DFT con el botón *FFT* o su función de energía y cruces por cero con *E_C0*.

Si el resultado es correcto pulsamos *Guardar* y el resultado se guardará, incrementándose el contador de repeticiones de la palabra junto al Listbox. También se mostrará en el gráfico la matriz MFCC generada.

Para agilizar el proceso podemos pulsar el botón *Grabar 1min* y grabar con el micrófono durante 1 minuto, repitiendo numerosas veces la palabra seleccionada. También es posible introducir estos datos mediante un archivo de audio pulsando el botón *Archivo*. El nombre del archivo debe ser el nombre de la palabra con la extensión *.mat*, y estar guardado en una carpeta llamada */grabaciones* en el directorio raíz. El audio debe estar almacenado en un vector columna llamado *audio*, con codificación de 16 bits monocanal y la frecuencia de muestreo introducida en el panel de valores iniciales.

Utiliza las funciones:

- ✓ grabar_voz.m
- ✓ filtro_preEnfasis.m
- ✓ analisis_coper.m
- ✓ recortar.m
- ✓ coeficientes_mel.m
- ✓ cepstrum.m
- ✓ analisis_palabra.m

Entrenamiento del codebook

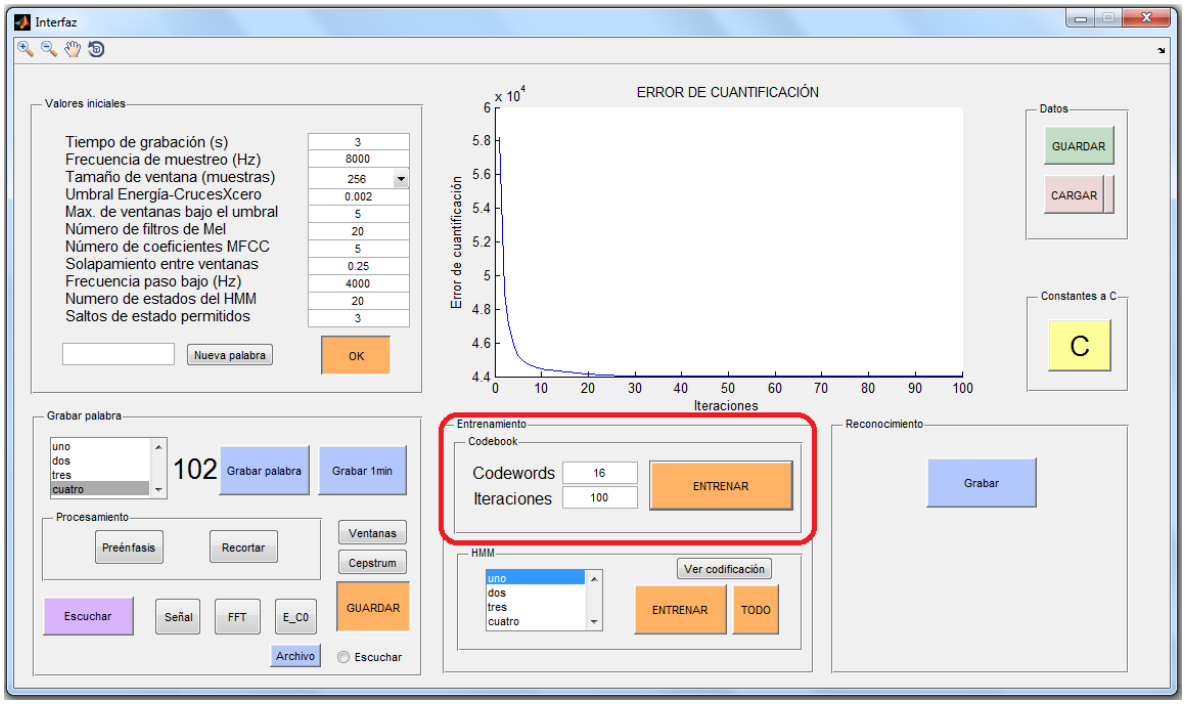

Figura 4.3: Interfaz para el entrenamiento del codebook.

La figura 4.3 muestra en rojo el panel de entrenamiento del codebook. El entrenamiento se hace con todas las matrices MFCC obtenidas de las grabaciones con el panel anterior. En los Edit Text de la parte izquierda podemos introducir el número de codewords que formarán el codebook y las iteraciones del algoritmo K-means con las que vamos a entrenarlo. Pulsando el botón *ENTRENAR* se genera el codebook y se muestra la gráfica con el error de cuantificación total en cada iteración.

Utiliza las funciones:

- ✓ algoritmo_Kmeans.m
- ✓ error_por_distorsion.m
- ✓ codebook_inicial.m

Entrenamiento de los HMM

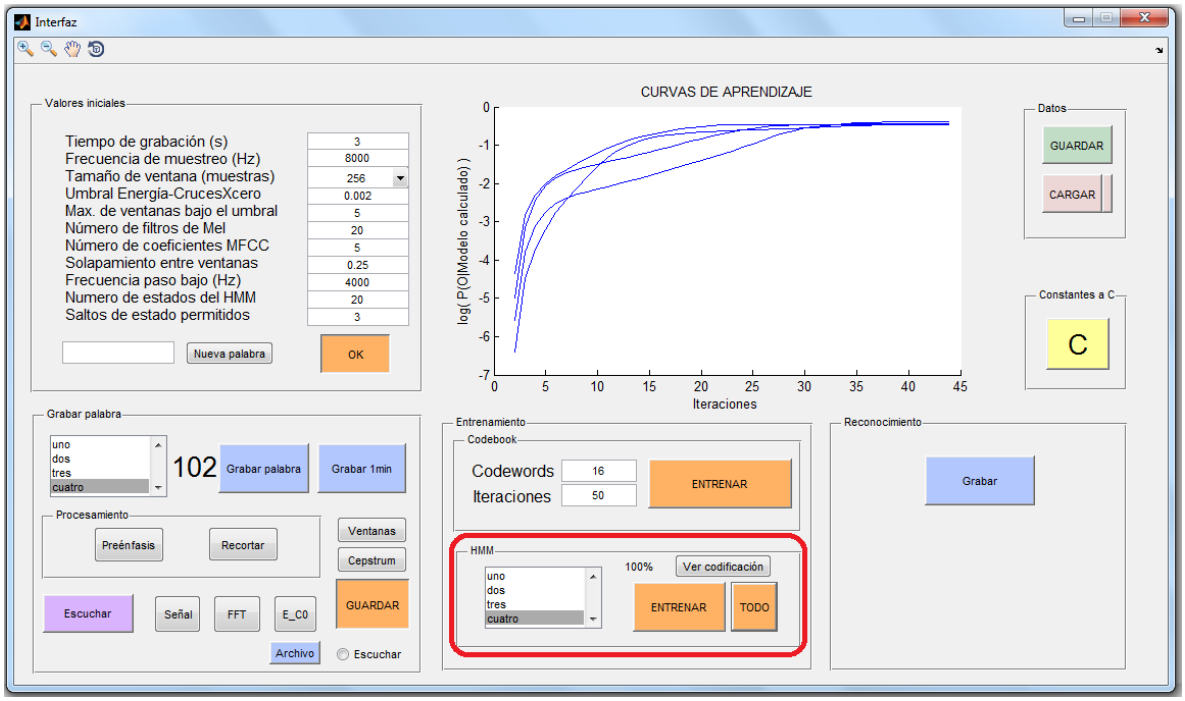

Figura 4.4: Interfaz para el entrenamiento de los HMM.

La figura 4.4 muestra en rojo el panel de entrenamiento de los HMM. El panel se puede utilizar una vez obtenido el codebook. Seleccionando una palabra del Listbox y pulsando el botón *ENTRENAR* se ejecuta una iteración del algoritmo Baum-Welch para el modelo de la palabra seleccionada. Marcando el botón *TODO* se iterará continuamente para todos los modelos, mostrando al final de cada iteración las curvas de aprendizaje como aparece en la figura 4.4. Para detener el proceso volvemos a pulsar el botón para desmarcarlo.

Utiliza las funciones:

- ✓ codificar_palabra.m
- ✓ error_por_distorsion.m
- ✓ algoritmo_forward.m
- ✓ algoritmo_backward.m
- ✓ variable_Epsilon.m
- ✓ variable_Gamma.m
- ✓ algoritmo_BaumWelch.m
- ✓ calculo_HMM_inicio.m
- ✓ iteracion_HMM.m

Reconocimiento de voz

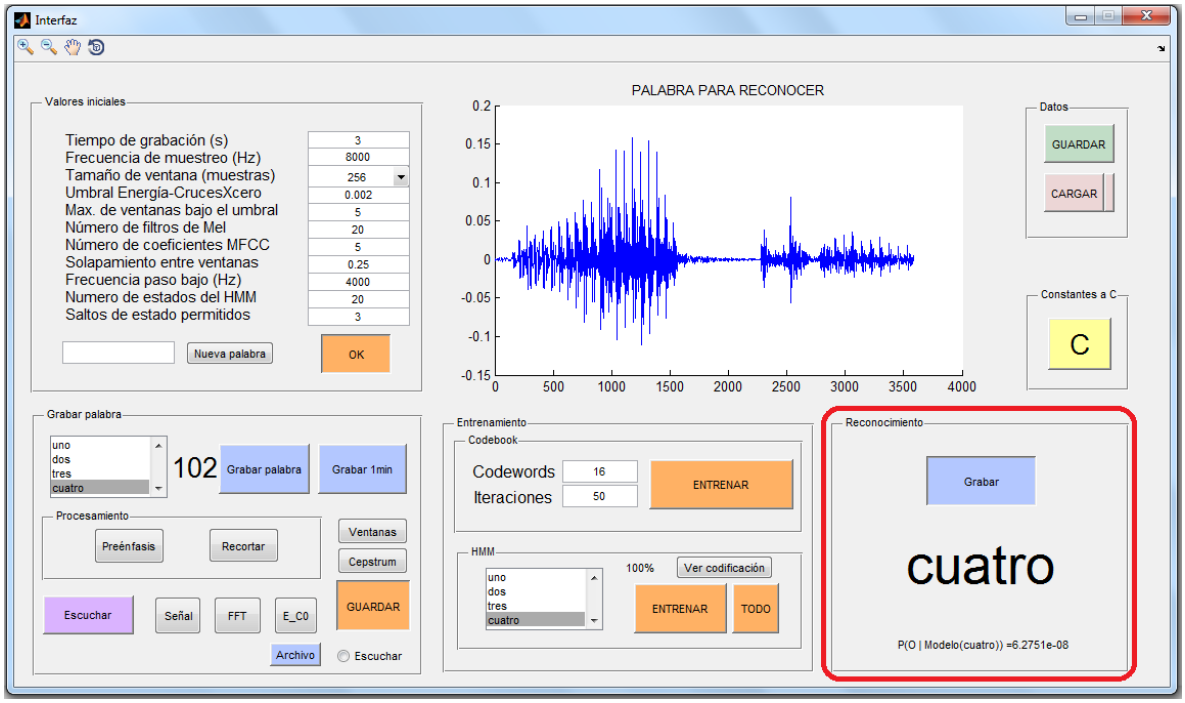

Figura 4.5: Interfaz para el reconocimiento de voz.

La figura 4.5 muestra en rojo el panel de reconocimiento de voz. Se puede utilizar una vez se han entrenado los HMM del vocabulario. Para reconocer una palabra pulsamos el botón *Grabar* y la pronunciamos en el micrófono hasta que se muestre la señal en el gráfico. Si se ha reconocido aparecerá el nombre de la palabra y la probabilidad obtenida del modelo con la que se ha reconocido, como muestra la figura 4.5. Si no, aparecerán signos de interrogación.

Utiliza las funciones:

- ✓ grabar_voz.m
- ✓ filtro_preEnfasis.m
- ✓ analisis_coper.m
- ✓ recortar.m
- ✓ coeficientes_mel.m
- ✓ cepstrum.m
- ✓ analisis_palabra.m
- ✓ codificar_palabra.m
- ✓ error_por_distorsion.m
- ✓ algoritmo_forward_reconocimiento.m

Guardar los resultados

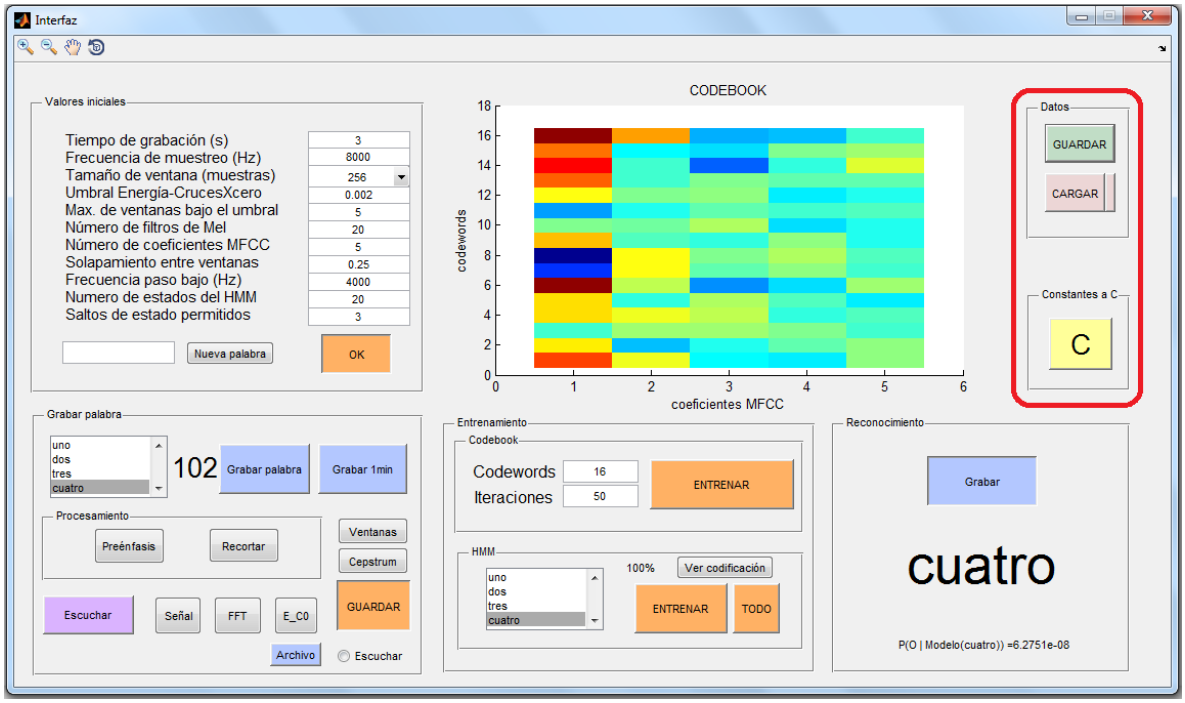

Figura 4.6: Interfaz para guardar los resultados.

Una vez entrenado y comprobado que el sistema de reconocimiento de voz funciona correctamente podemos guardar los resultados. La figura 4.6 muestra en rojo el panel para ello. El botón *GUARDAR* almacena en un fichero llamado *Datos.mat* los parámetros iniciales, la lista de repeticiones de audio grabadas y todos los resultados del entrenamiento de los HMM en formato *struct*. El codebook se guarda en un fichero aparte, llamado *Codebook.mat*.

Mediante el botón *CARGAR* se cargan los datos guardados en la sesión anterior para poder seguir entrenándolos o utilizarlos en reconocimiento.

El botón *C* genera el fichero *constantes.h*, explicado en el subapartado 3.1, en formato de texto con los datos y resultados del entrenamiento en lenguaje C.

4.2. Programa en C de reconocimiento de voz a partir de un archivo WAV para PC

Este programa utiliza las funciones de la librería en lenguaje C desarrollada en el apartado 3, y el fichero *constantes.h* ha sido generado mediante la aplicación Matlab del subapartado 4.1. El objetivo es, a partir de un fichero de audio con una lista de palabras pronunciadas, obtener como resultado la lista en pantalla con las palabras reconocidas. El programa se ejecutará en un PC (Intel Core i7-3612QM CPU@2.10GHz, RAM 6.0GB) con Windows 7.

Se ha utilizado el entorno de desarrollo integrado *Dev-C++* para elaborar el programa. Como ejemplo, en la figura 4.7, se muestra el resultado por consola para un sistema de reconocimiento de voz con un vocabulario de 4 palabras, utilizando un archivo de audio WAV con 33 palabras pronunciadas. Las líneas en blanco son los casos en que la palabra no se ha reconocido.

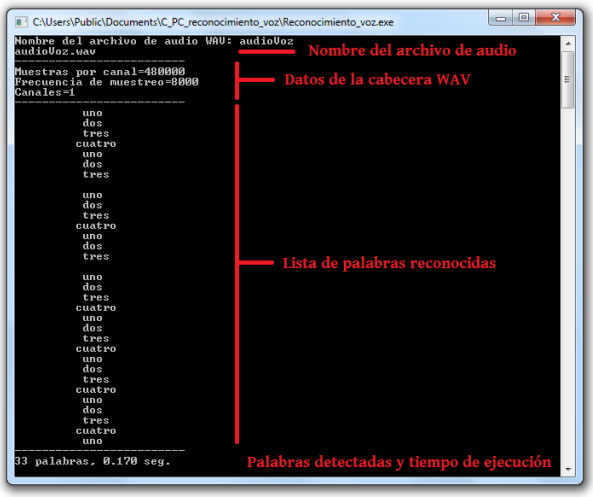

Figura 4.7: resultados por consola del reconocimiento de voz.

La figura 4.8 muestra el fichero con la función principal del programa, *main.c*. La función *leer_archivoAudio* utiliza la librería *sndfile* para leer el archivo WAV, y guardar todas las muestras en *buff_audio*. También muestra por pantalla los datos obtenidos de la cabecera del archivo [8]. Una vez guardadas las muestras, se recorre el buffer buscando el comienzo y final de cada palabra, analizándolas y escribiendo en pantalla el resultado del reconocimiento. Utiliza la librería estándar de C *time.h* para medir el tiempo de ejecución del reconocimiento de voz [9]. Para evaluar concretamente el tiempo de ejecución de las funciones de reconocimiento será necesario suprimir las líneas dedicadas a mostrar los resultados por pantalla.

```c
#include <sndfile.h>
#include <time.h>
#include "analisis_cepstral.h"
#include "codificar_palabra.h"
#include "reconocer_palabra.h"
#include "recortar_audio.h"

int *buf_audio;
int leer_archivoAudio();//Almacena en buff_audio mediante malloc las muestras del WAV
int main()
{
    int16_t palabra_a_reconocer[TAM_BUFF_REC];//<16,15>
    //buffer circular que almacena 1s de audio filtrado
    int tamano_palabra_a_reconocer=0;
    //Tamaño en muestras de la palabra recortada en el buffer
    int inicio_palabra_a_reconocer=0;
    //Inicio de la palabra recortada en el buffer
    int16_t MFCC[TAM_BUFF][TAMANO_CODEWORD];//<16,10>
    //Matriz de los Mel Frecuency Cepstral Coeficients de la palabra recortada
    int palabra_codificada[TAM_BUFF];
    //Palabra codificada con los indices del codebook
    int indice_caracter=0;
    //Indice de escritura del buffer0
    int inicio=1;
    //Indica si es el inicio del programa, para borrar el mensaje de inicio en el display
    int filas_MFCC;
    //Número de ventanas solapadas que tiene la palabra recortada
    int palabra;
    //Indice de la palabra reconocida. -1 si no se ha reconocido
    int *audio;
    clock_t c_ini, c_fin;
    int cont_palabras=0;
    int tamano_audio=0;
    tamano_audio=leer_archivoAudio();
    getch();
    int i;
    int contador_100=0;
    c_ini=clock(); //Tiempo de inicio
    for(i=0;i<tamano_audio;i++)
    {
        int16_t x=buf_audio[i]>>16;
        recortar_audio(x, palabra_a_reconocer, &inicio_palabra_a_reconocer,
                                                    &tamano_palabra_a_reconocer);
        if(tamano_palabra_a_reconocer>MIN_tam_palabra)//Se ha detectado la palabra
        {
            cont_palabras++;
            filas_MFCC=analisis_cepstral(palabra_a_reconocer,
                        inicio_palabra_a_reconocer, tamano_palabra_a_reconocer, MFCC);
            codificar_palabra( MFCC, filas_MFCC, palabra_codificada);
            palabra=reconocer_palabra( palabra_codificada, filas_MFCC);
            if (palabra>=0)//Se ha reconocido
            {
                printf("    ");
                int j;
                for(j=0;j<16;j++)
                    printf("%c", nombre[palabra][j]);
            }
            printf("\n");
            tamano_palabra_a_reconocer=0;
        }
    }
    c_fin=clock();//Tiempo final
    printf("------------------------\n");
    printf("%d palabras, %.3f seg.", cont_palabras, (double)(c_fin-c_ini)/CLOCKS_PER_SEC);
    getch();
    exit(0);
}
```

Figura 4.8: función principal *main*.

Debido a que el procesador del PC dispone de FPU, se pueden utilizar variables de tipo double y obtener mayor precisión en los cálculos del logaritmo y de la DFT. Para desarrollar el código de estas funciones se siguen las especificaciones de las cabeceras; subapartado 3.6. Es necesario hacer casting y multiplicar por una constante de escala para convertir los enteros a reales al comienzo de estas funciones, y los reales a enteros para devolver el resultado.

La función *abs_transformada_Fourier* se muestra en la figura 4.9. Utiliza la función *rdft* incluida en la librería *General Purpose FFT Package* para el cálculo de la DFT en coma flotante [10].

```
#include "abs_transformada_Fourier.h"

#define NMAX 512  //256 muestras Real y compleja
#define NMAXSQRT 16

void abs_transformada_Fourier(int32_t *ventana, uint32_t *abs2_fft)
{
     int ip[NMAXSQRT + 2];
     double a[NMAX + 1], w[NMAX * 5 / 4], temp;
     static int inicio=1;
     if(inicio)
     {
         ip[0]=0;
         inicio=0;
     }
     int i;
     for(i=0;i<NMAX/2;i++) //Convierte a double y escala cada muestra. <16,17> -> double
     {
         a[2*i]=(double)(ventana[i]>>15)*7.6294e-06;    //Parte real
         a[2*i+1]=0;                                    //Parte compleja
     }
     rdft(NMAX, 1, a, ip, w); //FFT
     temp=a[0]*a[0];
     abs2_fft[0]=(uint32_t)(temp*262144);
     for(i=1;i<NMAX/2;i++) //Valor absoluto cuadrado
     {
         temp=a[2*i]*a[2*i]+a[2*i+1]*a[2*i+1];
         abs2_fft[i]=(uint32_t)(temp*262144);//Escala. Convierte a entero. double-><32,18>
     }
}
```

Figura 4.9: función C en coma flotante *abs_transformada_Fourier*.

La función *logaritmo_10* utiliza la función *log10* de la librería estándar *math.h* en coma flotante. El código se muestra en la figura 4.10.

```
#include "logaritmo_10.h"

void logaritmo_10(int16_t *x, int32_t *log10_x, int tamano_x)
{
    double x_d;
    double log10_x_d;
    int i;
    for(i=0;i<tamano_x;i++) //Cada muestra del vector
    {
        x_d=(double)x[i]*3.0518e-05;      //Escala y convierte a double. <16,15> ->double
        log10_x_d=log10(x_d)*32768;       //Logaritmo en base 10 y escala. double-> <32,15>
        log10_x[i]=(int32_t)log10_x_d;    //Convierte a entero.
    }
}
```

Figura 4.10: función C en coma flotante *logaritmo_10*.

4.3. Programa en C de reconocimiento de voz en tiempo real para el ezdsp5535 de Texas Instruments

El *TMS320C5535* de Texas Instruments es un procesador digital de señales (DSP) de coma fija que incluye una unidad de aceleración de FFT (*HWAFFT*). El sistema incluye funciones en su memoria ROM para el uso de esta unidad [12], y el fabricante facilita además una librería con funciones optimizadas (*dsplib*) para este tipo de procesadores en coma fija [11]. También incluye el software IDE *Code Composer Studio* (*CCS*) con el compilador y las herramientas para el desarrollo y test del programa. La figura 4.11 muestra el *ezdsp5535* con el programa de reconocimiento de voz en funcionamiento.

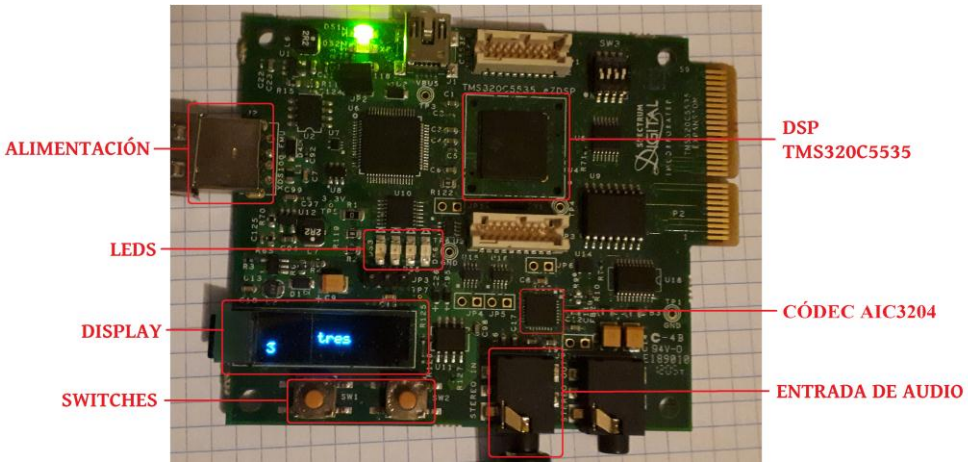

Figura 4.11: programa funcionando en el *ezdsp5535*.

El programa diseñado tiene como objetivo, mediante un micrófono conectado a la entrada de audio del sistema, procesar la señal entrante y detectar el principio y final de la palabra en tiempo real. Una vez detectada, procesarla aplicando las funciones de reconocimiento de voz, mostrar en el display el resultado y continuar analizando la señal de audio entrante. Utiliza las funciones de la librería C de reconocimiento de voz desarrolladas en el apartado 3, y el fichero *constantes.h* obtenido mediante la aplicación desarrollada en el apartado 4.1.

La figura 4.13 muestra la función principal *main* y la subrutina de interrupción para la recepción de audio *ISR_I2S_rx*.

Cuando arranca el programa, la función *inicializa_sistema* ejecuta lo siguiente:

1. Configura la posición en memoria de los vectores de interrupción.
2. Inicializa el puerto serie síncrono I^2C para la configuración del códec.
3. Configura el códec *AIC3204* para la conversión A/D con frecuencia de muestreo de 8.000 Hz, 30 dB de ganancia y recepción monocanal.
4. Inicializa el puerto serie síncrono I^2S para la recepción de datos del códec.
5. Habilita la interrupción *I2S2*.
6. Inicializa el display, los leds y los switches.

7. Habilita las interrupciones globales.

El programa entra en el bucle principal desde el que se producen interrupciones periódicas del puerto *I²S*. Estas interrupciones llaman a la subrutina *ISR_I2S_rx* en la que se lee un nuevo dato de audio y se ejecuta la función *recortar_audio*. Las muestras a la salida de la función se guardan en el buffer circular *palabra_a_reconocer*.

Cuando se detecta el final de la palabra se indica con los valores en las variables *inicio_palabra_a_reconocer* y *tamano_palabra_a_reconocer*. Se deshabilitan las interrupciones del *I2S2* y se analiza el segmento de voz mediante la función *analisis_cepstral*. La matriz MFCC se codifica con la función *codificar_palabra*, y el vector resultado se reconoce con la función *reconocer_palabra*. La palabra se muestra en el display mediante la función *escribe_en_display*. Esta función también ilumina el led verde para indicar que se ha reconocido.

Se vuelven a habilitar las interrupciones del *I2S2* y se continúa en el bucle principal recibiendo muestras de audio.

Pulsando uno de los dos switches, la función *lee_pulsadores* aumenta o disminuye en una unidad la ganancia del códec. Esto es necesario para ajustar el volumen del audio de entrada con el utilizado en el micrófono del PC durante el entrenamiento. Pulsando los dos switches a la vez se reinicia a 30dB.

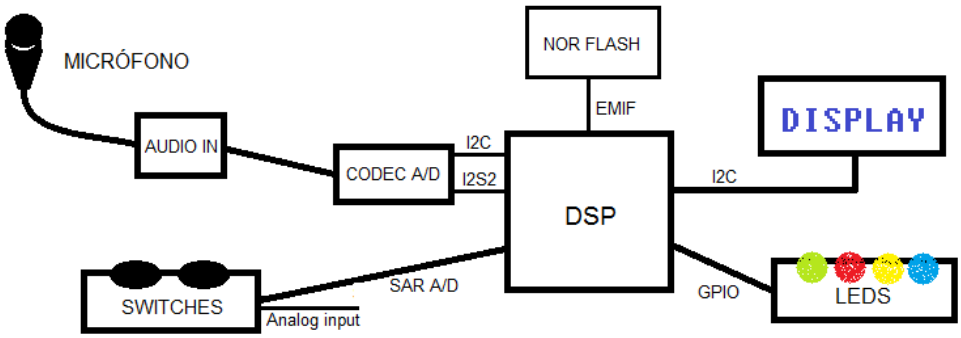

Figura 4.12: diagrama de bloques del demostrador en el *ezdsp5535*.

En la figura 4.12 se muestra el diagrama de bloques con el DSP y los periféricos principales empleados.

```
#include <analisis_cepstral.h>
#include <codificar_palabra.h>
#include <reconocer_palabra.h>
#include <recortar_audio.h>

int16_t palabra_a_reconocer[TAM_BUFF_REC];//<16,15> buffer circular.Almacena 1s de audio filtrado
int tamano_palabra_a_reconocer=0;       //Tamaño en muestras de la palabra recortada en el buffer
int inicio_palabra_a_reconocer=0;         //Inicio de la palabra recortada en el buffer

interrupt void ISR_I2S_rx(void);//Interrupción de recepción de audio
void inicializa_sistema(void);  //Inicializa el sistema y habilita la interrupción del micrófono
void escribe_en_display(int);   //Muestra en el display el resultado de la palabra de entrada
void lee_pulsadores(void);      //Lee pulsadores para controlar el display

void main(void)
{
    int16_t MFCC[TAM_BUFF][TAMANO_CODEWORD];//<16,10> Matriz de los MFCC
    int palabra_codificada[TAM_BUFF];       //Palabra codificada con los indices del codebook
    int filas_MFCC;                 //Número de ventanas solapadas que tiene la palabra recortada
    int palabra;                    //Indice de la palabra reconocida. -1 si no se ha reconocido
    inicializa_sistema();
    while(1)
    {
       if(tamano_palabra_a_reconocer>MIN_tam_palabra)//Se ha detectado la palabra
       {
           CPU->IER0=0;//Desabilita interrupción de recepción de I2S2
           filas_MFCC=analisis_cepstral(palabra_a_reconocer, inicio_palabra_a_reconocer,
                                                              tamano_palabra_a_reconocer, MFCC);
           codificar_palabra( MFCC, filas_MFCC, palabra_codificada);
           palabra=reconocer_palabra( palabra_codificada, filas_MFCC);
           if (palabra>=0)//Se ha reconocido
           {
               escribe_en_display(palabra);
           }
           tamano_palabra_a_reconocer=0;
           CPU->IER0=1<<15;//Habilita interrupción de recepción de I2S2
       }
       lee_pulsadores();
    }
}
interrupt void ISR_I2S_rx(void)//Se ejecuta a la frecuencia de muestreo Fs=8000Hz
{
    int16_t rx_R;
    int32_t datoR_32;
    I2S_read32(I2S2,&datoR_32,&datoR_32); //Lee de la salida del conversor A/D
    rx_R=datoR_32>>16;
    recortar_audio(rx_R, palabra_a_reconocer, &inicio_palabra_a_reconocer,
                                                                  &tamano_palabra_a_reconocer);
}
```

Figura 4.13: Funciones C *main* e *ISR_I2S_rx*.

La función *abs_transformada_Fourier* se muestra en la figura 4.14. Utiliza las funciones *hwafft_br* y *hwafft_256pts* para ventanas de 256 muestras. Utilizan la unidad hardware HWAFFT y sus códigos los incluye el fabricante en la memoria ROM del sistema. Para utilizarlas es necesario incluir el archivo *hwafft.h* en la cabecera de la función, y especificar en el *linker* del programa la posición en memoria ROM de las funciones. También es necesario colocar los buffers utilizados por las funciones en posiciones concretas de memoria [11, 12], como se muestra en la parte superior de la figura 4.13.

```c
#include "abs_transformada_Fourier.h"
#include "hwafft.h"

#define ALIGNMENT 2*256
// Almacena el buffer de entrada de la función hwfft_br en una posición alineada de memoria
#pragma DATA_SECTION(data_br_buf, "data_br_buf");
#pragma DATA_ALIGN (data_br_buf, ALIGNMENT);
int32_t data_br_buf[256];
int32_t scratch_buf[256];

void abs_transformada_Fourier(int32_t *ventana, uint32_t *abs2_fft)
{
    uint16_t out_sel;              //Indica donde está la salida de hwfft
    int32_t *data_br = data_br_buf;//buffer para la fft
    int32_t *scratch = scratch_buf;//buffer para la fft
    int16_t real;//<16,9>            Parte real de la transformada de Fourier
    int16_t imag;//<16,9>            Parte imaginaria de la transformada de Fourier

    hwafft_br(ventana, data_br, 256);// Bit reverse
    out_sel=hwafft_256pts(data_br, scratch, scratch, data_br, FFT_FLAG , SCALE_FLAG );
    int j;                                                          // EScala-> /256=2^8
    if (out_sel==OUT_SEL_DATA)//La transformada de Fourier está en data_br
    {
        for (j=0;j<128;j++)
        {
             imag=data_br[j] & 0x0000FFFF;
             real=data_br[j]>>16;
             abs2_fft[j]=(int32_t)real*(int32_t)real+(int32_t)imag*(int32_t)imag;
        }
    }
    else//La transformada de Fourier está en scratch
    {
        for (j=0;j<128;j++)
        {
             imag=scratch[j] & 0x0000FFFF;
             real=scratch[j]>>16;
             abs2_fft[j]=(int32_t)real*(int32_t)real+(int32_t)imag*(int32_t)imag;
        }
    }
}
```

Figura 4.14: función C en coma fija *abs_transformada_Fourier*.

La función *logaritmo_10* utiliza *log_10* de la librería *C55x DSPLIB* de Texas Instruments [11]. El código se muestra en la figura 4.15. Es necesario incluirla en la cabecera de la función.

```c
#include "logaritmo_10.h"
#include <dsplib.h>

void logaritmo_10(int16_t *x, int32_t *log10_x, int tamano_x)
{
    log_10 (x, log10_x, tamano_x);
}
```

Figura 4.15: función C en coma fija *logaritmo_10*.

4.4. Resultados experimentales

Para evaluar el funcionamiento de los demostradores desarrollados en este apartado se va a realizar una serie de pruebas experimentales y a exponer los resultados obtenidos. El objetivo es comprobar el funcionamiento, y observar cómo influye en el resultado el proceso de entrenamiento y las variables iniciales utilizadas. También se pretende comparar los resultados de los tres sistemas para exponer en las conclusiones sus ventajas e inconvenientes.

Los datos iniciales utilizados en los experimentos son los siguientes:

- Frecuencia de muestreo: 8.000 Hz.
- Tamaño de ventana: 256 muestras.
- Solapamiento entre ventanas: 25%.
- Umbral mínimo de energía y cruces por cero de la ventana: 0.002.
- Ventanas bajo el umbral para la detección del final de palabra: 5.
- Número de filtros de Mel: 20.
- Número de coeficientes cepstrales por ventana: 5.
- Frecuencia máxima paso bajo: 4.000 Hz.
- Estados de los HMM: 20.
- Saltos de estado permitidos: 3.

El micrófono utilizado para el entrenamiento y reconocimiento es el micrófono electret incorporado en el Smartphone Samsung Galaxy A5. El entrenamiento y reconocimiento se realizan en la misma sala por un único hablante. Se utiliza la misma grabación de audio con 23 repeticiones de cada palabra para evaluar los dos sistemas en lenguaje C. El sistema en lenguaje Matlab se evalúa directamente con el micrófono, repitiendo cada palabra 23 veces.

En primer lugar se comprueban las tasas de reconocimiento y el tiempo de ejecución medio, desde el momento en que la palabra está almacenada en memoria hasta que se reconoce. Se varía el número de entrenamientos por palabra y el tamaño del codebook. Las 3 palabras del vocabulario entrenado son las siguientes: “uno”, “dos” y “tres”. El DSP trabaja a 100MHz. Las tablas 4.1, 4.2 y 4.3 muestran los resultados.

10 Entrenamientos/palabra									
Codewords	**8**			**16**			**32**		
	PC (Matlab)	**PC (C)**	**DSP (C)**	**PC(Matlab)**	**PC (C)**	**DSP (C)**	**PC(Matlab)**	**PC (C)**	**DSP (C)**
uno	0.609	0.522	0.391	0.435	0.130	0.000	0.348	0.000	0.000
dos	0.478	0.000	0.174	0.609	0.000	0.043	0.000	0.000	0.043
tres	0.652	0.000	0.000	0.261	0.043	0.000	0.500	0.043	0.174
TOTAL	**0.580**	**0.174**	**0.188**	**0.435**	**0.058**	**0.014**	**0.283**	**0.014**	**0.072**
T. medio(ms)	7.67	1.82	30.4	8.306	2.12	31.1	8.425	2.12	36.28

Tabla 4.1: Tasas de reconocimiento y tiempo de ejecución para 10 entrenamientos/palabra.

50 Entrenamientos/palabra									
Codewords	**8**			**16**			**32**		
	PC (Matlab)	**PC (C)**	**DSP (C)**	**PC(Matlab)**	**PC (C)**	**DSP (C)**	**PC(Matlab)**	**PC (C)**	**DSP (C)**
uno	1.000	0.957	0.913	0.826	0.913	0.783	0.696	0.000	0.130
dos	0.609	0.435	0.130	0.652	0.348	0.304	0.826	0.304	0.217
tres	0.739	0.522	0.565	0.739	0.217	0.261	0.250	0.087	0.000
TOTAL	**0.783**	**0.638**	**0.536**	**0.739**	**0.493**	**0.449**	**0.591**	**0.130**	**0.116**
T. medio(ms)	6.779	1.82	26.17	8.019	2.12	27.56	8.548	2.12	31.27

Tabla 4.2: Tasas de reconocimiento y tiempo de ejecución para 50 entrenamientos/palabra.

100 Entrenamientos/palabra									
Codewords	**8**			**16**			**32**		
	PC (Matlab)	**PC (C)**	**DSP (C)**	**PC(Matlab)**	**PC (C)**	**DSP (C)**	**PC(Matlab)**	**PC (C)**	**DSP (C)**
uno	0.870	0.609	0.783	0.957	1.000	0.957	0.913	0.913	0.957
dos	1.000	0.913	0.696	0.739	0.565	0.522	0.696	0.348	0.304
tres	0.783	0.522	0.478	0.913	0.739	0.783	0.696	0.261	0.130
TOTAL	**0.884**	**0.681**	**0.652**	**0.870**	**0.768**	**0.754**	**0.804**	**0.507**	**0.464**
T. medio(ms)	8.264	1.82	20.83	8.452	2.12	22.356	9.069	2.12	33.44

Tabla 4.3: Tasas de reconocimiento y tiempo de ejecución para 100 entrenamientos/palabra.

Se observa cómo la tasa de reconocimiento aumenta con el número de entrenamientos. También se aprecia un mejor resultado con codewords pequeños. Esto puede deberse a que, al haber menos símbolos, las matrices requieren un menor entrenamiento. El tiempo de ejecución aumenta con el tamaño del codebook, debido a que requiere un mayor número de comparaciones en el proceso de codificación y los cálculos con las matrices de observaciones son más grandes.

A continuación se comprueban los resultados al variar el número de MFCCs por ventana. Se utiliza un codebook de 16 elementos y 100 entrenamientos por palabra. La tabla 4.4 muestra los resultados.

MFCCs	**3**			**5**			**7**		
	PC (Matlab)	**PC (C)**	**DSP (C)**	**PC(Matlab)**	**PC (C)**	**DSP (C)**	**PC(Matlab)**	**PC (C)**	**DSP (C)**
uno	0.783	1.000	0.826	1.000	0.913	1.000	0.826	0.783	0.913
dos	0.870	0.739	0.739	0.870	0.478	0.696	0.696	0.522	0.391
tres	0.478	0.391	0.174	0.739	0.652	0.261	0.565	0.565	0.261
TOTAL	**0.710**	**0.710**	**0.580**	**0.870**	**0.681**	**0.652**	**0.696**	**0.623**	**0.522**
T. medio(ms)	8.376	2.12	21.5	8.807	2.12	22.42	9.071	2.12	23.56

Tabla 4.4: Tasas de reconocimiento y tiempo de ejecución variando el número de MFCCs.

Se observa como los resultados son ligeramente mejores con un número de MFCCs en torno a 5 elementos. También se aprecia un aumento en el tiempo de procesamiento con el aumento de los MFCCs debido a que se requieren cálculos con vectores de mayor tamaño.

Por último se entrena un vocabulario de 10 palabras mediante 150 entrenamientos por palabra. El vocabulario entrenado es: “uno”, “dos”,…, “nueve” y “cero”. Se emplean 5 MFCCs por ventana y un codeword de 16 elementos. La tabla 4.5 muestra los resultados.

	PC (Matlab)	PC (C)	DSP (C)
uno	0.652	0.696	0.435
dos	0.783	0.739	0.826
tres	0.870	0.913	0.739
cuatro	0.783	0.652	0.696
cinco	0.696	0.609	0.435
seis	0.739	0.565	0.478
siete	0.739	0.696	0.739
ocho	0.957	0.913	0.826
nueve	0.826	0.783	0.565
cero	0.913	0.826	0.870
TOTAL	**0.796**	**0.739**	**0.661**
T. medio(ms)	11.815	5.21	42.32

Tabla 4.5: Tasas de reconocimiento y tiempo de ejecución para un vocabulario de 10 palabras.

Se comprueba como al aumentar el número de palabras en el vocabulario se reduce la tasa de reconocimiento. Esto se debe a que se producen confusiones entre palabras fonéticamente similares. También se aprecia un aumento en el tiempo de procesamiento debido a que la secuencia introducida se compara con un mayor número de modelos.

En general se observa que los mejores resultados se obtienen con las funciones en Matlab, ya que utilizan precisión de coma flotante de 64 bits en todos sus cálculos. Con los demostradores en C se obtienen mejores resultados en PC. Esto se debe a que dispone de FPU y los cálculos de la FFT y el logaritmo son mas precisos; en formato double de 64 bits. El DSP realiza éstos cálculos en coma fija con enteros de 32 bits.

5. Conclusiones

Durante este trabajo se ha hecho un estudio, análisis y desarrollo de funciones para el tratamiento de señales de audio y reconocimiento de voz en los lenguajes de programación Matlab y C. Se han aplicado los conocimientos teóricos obtenidos del estudio de los modelos ocultos de Markov y del análisis de señales de audio en la programación de una librería de utilidad práctica en futuros proyectos en los que sea necesario el reconocimiento de comandos de voz.

Desde el punto de vista académico se han desarrollado nuevos conocimientos en el tratamiento digital de señales, y se han reforzado y llevado a la práctica los ya adquiridos. Se han mejorado las habilidades de programación en los lenguajes C y Matlab, y ha sido necesario aprender a utilizar herramientas desconocidas de programación, como el entorno de desarrollo de GUI de Matlab y las librerías para el tratamiento de datos y audio en lenguaje C. También se ha profundizado en el conocimiento de los sistemas empotrados en coma fija, en concreto del ezdsp5535, de su arquitectura y capacidades, y de las herramientas disponibles para el desarrollo de proyectos. Ha sido necesario realizar un estudio previo y búsqueda en numerosas fuentes sobre la materia, y redactar la presente memoria utilizando el lenguaje y estructura de un documento técnico.

5.1 Valoración de resultados

A partir de los tres demostradores elaborados en el apartado 4 y de los resultados experimentales obtenidos, se puede afirmar que la librería funciona correctamente. Además se han cumplido los objetivos iniciales del proyecto expuestos en el apartado 1. El propósito del trabajo no era obtener una alta tasa de reconocimiento de voz sino, conociendo las limitaciones que ofrecen los sistemas empotrados en coma fija, elaborar una librería multiplataforma de reconocimiento de voz sencilla y funcional.

La tasa de reconocimiento varía dependiendo de los parámetros iniciales utilizados. Cuanto mayor es el número de repeticiones en el entrenamiento de una palabra, mayor es la tasa obtenida para ésta. El número de palabras en el vocabulario también influye, pues se producen errores por confusión entre palabras fonéticamente similares; esto puede reducirse mediante la utilización de un nivel de probabilidad de salida mínimo. También puede comprobarse como influye la precisión de los datos en el resultado. La utilización de precisión en coma fija limita en gran medida las posibilidades de mejora en la tasa de reconocimiento.

5.2 Líneas futuras de trabajo

El factor que más influye en los resultados del trabajo es el entrenamiento del sistema, y el éxito del reconocimiento de una palabra depende de que en el entrenamiento se hayan utilizado muestras de señal similares. Las muestras utilizadas para el entrenamiento son obtenidas mediante el micrófono conectado al PC, y no son exactamente como las utilizadas en el reconocimiento con el micrófono conectado al sistema empotrado. Una mejora que aumentaría la tasa de reconocimiento sería desarrollar un método de obtención de las muestras de audio directamente del sistema

empotrado en el que se va a reconocer, y transferirlas al PC en el que realizar el entrenamiento. En el caso del ezdsp5535 esto podría hacerse mediante el puerto USB disponible en la placa.

6. Referencias

[1] Speech recognition. En *Wikipedia, la enciclopedia libre*. Fecha de consulta: 24 de junio de 2018. Disponible en https://en.wikipedia.org/wiki/Speech_recognition

[2] L. R. Rabiner, A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition, 1989.

[3] F. Martinez, G. Portale, H. Klein, O. Olmos, «Reconocimiento de voz, apuntes de cátedra para Introducción a la Inteligencia Artificial»

[4] J. Gómez Durán, J. Simancas García, M. Acosta Coll, F. Melendez Pertuz, J. Vélez Zapata, «Algoritmo de reconocimiento de comandos voz basado en técnicas no-lineales», *Revista Espacios*, Vol. 38 (Nº17), 2017.

[5] B. Juang, L. Rabiner, Fundamentals of Speech Recognition, N.J.: Prentice Hall, 1993.

[6] F. Peralta Reyes, A. Cotrina Atencio, «Algoritmo coper para la detección de actividad de voz», facultad de Ingeniería Electrónica de la UNMSM, 2002.

[7] V. Tyagi, C. Wellekens, «On desensitizing the Mel-Cepstrum to spurious spectral components for Robust Speech Recognition», 2005.

[8] E. Castro Lopo. LIBSNDFILE. Fecha de consulta: mayo de 2008. Disponible en http://www.mega-nerd.com/libsndfile/

[9] The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004.

[10] Takuya Ooura. General Purpose FFT (Fast Fourier/Cosine/Sine Transform) Package. Fecha de consulta: mayo de 2018. Disponible en: http://www.kurims.kyoto-u.ac.jp/~ooura/fft.html

[11] TMS320C55x DSP Library. SPRU422J. May 2000 .Revised May 2013.

[12] TMS320C5505 DSP System User's Guide. SPRUGH5B. November 2010. Revised May 2014.

[13] TMS320C5535/34/33/32 Fixed-Point Digital Signal Processor Silicon Revision 2.2.

[14] C Coding Standards for EECS 381, 2016.

[15] David Kieras, Why and How You Should Comment Your Code, EECS Department, 2003.

[16] David Kieras, C Header File Guidelines, EECS Dept., University of Michigan, 2012.

[17] F. Javier Salcedo Campos. Modelos Ocultos de Markov: Del Reconocimiento de Voz a la Música. Universidad de Granada.

[18] The voice foundation 2017, Philadelphia (USA). Understanding voice production Fecha de consulta: 7 de junio de 2018. Disponible en www.voicefoundation.org

[19] Señal de voz. En *Wikipedia, la enciclopedia libre*. Fecha de consulta: 7 de junio de 2018. Disponible en https://es.wikipedia.org/wiki/Se%C3%B1al_de_voz

[20] Fonema. En *Wikipedia, la enciclopedia libre*. Fecha de consulta: 7 de junio de 2018. Disponible en https://es.wikipedia.org/wiki/Fonema

[21] C. George Boeree. Fonética. Fecha de consulta: 7 de junio de 2018. Disponible en http://webspace.ship.edu/cgboer/foneticaesp.html

[22] Micrófono. En *Wikipedia, la enciclopedia libre*. Fecha de consulta: 7 de junio de 2018. Disponible en https://es.wikipedia.org/wiki/Micr%C3%B3fono

Anexos

I. Características de la señal de voz

Una de las principales áreas de trabajo del procesamiento digital de señales es el procesado digital de voz. Para entender como funciona el reconocimiento de patrones de voz es necesario entender como se produce la voz en el aparato fonador humano y sus características básicas.

I.1. Producción de la voz en el aparato fonador humano

Una onda sonora es una onda longitudinal que se propaga con forma esférica en medios elásticos, produciendo una variación local de su presión y densidad. Las variaciones de presión producen un movimiento de vibración en las moléculas transmitiendo a su vez el movimiento a las moléculas vecinas. La percepción de estas variaciones de presión por el oído produce la sensación de sonido. En general, el oído humano puede percibir vibraciones entre 20 y 20.000 Hz, el llamado espectro audible [18].

El aparato fonador humano se compone del conjunto de órganos utilizados en la función del habla. El sonido es generado por el aire que es impulsado con fuerza desde los pulmones, recorre las cavidades laríngeas y supra-glóticas y es expulsado por la nariz y boca. En la cavidad laríngea, al final de la tráquea, se encuentran las cuerdas vocales, que son dos repliegues membranosos que vibran durante la expulsión del aire. Durante el proceso del habla las cuerdas vocales están continuamente abriendo y cerrándose, produciendo los llamados fonemas sonorizados y no sonorizados. El tamaño de las cuerdas vocales y sus pliegues caracterizan el tono de la voz, que permite distinguir a cada hablante, y si el hablante es un niño o un adulto, un hombre o una mujer [18].

En los fonemas sonorizados las cuerdas vocales están cerradas y vibran a la denominada frecuencia fundamental, entre 110 y 300 ciclos por segundo, dependiendo del tono. El sonido producido por las cuerdas vocales en tensión se debe al el efecto Bernoulli, al atravesar el aire el estrecho hueco entre las ellas. Es una especie de zumbido que se aproxima, en el modelado digital, a un tren de impulsos constante [3, 18].

En los no sonorizados las cuerdas están abiertas y el aire es expulsado sin resistencia por la tráquea, produciendo aproximadamente ruido blanco.

Las sucesiones de trenes de pulsos y ruido blanco producidos en la laringe son moldeados y filtrados a su paso por el resto de cavidades que funcionan como elementos resonantes. Este conjunto de cavidades se denomina tracto vocal y se compone de tres partes; la faringe, la cavidad oral y el tracto nasal. La fuerza con la que los pulmones expulsan el aire determina el volumen del sonido, y la posición de la lengua permite dirigir el aire por unos conductos u otros, produciendo diferentes tipos de resonancias. La lengua, el paladar y los labios intervienen en la producción consciente de los diferentes sonidos, son los elementos de articulación de las palabras [18].

En el modelo *Source-Filter* para la síntesis digital de voz, el tracto vocal se modela como un filtro lineal de parámetros variables, y un amplificador de ganancia variable a su entrada que determina la intensidad de la excitación. Los elementos del filtro varían de forma mucho mas lenta que la excitación, por lo que se considera cuasi-estático. La acción de los elementos de articulación está considerada en los parámetros variables del filtro. Los fonemas sonorizados y no sonorizados se producen en fuentes diferentes, que se conectan a la entrada mediante un interruptor [3].

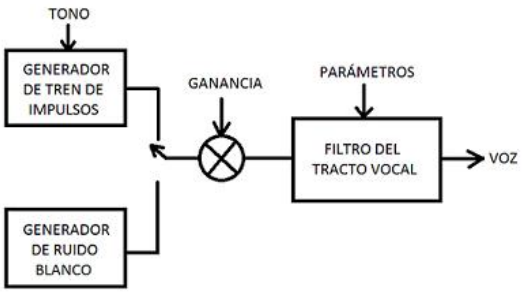

Figura A.1: Modelo Source-Filter para la síntesis digital de voz.

I.2. Tipos de fonemas y sus características principales

Los fonemas son las unidades básicas que componen el sonido hablado. Se dividen en dos tipos principales; vocálicos y consonantes [19].

Vocálicos

Los vocálicos son fonemas sonoros que se producen mediante la excitación de la cavidad oral por los pulsos aire, generados en las cuerdas vocales sin la obstrucción del tracto vocal. El periodo de estos pulsos es el tono. La vibración de las cuerdas vocales produce ondas sonoras con un espectro distribuido, y a su paso por el tracto vocal algunas frecuencias son atenuadas y otras reforzadas. La boca permanece abierta y la forma y tamaño de las cavidades atravesadas determinan las frecuencias resonantes, llamadas también formantes, que son las bandas de frecuencia que salen reforzadas, y en las que se concentra la mayor parte de la energía del espectro de la señal. Las vocales tienen tres formantes principales entre los 200 y 3.000 Hz, dependiendo de la persona. Cada sonido vocal produce un reparto y posición de la energía diferente en el espectro, siendo esto clave en el proceso de reconocimiento de voz. Existen tantas formantes como cavidades resonantes hay en el tracto, pero en la mayoría de los lenguajes basta con las dos primeras formantes para caracterizar un fonema vocal. El resto de formantes caracterizan otras propiedades del sonido, como el timbre. En síntesis digital de voz se modela como un generador de trenes de impulsos de frecuencia variable y filtrado con el filtro del tracto vocal [4].

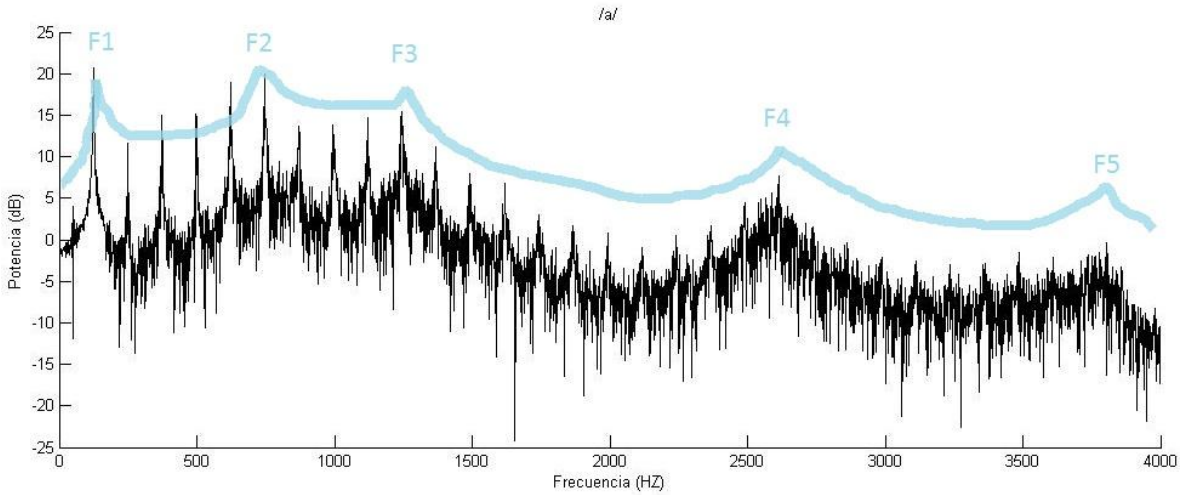

Figura A.2: Espectro del fonema vocálico /a/ y sus formantes.

El la figura A.2 observamos un primer pico en 125 Hz. Esta es la frecuencia fundamental generada por las cuerdas vocales, y los demás picos que se repiten cada 125 Hz son sus armónicos. Las formantes son las acumulaciones de energía que se aprecian como máximos locales en la envolvente del espectro [3].

Consonantes

En los fonemas consonantes las cuerdas vocales están relajadas y no producen vibración, con algunas excepciones. El aire sale con fuerza por la faringe y es obstruido a su paso por el tracto vocal, formando una turbulencia audible. Las consonantes quedan caracterizadas fonéticamente por el modo y lugar en el que se produce la obstrucción, y el sonido está determinado por la forma y la duración en el tiempo del ruido generado en la turbulencia. En síntesis digital de voz se modela como un generador de ruido aleatorio conectado a su salida a un interruptor y al filtro del tracto vocal.

Si las cuerdas vocales vibran o no tenemos consonantes sonoras y no sonoras. En castellano algunas consonantes sonoras son la m, la b y la d. Las consonantes sonoras se diferencian de las vocales en que hay una obstrucción en la salida del aire, la boca no permanece abierta.

Las consonantes plosivas se caracterizan en que el aire es acumulado durante un breve espacio de tiempo antes de ser expulsado, produciéndose un momento de silencio antes de escucharse el sonido. En castellano tenemos la p, la t y la k entre otras [20].

II. Factores que afectan a la calidad en grabadoras digitales de audio

Los factores que influyen en la calidad de recepción de audio en un sistema digital son; el ruido, la acústica del entorno y la calidad del micrófono, entre otros.

Ruido

El ruido en un sistema digital proviene de diversas fuentes. El producido en el entorno por aparatos electrónicos, murmullos, soplidos en el micrófono..., que se combinan con la señal de audio aditivamente y reducen la relación señal-ruido de entrada al sistema.

El proceso de cuantificación en la conversión analógico-digital produce una señal digital con una relación señal-ruido, respecto a la señal analógica, según la fórmula:

$$SNRQ(dB) = 6.0206 \cdot b + 1.7609$$

Donde *b* es el número de bits utilizados en la cuantificación. El valor mínimo requerido es de 8 bits para una calidad baja. Se obtiene una gran mejora en la calidad utilizando una cuantificación de 16 bits, que es la más habitual.

Acústica

La acústica del entorno en el que se está percibiendo la señal de voz puede producir alteraciones en el espectro debido a la resonancia de las paredes, en el caso de estar en una habitación, y producirse ecos por el impacto de la señal en superficies y objetos cercanos, que llegarán al receptor reflejados y con retardo respecto a la señal original. Esto influye negativamente en los sistemas de reconocimiento de voz, haciendo que la calidad dependa en gran medida del lugar en el que se lleve a cabo.

Micrófonos

El micrófono es el encargado de transformar la energía acústica en eléctrica. El modo más habitual de clasificarlos es según la forma en la que hacen esta transformación. Las características de un micrófono que nos interesan para el reconocimiento de voz son la sensibilidad, que es la capacidad de captar con precisión señales acústicas de baja potencia, y la respuesta en frecuencia, que es el rango de frecuencias que es capaz de capturar sin perder sensibilidad. Existe una amplia variedad de micrófonos, los dos mas utilizados actualmente son los dinámicos y de condensador [21].

El micrófono dinámico o de bobina móvil. El sonido se transforma mediante la vibración de una membrana, que hace mover un solenoide sujeto a ésta en el interior de un campo magnético, generando una corriente eléctrica. Admiten un nivel muy alto de presión sonora y son muy robustos, pero ofrecen una baja sensibilidad y pobre respuesta en altas frecuencias.

El micrófono de condensador o electret. Se basa en los cambios que se producen en la capacidad de un condensador de placas al variar la distancia de separación entre éstas. Una de las placas está sujeta a una membrana como en el caso anterior y la otra permanece fija. Son los más utilizados en estudios de grabación debido a su gran sensibilidad y a que producen una respuesta plana en un amplio rango de frecuencias. Su desventaja es que necesitan alimentación externa y son más frágiles [21].

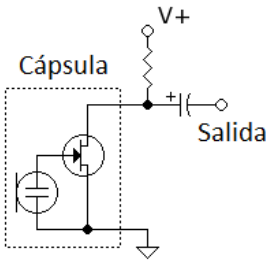

Figura A.3: Circuito típico de micrófono electret con preamplificador y alimentación.

Los micrófonos de los teléfonos móviles actuales son de tipo condensador. Durante el desarrollo de este trabajo se ha utilizado el micrófono incorporado en el Smartphone Samsung Galaxy A5 (2017).

III. Introducción a los modelos ocultos de Markov para reconocimiento de voz

Un proceso discreto de Markov está limitado a sistemas en los que cada estado corresponde a un evento observable. Este modelo es muy restrictivo para ser aplicado a otros muchos problemas reales; por ello, surge la necesidad de extender el concepto para incluir los casos en los que la observación es una función probabilística del estado. Un modelo oculto de Markov (HMM) describe un proceso doblemente estocástico, en el que uno de sus procesos no es observable y sus características solo pueden determinarse mediante otro proceso estocástico, que son los resultados observables que produce el modelo.

Las características de un HMM de tiempo discreto son [2]:

1. N, es el número de estados en el proceso. A pesar de que los estados están ocultos tienen un significado físico y resulta relevante el modo en que se interconectan entre si. Los estados individuales se denotan como $S=\{S_1,\ldots,S_N\}$ y el estado del sistema en el momento *t* como q_t.

2. M, es el número de símbolos generados en cada estado. Estos son los resultados observables que produce el modelo. Se denotan como $V=\{V_1,\ldots,V_M\}$.

3. A, es la matriz NxN de probabilidades de transición entre estados. El elemento a_{ij} se define como la probabilidad de transición del estado S_i, al S_j:

$$a_{ij} = P[q_{t+1} = S_j | q_t = S_i], \quad 1 \leq i, j \leq N$$

   La suma de los elementos de cada fila es igual a uno.

4. B, es la matriz NxM de probabilidades de las observaciones. El elemento $b_j(k)$ se define como la probabilidad de observar el símbolo *k* cuando el sistema se encuentra en el estado S_j:

$$b_j(k) = P[v_k \text{ en } t | q_t = S_j], \quad 1 \leq j \leq N, \quad 1 \leq k \leq M$$

   La suma de los elementos de cada fila es igual a uno.

5. Π, es la matriz Nx1 de probabilidades de estado iniciales. El elemento π_i es la probabilidad de que en el momento inicial, t=1, el sistema se encuentre en el estado S_i:

$$\pi_i = P[q_1 = S_i], \quad 1 \leq i \leq N$$

   La suma de sus elementos es igual a uno.

La notación compacta utilizada para definir un HMM es $\boldsymbol{\lambda}=(A, B, \pi)$ y la cadena de observaciones generada por el modelo hasta el instante t=T, $\mathbf{O}=O_1, O_2 \ldots O_T$.

En reconocimiento de voz se imponen una serie de restricciones iniciales al modelo que simplifican el análisis. Debido a que la señal de voz es un proceso temporal se supone que siempre se comienza en el estado S_1, y se avanza de izquierda a derecha, nunca al revés. También se impone un número máximo de transiciones entre estados. Típicamente de cada estado se puede saltar al siguiente y al

posterior al siguiente, en caso de que exista, o quedarse en el mismo. La figura A.4 muestra un ejemplo del diagrama de estados obtenido tras estas restricciones.

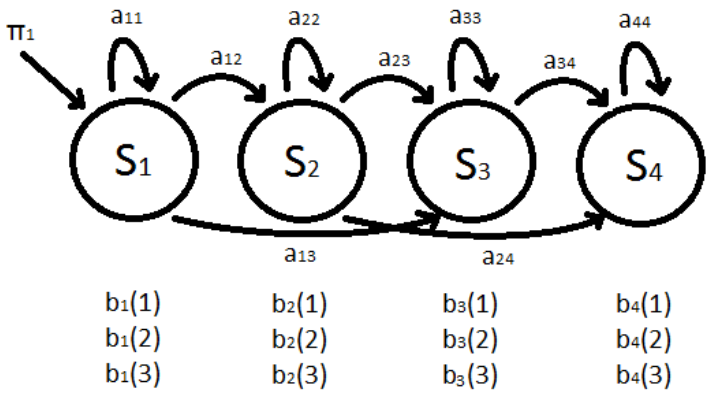

Figura A.4: Cadena de HMM con 4 estados y 3 observaciones posibles por estado.

Esto reduce el número de componentes no nulas de las matrices A y Π. Con estas restricciones la matriz A queda como una matriz triangular superior en la que cada fila tiene tres elementos no nulos, seguidos desde la diagonal hacia su derecha, excepto las dos últimas filas que tienen dos y uno respectivamente. La matriz Π tiene el primer elemento de valor uno y el resto ceros.

Una vez explicado el modelo, los problemas que son necesarios abordar para el reconocimiento de voz son dos; el problema de evaluación y el de optimización.

III.1. Problema de evaluación

Dado un determinado modelo $\lambda=(A, B, \pi)$ y un conjunto de observaciones $\mathbf{O}=O_1, O_2...O_T$ hasta un instante T, ¿cuál es la probabilidad de que esas observaciones hayan sido generadas por dicho modelo? Es decir, $P(\mathbf{O}|\lambda)$ [2].

La solución a este problema permite comparar entre varios modelos y decidir cuál de ellos se aproxima mas al modelo generador del conjunto de observaciones $\mathbf{O}$.

La forma directa de resolver el problema es enumerar cada posible secuencia de estados alcanzables hasta el instante T, $\mathbf{Q}=q_1, q_2,...q_T$ y, suponiendo independencia estadística entre observaciones, aplicar la fórmula:

$$P(\boldsymbol{O}|\lambda) = \sum_{todo\ \boldsymbol{Q}} P(\boldsymbol{O}|\boldsymbol{Q},\lambda) \cdot P(\boldsymbol{Q}|\lambda)$$

Resolver la ecuación para un modelo de N estados conlleva un total de $2\cdot T\cdot N^T$ cálculos; concretamente $2\cdot(T-1)\cdot N^T$ multiplicaciones y N^T-1 sumas. Para un número alto de observaciones el cálculo resulta prácticamente imposible.

Para resolver el problema de forma eficiente se utiliza el algoritmo Forward, con el cual se soluciona mediante un número del orden de $N^2 \cdot T$ cálculos.

III.2. Problema de optimización

Dado un conjunto de observaciones $\mathbf{O}=O_1, O_2 \ldots O_T$, ¿cómo obtengo un modelo $\lambda=(A, B, \pi)$, y como ajusto sus parámetros para que $P(\mathbf{O}|\lambda)$ sea máxima [2]?

La solución de este problema permite encontrar, dada una serie de observaciones procedentes de una determinada fuente, el modelo óptimo que mejor describe el comportamiento de esa fuente. Este es el llamado "entrenamiento" de un HMM.

El entrenamiento es el problema más difícil que presentan los HMM y no existe una manera óptima de resolverlo analíticamente. El método que utilizaré es el algoritmo de Baum-Welch, que obtiene un máximo local como solución a partir de un modelo inicial y mediante sucesivas iteraciones del algoritmo.