Tecnológico de Costa Rica

Escuela de Ingeniería Electrónica

Implementación de un sistema de reconocimiento de patrones acústicos de disparos y motosierras en un sistema embebido.

Informe de Proyecto de Graduación para optar por el título de

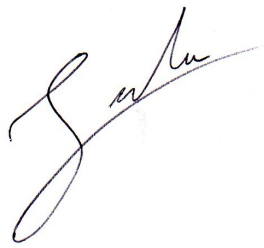
Ingeniero en Electrónica con el grado académico de Licenciatura

Jordan Montero Aragón

Cartago, 3 de abril de 2013

Declaro que el presente Proyecto de Graduación ha sido realizado enteramente por mi persona, utilizando y aplicando literatura referente al tema e introduciendo conocimientos propios.

En los casos en que he utilizado bibliografía he procedido a indicar las fuentes mediante las respectivas citas bibliográficas. En consecuencia, asumo la responsabilidad total por el trabajo de graduación realizado y por el contenido del correspondiente informe final.



Jordan Montero Aragón

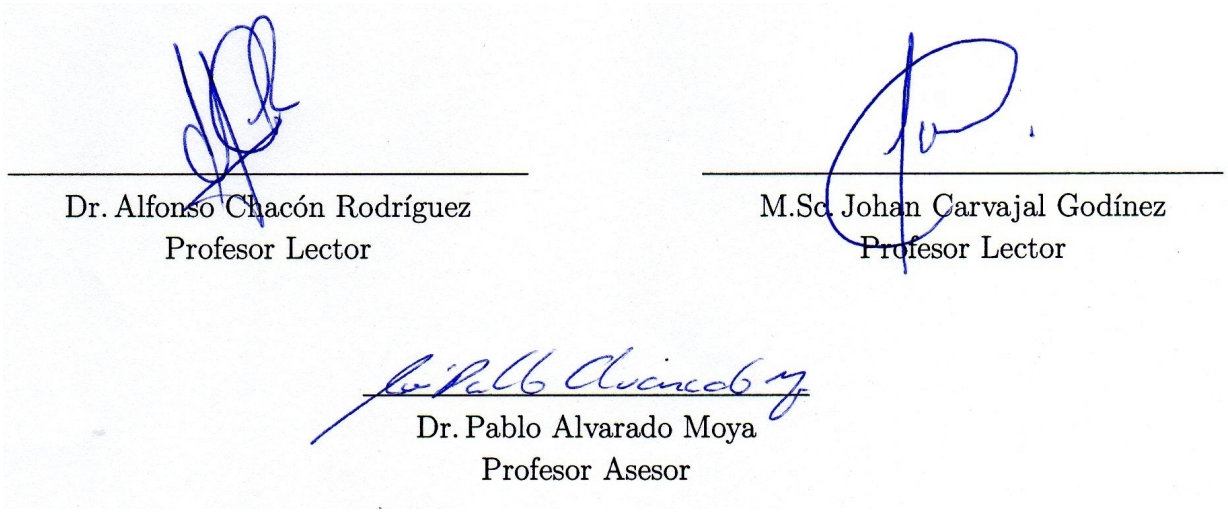
Cartago, 3 de abril de 2013

Céd: 1-1435-0390

Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Electrónica
Proyecto de Graduación
Tribunal Evaluador

Proyecto de Graduación defendido ante el presente Tribunal Evaluador como requisito para optar por el título de Ingeniero en Electrónica con el grado académico de Licenciatura, del Instituto Tecnológico de Costa Rica.

Miembros del Tribunal



Dr. Alfonso Chacón Rodríguez
Profesor Lector

M.Sc. Johan Carvajal Godínez
Profesor Lector

Dr. Pablo Alvarado Moya
Profesor Asesor

Los miembros de este Tribunal dan fe de que el presente trabajo de graduación ha sido aprobado y cumple con las normas establecidas por la Escuela de Ingeniería Electrónica.

Cartago, 3 de abril de 2013

Resumen

Actividades humanas como la tala de árboles y la caza ilegal de especies en peligro de extinción han provocado daños en los ecosistemas naturales, por lo que se han propuesto soluciones de bajo costo y consumo energético para su detección, mediante el uso de redes inalámbricas de sensores. Estas se conforman por nodos descentralizados que integran los elementos necesarios para llevar a cabo la extracción de patrones acústicos del medio, realizar reconocimiento sobre ellos y determinar así el estado actual del entorno, procediendo a comunicar a las autoridades pertinentes sobre cualquier eventualidad.

El presente proyecto se enfoca en la sección de extracción de los patrones acústicos del medio y entrenamiento de la cadena de procesamiento, en base a los valores capturados a partir de ejemplos de sonidos similares a los que se esperaría detectar, mediante algoritmos de análisis de discriminantes lineales, análisis de conglomerados de k-medias y diferentes variantes de entrenamiento para modelos ocultos de Markov. Todos estos implementados en una aplicación con interfaz gráfica desarrollada en C/C++.

Se desarrolla luego la implementación de la cadena de procesamiento en un sistema embebido que ejecuta un sistema operativo GNU/Linux, usando el lenguaje de programación C. Esta aplicación abarca desde la extracción de características hasta la estimación probabilística del estado del medio, usando para ello los resultados generados a partir del entrenamiento del sistema. Este prototipo en software permite evaluar de manera rápida cambios en los valores de las constantes usadas en cada sección del sistema, y constituye un punto de comparación para la implementación en FPGA y/o ASIC.

Palabras clave: C/C++, HMM, Linux, Reconocimiento de patrones, Sistemas Embebidos, Software, Programación, Qt, Procesamiento Digital de Señales.

Abstract

Human activities such as illegal hunting and deforestation have led to damage to natural ecosystems, evidencing the need for solutions that have to be cost affordable and energy efficient. One of the proposals uses wireless sensor networks, which are composed of descentralized nodes that incorporate all the necessary elements required for the tasks of acoustic pattern extraction and recognition, and for the transmission of alerts between nodes and to a monitoring PC.

This project focuses on the acoustic pattern extraction and recognition section, and on the training of the different elements of the processing chain by using sound samples of various weapons and chainsaws usually involved in the mentioned activities. Algorithms like linear discriminant analysis, k-means clustering and various hidden Markov models training methods are used to determine the values that need to be used at every each stage. All this training is done with the aid of an application developed in C/C++, which incorporates a graphical user interface.

An implementation of the main processing chain on an embedded system, using a GNU/Linux operating system and developed in the C programming language, is presented. This application does everything from the extraction of the acoustic patterns to the probabilistic estimation of the current state of the surrounding medium, using the results from the previous training. This software prototype makes it easier to study the effects of variations of these values on the recognition rate while dealing with field testing, and also gives a point of comparision for the FPGA/ASIC system implementation.

Keywords: C/C++, HMM, Digital Signal Processing, Embedded Systems, Linux, Pattern Recognition, Programming, Qt, Software.

a mis queridos padres, abuelos y hermanos

Agradecimientos

Este trabajo representa la culminación de varios años de esfuerzo y trabajo continuo, que me han permitido llegar hasta este punto que en algún momento se veía como muy lejano y difícil de alcanzar.

El agradecimiento va dirigido primero hacia mis padres, quienes han estado siempre ahí, dispuestos a ayudarme en todo lo que les ha sido posible. Debo agradecer también a mi abuela, y en especial a mi abuelo, que en paz descanse; definitivamente me hubiera gustado que estuviera acá para ver cómo alcanzaba esta meta, y todo lo que está por venir.

Agradecer también a mis hermanos, primos, tíos y al resto de mi familia, así como a amigos y a compañeros del TEC, muchos de los cuales finalizamos esta etapa juntos este mismo semestre. Todos me han ayudado a llegar a acá de una u otra forma, por lo que no puedo dejar de mencionarlos.

Los agradecimientos se extienden también a todo el personal educativo del TEC, pues son quienes me han dado la preparación académica requerida para llegar aquí. Agradecezco especialmente a todos los profesores de la Escuela de Ingeniería Electrónica, pues son quienes me enseñaron y permitieron comprender sobre esta área tan interesante, extensa y útil de la ingeniería. Recién cuando inicié la carrera no conocía mucho de ella, y para este momento sólo puedo concluir que me ha gustado mucho todo lo que he aprendido.

Finalizo agradeciendo especialmente al profesor Pablo Alvarado, pues su guía fue muy importante a través del desarrollo de este trabajo, y también al profesor Alfonso Chacón, pues ambos me han ayudado mucho a mí, y a otros compañeros, tanto a nivel de enseñanza como en la realización de proyectos, como referencias profesionales y ofreciéndonos invaluable ayuda y oportunidades.

Muchas gracias a todos por ser parte de la realización de esta meta.

Jordan Montero Aragón

Cartago, 3 de abril de 2013

Índice general

Índice de figuras

Índice de tablas

Lista de símbolos y abreviaciones

Abreviaciones

ASIC	Circuito integrado de aplicación específica
DSP	Procesador Digital de Señales
FFT	Transformada rápida de Fourier
HMM	Modelos Ocultos de Markov
HPF	Filtro pasa altas
LDA	Análisis de discriminantes lineales
LPF	Filtro pasa bajas
PCA	Análisis de componentes principales
RIS	Red Inalámbrica de Sensores
SiRPA	Sistema de Reconocimiento de Patrones Acústicos
SoC	Sistema en chip
VHDL	VHSIC (Very High Speed Integrated) Hardware Description Language
VLSI	Very Large Scale Integration

Notación general

$\mathbf{A}$	Matriz. $\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix}$
$\mathbb{R}$	Conjunto de los números reales.
$\underline{\mathbf{x}}$	Vector. $\underline{\mathbf{x}} = [x_1 \; x_2 \; \ldots \; x_n]^T = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$
$\mathbb{Z}$	Conjunto de los números enteros.

Capítulo 1

Introducción

Los ecosistemas naturales proveen servicios ambientales y recursos naturales indispensables para el desarrollo de la vida humana. Sin embargo, estos han sido víctimas de los procesos de desarrollo económico y social, así como de la explotación de los recursos que ellos proveen. Países como Costa Rica, entre otros, conscientes del daño provocado, han establecido reglamentaciones y regulaciones con el fin de protegerlos, permitiendo capturar y penalizar a las personas que violenten la legislación.

Si bien estos mecanismos legales han sido aprobados y ratificados en los respectivos países, su aplicación efectiva dista de ser ideal, ya que la falta de recursos tanto económicos como humanos dificulta detectar a los infractores, y por lo tanto aplicar la jurisdicción. Tal es el caso de Costa Rica, donde se cuenta con más de 160 zonas protegidas [40, 39], las cuales albergan miles de especies de plantas y animales. Así, por ejemplo, el parque nacional Braulio Carrillo cuenta con más de 47500 hectáreas de terreno protegido, con menos de 25 guardaparques o personal disponible para su vigilancia. Esto implica que cada miembro tendría a su cargo la vigilancia de más de 1900 hectáreas.

Como parte de las alternativas propuestas para ayudar en la detección de actividades ilegales, tales como la caza de especies en peligro y la tala de árboles, se tiene el uso de redes inalámbricas de sensores (RIS). Éste tipo de redes se componen por nodos que integran todos los componentes requeridos para realizar la captura de señales del medio, procesamiento sobre ellas y la comunicación hacia el resto de la red y a sus operarios en caso de presentarse una situación de alarma. Sin embargo, el diseño de estos dispositivos se encuentra sujeto a requisitos tales como bajo consumo energético, suficiente poder de procesamiento para ejecutar de manera correcta los algoritmos de reconocimiento utilizados y un radio de detección y comunicación tal que se pueda utilizar un menor número de terminales y se pueda reducir así el costo de implementación y mantenimiento de la red.

El presente trabajo forma parte de una propuesta que utiliza una RIS cuyos nodos realizan la captura de señales acústicas del medio y realizan reconocimiento de patrones sobre ellas, con el fin de determinar el estado actual del entorno. En caso de que se determine que el medio, en este caso un bosque, se encuentre en un estado de alarma, que podría

ser representado por la presencia de disparos o de una motosierra, se procedería con la activación de la sección de transmisión y el envío de la notificación a un sumidero desde donde se monitoriza la actividad de la red. En la figura 1.1 se muestra un diagrama general que explica este procedimiento.

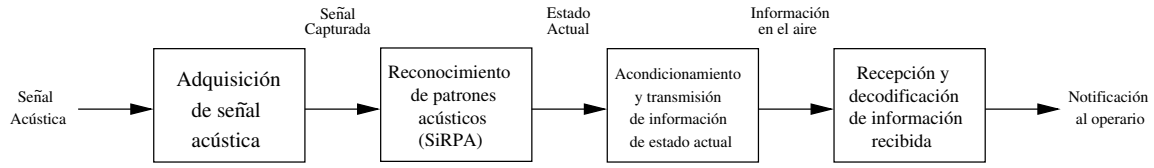


Figura 1.1: Cadena de procesamiento realizada por cada nodo de la red.

Específicamente, este trabajo se concentra en la implementación del bloque denominado SiRPA (Sistema de Reconocimiento de Patrones Acústicos), el cual utiliza técnicas de procesamiento digital de señales, una herramienta estadística conocida como modelos ocultos de Markov (HMM) y estrategias de entrenamiento para cada sección del sistema como el análisis de discriminantes lineales (LDA), análisis de conglomerados de k-medias y diferentes variantes de entrenamiento para los HMM.

La totalidad de este bloque se encuentra desarrollada en software, y se puede dividir en dos secciones: entrenamiento y ejecución. El diagrama general para el primer caso se presenta en la figura 1.2.

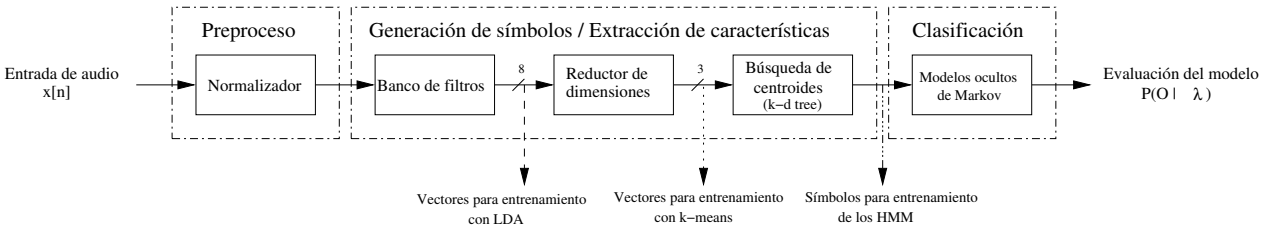


Figura 1.2: Cadena de procesamiento del SiRPA para entrenamiento.

Este sistema opera de la siguiente manera: se obtiene primero la señal acústica de entrada, que luego debe ser discretizada. Esta señal se normaliza para llevarla a un nivel de referencia común y prepararla así para la *extracción de características* y su final *clasificación.*

La extracción de características se realiza separando la señal normalizada en ocho diferentes bandas de frecuencia, a partir de las que se estima la energía en cada banda. El vector de ocho dimensiones resultante es luego proyectado hacia un espacio de tres dimensiones, para simplificar el procesamiento en etapas posteriores. Este vector en tres dimensiones es luego comparado contra los centroides que se encuentran almacenados en el árbol k-d, con el fin de determinar cuál de ellos (de un total de 32) es el más cercano al punto descrito por el vector. La salida del árbol es entonces el identificador, o *id*, del símbolo de observación representado por este centroide.

Los símbolos de observación resultantes se pasan finalmente al módulo de clasificación, donde una vez que se ha capturado el número requerido para formar una cadena de

observaciones se utilizan los denominados Modelos Ocultos de Markov (HMM por sus siglas en inglés) para asignarla a una de las tres clases predefinidas, que pueden ser *bosque*, *disparo* o *motosierra*. En la implementación de entrenamiento se utiliza esta clasificación para obtener las denominadas *matrices de confusión*, a partir de las cuales se pueden estudiar las tasas de reconocimiento de los modelos utilizados y verificar así la efectividad del entrenamiento realizado.

Dentro del SiRPA existen diferentes bloques que deben ser entrenados previo a su utilización para reconocimiento en línea. Esto se hace por medio de archivos de audio de ejemplo para las tres clases mencionadas, y que fueron capturados para su uso en trabajos previos. Específicamente, deben entrenarse los bloques del reductor de dimensiones, árbol k-d y los HMM dentro del módulo de clasificación, para lo que se toman muestras en los puntos mostrados en la figura 1.2. Los resultados del entrenamiento pueden ser posteriormente exportados para usarlos en la implementación de ejecución.

La sección de entrenamiento se encuentra desarrollada en C/C++ e integra una interfaz gráfica para facilitar la extracción de patrones acústicos desde archivos de audio, almacenamiento y administración de las muestras utilizadas por cada algoritmo de entrenamiento, y evaluación de los resultados obtenidos para cada uno de los bloques que integran el sistema.

Por otro lado, la sección de ejecución es implementada en un sistema embebido Beagleboard-xM, y se encuentra desarrollada en el lenguaje de programación C. Esta ha sido desarrollada para contar con un prototipo de software que permita evaluar cambios de manera flexible, realizar pruebas de campo y para contar con un punto de comparación con la implementación de hardware, en cuanto a funcionamiento y rendimiento. La cadena de procesamiento realizada por el sistema embebido se muestra en la figura 1.3.

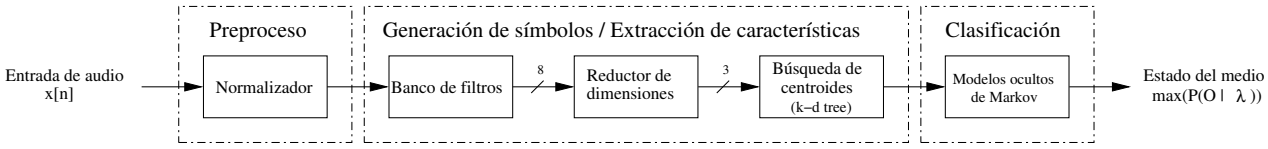


Figura 1.3: Cadena de procesamiento del SiRPA para ejecución.

El funcionamiento para el sistema es el mismo que para la implementación de entrenamiento, con la excepción de que la salida del módulo de clasificación da una estimación probabilística del estado del medio de acuerdo a los resultados obtenidos con los HMM. En base a esto se tomarían acciones específicas en etapas posteriores, pues podría requerirse notificar a través de la red sobre la presencia de eventos sospechosos.

El sistema embebido utilizado incluye dos procesadores diferentes dentro de su sistema en chip (SoC), siendo uno un procesador de propósito general con arquitectura ARM y el otro un procesador digital de señales (DSP). Con el fin de aprovechar este último y reducir la carga de trabajo en el procesador ARM, se modifica el banco de filtros del SiRPA para que sea ejecutado en el DSP, que trabaja con aritmética de punto fijo. Los algoritmos involucrados deben ser optimizados para este tipo de aritmética, pues si se dejan tal

a como se encontraban desarrollados originalmente (con aritmética de punto flotante) podrían no funcionar adecuadamente o presentar tiempos de ejecución excesivos.

El código modificado para ser ejecutado en el DSP se desarrolla también en lenguaje C, y se compila con la herramienta C6RunLib de Texas Instruments. Esta se encarga de generar bibliotecas con las funciones que se desean trasladar a este procesador, e introduce secciones de código en la aplicación del procesador ARM, con el fin de que al llamar a cualquiera de estas funciones se traslade la ejecución al DSP.

La aplicación se optimiza en cuanto a tiempos de ejecución utilizando los *intrinsic* que provee C6RunLib, que permiten llamar instrucciones del lenguaje ensamblador del DSP desde el código en C. Estas instrucciones se ejecutan usualmente en un solo ciclo de reloj, por lo que deben usarse siempre que se pueda para reducir estos tiempos. Además, se estructura y analiza el código para aprovechar las diferentes unidades de ejecución del procesador, permitiendo paralelizar operaciones y reducir aún más los tiempos de cálculo.

1.1 Trabajos realizados previamente

Existen varios proyectos realizados previamente que han explorado o investigado diferentes aspectos de implementación o mejora para uno o más de los bloques que componen el SiRPA, tanto a nivel de hardware como software. Entre ellos se tienen los siguientes:

- En [44] se realizó una primera implementación del SiRPA en software, utilizando MATLAB como plataforma de desarrollo y ejecución, y realizando el análisis de la señal acústica, en la etapa de extracción de características, mediante teoría de onditas (*wavelets* en inglés).
- En [41] se realizó otra implementación del SiRPA en software, utilizando también MATLAB pero descartando el uso de onditas con el fin de simplificar la implementación en hardware. Esto se debe a que ciertos tipos de onditas son más complejas que filtros digitales, y aquellas que son más sencillas tienen respuestas de frecuencia que podrían ser inadecuadas para la aplicación.
- En [34] se realizó la primera implementación parcial del SiRPA en hardware, utilizando una FPGA y el lenguaje de descripción de hardware VHDL, y basándose en la implementación del punto anterior. En este trabajo se realizaron cambios en el sistema con el fin de adaptarlo a los recursos disponibles, y para acercarlo al comportamiento deseado en cuanto a rendimiento y consumo energético. Esta implementación es la base para el actual desarrollo en hardware del sistema.
- En [36] se investigó y desarrolló una manera de reducir el número de número de cálculos requeridos para obtener la salida final de la sección de extracción de características del SiRPA, lo que disminuye tiempos de ejecución y consumo energético. El resultado fue un módulo de reducción de dimensiones implementado en hardware (VHDL) y en software (C++), donde se determinó a su vez que para obtener una menor pérdida de información durante el proceso, era mejor el entrenamiento con

LDA que con PCA.

- En [7] se desarrolló una aplicación con interfaz gráfica, usando C/C++, cuya finalidad es el entrenamiento de los HMM utilizados en la sección de clasificación del SiRPA. Para realizar esto se desarrollaron algoritmos de entrenamiento simple, múltiple y negativo. Esta aplicación permite además realizar la extracción de patrones acústicos desde archivos de audio y realizar la evaluación de la salida de los HMM para verificar la efectividad del reconocimiento de sonidos. El trabajo realizado en el presente documento toma como base esta implementación.
- Existen además varios trabajos cuya finalidad ha sido el desarrollo de la implementación en ASIC del SiRPA, usando tecnología VLSI. En [1] se obtuvo el trazado físico, utilizando herramientas que lo generan automáticamente, para los módulos de reducción de dimensiones, árbol k-d y el de clasificación (para el que se diseñó un procesador de nombre MAP). En [49] se hizo lo mismo pero para el banco de filtros. Además, en [50] se llevó a cabo el trazado físico para el bloque normalizador. Existen trabajos adicionales que por cuestiones de espacio no se incluyen.

1.2 Objetivos y estructura del documento

Este trabajo cuenta con dos objetivos principales; el primero consiste en desarrollar una aplicación de software que integre los elementos necesarios para realizar el entrenamiento de los diferentes bloques que componen el SiRPA, mientras que el segundo consiste en desarrollar un prototipo de software que se ejecute en un sistema embebido, utilizando los resultados del entrenamiento, y realizando procesamiento en línea.

Además, se busca modificar parte de la cadena de procesamiento del SiRPA en el sistema embebido, con el fin de que sea ejecutada por su procesador digital de señales y se reduzca así el tiempo de procesamiento por bloque y la carga de trabajo en el procesador de propósito general.

La estructura del documento es la siguiente: en el capítulo 2 se presenta la teoría y fundamentos matemáticos en que se basa el desarrollo del proyecto, junto a una explicación de los diferentes algoritmos utilizados tanto para realizar el entrenamiento de cada bloque como para realizar la extracción de características y evaluación del estado del medio.

En el capítulo 3 se presenta de manera detallada la solución tanto para la sección de entrenamiento como la de ejecución. Se estudian las diferencias entre implementaciones y las características propias de cada una.

En el capítulo 4 se presentan los resultados obtenidos a partir del entrenamiento, las tasas de reconocimiento obtenidas para los modelos utilizados, y los resultados de evaluación y tiempos de ejecución para la implementación en el sistema embebido. Se finaliza presentado las conclusiones y recomendaciones pertinentes en el capítulo 5.

Capítulo 2

Marco teórico

2.1 Reconocimiento de patrones

El área de reconocimiento de patrones busca asignar un objeto, también referido como *patrón*, a una categoría o *clase* específica dentro de un determinado número de posibilidades [45]. Estos patrones pueden ser cualquier tipo de señal que necesite ser clasificada, tal como las señales acústicas capturadas en el bosque. En base al resultado de esta clasificación, podría requerirse tomar acciones específicas.

Los patrones, denotados como $\underline{\mathbf{x}}$, se componen de elementos denominados *características*, que corresponden a cada uno de los elementos del vector. Así, al patrón se le puede llamar también vector de características. De esta forma, el problema matemático principal del reconocimiento de patrones sería asignar $\underline{\mathbf{x}}$ a una de M posibles clases. Para esto se utiliza la probabilidad de que el vector pertenezca a cada una de las clases, que se expresa como

$$P(\omega_i|\underline{\mathbf{x}}) \quad , i = 1, 2, .., M \tag{2.1}$$

Para realizar esta clasificación se requiere primero aprender sobre las diferentes clases, para lo que usualmente se utilizan patrones de ejemplo similares a los que se busca reconocer. Esto se conoce como *aprendizaje supervisado*. En el caso de este trabajo, las señales acústicas de ejemplo corresponden a sonidos normales en el bosque, así como sonidos de disparos y motosierras.

El área de reconocimiento de patrones acústicos ha sido de utilidad en aplicaciones de reconocimiento del habla [33], análisis de fallas en las micro estructuras de guías de onda [30] y reconocimiento de disparos para ayudar a la policía a ubicar infractores de la ley [8]. Aplicaciones en otras áreas incluyen reconocimiento facial, análisis de huellas digitales, identificación de placas de automóviles, entre otras.

2.2 Estimación de energía de señales

En el bloque del normalizador y a la salida del bloque del banco de filtros en las figuras 1.2 y 1.3 se debe estimar la energía de la señal, que para una señal de variable discreta es [28]

$$E_x = \sum_{n=-\infty}^{\infty} |x(n)|^2 \tag{2.2}$$

o en caso de limitarla a un rango finito de tiempo

$$E_N = \sum_{n=-N}^{N} |x(n)|^2 \tag{2.3}$$

La energía puede ser también obtenida en el dominio de la frecuencia, utilizando la relación de Parseval, como

$$E_x = \frac{1}{2\pi} \int_{-\pi}^{\pi} |X(\omega)|^2 d\omega \tag{2.4}$$

Si bien (2.2) y (2.4) permiten obtener la energía de la señal de manera exacta, presentan el problema de que se requiere la totalidad de la señal disponible para realizar el cálculo, lo que en un sistema que opera en línea no es posible. Además, aún cuando (2.3) permite obtener la energía de la señal en un intervalo que incluya sólo muestras previas, resulta computacionalmente pesada.

Dentro de la cadena de procesamiento del SiRPA existen diversos bloques que requieren obtener un estimado de la energía de una señal. El bloque normalizador lo requiere para obtener un promedio de su señal de entrada, mientras que el banco de filtros lo requiere para estimar la energía por banda. Es por esto que se han buscado estrategias de estimación de energía que sean más sencillas de implementar, y presenten menores requisitos de tiempo de cálculo, aunque ello signifique una pérdida de precisión. Una de las opciones corresponde al uso de un seguidor de envolvente, también conocido como demodulador AM [43], que se muestra en la figura 2.1.

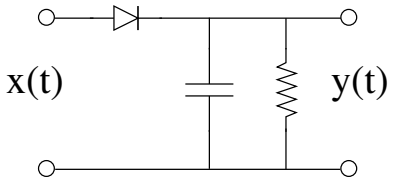


Figura 2.1: Circuito seguidor de envolvente.

Este circuito puede ser visto como un rectificador seguido por un filtro paso bajo, representado por el circuito tanque de donde se toma la salida. Este filtro presenta un

comportamiento de media móvil, lo que permite obtener un estimado de la energía de la señal en cada momento [32]. El diodo realiza en este caso una rectificación de media onda, pero se puede modificar el esquema para realizar una rectificación de onda completa al cambiar este elemento por un puente rectificador.

Suponiendo que la rectificación es de onda completa, el funcionamiento del circuito sería el siguiente: inicialmente no se tiene entrada aplicada y el condensador se encuentra descargado. Una vez que se aplica una entrada, ésta es rectificada y comienza a cargar el condensador hasta llegar a su valor, con el fin de seguirla. Cuando el valor de entrada pasa a ser más bajo que el que se encuentra almacenado en el condensador, la salida del circuito no disminuye inmediatamente, sino que comienza a descargarse de manera exponencial a una tasa dada por la constante RC del circuito tanque, hasta que nuevamente se tenga una entrada de mayor valor que el almacenado. De esta forma se consigue un valor promedio de la entrada y sus valores previos. La figura 2.2 muestra este proceso de manera gráfica, donde la salida es $v_o(t)$, y se muestra el efecto de la constante RC.

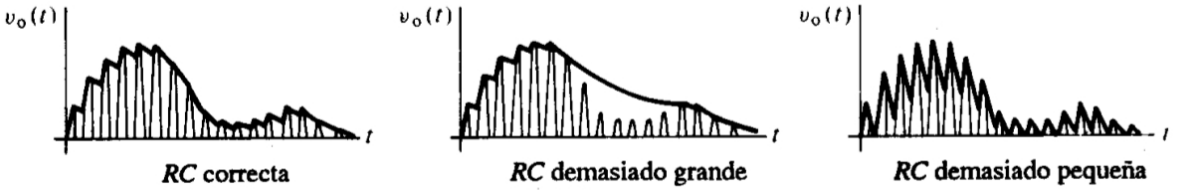


Figura 2.2: Proceso de demodulación con detector de envolvente. Tomada de [43].

La salida del circuito, una vez discretizadas las ecuaciones que la describen, puede ser expresada como

$$y[n] = \begin{cases} |x[n]| & \text{si } |x[n]| \geq y[n-1] \\ \alpha \cdot y[n-1] & \text{si } y[n-1] > |x[n]| \end{cases} \tag{2.5}$$

donde el parámetro α se relaciona con la constante RC como

$$\alpha = \exp\left(-\frac{T}{RC}\right) \tag{2.6}$$

2.3 Normalización de señales

El proceso de normalización consiste en llevar una señal pasada como entrada a un nivel de referencia común, con el fin de prepararla para su posterior procesamiento y análisis probabilístico [37, 4]. Esto se hace dado que un sistema podría ser sometido a señales que presentan variación, por lo que se deben normalizar para reducir los efectos de estas variaciones sobre los resultados de salida, así como para capturar eventos o cambios súbitos en la señal de entrada.

En [20] se presenta un ejemplo de la importancia de normalizar las señales capturadas a partir de electroencefalogramas de diferentes pacientes, como preparación previa a su extracción de características. El objetivo de este proceso es utilizar algoritmos que sean independientes de cada paciente, dado que entre cada uno de ellos se obtendrán señales cuyos valores presentan variaciones, haciendo necesario que sean llevadas a un nivel de referencia común.

Existen diferentes estrategias para realizar el proceso de normalización, las cuales varían en cuanto a complejidad y efectividad de la normalización resultante. Uno de los métodos comúnmente utilizados es el control automático de ganancia (AGC), el cual hace uso de un amplificador de ganancia variable que ajusta su ganancia de acuerdo a una señal de realimentación; de esta forma se reducen las variaciones en la señal de entrada. La figura 2.3 muestra el diagrama de bloques para un ejemplo de AGC.

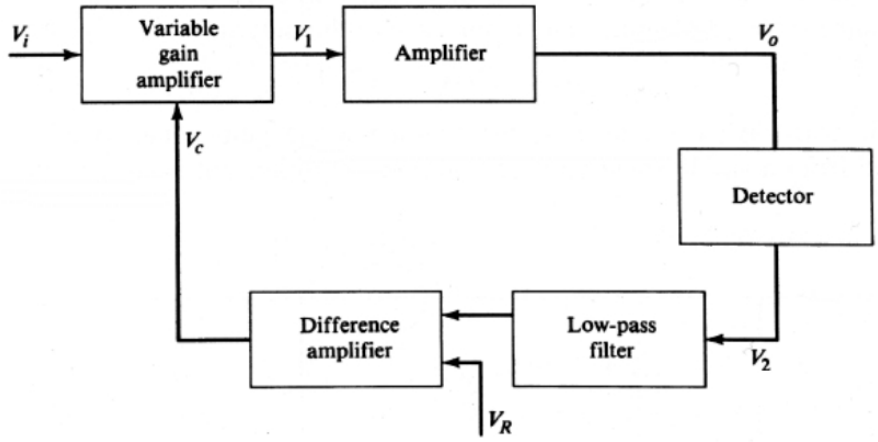


Figura 2.3: Diagrama de bloques para AGC con lazo de realimentación. Tomada de [42].

La señal de realimentación utilizada en este caso es tomada de la salida, la cual es filtrada para obtener un valor promedio que es luego comparado contra un referencia predefinida, dando así la señal de control para el amplificador de ganancia variable. Este esquema ha sido utilizado para normalizar señales de entrada en sistemas de comunicación [47], permitiendo que señales débiles obtenidas por el dispositivo de recepción no queden enmascaradas por el ruido, al amplificarlas hasta un nivel adecuado, a la vez que se disminuye la ganancia de señales fuertes que podrían provocar la saturación de los amplificadores.

Este esquema puede modificarse para utilizar un promedio de la señal de entrada para controlar la ganancia aplicada a la salida, lo que evita los problemas de estabilidad que pueden surgir a partir del lazo de realimentación negativa. Tiene además un tiempo de asentamiento de la salida más rápido, haciéndole más eficiente que el otro esquema, a costa de un mayor consumo energético al implementarlo en hardware [2]. Este tipo de implementación es llamada en inglés *feedforward*, y su diagrama de bloques se muestra en la figura 2.4.

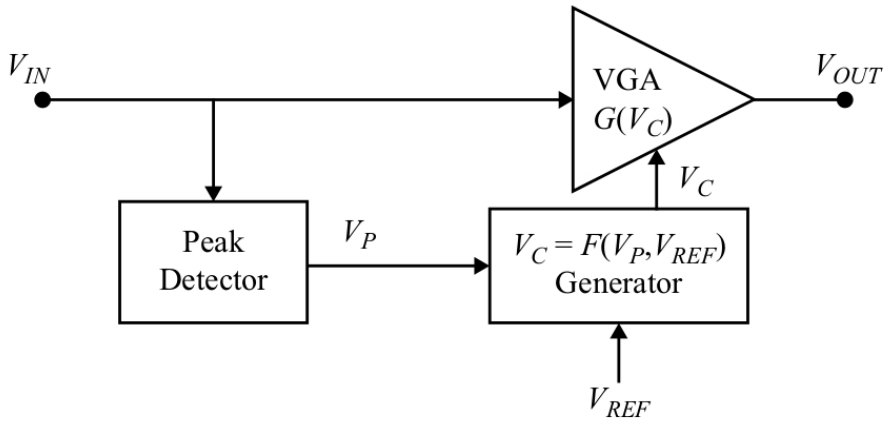


Figura 2.4: Diagrama de bloques para AGC con lazo de feedforward. Tomada de [2].

2.4 Filtros digitales y bancos de filtros

En el sistema propuesto (figuras 1.2 y 1.3) se utiliza un banco de filtros digitales para separar la señal capturada en bandas espectrales.

2.4.1 Filtros

Un filtro es un tipo de sistema que discrimina o extrae algún atributo de una señal aplicada a su entrada [28]. En procesamiento de señales, filtro es un término utilizado usualmente para indicar un sistema lineal e invariante en el tiempo (LTI) capaz de filtrar diferentes componentes de frecuencia de la señal aplicada a su entrada, de acuerdo a su respuesta de frecuencia $H(\omega)$. Un *filtro digital* es un sistema selectivo de frecuencia que opera sobre datos muestreados, o en tiempo discreto. En la figura 2.5 se muestra la transformación que realiza este sistema sobre su señal de entrada.

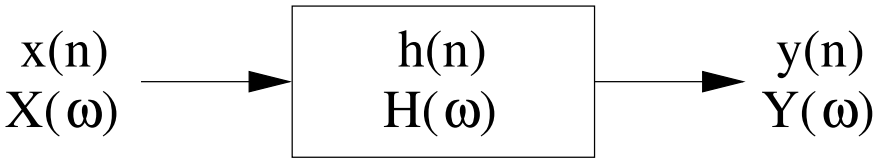


Figura 2.5: Transformación realizada por un filtro digital.

Los filtros digitales pueden ser separados en dos categorías [28], los de respuesta finita al impulso (FIR) y los de respuesta infinita al impulso (IIR). Además, puede distinguirse entre sistemas causales y anticausales [3], donde acá sólo se considerarán los primeros al desarrollarse un sistema que opera en línea.

Los filtros FIR tienen la característica de que su respuesta al impulso $h(n)$ es representada por un número finito de M valores. Estos son implementables directamente por medio de la suma de convolución, dada por

$$y(n) = \sum_{k=0}^{M-1} h(k)x(n-k), \quad h(k) = 0 \text{ para } k < 0 \wedge k \geq M \tag{2.7}$$

Por otro lado, los filtros IIR tienen un $h(n)$ con extensión infinita, por lo que (2.7) no puede ser utilizada para calcular la salida al requerir un número infinito de operaciones. En su lugar, se implementan por medio de las denominadas *ecuaciones de diferencias*, que son el equivalente discreto de las ecuaciones diferenciales; estas describen los sistemas IIR de manera recursiva como

$$y(n) = -\sum_{k=1}^{N} a_k y(n-k) + \sum_{k=0}^{M} b_k x(n-k) \tag{2.8}$$

donde N recibe el nombre de orden del sistema e indica el número de polos, mientras que M indica el número de ceros. Los coeficientes a_k y b_k provienen de la denominada función de transferencia del sistema $H(z)$, y de ellos depende la ubicación de cada uno de los polos y los ceros que describen al sistema. Es posible obtener $H(z)$ al aplicar la herramienta matemática conocida como transformada z a la ecuación de diferencias del sistema analizado.

Los filtros IIR son más simples de implementar que los FIR. El único escenario donde estos últimos suelen ser utilizados es cuando requisitos de la aplicación exigen un comportamiento de fase lineal [28], ya que en los IIR no es posible obtenerlo. Sin embargo, en aplicaciones de audio, o en aquellas donde solamente interesa determinar características como la energía de la señal, que viene dada por su amplitud, la fase lineal no es importante, por lo que se prefieren los filtros IIR.

Existen diferentes formas de implementar los filtros IIR de acuerdo a como se manipule (2.8); estas buscan reducir requisitos de memoria, efecto de errores o ruido de cuantización de los coeficientes en la ubicación de los polos y ceros, entre otros factores. La denominada forma directa II [28] permite reducir el número de elementos de memoria requeridos al mínimo, al introducir en (2.8) una variable intermedia $\omega(n)$. El diagrama de bloques que describe esta implementación se muestra en la figura 2.6, donde el bloque z^{-1} representa un retardo de una muestra y deriva de las propiedades de la transformada z.

2.4.2 Bancos de filtros

A la organización de un conjunto de filtros, digitales en este caso, se le denomina banco de filtros. Esta estructura toma una señal de entrada y la separa en diferentes bandas de frecuencia, que pueden o no traslaparse. Así, la salida de cada uno de los elementos que componen el banco representa una banda específica.

Existen diferentes estructuras para construir el banco [35], así como diferentes maneras de segmentar el espectro de frecuencia. Una de las alternativas es la separación lineal o

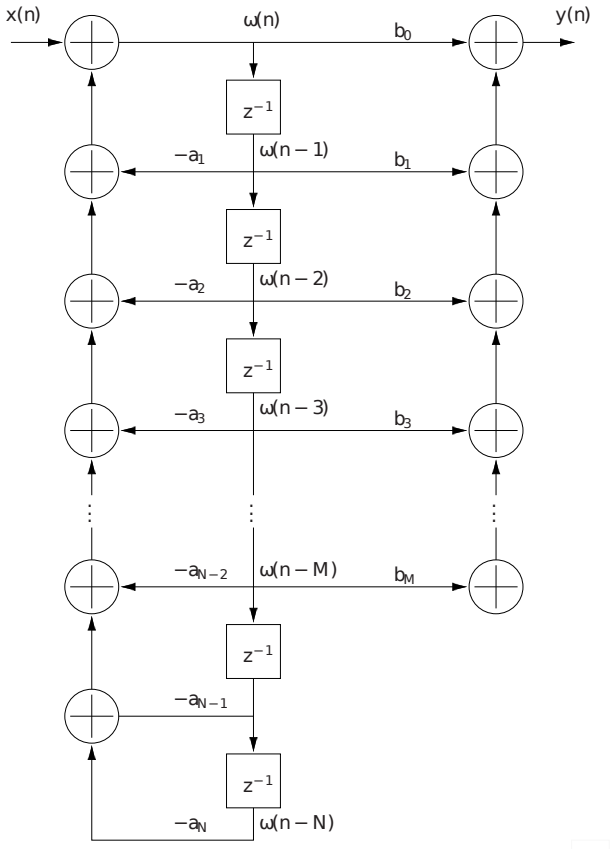


Figura 2.6: Estructura de forma directa II.

uniforme, que lo hace separando el espectro en bandas de frecuencia que comparten el mismo ancho de banda, contando con igual número de muestras a la salida de cada filtro. En la figura 2.7 se muestra un ejemplo de este tipo de separación, donde en esta caso se ha segmentando en M bandas. Cada función de transferencia es diferente, ya que afecta un rango específico de frecuencias.

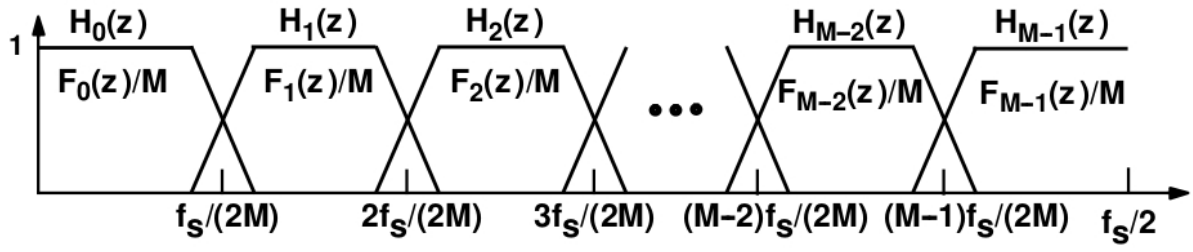


Figura 2.7: Separación uniforme del espectro. Tomada de [35].

Otra estrategia es la separación en octavas, que consiste en realizar una segmentación del espectro de manera tal que cada banda tenga la mitad del ancho de banda de la anterior. Esta separación se muestra en la figura 2.8.

Este tipo de separación [35] se ha usado en aplicaciones de compresión de señales de audio, y se implementa mediante el uso de filtros espejo en cuadratura (QMF). Estos últimos se dimensionan con el fin de que tengan una respuesta que abarca en conjunto todo el

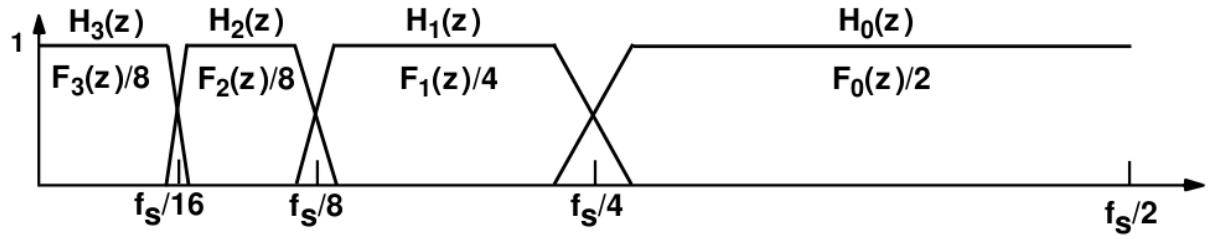


Figura 2.8: Separación del espectro en octavas. Tomada de [35].

espectro de frecuencias, permitiendo así separar el espectro de la señal de entrada en su mitad superior e inferior. El diagrama de bloques para este tipo de banco se muestra en la figura 2.9.

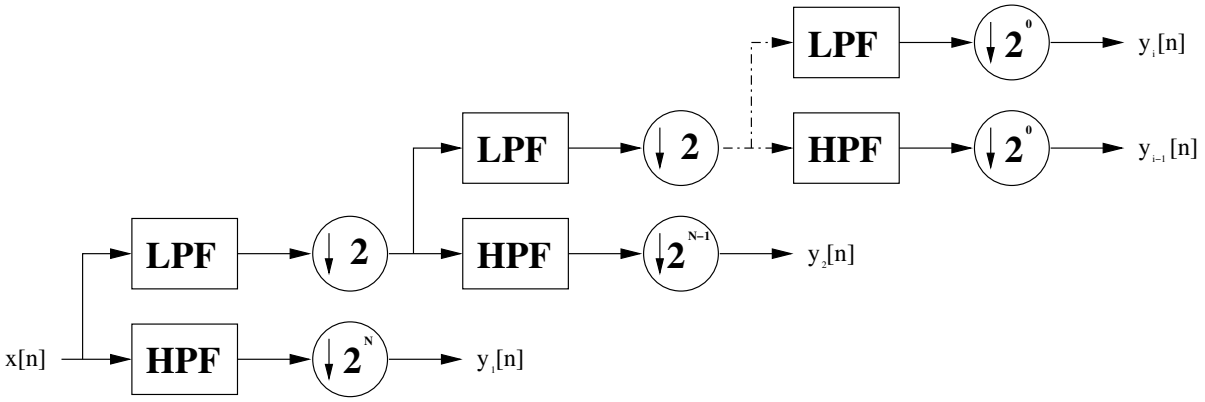


Figura 2.9: Banco de filtros para codificación subbanda.

Este representa un sistema multitasa, dado que no todos los filtros trabajan a la misma frecuencia de muestreo. Esta es dividida en dos por el bloque de diezmado intermedio cada vez que se pasa a la siguiente banda.

Este banco funciona separando el espectro de la señal de entrada en dos; la mitad superior es la generada por el filtro pasa altas (HPF), y corresponde a la octava de salida deseada. La mitad inferior corresponde a la salida del filtro pasa bajas (LPF), y es pasada como entrada al siguiente par de filtros del banco, repitiendo el proceso hasta generar el número de bandas deseadas. Los bloques de diezmado tienen la finalidad de tomar el espectro de la mitad inferior, que abarca idealmente el rango de frecuencias normalizadas [28] de $-\frac{1}{4}$ a $\frac{1}{4}$, y expandirlo al rango de $-\frac{1}{2}$ a $\frac{1}{2}$ para que divisiones subsiguientes den los resultados correctos.

La ventaja de este tipo de banco es que gracias al diezmado y a la división del espectro realizados, todos los pares de filtros LPF y HPF son iguales, por lo que sólo debe implementarse uno de cada uno, lo que es útil en la implementación de hardware. Además, la reducción del número de muestras que se pasan a la salida del banco permite que etapas posteriores trabajen a frecuencias menores, reduciendo así las exigencias de procesamiento. El banco tiene también un tiempo de procesamiento menor, dado que no todos los pares de filtros del banco trabajan continuamente.

2.5 Reducción de dimensiones

2.5.1 Transformación entre espacios dimensionales

El proceso de reducción de dimensiones busca pasar un conjunto de datos descritos por vectores de n dimensiones a un espacio de m dimensiones, por medio de una transformación lineal del vector de entrada $\underline{\mathbf{x}}_i$ al vector de salida $\underline{\mathbf{y}}_i$ [36]. Para esto se realiza el producto matriz-vector expresado como

$$\underline{\mathbf{y}}_i = \mathbf{W}\underline{\mathbf{x}}_i \tag{2.9}$$

donde $\mathbf{W}$ se conoce como matriz de transformación, y tiene tamaño $n \times m$.

Esta transformación asume que los datos tienen media cero, lo que se puede forzar restando la media $\underline{\boldsymbol{\mu}}$ a cada dato como

$$\hat{\underline{\mathbf{x}}}_i = \underline{\mathbf{x}}_i - \underline{\boldsymbol{\mu}}; \quad \underline{\boldsymbol{\mu}} = \frac{1}{N}\sum_{i=1}^{N} \underline{\mathbf{x}}_i \tag{2.10}$$

lo que es necesario realizar con el fin contribuir a la estabilidad numérica del proceso.

El objetivo del procedimiento de reducción de dimensiones es simplificar el procesamiento en etapas posteriores del sistema, dado que según lo predice la *maldición de la dimensionalidad* [36] el conjunto de símbolos necesarios para describir de manera adecuada las observaciones realizadas es 3 órdenes de magnitud mayor al trabajar en un espacio de 8 dimensiones en comparación a uno de 3 dimensiones. Además de los requisitos de memoria adicionales, se requeriría de alrededor de 3 veces más tiempo de procesamiento para realizar búsquedas en el árbol k-d. Todo esto justifica la realización de esta reducción, aún cuando conlleva una pérdida de parte de la información contenida en el espacio de mayor dimensionalidad.

2.5.2 Análisis de discriminantes lineales

El análisis de discriminantes lineales, conocido también como discriminantes lineales de Fisher, busca realizar una proyección del vector $\underline{\mathbf{x}}$ a otro vector $\underline{\mathbf{y}}$ en un espacio de menores dimensiones, de manera tal que se maximice la varianza entre los vectores de diferentes clases, a la vez que se minimiza la varianza entre vectores de una misma clase.

Este análisis identifica la matriz $\mathbf{W}$ que realiza de manera óptima esta transformación, al resolver un problema de eigenvalores y eigenvectores que da por resultado cada uno de los coeficientes que la componen. Los detalles para este análisis pueden ser consultados en [36, 9].

2.6 Organización de datos y árboles k-d

Para encontrar símbolos discretos asociados a la señal acústica, se deben identificar centroides en la etapa de entrenamiento, para lo que se usa un algoritmo de aglomeración (k-medias). Posteriormente, debe identificarse eficientemente cuál de los centroides es el más cercano a la señal de salida del módulo de reducción de dimensiones. Esta sección describe el proceso.

2.6.1 Árboles k-d y ordenamiento

Los árboles k-d (k-d tree en inglés) son árboles binarios donde cada uno de sus nodos tiene componentes en k dimensiones diferentes. Esta es una estructura de datos que permite ordenar un conjunto de N elementos en un árbol balanceado con profundidad $\log_2(N)$ (redondeado a la unidad superior). Este ordenamiento permite agilizar los procedimientos de búsqueda, que resultan en algoritmos que exhiben un comportamiento $O(\log n)$, y que reducen significativamente el número de comparaciones requeridas para alcanzar la solución buscada en comparación al conjunto de datos sin ordenar.

Existen diferentes maneras de construir un árbol k-d balanceado a partir del conjunto de datos introducido [25]. Una de ellas consiste en ir alternando entre cada una de las k dimensiones en orden secuencial, conforme se cambia entre los niveles de profundidad del árbol; en cada caso, se toma como punto de pivote la mediana de los datos en esa dimensión, se introduce el nodo en el árbol y se divide el conjunto de datos en dos, con los elementos menores al pivote en un grupo y los mayores en el otro. El proceso se repite hasta insertar todos los datos en el árbol. Otra alternativa consiste en tomar el conjunto de datos, determinar la dimensión con mayor variación de valores y tomar la mediana en ella. Se subdivide el conjunto igual que antes y se repite el procedimiento.

La última alternativa presentada, y que es la utilizada en esta implementación, consiste en determinar la dimensión de mayor varianza, tomar la mediana en ella y agregarla al árbol, y dividir luego el conjunto de acuerdo a si cada elemento es mayor o menor al valor del pivote en la dimensión determinada, procediendo a repetir el procedimiento en cada subconjunto. El objetivo de este tipo de construcción del árbol es reducir el número de comparaciones requeridas para obtener el nodo más cercano al punto de entrada en los algoritmos de búsqueda, dado que al segmentar el espacio de esta manera se reduce la probabilidad de que se tenga que visitar innecesariamente ramas del árbol y por ende se reduce el número de pasos a realizar para obtener el resultado final [27].

Las búsquedas en el árbol se realizan por la denominada *búsqueda del vecino más cercano*, o "nearest neighbor search" en inglés. Esta consiste en determinar cuál de los N elementos que componen el árbol se encuentra más cercano a un punto pasado como entrada, de acuerdo a una métrica predefinida tal como la distancia euclidiana. Se comparan los resultados obtenidos contra todos aquellos elementos del árbol que deben ser revisados, y se selecciona como resultado final aquel que minimiza esta métrica.

Al igual que con la construcción del árbol, existen diferentes maneras de realizar la búsqueda [46, 25] , donde el objetivo final de cada una es hacerla más eficiente. La elegida acá hace uso de un hiper rectángulo que encierra o contiene todo el conjunto de datos, donde sus límites vienen dados por los nodos con el menor y mayor valor en cada dimensión.

Al realizar la búsqueda, este hiper rectángulo es dividido de manera recursiva por un hiper plano, hasta llegar a un nodo hoja. Una vez hecho esto se calcula la distancia del punto hasta el nodo y se van deshaciendo las divisiones realizadas. En cada caso se compara el mejor resultado actual contra la distancia que se tiene desde el punto de entrada hasta los otros nodos, para verificar cuál de ellos presenta la distancia mínima. Además, se compara la mejor distancia contra la que se tiene entre el punto y los lados del hiper rectángulo, donde en caso que la última sea mayor se pueden descartar ramas enteras para la búsqueda. Esto hace que el algoritmo reduzca el número de comparaciones requeridas para obtener la salida, lo que reduce tiempos de procesamiento.

2.6.2 Análisis de conglomerados de k-medias

El análisis de conglomerados de k-medias (k-means clustering en inglés) permite obtener los centroides que mejor agrupan los vectores o datos de entrada denotados como $\underline{\mathbf{x}}_i$, donde cada uno tiene I componentes. Cada uno de los grupos, o "cluster" en inglés, es descrito por un vector $\underline{\mathbf{m}}_k$ que describe su media en cada dimensión.

Para realizar la agrupación de cada uno de los datos de entrada se utiliza algún tipo de métrica, tal como la mínima distancia euclidiana desde el punto dado por sus vectores hasta cada uno de los grupos. Definiendo este criterio se realiza el agrupamiento en dos pasos: *asignación* y *actualización*.

La *asignación* toma cada uno de los vectores de $\underline{\mathbf{x}}_i$ y los asigna a uno de los grupos descritos por $\underline{\mathbf{m}}_k$, de acuerdo a la métrica utilizada. La *actualización* utiliza todos los valores asignados a cada grupo y por medio de las componentes individuales actualiza la media de los vectores de $\underline{\mathbf{m}}_k$. Ambos pasos se repiten hasta que no existan cambios en los valores asignados a cada uno de los grupos, pues luego del paso de actualización podrían variar. Además, al inicio del proceso se generan valores aleatorios para cada uno de los grupos, que son actualizados posteriormente.

Una descripción detallada para este análisis, así como de las ecuaciones involucradas en cada uno de los pasos del proceso, pueden ser consultados en [21].

2.7 Modelos Ocultos de Markov

Los modelos ocultos de Markov (HMM) son una herramienta estadística que permite modelar sistemas y procesos, y que han sido utilizados en aplicaciones de reconocimiento

del habla, reconocimiento de escritura, reconocimiento facial, análisis de secuencias de ADN, estimación o identificación de acordes musicales, entre otras. Estos constituyen el último paso del modelo SiRPA (figuras 1.2 y 1.3).

Existe mucha literatura referente al tema, la cual detalla tanto aspectos teóricos como de aplicaciones de los HMM. Acá se detallará solamente aspectos prácticos de los HMM, así como los algoritmos involucrados para realizar la evaluación de los modelos. Para detalles teóricos puede consultarse [33], mientras que para las diferentes estrategias de entrenamiento puede utilizarse la misma lectura y [23, 7].

2.7.1 Definiciones básicas para los HMM

Los HMM se definen a partir de los siguientes elementos, que describen cada una de sus características:

- N, que corresponde al número de estados que componen el modelo. Los estados pueden o no tener algún significado físico. Usualmente un estado puede ser alcanzado por cualquier otro (modelo ergódico), pero esto puede variar.
- M, que indica el número de símbolos diferentes que se pueden emitir u observar por el modelo. Estos símbolos corresponden a la salida del sistema que se está modelando. Este parámetro indica el tamaño del alfabeto discreto, donde cada elemento se identifica como $V = \{v_1, v_2, \ldots, v_M\}$.
- La matriz $\mathbf{A}$, que indica la distribución de probabilidad de transición de estados. Esta matriz es descrita por sus coeficientes como $\mathbf{A} = \{a_{ij}\}$, donde cada uno de ellos define la probabilidad de pasar del estado i al estado j. Esta matriz es de tamaño $N \times N$.
- La matriz $\mathbf{B}$, que indica la distribución de probabilidad de observación de símbolos. Esta se describe como $\mathbf{B} = \{b_j(k)\}$, donde cada coeficiente indica la probabilidad de observar el símbolo k en el estado j. Esta matriz es de tamaño $N \times M$.
- El vector $\underline{\boldsymbol{\pi}}$, que da la distribución inicial de estados. Este es de tamaño $1 \times N$, y se describe como $\underline{\boldsymbol{\pi}} = \{\pi_i\}$. Así, cada coeficiente indica la probabilidad de que al inicializarse el modelo se esté en el estado i.

Los HMM pueden ser descritos en su totalidad por las matrices $\mathbf{A}$, $\mathbf{B}$ y el vector $\underline{\boldsymbol{\pi}}$, los cuales suelen resumirse identificando un modelo dado como $\lambda = (\mathbf{A}, \mathbf{B}, \underline{\boldsymbol{\pi}})$ [33]. Los modelos tienen como entrada una cadena de observaciones, la cual se denota como $O = O_1, O_2, \ldots, O_T$.

2.7.2 Algoritmo de evaluación

Existen tres problemas que se presentan a la hora de utilizar los HMM para modelar un proceso o sistema, y que deben resolverse para que sean útiles en aplicaciones prácticas [33]. En este caso, el interés se centra en el primero de ellos, que plantea lo siguiente: *dada*

una cadena de observaciones O y un modelo λ, ¿cómo se puede determinar de manera eficiente $P(O|\lambda)$, la probabilidad de que la cadena haya sido generada por este modelo?

Para resolver este problema se utiliza el denominado algoritmo de evaluación hacia adelante (forward en inglés). Este se divide en tres pasos diferentes. Se tiene primero la *inicialización*, dada por

$$\alpha_1(i) = \pi_i b_i(O_1) \ , \ 1 \leq i \leq N \tag{2.11}$$

Una vez realizado este paso, se realiza la *inducción*, expresada como

$$\alpha_{t+1}(j) = \left[\sum_{i=1}^{N} \alpha_t(i) a_{ij}\right] b_j(O_{t+1}) \ , \ 1 \leq t \leq T-1 \ , \ 1 \leq j \leq N \tag{2.12}$$

Finalmente, se realiza el paso denominado *terminación*, dado por

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i) \tag{2.13}$$

Este procedimiento presenta un problema numérico para su implementación, dado que conforme crece el número de observaciones de la cadena (T), el número de estados del modelo (N) o el tamaño del alfabeto discreto (M), las probabilidades de salida resultantes tienden rápidamente a cero. Esto se debe a que las matrices que describen al sistema se componen de coeficientes con valores pequeños (mucho menores a la unidad por lo general), por lo que al realizar todos los productos necesarios se llega a este problema. Por ello, debe realizarse un procedimiento de escalado [33, 24] en cada una de estas ecuaciones.

Se introduce entonces el factor de escala c_t, el cual se obtiene como

$$c_t = \frac{1}{\sum_{i=1}^{N} \alpha_t(i)} \tag{2.14}$$

Haciendo esto, (2.11) y (2.12) se modifican solamente cambiando α por α', donde este último se obtiene como

$$\alpha_t'(i) = c_t \alpha_t(i) \ , \ 1 \leq i \leq N \tag{2.15}$$

Así, se efectúan los cálculos tal como se muestra en (2.11) y (2.12), pero debe aplicarse el escalado dado por (2.15) *antes* de pasar al siguiente paso, o realizar la siguiente iteración, con el fin de que los valores α sean los correctos. Con este procedimiento ya no es posible utilizar (2.13), y en su lugar se utiliza el valor de su logaritmo, el cual se obtiene usando los c_t como

$$\log(P(O|\lambda)) = -\sum_{t=1}^{T} \log(c_t) \tag{2.16}$$

por lo que los resultados de la evaluación han pasado de estar en el rango de 0 a 1, y se ubicarán ahora entre -∞ y 0.

2.8 Matrices de confusión

Las matrices de confusión son una herramienta que permite validar, o estudiar, la precisión de un clasificador multiclase. Estas son las que dan información sobre las tasas de reconocimiento de los HMM utilizados en la etapa de clasificación, y son usadas durante el entrenamiento para verificar el efecto de las variaciones de los diferentes parámetros de los modelos en sus tasas de reconocimiento.

La matriz de confusión [52], conocida como $\mathbf{C}$, es una matriz de tamaño $N \times N$, donde N es el número de clases usadas en la etapa de clasificación. Las columnas de esta matriz indican la clase *real* a la cual pertenecen los objetos, mientras que las filas indican a cuál de estas clases fueron asignados por el clasificador. Así, el elemento c_{ij} de la matriz de confusión indica un objeto que fue asignado a la clase i, pero pertenece realmente a la clase j.

La matriz de confusión puede resumir los resultados de la clasificación de diferentes formas. La primera es la matriz de confusión *en bruto*, donde se llena $\mathbf{C}$ indicando cuantas veces se dio una determinada situación de clasificación; es decir, cuantas veces el resultado del clasificador fue cada elemento c_{ij}. A partir de esta matriz se pueden obtener dos valores que aportan más información sobre los resultados, que son la *sensitividad* y el *valor predictivo positivo* (VPP).

La sensitividad indica cuál es la probabilidad de que un objeto que pertenece a una determinada clase sea correctamente clasificado. Este valor puede ser estimado al dividir los elementos de la matriz de confusión en bruto entre el total de objetos que componen cada clase real. Es decir, se divide cada elemento de una columna entre la suma del total de elementos de esa columna. De esto último se desprende que la suma de los elementos de cada columna es la unidad.

El VPP indica cuál es la probabilidad de que un objeto asignado a una clase pertenezca verdaderamente a esa clase. Este valor se puede estimar tomando los elementos de la matriz de confusión en bruto y dividiéndolos entre el número total de elementos que se asignaron a cada clase. Es decir, se dividen los elementos de cada fila entre la suma del total de elementos de su fila respectiva, por lo que la suma de los elementos de cada fila suma uno.

Tanto la sensitividad como el VPP son útiles para estudiar la efectividad del clasificador, y lo ideal sería que ambos valores sean altos para todas las clases o elementos

$c_{11}, c_{22}, \ldots, c_{NN}$ de la matriz. En este trabajo, la sensitividad indica qué tan efectiva es la clasificación de las cadenas de observaciones, mientras que el VPP indica qué tan efectivo es el modelo para reconocer las cadenas de su propia clase.

Capítulo 3

Sistema de reconocimiento de patrones acústicos

3.1 Entrenamiento

El entrenamiento busca determinar los valores de las constantes que deben utilizarse en cada uno de los bloques del sistema, y que permitan maximizar las tasas de reconocimiento para cada evento que se busca detectar, permitiendo determinar de manera más precisa el estado del medio y evitar así falsos positivos y falsos negativos.

Como se verá posteriormente, la sección de ejecución (que corre en el sistema embebido) comparte la mayor parte de los bloques explicados en esta subsección, por lo que luego se explicarán sólo las diferencias de implementación entre ambas. A continuación se presenta de manera detallada la solución completa, explicando cada elemento de manera individual; en aquellas secciones que involucran entrenamiento, se presenta el algoritmo que lo realiza antes de presentar el bloque a entrenar.

Todos los bloques llevan a cabo sus cálculos utilizando valores en punto flotante, y la implementación es desarrollada en C/C++ usando el entorno de desarrollo Qt Creator [26]. La aplicación base para este trabajo es la propuesta en [7].

3.1.1 Promediador de señal

El objetivo del promediador es obtener el valor medio de la señal de entrada, con fines de estimación de energía y de normalización de valores (ver sección 2.2).

El comportamiento de descarga exponencial que se presentó es modificado para adaptarlo a las limitaciones impuestas para el desarrollo del SiRPA, sustituyéndola por una carga/descarga lineal dada por las constantes k_{crec} y k_{decrec}. Esto permite reducir el consumo energético y de área para la implementación en ASIC, al evitar la implementación de multiplicadores.

El objetivo de estas constantes es limitar tanto el crecimiento como el decrecimiento de la señal, con el fin de capturar eventos que ocurren en un corto periodo de tiempo (impulsivos) tal como la detonación de un disparo, y mantenerlo presente por el tiempo suficiente para extraer los patrones acústicos característicos, buscando a la vez que la salida no se vea saturada en un valor alto por un periodo prolongado. El diagrama de la figura 3.1, muestra la manera en que este bloque realiza su procesamiento.

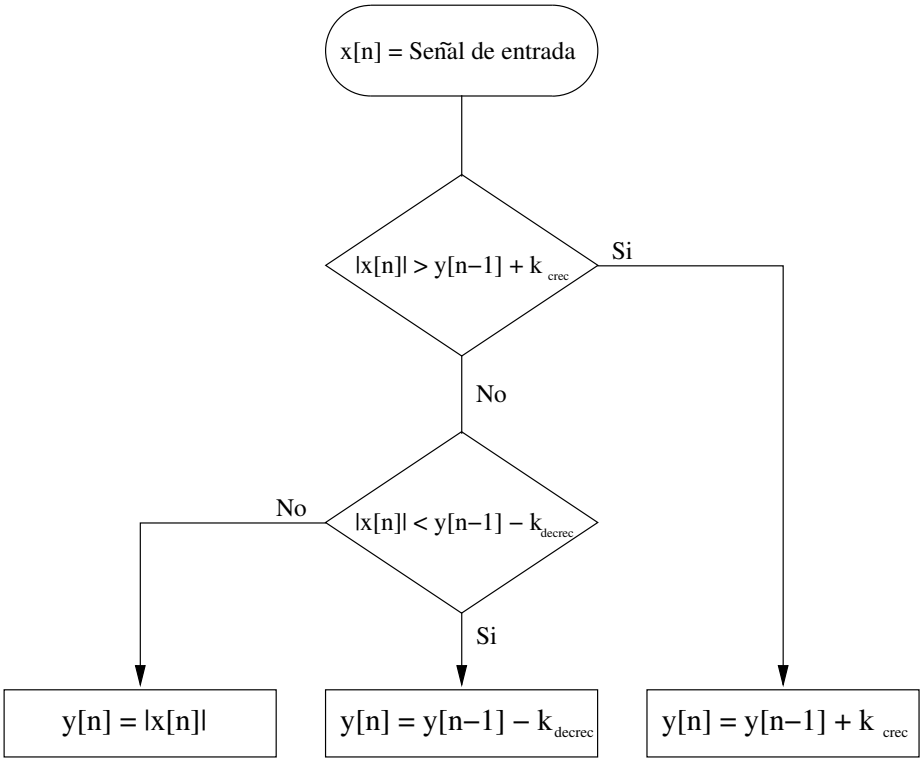


Figura 3.1: Diagrama de flujo para promediador.

En caso de que se busque estimar la energía contenida en la señal, se utiliza el mismo esquema pero con la modificación de que la constante de crecimiento (k_{crec}) es infinita [4], por lo que la primera condición del diagrama de flujo de la figura 3.1 nunca se cumple. Esto se hace con el fin de seguir la envolvente de la señal pero con una descarga lineal, dada por k_{decrec}.

3.1.2 Normalizador de señal

El proceso de normalización consiste en llevar la señal de entrada a un nivel de referencia deseado, con el fin de prepararla para el procesamiento en las siguientes etapas. Este se basa en el diagrama de bloques que se muestra en la figura 3.2.

El bloque promediador es el mismo que se presentó en la sección 3.1.1, y se incluye una constante llamada k_{nor}, cuyo objetivo es evitar la división por cero y limitar el valor de salida a un valor predefinido.

El funcionamiento de este bloque puede ser descrito matemáticamente como

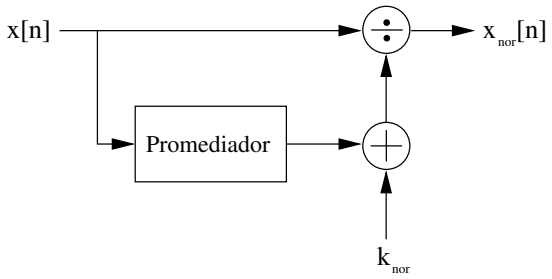


Figura 3.2: Diagrama de bloques para el normalizador.

$$x_{nor}[n] = \frac{x[n]}{x_{prom}[n] + k_{nor}} \tag{3.1}$$

La idea básica detrás de este bloque consiste en que ante un estado normal del medio se tendrán valores de entrada que se mantendrán dentro de una banda reducida, por lo que el promedio de la señal será relativamente constante y la salida del normalizador estará cerca de la unidad. En caso que se presente un evento impulsivo la señal de entrada crecerá rápidamente (posiblemente saturándose), por lo que al estar dividiéndose entre un valor menor a la unidad (pues en el estado normal del medio se tendrá un promedio bajo) la salida del normalizador se disparará por encima de 1, lo que se reflejará en los patrones de salida obtenidos en etapas posteriores.

La selección adecuada de la constante k_{nor} es importante, pues si se usa un valor grande dominará la expresión del denominador, provocando que los valores de salida del normalizador sean por lo general bajos y no se pueda observar el efecto impulsivo de la salida ante la presencia de sonidos de alta energía como disparos, además de que pequeñas variaciones de la entrada quedarían enmascaradas por esta constante. Por otro lado, un valor pequeño provocaría la presencia de ruido indeseado y valores muy altos a la salida, pues si el promedio es muy bajo (inclusive puede ser cero), la división se realizaría por un valor cercano a cero. Lo ideal es analizar ejemplos de la señal de entrada para seleccionar este valor un poco por encima del piso de ruido, verificando que también la salida máxima esperada no supere un techo dado.

3.1.3 Banco de filtros

El banco de filtros implementado utiliza codificación subbanda, separando la señal de entrada en diferentes bandas de frecuencia, cada una con un tamaño de una octava menor a la anterior. En la figura 3.3 se muestra el diagrama de bloques del sistema, en donde se han colocado estimadores de energía a la salida de cada banda.

La tabla 3.1 muestra la distribución de frecuencias por banda, según se dimensionaron.

En este caso, y siguendo el diagrama de la figura 3.3, se tendría que $i = 9$ pues se tienen 9 salidas del banco, una por cada banda; además, $N = 7$, pues existen 7 bloques de diezmado desde el punto donde entra la señal hasta el último par de filtros del banco.

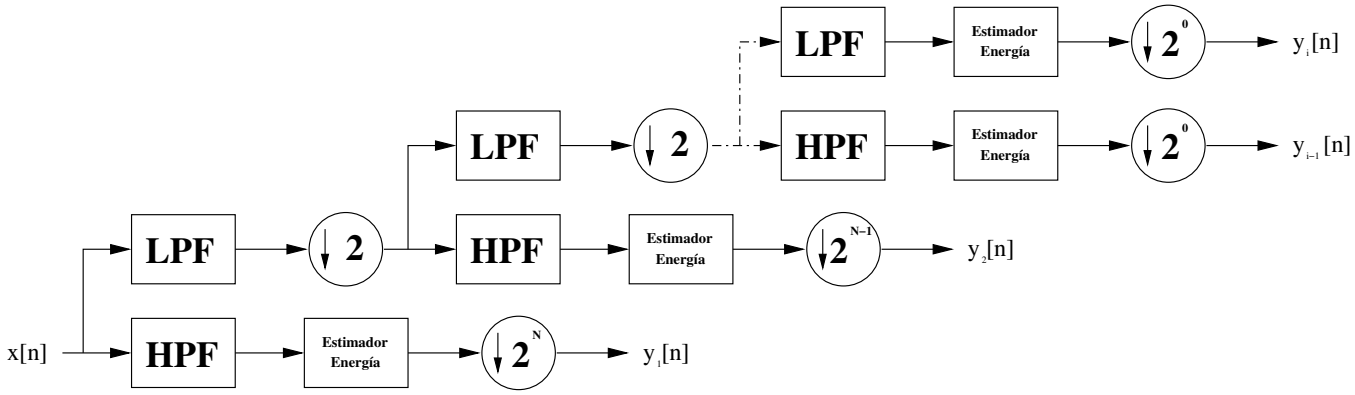


Figura 3.3: Diagrama de bloques para el banco de filtros.

Tabla 3.1: Distribución de frecuencias en el banco de filtros.

Banda	Frec. Muestreo (Hz)	Frec. Superior (Hz)	Frec. Inferior (Hz)
1	44100	22050	11025
2	22050	11025	5512,5
3	11025	5512,5	2756,25
4	5512,5	2756,25	1378,125
5	2756,25	1378,125	689,0625
6	1378,125	689,0625	344,53125
7	689,0625	344,53125	172,265625
8	344,53125	172,265625	86,1328125
9	344,53125	86,1328125	0

Sin embargo, la salida del último LPF del banco, en la banda 9, es descartada, pues un análisis previo [41] mostró que no aporta información relevante para la transformación realizada por el siguiente bloque en la cadena de procesamiento.

Los estimadores de energía mostrados fueron descritos en la sección 3.1.1, y corresponden a un bloque similar al promediador de la señal. El estimador conectado a la salida del primer HPF usa la misma k_{decrec} que se utiliza para el bloque promediador del normalizador, pero este valor se incrementa en un factor de 2 conforme se avanza en cada banda (donde la frecuencia de muestreo disminuye en el mismo factor). Esto se hace con el fin de que el efecto del estimador de energía sea similar en todas las bandas, pues conforme se disminuye la frecuencia de muestreo, cada banda tiene un menor número de muestras para procesamiento, y sin este ajuste no se obtendrían niveles de energía equivalentes en cada salida del banco.

La inclusión de los bloques de diezmado adicionales a la salida de cada estimador se hace con el fin de que la salida del banco se actualice hasta el momento en que todos los estimadores que componen el banco hayan actualizado su salida, pues de otra forma no se capturaría la variación en todas las bandas y se provocaría un despilfarro energético en la implementación en ASIC [34]. Así, la salida del banco debe actualizarse a una frecuencia

igual o menor que la última frecuencia de muestreo del banco, donde se elige acá que sea igual para contar con un número mayor de muestras para entrenamiento y evaluación. Etapas posteriores al banco de filtros operan a la misma frecuencia seleccionada.

En la implementación en la que se basa este trabajo, se olvidó incluir a la salida del banco estos bloques de diezmado, lo que provocaba que la salida se actualizara a la misma frecuencia de muestreo de la entrada. Los bloques del reductor de dimensiones y del árbol k-d también funcionaban a esta frecuencia, provocando que se diera una repetición de símbolos en las cadenas de observaciones por periodos prolongados de tiempo, al no capturar la variación en todas las bandas de frecuencia. Por ello, fue necesario agregar estos bloques.

Además, se corrigió el punto donde se toma la salida del banco, pues en esa misma implementación se tomaba directamente de los HPF y estos presentan valores de salida volátiles o cambiantes. Así, se agregaron los estimadores de energía previo a la salida del banco, calculando e introduciendo al mismo tiempo la constante de decrecimiento k_{decrec} adecuada para cada banda.

3.1.4 LDA

El entrenamiento con LDA se realiza mediante una aplicación externa al sistema implementado, utilizando los vectores de salida del banco de filtros. Cada uno se toma a la misma frecuencia con la que se actualiza la salida del banco, y son vectores en un espacio de 8 dimensiones.

El análisis de discriminantes lineales (LDA) busca realizar una proyección precisa del espacio de 8 dimensiones a uno de 3 dimensiones, maximizando la varianza interclase y minimizando la varianza intraclase [9], con el fin de facilitar la discriminación de los patrones característicos de cada clase que se busca reconocer. Esto se hace por medio de la matriz de transformación $\mathbf{W}$ y el vector de media $\underline{\boldsymbol{\mu}}$.

Los vectores usados para entrenamiento son extraídos y almacenados en archivos de texto, utilizando una base de datos basada en MySQL para realizar su administración y consulta. Cuando se desea realizar el entrenamiento se lleva a cabo la preparación de matrices individuales para cada clase, utilizando las muestras seleccionadas. La matriz de transformación resultante es almacenada en la misma base de datos, y una vez obtenida puede ser utilizada en el módulo de reducción de dimensiones. La aplicación externa, basada en la LTI-Lib [12], fue el resultado de [36].

3.1.5 Reductor de dimensiones

El objetivo del reductor de dimensiones es transformar el espacio de valores de salida del banco de filtros, de ocho dimensiones, a un espacio tridimensional. Esto tiene como fin reducir tanto el tiempo de procesamiento como el consumo energético en la implementación

en ASIC, pues aunque el procedimiento conlleva una pérdida de información, se compensa con una optimización de estos factores. La transformación se hace mediante un producto matriz-vector, donde el diagrama de bloques de la figura 3.4 muestra cómo se realiza esta operación.

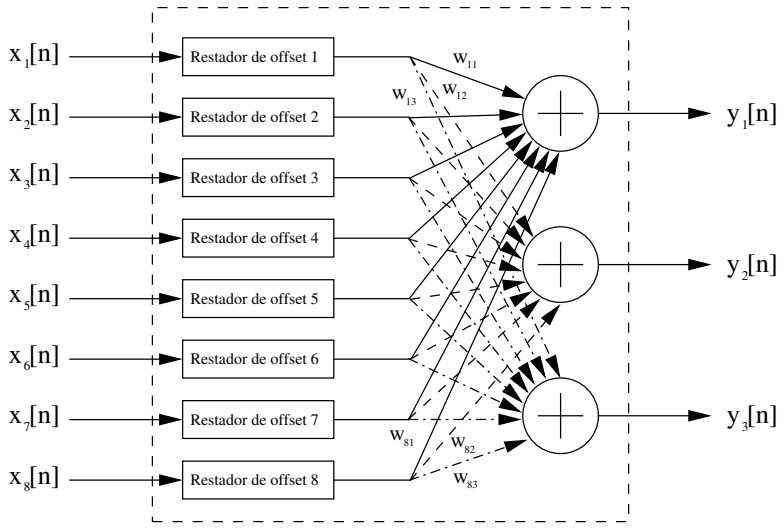


Figura 3.4: Diagrama de bloques para el reductor de dimensiones.

Este bloque carga los valores usados por la matriz de transformación y por el vector de media desde archivos generados a partir del entrenamiento con LDA, en caso que hayan sido exportados primero. Estos valores son almacenados en arreglos internos con el fin de realizar primero la resta de la media y luego aplicar el producto matriz-vector a todas las muestras de salida del banco de filtros.

3.1.6 k-medias

El algoritmo de análisis de conglomerados de k-medias (k-means en inglés), es utilizado con el fin de generar los centroides en el espacio tridimensional que identifican a cada uno de los símbolos que componen el alfabeto discreto utilizado en la etapa de clasificación.

Para la administración y extracción de los vectores se utilizan archivos de texto y una base de datos basada en MySQL. El entrenamiento se realiza llamando a una aplicación externa a la cual se le pasa como parámetro un archivo que contiene la totalidad de los vectores seleccionados para entrenar. La aplicación genera un archivo que contiene los centroides calculados, que son posteriormente cargados y almacenados en la base de datos, y que además pueden ser vinculados con el sistema y utilizarlos para generar el árbol k-d.

3.1.7 Árbol k-d

En esta implementación se utiliza un árbol k-d para organizar los 32 centroides, de 3 dimensiones cada uno, generados a partir del entrenamiento con el algoritmo de k-medias,

permitiendo así realizar la búsqueda del centroide cuyas coordenadas se encuentran más cerca del punto pasado como entrada, que es la salida del reductor de dimensiones. Se utiliza como métrica la distancia euclidiana al cuadrado desde cada uno de ellos hasta el punto en análisis, expresada como

$$d = \sum_{i=1}^{k} (\underline{\mathbf{m}}_i - \underline{\mathbf{x}}_i)^2 \tag{3.2}$$

donde $\underline{\mathbf{x}}$ es el vector de entrada y $\underline{\mathbf{m}}$ el vector del centroide en análisis. Se usa esta métrica en lugar de la distancia euclidiana para optimizar el proceso al evitar el cálculo de la raíz cuadrada, dado que los resultados de las comparaciones no se ven afectados.

Una vez que se ha localizado el nodo, se obtiene como salida su identificador o símbolo, que es utilizado para generar la cadena de observaciones que se pasa a los HMM. Cada nodo del árbol almacena los siguientes datos: coordenadas del centroide en 3D, identificador único del nodo, dirección de partición del espacio (que en este caso puede ser x, y ó z) y vecinos a su izquierda y derecha (de acuerdo a la dimensión de partición actual).

El algoritmo de búsqueda del vecino más cercano trabaja de manera recursiva, realizando una división del hiper rectángulo (o al ser en 3D, el cubo) que contiene todos los centroides conforme se profundiza en cada nodo hasta alcanzar una hoja. Esto se realiza verificando si la coordenada del punto de entrada es menor o mayor a la del centroide con el que se compara en cada momento, de acuerdo a la dimensión de partición de este último, decidiendo así si se mueve hacia la rama izquierda o derecha.

Cuando se alcanza la hoja, se calcula (3.2) usando las coordenadas del centroide asociado a este nodo y las del punto de entrada, y se comienzan a deshacer las particiones realizadas al hiper rectángulo, verificando nuevamente la métrica y comparándola contra los otros nodos del árbol para obtener cuál centroide la minimiza, que determinaría el símbolo de salida.

La partición del hiper rectángulo se realiza con el fin de determinar si debe descartarse o no la revisión de las ramas no visitadas, calculando (3.2) con el punto en análisis y los "lados" del hiper rectángulo que encierra el subespacio no visitado, el cual contiene todos los centroides de la rama analizada. Si la métrica resultante es menor a la mejor calculada hasta el momento debe revisarse esta rama, al existir la posibilidad de que el subespacio que ella encierra contenga un centroide que esté más cerca del punto de entrada.

Los algoritmos utilizados por el k-d tree fueron tomados de [48], pero adaptados para los requisitos de esta aplicación. El algoritmo de construcción del árbol en la implementación en que se basa este trabajo presentaba el problema de que no se actualizaban los "lados" del hiper rectángulo al introducir un nuevo nodo al árbol, lo que provocaba que al realizar las búsquedas se descartaran ramas que no debían descartarse. Esto fue corregido para que el hiper rectángulo se actualice conforme se insertan los nodos al árbol.

3.1.8 Entrenamiento de los HMM

Con el fin de determinar los valores de las matrices $\mathbf{A}$, $\mathbf{B}$ y el vector $\underline{\boldsymbol{\pi}}$ que caracterizan a cada modelo λ, se utilizan algoritmos de entrenamiento simple, múltiple y negativo. El objetivo de estos es maximizar las tasas de reconocimiento del modelo para eventos de su propia clase, a la vez que se disminuyen para los de las otras clases.

El entrenamiento se lleva a cabo capturando muestras de observaciones, utilizando archivos de audio de ejemplo para las diferentes clases o eventos que se busca reconocer. En la figura 1.2 se muestra el punto desde el cual se extraen los símbolos para realizar el entrenamiento, los cuales son almacenados en archivos de texto, y administrados por la base de datos implementada en MySQL.

Se utiliza el algoritmo de Baum-Welch [23] para el caso de entrenamiento simple y múltiple, donde para este último se debe modificar el algoritmo tal como se explica en [33]. Para el caso de entrenamiento negativo se utiliza el denominado Bold's Smooth Algorithm [23]. Todas estas variantes de entrenamiento fueron implementadas en un trabajo previo [7]. Sin embargo, tenían problemas de estabilidad al intentar utilizar para entrenamiento o evaluación cadenas de observaciones que no contaban con una longitud suficiente de acuerdo a determinados valores de muestreo y/o longitud de cadena, lo que llevaba al cierre de la aplicación. Esto se corrigió para que aquellas cadenas que no cumplan con la longitud requerida sean descartadas de estos procedimientos.

3.1.9 Evaluación de los HMM

Con el fin de realizar la evaluación de los HMM se utilizan las matrices de confusión, prestando especial atención a los valores de sensitividad y VPP obtenidos. Estos valores permiten estudiar el efecto en las tasas de reconocimiento de la variación de los parámetros de número de estados, muestreo, longitud de cadena y razón de aprendizaje (para el entrenamiento negativo), así como de las cadenas seleccionadas para entrenar. La idea es maximizar los casos donde la clase asignada i y la clase real j sean iguales, lo que denota una clasificación correcta.

Para formar las matrices se toma una cadena de observaciones para una clase determinada j y se pasa al módulo de clasificación, donde se tienen todos los HMM que se busca evaluar. Estos utilizan el algoritmo de evaluación hacia adelante para obtener $\log(P(O|\lambda))$, y se selecciona como clase asignada i a la clase del modelo que tenga un valor más cercano a cero, pues por el mapeo realizado al introducir los factores de escala descritos en la sección 2.7.2, valores cercanos a cero indican una mayor probabilidad de que el modelo haya sido el que generó esa cadena de observaciones. De esta forma se completa cada elemento c_{ij} de la matriz de confusión en bruto $\mathbf{C}$, a partir de la cual se obtienen luego los valores de sensitividad y VPP.

La implementación de la función de evaluación, junto a otras de entrenamiento y administración para los HMM, se realizó utilizando la biblioteca provista en [10], que se encuentra

desarrollada en C.

3.1.10 Clasificación

La etapa de clasificación es la que contiene los HMM utilizados para evaluación, los cuales son el resultado del entrenamiento realizado con los algoritmos previamente mencionados. En la implementación en el computador de propósito general, o de entrenamiento, este bloque no existe como parte de la cadena de procesamiento del SiRPA, pero se incluye en el diagrama de la figura 1.2 por claridad.

Cada vez que se lleva a cabo la evaluación de un HMM, o un grupo de ellos, se realiza la carga de los parámetros para cada uno desde la base de datos, junto con las cadenas que hayan sido elegidas. Se procede luego a verificar las probabilidades de salida para cada modelo de acuerdo a la cadena que se le pasa como entrada. Una vez que se realiza este procedimiento con todas las observaciones, se almacenan los valores resultantes en la base de datos, se generan las matrices de confusión por modelo y se generan las estadísticas de reconocimiento, quedando estos resultados disponibles para la revisión por el usuario.

3.2 Ejecución

La sección de ejecución corresponde a una versión reducida de la cadena de procesamiento utilizada para entrenamiento, la cual se optimiza para reducir el tiempo de procesamiento por cada bloque de señal, permitiendo al sistema embebido operar en línea a una frecuencia de muestreo definida. En la implementación de hardware del SiRPA, la reducción en los tiempos de procesamiento y el número de pasos requeridos para obtener la salida permiten reducir a su vez el consumo energético del sistema.

La implementación de ejecución hace uso de las diferentes matrices y vectores generados durante el entrenamiento, y solamente tiene la salida de la etapa de clasificación. Dado que la mayor parte de los bloques ya fueron detallados, se explican a continuación solamente las diferencias con respecto a la implementación de entrenamiento.

3.2.1 Implementación en sistema embebido

La implementación en el sistema embebido cuenta con dos modos de reconocimiento: utilizando archivos de audio (en formato WAV), o mediante la entrada de audio (de línea) que incluye la plataforma Beagleboard-xM utilizada [6]. La selección entre un modo u otro se da de acuerdo al número de argumentos pasados a la aplicación a la hora de llamarla desde la línea de comandos.

El modo de reconocimiento en base a archivos de audio se incluye con el fin de realizar pruebas de reconocimiento de manera rápida y flexible. Este modo permite comparar los

resultados de salida de cada etapa y de las probabilidades finales contra la implementación en el computador de propósito general. La aplicación lee la totalidad del archivo o sólo una parte de él, de acuerdo a la configuración elegida a la hora de compilarla, y genera archivos de texto donde se almacenan los valores de salida de cada una de las etapas de la cadena de procesamiento. Además, imprime en consola el estado determinado a partir de la evaluación de los modelos pasados como parámetros. Se almacena también información sobre tiempos de ejecución por etapa o por bloque procesado (cuyo tamaño es también configurable), lo que permite estimar los tiempos de procesamiento que se tendrán al utilizar el modo de reconocimiento en línea.

El modo de reconocimiento mediante la entrada de audio corresponde al modo de reconocimiento en línea. Este permite que se conecte un micrófono (que debe ser preamplificado, pues el sistema embebido no realiza amplificación alguna) a la plataforma, para realizar pruebas de campo. La captura de los datos desde esta entrada se hace mediante un cliente de JACK [13], que se encarga de tomar bloques de muestras de audio y realizar llamadas a una función de procesamiento específica, en un entorno de baja latencia. Parámetros como la frecuencia de muestreo, tamaño de bloque, número de búferes usados, entre otros, son configurables al iniciar el cliente.

En esta implementación se utiliza un tamaño de bloque de 256 muestras (que por limitaciones del sistema embebido es el máximo utilizable), con una frecuencia de muestreo de 48 kHz (cuyo valor no es configurable en este caso, y es impuesto por el hardware de audio de la plataforma), lo que implica una ventana de tiempo para procesamiento de menos de 5,3 ms, luego de la cual se llama a procesar el bloque siguiente.

Este modo de operación no almacena información en archivos, pues la apertura, escritura y cierra de ellos conlleva tiempos en el rango de los milisegundos, que por las limitaciones de tiempo mencionadas no son tolerables. En su lugar, se imprime solamente en consola información sobre el estado actual del medio con su probabilidad asociada (de acuerdo a evaluación del modelo), actualizando el estado sólo en caso de que éste cambie.

3.2.2 Procesamiento en punto fijo

Con el fin de estudiar maneras de acelerar el procesamiento de los datos y aprovechar los elementos de hardware provistos por el sistema en chip (SoC) DM3730 [18] de la plataforma utilizada, se hace uso del DSP (TMS320C64x+) incluido. Este procesador realiza sus cálculos mediante una representación numérica en punto fijo, por lo que es necesario adaptar los algoritmos que se desean ejecutar en el procesador a este formato. De no hacerlo así el compilador generará el código requerido para realizar la emulación del formato en punto flotante, lo que conlleva tiempos de procesamiento excesivos (decenas de veces más lentos).

Para determinar cuál sección de la cadena de procesamiento del SiRPA transferir al DSP se analizó la frecuencia a la que opera cada uno de los bloques de la etapa de *extracción de características*, que se presentó en la figura 1.2. Estas frecuencias se muestran en la

tabla 3.2. La frecuencia de muestreo utilizada como referencia es la de la implementación en ASIC (44,1 kHz), aunque podría utilizarse también la del sistema embebido (48 kHz).

Tabla 3.2: Frecuencia de operación de bloques de la cadena del SiRPA.

Bloque	Frec. de operación (Hz)
Banco de filtros	44100
Reductor de dimensiones	344,53125
Árbol k-d	344,53125

A partir de esto se decide transferir el bloque del banco de filtros al DSP, ya que es el único bloque de esta etapa que opera a la misma frecuencia con que se toman las muestras de la señal de audio, por lo que se le transfieren al procesador bloques de señal de gran longitud (256 muestras en este caso) y se evita así el intercambio frecuente de datos entre procesadores, que conlleva un *overhead*. Así, se hace el cambio de todos sus algoritmos para operar usando una representación en punto fijo de 32 bits, con un número de bits configurable para la parte fraccionaria y entera. No es posible utilizar un menor número de bits, pues conllevaría una pérdida de precisión importante respecto a la implementación en punto flotante.

Con el fin de llevar a cabo la compilación de los ejecutables para el DSP, se utiliza código desarrollado en C que se compila mediante la herramienta C6RunLib [17], la cual se encarga de que a la hora de ejecutar la aplicación en el sistema embebido se realice la carga del código necesario en el DSP. Este también introduce *stubs* con el fin de que las llamadas a las funciones transferidas al DSP sean desviadas hacia éste, pasándole el control de los datos y punteros a ellos; los resultados son almacenados también en memoria, retornando el DSP punteros hacia ellos.

La memoria utilizada por el DSP debe ser colocada en un espacio de memoria específico, con el fin de que pueda ser utilizada por el DSP, por lo que C6Run cambia las funciones de solicitud y liberación de memoria por sus propias versiones. Esta necesidad surge a partir del hecho de que el procesador de propósito general (GPP) ARM cuenta con una unidad de administración de memoria (MMU), que es utilizada por el sistema operativo para operar en un contexto de memoria virtual, mientras que el DSP no cuenta con una de estas unidades y trabaja por lo tanto con memoria física. De esta forma, debe realizarse algún tipo de conversión entre los punteros usados por ambos procesadores, pues de otra forma una dirección de memoria apuntada por el GPP por lo general no concordaría con la apuntada por el DSP, resultando en errores de acceso a los datos.

Generación y optimización de código para el DSP

Para el desarrollo de la sección de código que se ejecuta en el DSP se utiliza el lenguaje de programación C, donde los archivos fuente que se generan se compilan por medio de la herramienta C6RunLib. Además de cambiar todos los algoritmos usados por el banco

de filtros y los estimadores de energía conectados a su salida al formato de representación numérica en punto fijo, se utilizaron los denominados *intrinsic* [19] del DSP para mejorar el rendimiento de las funciones implementadas. Estos son instrucciones de ensamblador del DSP que pueden ser llamadas directamente desde el código en C, y que usualmente se ejecutan en un solo ciclo de reloj de este procesador.

En [19] se presentan otras alternativas para el desarrollo de aplicaciones para el DSP, que son el uso de ensamblador lineal y el ensamblador propio del procesador. La diferencia entre ambos es que para el primero debe especificarse solamente las instrucciones específicas del DSP que se desean utilizar, y el compilador se encarga automáticamente de los detalles de paralelismo, asignación de registros y uso de las unidades de ejecución (al tener una arquitectura *pipeline*). Por otro lado, el uso de ensamblador propio del sistema requiere que el programador indique explícitamente estos detalles.

El problema principal con estas alternativas es que la herramienta de compilación (C6RunLib) no permite utilizarlas, obligando a desarrollar el código tal como se describió previamente, donde el compilador genera el código ensamblador a partir del código desarrollado en C.

Dado que se desea estudiar el uso de las unidades del DSP para explotar el paralelismo entre ellas, se agregan banderas al compilador (ver configuración en la sección de apéndices) para que al crear el archivo ejecutable se genere también el archivo que contiene las instrucciones de ensamblador requeridas para realizar cada instrucción de código C. Así, puede estudiarse de qué manera se utilizan las unidades de ejecución del procesador al cambiar el código C y/o al utilizar los *intrinsics*. Esta es la estrategia recomendada en [19] para el desarrollo de aplicaciones, antes de recurrir al uso de código ensamblador que es menos "portátil", y que si no se utiliza correctamente no tendría un mejor rendimiento respecto al obtenido con herramientas automatizadas.

Un ejemplo de código ensamblador generado de manera automática se muestra en la figura 3.5. En este caso, la primera columna indica la instrucción de ensamblador del DSP utilizada, mientras que la segunda columna indica cuál unidad de ejecución se está utilizando. La tercera columna indica los registros utilizados en esa operación, y la última columna indica cuál es la instrucción de código C asociada a esta instrucción de ensamblador. Por medio de esto se puede reestructurar el código C y utilizar *intrinsics* para lograr que al pasar el código al compilador se utilicen mejor las unidades del procesador, y se pueda paralelizar funciones. En [15] puede consultarse más información sobre la arquitectura interna del DSP utilizado y sus unidades de ejecución.

Optimización de algoritmos para ejecución en el DSP

Dado que el bloque normalizador previo al banco de filtros y el reductor de dimensiones que se ubica luego de él trabajan con una representación numérica en punto flotante, se deben introducir bloques para realizar la conversión de punto flotante a punto fijo y viceversa. El diagrama resultante de esto se muestra en la figura 3.6.

```
EXCLUSIVE CPU CYCLES: 33
LDW     .D2T2   *B10,B4          ; |dsp_lib/filterDSP.dsp_stub.c:55|
ADD     .D2     SP,16,B5         ; |dsp_lib/filterDSP.dsp_stub.c:55|
ADD     .L2     4,B10,B31        ; |dsp_lib/filterDSP.dsp_stub.c:56|
ADD     .L2     12,SP,B30        ; |dsp_lib/filterDSP.dsp_stub.c:56|
ADD     .L1X    8,B10,A3         ; |dsp_lib/filterDSP.dsp_stub.c:56|
STW     .D2T2   B4,*B5           ; |dsp_lib/filterDSP.dsp_stub.c:55|
LDW     .D2T2   *B31,B5          ; |dsp_lib/filterDSP.dsp_stub.c:56|
ADD     .L1X    8,SP,A4          ; |dsp_lib/filterDSP.dsp_stub.c:57|
ADD     .L2     12,B10,B10       ; |dsp_lib/filterDSP.dsp_stub.c:56|
NOP             2
STW     .D2T2   B5,*B30          ; |dsp_lib/filterDSP.dsp_stub.c:56|
LDW     .D1T1   *A3,A3           ; |dsp_lib/filterDSP.dsp_stub.c:57|
ADD     .L2     4,SP,B5          ; |dsp_lib/filterDSP.dsp_stub.c:58|
NOP             3
STW     .D1T1   A3,*A4           ; |dsp_lib/filterDSP.dsp_stub.c:57|
LDW     .D2T2   *B10,B4          ; |dsp_lib/filterDSP.dsp_stub.c:58|
NOP             4
STW     .D2T2   B4,*B5           ; |dsp_lib/filterDSP.dsp_stub.c:58|
LDW     .D2T2   *+SP(4),B6       ; |dsp_lib/filterDSP.dsp_stub.c:59|
LDW     .D2T1   *+SP(8),A6       ; |dsp_lib/filterDSP.dsp_stub.c:59|
LDW     .D2T1   *+SP(16),A4      ; |dsp_lib/filterDSP.dsp_stub.c:59|
```

Figura 3.5: Código ensamblador generado por herramienta C6RunLib.

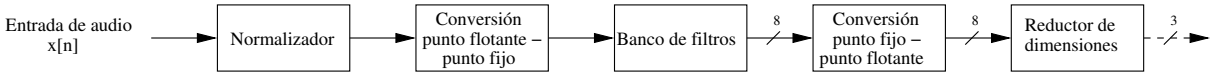


Figura 3.6: Cadena modificada para conversión entre formatos numéricos.

Lo primero que se realizó antes de comenzar con las modificaciones del resto del sistema fue determinar el número de bits requeridos para la parte entera y fraccionaria de la representación en punto fijo, para lo que se realizó un *profiling* del código desarrollado en formato punto flotante, y se estudiaron los valores máximos esperados y el tipo de precisión requerida.

Por el tipo de entradas que se esperan desde el bloque normalizador se determinó que se requiere como mínimo de 8 bits en la parte entera, y podrían requerirse más en caso de que se modifiquen las constantes usadas en él. De no usarse esta cantidad de bits se corre el riesgo de que se presente un desbordamiento u *overflow* de los datos, dando lugar a valores erróneos en caso que el resultado salga del rango de representación que da el número de bits elegido.

Por otro lado, se realizaron pruebas para determinar de qué manera varían los símbolos de salida de la etapa de extracción de características con respecto a la implementación de entrenamiento, de acuerdo al número de bits usados para la parte fraccionaria. Se encontró que como mínimo deben utilizarse 20 bits en la parte fraccionaria para que no varíen las cadenas de observaciones generadas y se provoque por lo tanto una clasificación incorrecta de la señal debido a esta variación.

Así, se utiliza una representación en punto fijo de 32 bits, lo que repercute sobre las posibilidades de optimización del código en el DSP dado que este tiene una arquitectura con un tamaño de bus de 16 bits [15], y se encuentra optimizado para trabajar con paquetes de datos ya sea de 8 o 16 bits. Esto afecta también el paralelismo, dado que el mayor número de bits obliga a utilizar varias unidades de ejecución para una sola operación aritmética, que de otra forma hubieran podido ser utilizadas por dos operaciones en

paralelo. Por ejemplo, una multiplicación entre dos valores de 32 bits tomará cuatro veces más tiempo que el que tomaría multiplicar dos valores de 16 bits [19].

Definiendo esto, se realizaron las modificaciones requeridas para optimizar el código en C, a la vez que se introdujeron secciones de código para que el compilador optimice secciones de la aplicación. Para esto último se introducen directivas para indicarle al compilador el número de veces mínimo y máximo que se espera se ejecute un ciclo en el código, lo que permite estimar cuándo se presentará una ramificación en su ejecución. Para esto se requiere analizar primero el código y estimar estos valores. Por ejemplo, la siguiente directiva indica que el código se ejecutará exactamente 256 veces, lo que se sabe previamente por el tamaño de bloque elegido.

```
#pragma MUST_ITERATE (256, 256);
```

Además de esto, se le indica al compilador de qué manera se utilizan variables de la función, con el fin de que puedan ser optimizadas en cuanto a colocación en espacios de memoria, acceso a los datos y uso de sus punteros, utilización de la caché en cuanto a localidad espacial y temporal, entre otras cosas. Por ejemplo, en la declaración de la función se colocan los argumentos como

```
INBUF int * restrict inBuffer, OUTBUF int * restrict outBuffer
```

donde INBUF indicaría que el búfer es sólo de entrada, mientras que OUTBUF indica que es de sólo lectura. Esto evitaría que el compilador intente agregar instrucciones y crear variables para escribir o leer de ellos, respectivamente. El término *restrict* indica además que solo el puntero pasado como argumento, y aquellos que se puedan derivar de él, tiene acceso a los datos en todo momento. De esta manera, se dispone libremente de los datos y el compilador hace optimizaciones adicionales.

El código en C se reestructuró además para reducir el número de variables requeridas en las operaciones, y evitar así la asignación innecesaria a registros. Sin embargo, existen casos donde se dejan variables temporales para ciertas operaciones aritméticas de multiplicación y acumulación, que evitan que se acceda de manera constante a memoria y se utilice en su lugar un registro del procesador, lo que disminuye tiempos de operación. Además, los "objetos" de filtros y el banco de filtros en sí fueron modificados para que a la hora de compilar la aplicación se sepa previamente el tamaño en memoria que ocupan, pues de otra forma la aplicación no puede ser compilada.

Finalmente, se introducen los *intrinsic* para optimizar ciertas operaciones realizadas. Por ejemplo, al realizar la multiplicación de dos valores de 32 bits, el resultado es un número de 64 bits que luego debe ser desplazado para obtener nuevamente un valor de 32 bits que represente el resultado. Esto se puede realizar con dos *intrinsic* como

```
tmpM = _mpy32ll(tmpOp1, tmpOp2);
tmpMul  = _hill((tmpM << INT_BITS));
```

de manera que en la variable tmpMul se tiene el resultado de multiplicar las variables de 32 bits tmpOp1 y tmpOp2. Esto se traduce en las dos instrucciones de ensamblador

```
MPY32
SHRU
```

que toman 4 ciclos y 1 ciclo de reloj del procesador para ejecutarse, respectivamente [16].

Es posible realizar optimizaciones adicionales al código en caso que se reduzca el número de bits usados. Si se utilizan 16 bits en total para la parte entera y fraccionaria se podrían utilizar un mayor número de *intrinsic*, además de que se podrían paralelizar un mayor número de instrucciones. Sin embargo, debe evaluarse la pérdida de precisión que se tendrá en los resultados con respecto a la implementación en punto flotante, y también considerarse los efectos de desbordamiento. Para más información respecto a las posibles optimizaciones se puede consultar [19].

3.2.3 Ordenamiento del árbol k-d

La implementación para la construcción del árbol k-d utilizada en la sección de entrenamiento produce estructuras desbalanceadas, lo que no es un factor crítico debido a la naturaleza de operación fuera de línea de esta etapa. Sin embargo, dado que el sistema embebido trabaja en línea se deben disminuir los tiempos de procesamiento al máximo, por lo que no son tolerables los retardos adicionales debido a esto.

Por ello, se implementa un algoritmo que realiza el ordenamiento del árbol de manera recursiva, ejecutándose al momento de inicializar los parámetros del sistema para que cuando se inicie la captura y procesamiento de datos ya se encuentre balanceado. Esto permite que el algoritmo de búsqueda sea efectivamente del tipo $O(\log n)$, haciendo que se requiera de tan sólo alrededor de 5 comparaciones como máximo para encontrar el centroide más cercano al punto de entrada [36].

El algoritmo de ordenamiento realiza la partición en base a la dimensión de mayor varianza, utilizando el algoritmo Quicksort para organizar el conjunto de datos. El funcionamiento es el siguiente: se toma el conjunto de datos y se determina cuál de las k dimensiones (tres en este caso) presenta la mayor varianza, usando las coordenadas de todo el conjunto para realizar el cálculo. Se ordenan los centroides de menor a mayor según la coordenada de cada uno en la dimensión determinada y se toma la mediana del conjunto, que se inserta inmediatamente al árbol. El algoritmo se repite de manera recursiva sobre cada uno de los subconjuntos generados, hasta que se hayan insertado todos los nodos en el árbol, resultando en un árbol balanceado.

3.2.4 Clasificación

La etapa de clasificación consiste en un modelo para cada clase a identificar, donde cada modelo estima la probabilidad de que la secuencia de observaciones a su entrada sea emitida u observada por él. Se asume que la clase correcta para la cadena de observaciones es aquella cuyo modelo produzca la mayor probabilidad; en este caso, la que se encuentre más cercana a cero al utilizarse $\log(P(O|\lambda))$.

Al inicializar se cargan todos los parámetros de los modelos desde un archivo que contiene las matrices $\mathbf{A}$, $\mathbf{B}$ y el vector $\underline{\boldsymbol{\pi}}$, junto a los parámetros N, T, F y V. Todos estos valores son generados de manera automática al exportar los modelos desde la implementación de entrenamiento. La figura 3.7 ilustra la estructura interna que se tendría en este bloque al exportar un modelo de cada clase.

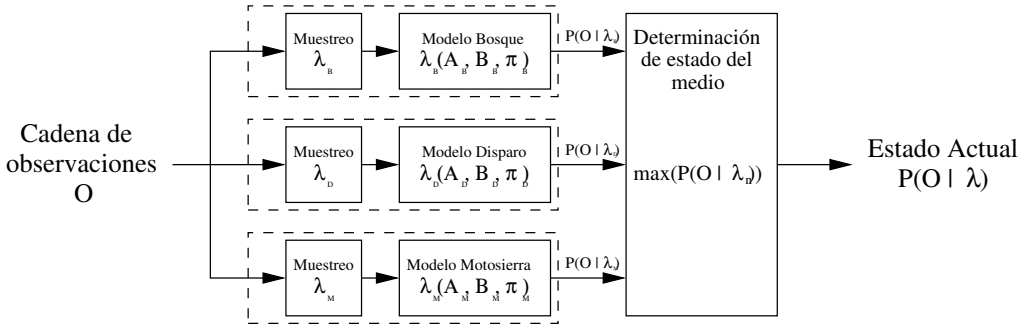


Figura 3.7: Etapa de clasificación en implementación de ejecución.

Para realizar la clasificación se utiliza el algoritmo hacia adelante, utilizando las modificaciones presentadas en [33], las cuales introducen factores de escala dentro de los cálculos. El algoritmo original fue tomado de [11], pero fue luego modificado para reducir tiempos de procesamiento y uso de memoria, así como para orientarlo al uso de objetos para facilitar administración y evaluación de los modelos. Se removieron también funciones y secciones de código no requeridas.

Con el fin de realizar la operación en línea, se realiza una adaptación a la forma en que se realiza el proceso de evaluación [38]. Se realiza el muestreo de símbolos de manera normal, hasta que se alcanza la longitud requerida por el modelo (T); una vez que se cumple esto se llama a la función de evaluación, la cual realiza los tres pasos presentados en la sección 2.7.2 para obtener $\log(P(O|\lambda))$. Luego de calcular esta salida, se "reinicia" el tiempo, procediendo a muestrear símbolos nuevamente hasta que se vuelva a contar con la cantidad de muestras requeridas para evaluar, repitiendo todos los pasos de evaluación.

Este procedimiento puede verse como la aplicación de una ventana a la cadena de observación, la cual inicia en cero y finaliza en $T-1$. Una vez evaluada esta cadena, se desplaza la ventana para iniciar en T y finalizar en $2T-1$. Se muestrea la cadena y se evalúa, repitiendo el proceso hasta llegar al final de la cadena de observación completa, o hasta que se detenga ejecución de la aplicación que opera en línea.

Este método de evaluación tiene sus desventajas. La primera de ellas corresponde al hecho

de que los HMM son sensibles a variaciones en los puntos donde se toman los símbolos que componen la cadena. Esto haría que si se muestrea la cadena unos cuantos símbolos antes o después de que se presente un evento de alarma las probabilidades de salida resultantes provoquen una clasificación incorrecta del medio, llevando a falsos positivos o a falsos negativos. Otra desventaja es el hecho de que el proceso de evaluación completo se puede repetir con mucha frecuencia, por lo que en caso de que se usen configuraciones de modelos que sean exigentes en cuanto a tiempo de procesamiento (con muchos estados y longitudes de cadena grandes), podrían llegar a presentarse violaciones al tiempo de procesamiento disponible.

Sin embargo, se elige este método de evaluación dado que es más sencillo y eficiente que otras alternativas, y dado que las limitaciones para la implementación en ASIC impiden el uso de algoritmos más complejos.

Capítulo 4

Resultados y análisis

4.1 Entrenamiento

Para realizar el entrenamiento de cada una de las etapas que componen el SiRPA, así como la evaluación de las tasas de reconocimiento y la generación de las matrices de confusión en la etapa de clasificación se cuenta con un total de 192 archivos de audio. Estos se clasifican de acuerdo a su clase, y se separan entre cadenas de entrenamiento y evaluación. La tabla 4.1 muestra la distribución de todos los archivos.

Tabla 4.1: Distribución de archivos para entrenamiento y evaluación.

Clase	Entrenamiento	Evaluación	Total
Bosque	43	38	81
Disparo	24	21	45
Motosierra	36	30	66
Total	103	89	192

La selección de los archivos usados para entrenamiento tiene efectos en las tasas de reconocimiento obtenidas, por lo que deben realizarse pruebas empíricas para mejorarlas. Factores como el muestreo y longitud de cadena son también relevantes, pues además de afectar estas tasas, ciertas combinaciones de estos valores provocarían que deban descartarse cadenas que no cuentan con la longitud suficiente para ser usadas en el entrenamiento o evaluación, reduciendo el número de ejemplos disponibles.

4.1.1 Constantes del normalizador

Las constantes utilizadas por el normalizador son dimensionadas en base a la clase de disparo, con el fin de que se pueda capturar el pico de amplitud producido por la detonación de un arma, pero que la salida del bloque no se vea saturada a un nivel de

amplitud alto por un tiempo prolongado. De no cumplirse esto la salida de la etapa de generación de símbolos podría verse saturada a uno o dos valores, lo que no permitiría reconocer el momento en que se dio la transición del estado de disparo al estado normal del medio, dando lugar a un entrenamiento inadecuado de los HMM, y por lo tanto a tasas de reconocimiento pobres.

Así, se seleccionan las constantes k_{crec} y k_{decrec} del promediador con el fin de que se alcance la amplitud máxima del pico, que en este caso es 1, en el tiempo de duración total de la detonación del disparo, y no del eco remanente. En [22] se explica que la duración de esta explosión sónica es de 3-5 ms, por lo que en esta implementación se trabaja con una duración de 5 ms. La figura 4.1 muestra el comportamiento que se busca para el promediador, donde se asume que una vez que se alcanza el pico de amplitud de entrada de 1, éste desaparece y se procede con la descarga lineal.

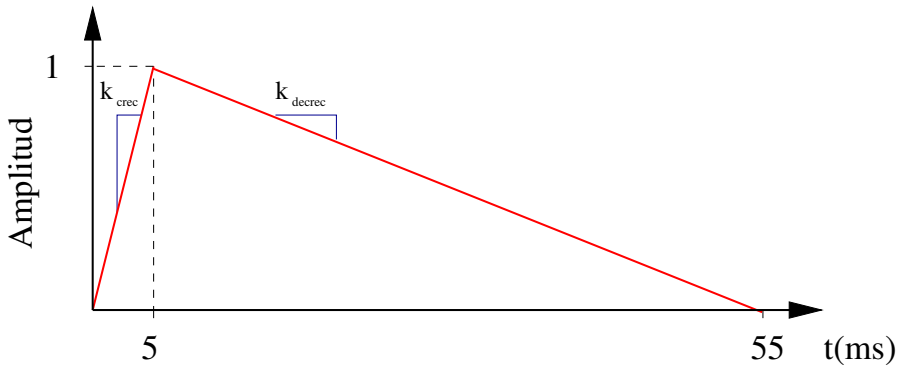


Figura 4.1: Curva usada para determinación de constantes de normalizador.

Otra constante importante para el normalizador es k_{nor}, que evita la división por cero, y también disminuye el efecto de ruido de alta frecuencia. Para su determinación se usaron archivos de audio de ejemplo y se empleó la FFT para estimar el valor del piso de ruido en ellos. La constante se selecciona para ser un poco mayor a este valor.

Los valores obtenidos para cada una de las tres constantes se resumen en la tabla 4.2. Se asume que la frecuencia de muestreo es de 44,1 kHz, que es la utilizada por los archivos de audio de entrenamiento y en la implementación en ASIC.

Tabla 4.2: Constantes utilizadas en el normalizador, con $F_S = 44,1$ kHz.

Constante	Valor
k_{crec}	0,004524887
k_{decrec}	0,000452489
k_{nor}	0,000030518

Se calculan además estas constantes para el caso donde la frecuencia de muestreo sea de 48 kHz, que es la utilizada por el sistema embebido. Estas se muestran en la tabla 4.3. En caso que se utilice el modo de reconocimiento en línea deben utilizarse estas

constantes, mientras que si se utiliza el modo de reconocimiento sobre archivos, y estos fueron muestreados a 44,1 kHz, deben utilizarse las constantes que se presentaron en la tabla 4.2.

Tabla 4.3: Constantes utilizadas en el normalizador, con $F_S = 48$ kHz.

Constante	Valor
k_{crec}	0,004166667
k_{decrec}	0,000416667
k_{nor}	0,000030518

4.1.2 Resultados para LDA

El objetivo del entrenamiento con LDA es obtener la matriz de transformación **W** y el vector de media que son utilizados por el bloque reductor de dimensiones para pasar de un espacio de entrada de 8 dimensiones a uno de 3 dimensiones, por lo que la matriz resultante es de tamaño 8 por 3.

La siguiente matriz fue obtenida mediante el entrenamiento con la totalidad de los archivos disponibles para todas las clases, y utilizando el muestreo por defecto de 1 con el fin de contar con un mayor número de muestras.

$$\mathbf{W} = \begin{bmatrix} -0.407052 & 0.247081 & -0.890227 \\ -0.337315 & 0.071749 & 0.139704 \\ -0.049092 & 0.064765 & 0.043157 \\ 0.205715 & -0.233988 & -0.165127 \\ 0.778149 & 0.022167 & -0.243426 \\ 0.018490 & -0.297322 & -0.130926 \\ 0.051397 & 0.258108 & 0.091751 \\ 0.259423 & 0.848143 & 0.272091 \end{bmatrix}$$

Esta matriz tiene asociado el siguiente vector de media en 8 dimensiones.

$$\underline{\boldsymbol{\mu}} = \begin{bmatrix} 0.648541 & 0.537859 & 0.547184 & 0.526413 & 0.494256 & 0.405472 & 0.308410 & 0.279029 \end{bmatrix}$$

4.1.3 Resultados para k-medias

El objetivo del entrenamiento con el algoritmo de k-medias es obtener la matriz de centroides en 3 dimensiones que componen el alfabeto discreto utilizado por los HMM en la etapa de clasificación. Estos componen también los diferentes nodos del árbol k-d, utilizado como parte de la cadena de procesamiento del SiRPA.

Para fines de este proyecto se utiliza un alfabeto discreto compuesto por un total de 32 centroides, pues en un trabajo previo [41] se determinó que este valor es suficiente para realizar un reconocimiento adecuado, lo que se fundamenta en la reducción de la distorsión de los datos expuesta en [33]. Así, el resultado del entrenamiento será una matriz de 32 por 3.

La matriz obtenida se muestra en la tabla 4.4, donde se ha organizado de esta manera con el fin de mostrar el número de identificación del símbolo asociado a cada uno de los centroides. El entrenamiento se realizó utilizando todos los archivos, y con un muestreo de 1 para contar con un mayor número de muestras.

Con el fin de observar de manera gráfica la distribución de los centroides en 3 dimensiones se generó la figura 4.2, donde se muestran los puntos descritos en la tabla 4.4.

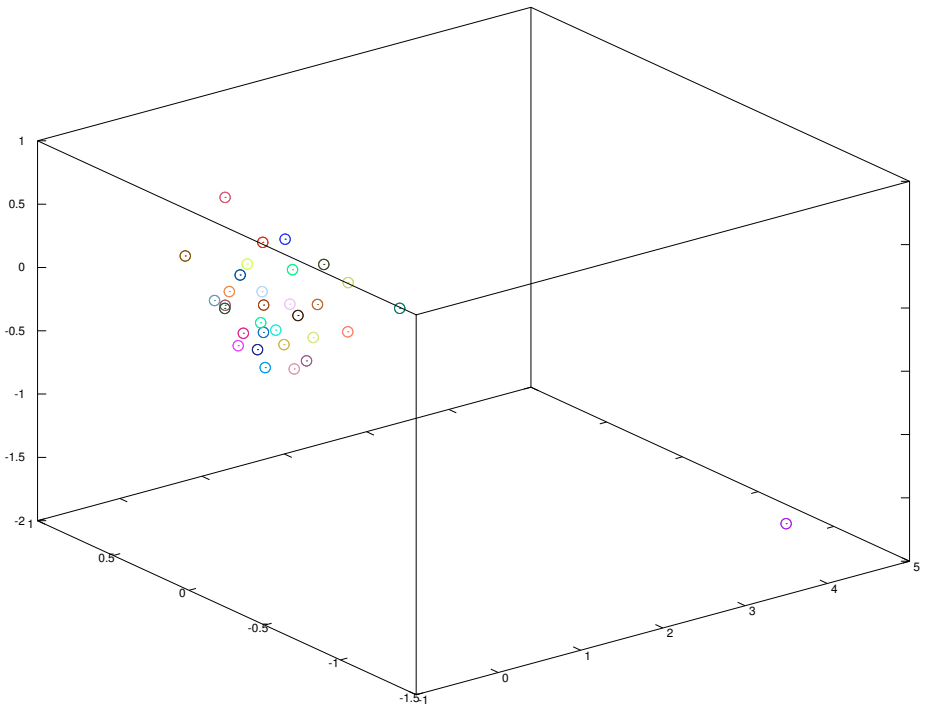


Figura 4.2: Distribución de los centroides generados en espacio de 3 dimensiones.

4.1.4 Resultados de clasificación

Se realizaron pruebas no exhaustivas (usando todas las cadenas disponibles para el entrenamiento, y variando parámetros en alrededor de 3-5 valores diferentes) con los algoritmos de entrenamiento múltiple y negativo, variando el número de estados, la longitud de cadena y el muestreo para verificar efectos sobre la tasa de reconocimiento de las cadenas.

Tabla 4.4: Centroides que componen alfabeto discreto **V**.

ID	Coordenadas (x, y, z)
0	-0.200390, 0.095361, 0.297638
1	0.128159, 0.227361, 0.253334
2	0.182770, 0.006590, 0.561523
3	-0.470512, 0.048084, 0.132072
4	-0.407849, -0.004194, -0.169165
5	0.016712, -0.143028, -0.351365
6	0.015306, 0.314037, -0.124772
7	0.318917, 0.449405, -0.119228
8	0.281760, 0.203966, -0.300115
9	0.026294, 0.104181, -0.336463
10	-0.157564, 0.097897, -0.171364
11	0.353007, 0.068058, -0.015132
12	-0.125777, -0.152615, -0.129345
13	0.088305, -0.230108, -0.067904
14	-0.277249, -0.081856, 0.031854
15	-0.211896, -0.075756, -0.337535
16	0.267882, -0.088447, -0.362031
17	0.096316, 0.021522, -0.148992
18	0.027619, 0.581800, 0.139942
19	-0.243702, 0.241544, 0.022696
20	-0.025749, 0.035382, 0.062096
21	0.139876, -0.229726, 0.183623
22	-0.095523, -0.228818, 0.137479
23	-0.327123, -0.117944, 0.305483
24	4.123076, -1.160674, -1.734850
25	0.380117, -0.299191, -0.035840
26	0.550334, 0.353983, 0.281069
27	0.684771, -0.477256, 0.193790
28	0.316508, 0.476075, 0.609859
29	0.476441, -0.088568, 0.362999
30	0.260909, -0.366663, 0.410339
31	-0.133469, -0.213828, 0.497775

Para cada una de las pruebas se obtiene la matriz de confusión, a partir de la cual se extraen los valores de sensitividad y el VPP para cada clase individual, que corresponden a los elementos c_{11}, c_{22} y c_{33} de la matriz **C**. Dado que los resultados son numerosos, se coloca la tabla con todos los valores en la sección de apéndices, mostrando en la tablas 4.5 y 4.6 solamente un resumen de los mejores valores de reconocimiento obtenidos en cada entrenamiento. Además, tal como se mencionó previamente, conforme se incrementa el

valor del muestreo y/o la longitud de cadena, el número de archivos útiles para entrenar y evaluar disminuye, y de ahí que en algunos casos se indique con N.A. que no existe una cadena para evaluar esta clase. Si un valor es cero, quiere decir que no se clasificó ninguna cadena de una clase determinada correctamente.

Tabla 4.5: Tasas de reconocimiento obtenidas mediante entrenamiento múltiple.

			Sensitividad (%)			VPP (%)		
Estados	Muestreo	Long. Cadena	Bosque	Disparo	Motosierra	Bosque	Disparo	Motosierra
3	10	25	89,47	90	86,67	89,47	100	83,87
3	10	10	57,89	**94,44**	73,33	73,33	**94,44**	57,89
10	10	50	94,74	0	**93,33**	94,74	0	**87,5**
15	20	25	**97,37**	0	60	**74**	0	90

Tabla 4.6: Tasas de reconocimiento obtenidas mediante entrenamiento negativo.

			Sensitividad (%)			VPP (%)		
Estados	Muestreo	Long. Cadena	Bosque	Disparo	Motosierra	Bosque	Disparo	Motosierra
3	1	10	44,74	**4,76**	0	30,36	**3,23**	0
10	1	5	**97,37**	0	6,67	**46,84**	0	50
10	20	50	18,42	N.A.	**58,62**	36,84	N.A.	**62,96**

En la tabla 4.5 se observa que los resultados con entrenamiento múltiple dan valores de sensitividad y VPP por encima del 80% para las tres clases, aún cuando se utilizan pocos estados. Además, es posible obtener valores de sensitividad por encima del 93% para cada clase de manera individual, al variar los parámetros del modelo. Si se realizan pruebas exhaustivas es posible que se obtengan valores aún mayores a estos.

A partir de los resultados de la tabla 4.6 podría pensarse que el entrenamiento negativo tiene un rendimiento inferior, pero este no es el caso. Tal como se explica en [7], el entrenamiento negativo requiere de la utilización de cadenas de observación con longitudes cortas, pues de otra forma el entrenamiento agrega ruido a los modelos y se disminuyen las tasas de reconocimiento. Además, la selección de las cadenas de observación utilizadas para entrenar es más delicada que con el entrenamiento múltiple, y se requiere evaluar los resultados y agregar o quitar cadenas que los afecten, hasta obtener tasas satisfactorias, lo que aunado a la variación que debe realizarse de los otros parámetros de los modelos hace el proceso más trabajoso.

En la figura 4.3 se muestra un ejemplo de diagrama de transición de estados para las 3 clases involucradas, utilizando solamente 3 estados. Estos fueron obtenidos mediante entrenamiento múltiple, y tal como se presentó en la tabla 4.5, ofrece tasas de reconocimiento aceptables (mayores al 80% en las tres clases) con pocos estados, lo que en la implementación de ejecución es importante, ya que el tiempo de procesamiento aumenta conforme se incrementa el número de estados y la longitud de las cadenas de observación.

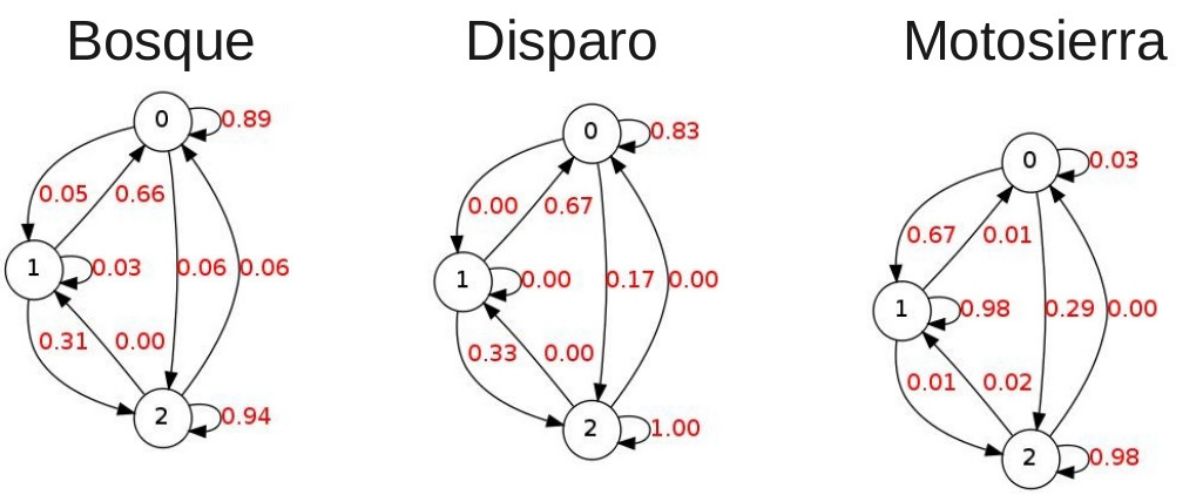


Figura 4.3: Ejemplos de HMM de 3 estados, entrenados con múltiples cadenas.

El número de estados, el muestreo y la longitud de las cadenas de observación no tienen que ser necesariamente igual entre los modelos, por lo que es posible utilizar combinaciones de ellos con el fin de maximizar las tasas de reconocimiento. Para ello, las tablas en los apéndices pueden dar una guía sencilla de los efectos de cada uno de estos valores sobre un modelo en específico. Sólo es importante evitar utilizar valores que provoquen tiempos de procesamiento excesivo, que harían que la implementación de ejecución llegue a fallar en algunos casos.

4.2 Ejecución

Las pruebas realizadas en el sistema embebido, para verificar la implementación de ejecución, han sido realizadas conforme era desarrollada y depurada la aplicación, dado que en un inicio se tuvieron errores estructurales que obligaron a reestructurar algunas etapas de procesamiento. Se hará mención específica de esto en los casos que corresponda.

Se presentan resultados para observar el comportamiento de diferentes secciones del sistema, así como para verificación de la variación de los valores obtenidos entre la implementación de ejecución y entrenamiento, que se deben entre otros factores al procesamiento realizado en formato punto fijo en una de las etapas del sistema.

4.2.1 Resultados gráficos por etapa

En las siguientes figuras se presenta de manera gŕafica el efecto de algunas de las etapas del SiRPA, con el fin de observar el efecto del normalizador, estimación de energía por banda y símbolos generados. Se presentan los resultados para los primeros 100 ms de audio para un disparo de un revólver calibre 32 a una distancia de 30 m, con angulación de 90° respecto al dispositivo de captura. Se toman las primeras 4411 muestras de audio,

que producen en total 36 símbolos discretos. En la figura 4.4 se muestra la señal de entrada tal a como se lee del archivo (en la parte superior) y la señal resultante luego de la normalización (en la parte inferior).

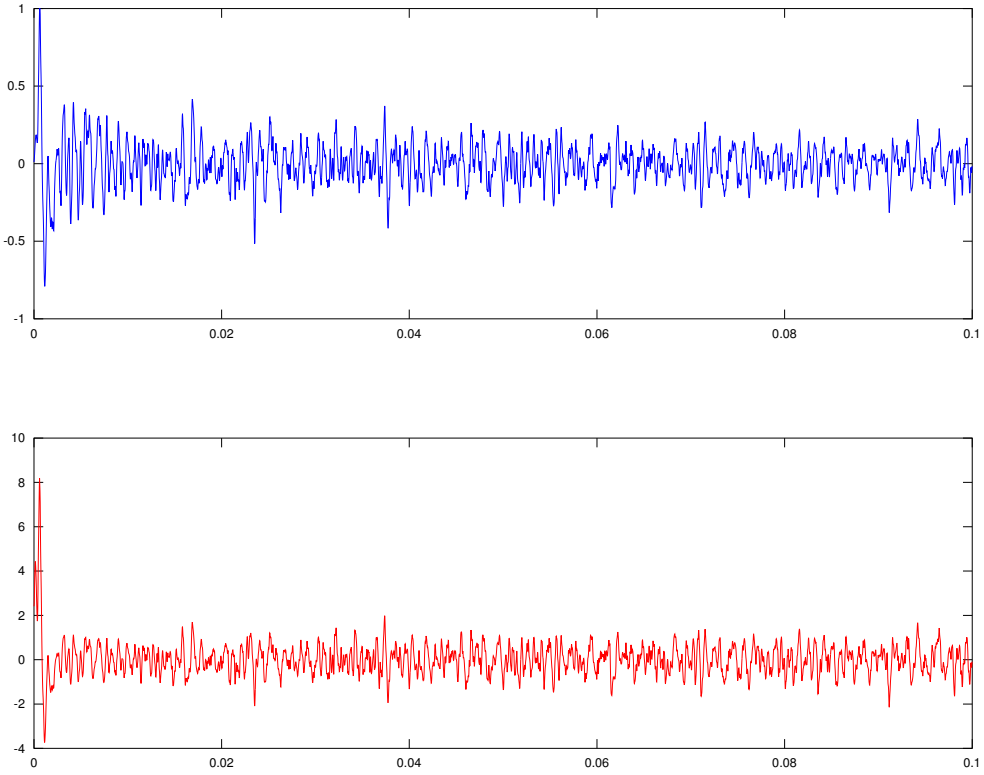


Figura 4.4: Señal de entrada (arriba) y señal normalizada resultante (abajo). [1]

Puede observarse la presencia de picos de gran amplitud (cercanos a ±1) en los primeros 5 ms del gráfico de la entrada, que corresponde a la detonación del arma. Estos picos generan que la salida del normalizador se dispare, alcanzando en este caso valores 8 veces mayores a la máxima amplitud de la entrada. Posteriormente, la amplitud de la entrada baja, procediendo el normalizador a seguirla pero con una amplitud mayor. Una vez que desaparecen estos picos, la señal de entrada disminuye y se mantiene en niveles menores a ±0,5. Esto hace que la salida del normalizador se mantenga en niveles cercanos a ±1, que serían los límites esperados cuando el medio se encuentra en un estado normal.

La presencia de los picos de gran amplitud a la salida del normalizador sería el comportamiento esperado cuando el medio se encuentra en un estado normal y se presenta súbitamente el disparo de un arma de fuego. Dado que el valor promedio en el estado normal sería aproximadamente constante, y de amplitud baja en comparación al pico para la detonación, la salida del normalizador se eleva y se mantiene en un valor alto hasta que el promediador reduzca su ganancia y por lo tanto la amplitud de salida.

[1] Se analiza una señal acústica para un disparo de un revólver calibre 32 a 30 m de distancia.

La señal normalizada es luego pasada al banco de filtros, donde la figura 4.5 muestra las salidas obtenidas para los estimadores de energía que lo componen.

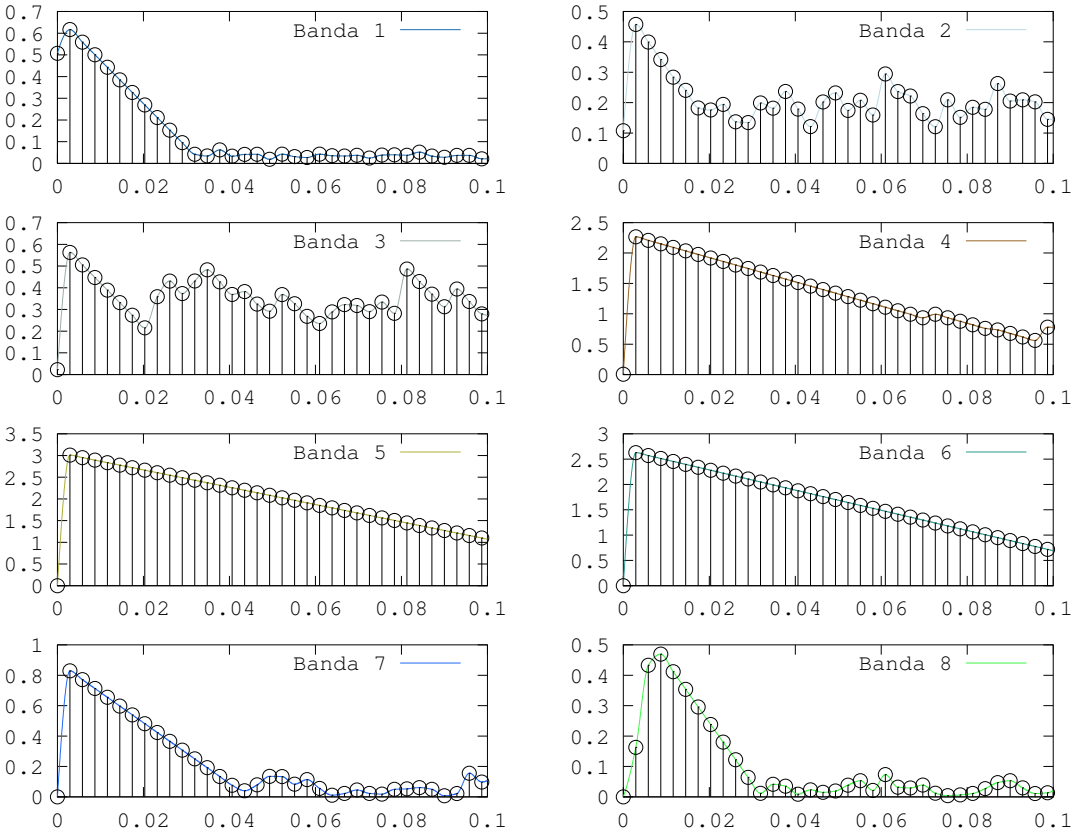


Figura 4.5: Salida del banco de filtros con estimación de energía por banda.

En esta figura puede notarse cómo, para este caso, las bandas 4, 5 y 6 son las que presentan los mayores niveles de energía, alcanzando valores máximos en las primeras muestras y procediendo luego a descargarse de manera lineal, lo que corresponde al seguimiento de la envolvente. En cada gráfica se indican los puntos de muestreo en color negro, y superpuesta se tiene la señal continua interpolada a partir de estas muestras, utilizando interpolación *cubic* o *pchip*.

Los vectores de ocho dimensiones de los estimadores de energía son luego pasados por el reductor de dimensiones, y el vector de tres dimensiones resultante se utiliza para realizar la búsqueda en el árbol k-d. Esto permite determinar el centroide que se encuentra más cercano a este vector, y por lo tanto la salida del árbol sería el identificador para ese centroide, que corresponde también al símbolo de observación usado como entrada del módulo de clasificación. El tren de símbolos generado por los vectores mostrados en la figura 4.5 se muestra en la figura 4.6.

Estos símbolos serían los que se pasan posteriormente a la etapa de clasificación para

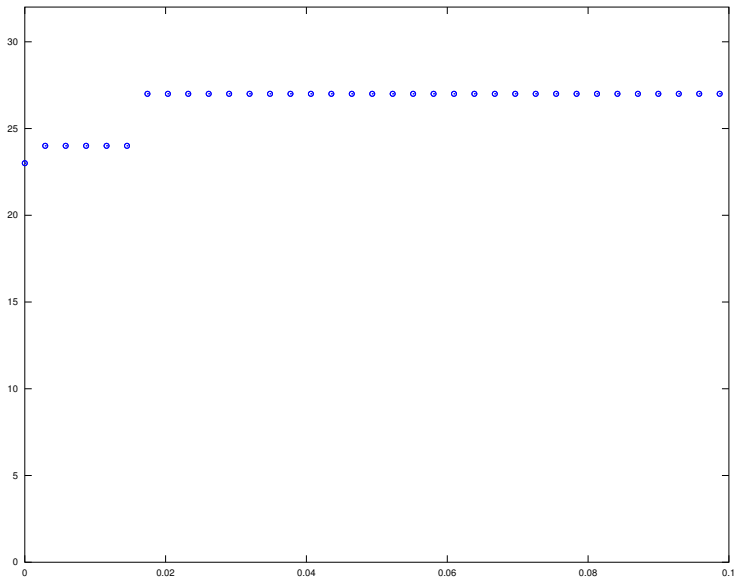


Figura 4.6: Tren de símbolos durante ventana de tiempo elegida.

realizar la evaluación, y podría aplicarse un muestreo adicional sobre ellos dependiendo de la configuración de los modelos. La distribución de los símbolos es similar para los otros ejemplos de disparos con los que se cuenta, pero varía de acuerdo a la distancia y la inclinación con respecto al dispositivo de captura [22].

4.2.2 Resultados gráficos para diferentes clases

Con el fin de observar la separación de los datos de diferentes clases que se obtiene luego de realizar la reducción de dimensiones, se presenta la figura 4.7. En esta se muestran puntos en tres dimensiones para las clases de disparo, motosierra y bosque, donde se utilizó un archivo de audio de ejemplo para cada una.

Puede notarse que los datos en tres dimensiones se separan espacialmente para vectores de diferentes clases, mientras que los vectores de la misma clase se encuentran agrupados. Este es justamente el efecto buscado, pues de esta forma las cadenas de observaciones generadas para cada clase facilitan la clasificación del patrón.

En la figura 4.7 se observa que la clase de disparo es la que presenta mayor variación en sus valores, lo cual se debe a que sus vectores de salida en ocho dimensiones del banco de filtros presentan altos valores de energía en diferentes bandas, tal como se observa en la figura 4.5. Conforme la salida de los estimadores de energía se descarga, los datos comienzan a "desplazarse" hacia la región de datos del bosque, que es el efecto buscado y para el cual se dimensionaron las constantes mostradas en la sección 4.1.1. De esta forma se captura el sonido de la detonación del disparo, se extraen sus características y se vuelve nuevamente a un estado normal del bosque.

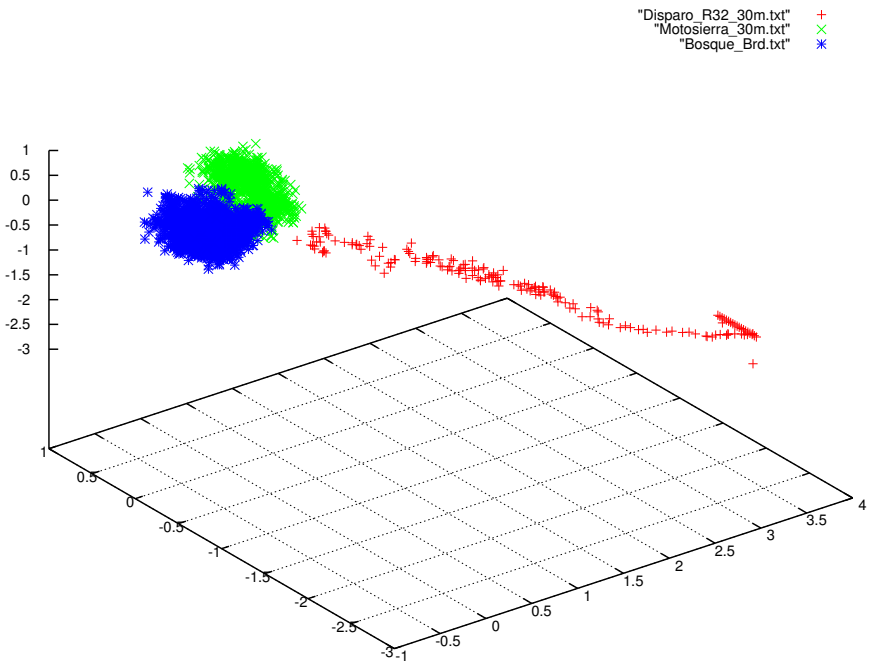


Figura 4.7: Datos en tres dimensiones para las diferentes clases.

La separación presente entre los datos de la clase de disparo con respecto a las otras dos es lo que permite que en las matrices de confusión esta clase muestre valores de sensitividad por encima del 90%, y un VPP de inclusive el 100%. Sin embargo, conforme los sonidos capturados se ubican a una mayor distancia, los niveles de energía obtenidos en cada banda son más bajos y se da un traslape de los vectores de diferentes clases.

Este efecto es más notable entre las clases de bosque y motosierra, pues si bien en la figura 4.7 se encuentran debidamente separadas, la distancia entre los datos de ambas clases no es tan notable como para la clase de disparo, y conforme los sonidos se ubiquen más lejos del dispositivo de captura comienzan a traslaparse los datos entre ambas. Para ilustrar esto, se muestra en la figura 4.8 los datos obtenidos para sonidos de las clases de disparo y motosierra a 600 m de distancia.

Finalmente, se muestra en la figura 4.9 un ejemplo de tren de símbolos para las tres diferentes clases. Estos son los símbolos que componen las cadenas de observaciones que se pasan posteriormente al módulo de clasificación. Se muestran los primeros 100 símbolos (equivalente a aproximadamente 300 ms de audio) generados a partir de los datos que se mostraron en la figura 4.7, y utilizando los mismos colores para cada clase que se utilizaron en ella.

En esta figura puede notarse que el tren de símbolos para la clase de disparo (en rojo) es repetitivo, lo cual se debe a que sus datos, según se muestran en la figura 4.7 se ubican en su mayoría cerca de los centroides con el identificador 24 y 27, mostrados en la tabla 4.4. Los trenes para las clases de bosque y motosierra cambian repetidamente de valores,

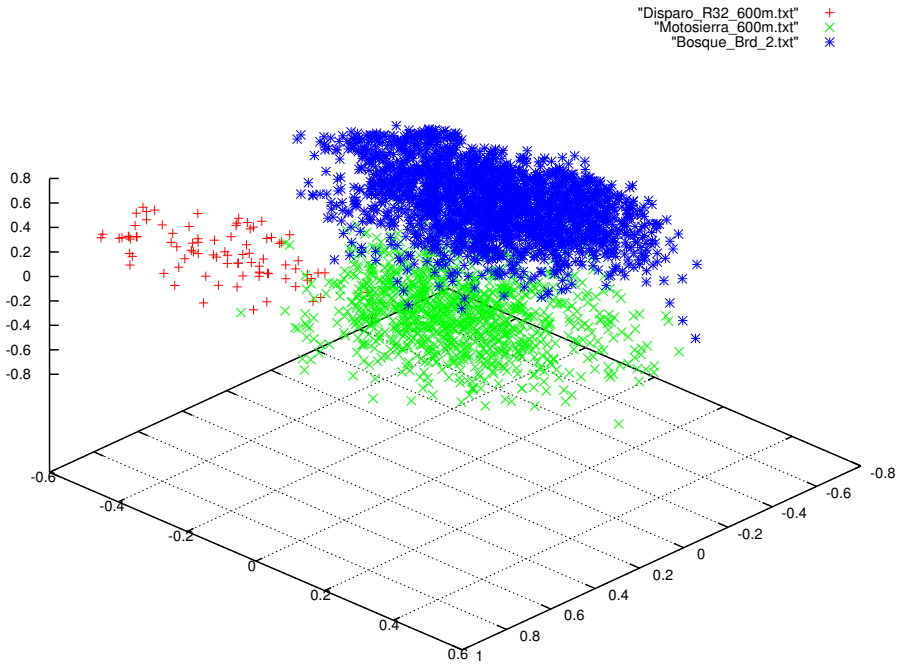


Figura 4.8: Datos en tres dimensiones con sonidos a mayor distancia.

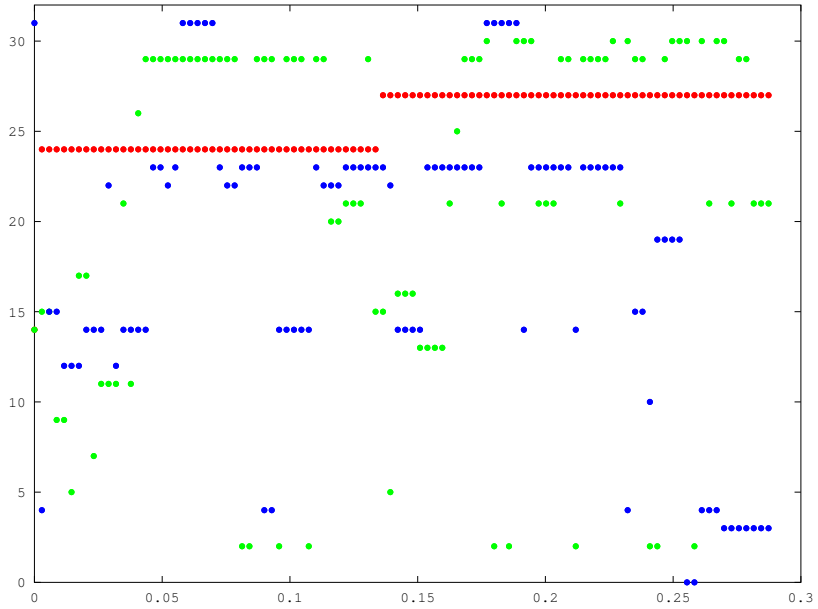


Figura 4.9: Ejemplo de tren de símbolos para las tres clases.

pues tal como se observa en la figura 4.7 sus datos se concentran en una región del espacio tridimensional donde se tienen múltiples centroides cerca, por lo que la búsqueda del vecino más cercano no siempre da el mismo resultado.

A partir de esto es posible notar que se requiere de un mayor número de centroides para describir las clases de bosque y motosierra, dado que los vectores para estas se ubican en

regiones concentradas del espacio, y de ahí que 31 de los 32 centroides mostrados en la tabla 4.4 se ubiquen dentro de un cubo delimitado en todas las dimensiones por -1 y 1, con sólo un centroide fuera de esa región. Este último describe por lo general salidas de alta energía del banco de filtros, tal como son los sonidos de disparos.

4.2.3 Resultado de ordenamiento del árbol k-d

Tal como se mencionó en un capítulo previo, el árbol k-d de la implementación de entrenamiento previa se encuentra no balanceado. Dado que la aplicación para el sistema embebido opera en línea, el retraso adicional que se genera por esto es un factor crítico, y de ahí la inclusión de algoritmos para balancearlo. La construcción, tal como se mencionó en la sección 2.6.1, se hace en base a la división del espacio en la dimensión de mayor varianza, para que los algoritmos de búsqueda reduzcan así el número de comparaciones requeridas para obtener el símbolo de salida.

El resultado de esto se muestra de manera gráfica en la figura 4.10, donde es posible observar que el árbol se encuentra balanceado con un total de 6 niveles. El número en cada nodo corresponde al identificador, o símbolo, que le fue asignado en la tabla 4.4.

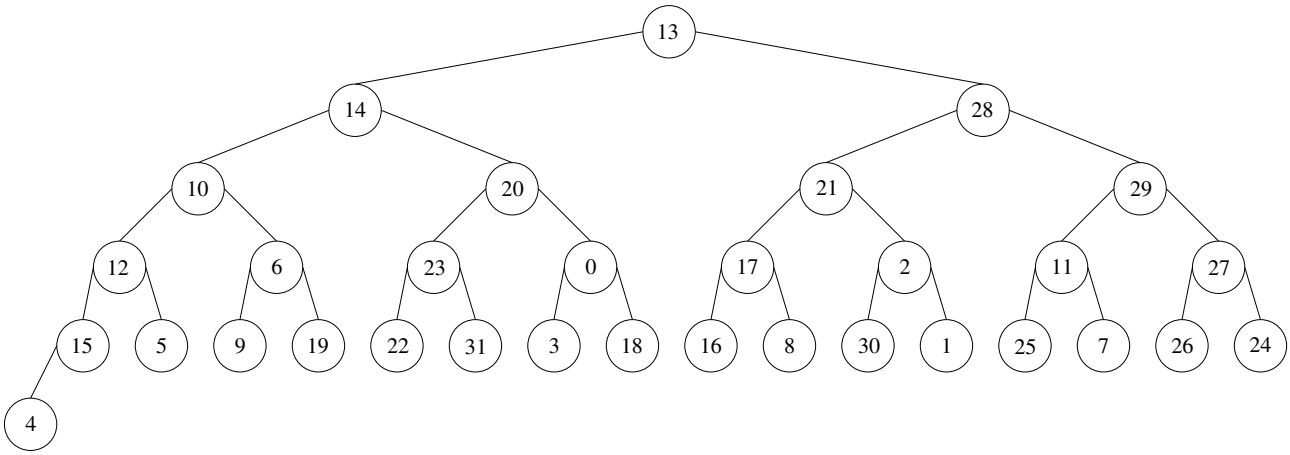


Figura 4.10: Ordenamiento resultante de balancear el árbol k-d.

4.2.4 Resultados de clasificación

Con el fin de evaluar los resultados de la etapa de clasificación, se exportaron dos juegos de modelos diferentes, obtenidos ambos utilizando entrenamiento múltiple. El primero consiste en modelos de 3 estados, muestreo de 10 y longitud de cadena de 25, mientras que el segundo son modelos de 10 estados, muestreo de 5 y longitud de cadena de 50. Se eligieron estos 2 juegos con el fin de evaluar tanto los efectos de la variación de los parámetros mencionados sobre el reconocimiento, como para determinar la variación en el tiempo de procesamiento requerido para obtener la salida en cada caso, pues los modelos del segundo juego requieren considerablemente más cálculos para realizar la evaluación.

Se utilizan las misma cadenas de evaluación que para la implementación de entrenamiento, y en base a los resultados de clasificación se construyen las matrices de confusión, cuyos resultados se organizan en la tabla 4.7.

Tabla 4.7: Tasas de reconocimiento obtenidas mediante con sistema embebido.

			Sensitividad (%)			VPP (%)		
Estados	Muestreo	Long. Cadena	Bosque	Disparo	Motosierra	Bosque	Disparo	Motosierra
3	10	25	81,58	90	83,33	86,11	100	75,76
10	5	50	86,84	80	83,33	84,62	100	80,65

4.2.5 Tiempos de ejecución por bloque

Se llevaron a cabo mediciones del tiempo de ejecución requerido para procesar cada bloque de señal, así como el tiempo que toma en promedio cada etapa para realizar sus operaciones. Dado que uno de los objetivos principales de la implementación en el sistema embebido es reducir estos tiempos para realizar la operación en línea de manera adecuada, varios de los bloques y sus funciones asociadas han sido optimizadas, verificando siempre que los resultados no se vean afectados.

Para fines de comparación se tienen tres aplicaciones diferentes; dos de ellas realizan el procesamiento utilizando solamente el procesador ARM, pero una es compilada utilizando gcc [31] y el entorno de desarrollo Eclipse [14], mientras que la otra se compila con un Makefile y herramientas propias de Texas Instruments. La tercera utiliza la combinación del procesador ARM y el DSP, ejecutando en este último todo lo relativo al banco de filtros y estimación de energía en cada banda. Se utilizan los mismos modelos de la subsección previa para verificar cambio en tiempos de procesamiento para cada uno, así como un tamaño de bloque de 256 muestras para estimar tiempos de procesamiento que se presentarán en la operación en línea.

La implementación inicial del sistema contenía errores estructurales heredados de la aplicación para entrenamiento, que fue desarrollada originalmente en un proyecto previo [7]. Estos errores fueron detallados en el capítulo anterior. Sin embargo, las mediciones de tiempo realizadas se llevaron a cabo conforme se corregían y optimizaban diferentes partes del sistema, lo que permite observar la mejora en los tiempos de procesamiento conforme se finalizaba la aplicación.

En la tabla 4.8 se muestran los resultados de tiempo de procesamiento promedio obtenidos para cada una de las aplicaciones previamente mencionadas. En cada nueva fila se especifica cuál era la condición de la implementación del sistema, para verificar así el efecto de las correcciones y optimizaciones realizadas. En orden se tendrían las siguientes condiciones: *Inicial*, donde se tenían las tasas de muestreo incorrectas para el banco de filtros, y no se tomaba la salida en los estimadores de energía. *Corrección 1* donde se atacó este problema y a su vez se llevó a cabo la implementación de los algoritmos de filtra-

do en el DSP. *Corrección 2* donde se obtuvo un árbol k-d balanceado, y por último la implementación *Final*, donde se realizaron últimas modificaciones al código en diferentes bloques para mejorar los tiempos de procesamiento.

Tabla 4.8: Tiempos de procesamiento para aplicaciones en sistema embebido.

	Tiempo por tipo de aplicación (ms)		
Estado de implementación	ARM (GCC)	ARM (TI)	ARM + DSP
Inicial	4,4768	1,5595	1,7912
Corrección 1	1,2771	0,4274	0,5492
Corrección 2	1,0997	0,3142	0,4576
Final	0,8851	0,3051	0,3660

Al realizar mediciones de tiempo etapa por etapa, se observa que las implementaciones compiladas con herramientas de TI tienen tiempos similares en todas ellas excepto en el banco de filtros, que toma en promedio 80 μs más para obtener la salida. Este tiempo viene asociado al *overhead* generado al llamar a la función de filtrado que se ejecuta en el DSP, ya que debe transferirse el control de los datos a éste y esperar posteriormente los resultados. Esto, junto al tiempo que toma realizar la conversión de memoria virtual a física y viceversa, provoca que el tiempo de procesamiento se vea afectado. Si el tamaño de bloque utilizado fuera mayor, estos efectos podrían verse mitigados, al realizar menos llamadas a esta función.

Para la medición del tiempo de procesamiento de la etapa de clasificación se utilizó la aplicación ARM compilada con las herramientas de TI. En el caso del modelo de 3 estados, el tiempo de procesamiento para el bloque donde se alcanza el número de muestras requeridas para realizar la evaluación es de 549 μs, por lo que tomando como referencia el tiempo de procesamiento de bloque de 305 μs indica que la evaluación para los 3 modelos toma alrededor de 250 μs. Por otro lado, para el modelo de 10 estados se obtiene un tiempo de 1,312 ms, por lo que en este caso la evaluación toma un poco más de 1 ms, lo que es de esperar dado que con este modelo se tienen matrices más grandes, así como una cadena de observación el doble de larga.

Estos resultados muestran que el prototipo de software del SiRPA implementado en el sistema embebido permite realizar pruebas con modelos de un tamaño considerable, manteniéndose por debajo del tiempo límite para el procesamiento en línea que es de aproximadamente 5,3 ms. Esto permite estudiar sin problema las tasas de reconocimiento para modelos de gran tamaño. Solamente debe garantizarse que no se exceda el tiempo indicado, pues conllevaría a que no se atiendan las respuestas a tiempo en el cliente de JACK, provocando que la aplicación falle.

Capítulo 5

Conclusiones y recomendaciones

5.1 Conclusiones

Se logró la integración de todas las fases de procesamiento del SiRPA, tanto para la cadena de entrenamiento como para la de ejecución (o reconocimiento). Se estableció también el formato para pasar los datos resultantes del entrenamiento a la cadena de reconocimiento en el sistema embebido. Además, la integración permite utilizar este último para procesar cadenas de observaciones en tiempo real, mientras que la implementación en la PC permite realizar "procesamiento por lotes", para la generación de estadísticas.

La aplicación de entrenamiento desarrollada permitió obtener de manera correcta la matriz de transformación y el vector de media para el reductor de dimensiones. Además, permitió el cálculo adecuado de los 32 centroides que describen el alfabeto discreto para los HMM. Todos estos valores pudieron ser además exportados, cargados y utilizados por el sistema embebido para realizar la generación de símbolos, dando por resultados los mismos trenes entre ambas implementaciones.

Los mejores valores se sensitividad obtenidos con entrenamiento múltiple fueron de 89,47%, 90% y 86,67%, para las clases de bosque, disparo y motosierra, respectivamente. Estas se obtuvieron con modelos de 3 estados, muestreo de 10 y longitud de cadena de 25. Sin embargo, debe notarse que podrían existir valores de estos parámetros que lleven a tasas de reconocimiento mayores, para lo que es necesario hacer un mayor número de pruebas.

Los parámetros de número de estados, muestreo y longitud de cadena afectan de diferente manera las tasas de reconocimiento de los modelos. Las tasas suelen mejorar para todas las clases conforme se incrementa la longitud de la cadena, hasta que se alcanza un punto máximo alrededor de longitudes de 15 o 20 símbolos, a partir de donde un incremento beneficia el reconocimiento para clases como la de bosque, y en ocasiones la de motosierra, pero suele afectar negativamente a la de disparo.

El muestreo también influye sobre las tasas de reconocimiento, pues conforme se incrementa su valor se mejoran estas tasas para las tres clases, hasta un punto máximo alrededor

de valores de muestreo de 5 o 10. Luego de este último valor las tasas de reconocimiento disminuyen para las tres clases. Esto es de esperar dado que con valores superiores a 10 se descartan símbolos que aportan información característica de los patrones, además de que se reduce el número de cadenas útiles para entrenar y evaluar dependiendo también de la longitud de cadena usada. Así, en futuros entrenamientos el valor de muestreo debe ser igual o menor a 10.

Las tasas de reconocimiento para las tres clases se incrementan conforme se aumenta el número de estados, hasta alcanzar valores máximos para modelos con 10 estados, aunque esto depende también de los valores del muestreo y longitud de cadena usados. Para modelos con 15 estados las tasas de reconocimiento para las clases de bosque y motosierra pueden disminuir o incrementarse dependiendo también de estos otros valores. Por el contrario, la clase de disparo en general disminuye sus tasas de reconocimiento al pasar de modelos de 10 estados a los de 15 estados. Sin embargo, y tal como se indicó previamente, es posible obtener tasas de reconocimiento altas con modelos de apenas 3 estados, lo que en la implementación de hardware es beneficioso al reducir el tiempo necesario para hacer la clasificación a cerca de un cuarto del tiempo requerido para hacerlo con modelos de 10 estados.

El entrenamiento negativo dio tasas de reconocimiento bajas, pero los procedimientos de entrenamiento y evaluación realizados no fueron exhaustivos. Este entrenamiento requiere de una selección delicada de las cadenas de observación utilizadas y de los parámetros de longitud de cadena, muestreo, número de estados y razón de aprendizaje. Si todos estos no se seleccionan con cuidado se obtendrán tasas como las mostradas, por lo que los resultados obtenidos en este caso no son concluyentes respecto a la efectividad de esta variante de entrenamiento.

La exportación de los modelos pudo ser realizada correctamente, dando por resultado las mismas probabilidades al realizar la evaluación en el sistema embebido con respecto a la aplicación de entrenamiento. Los modelos pudieron ser cargados por el sistema embebido sin ningún tipo de error, y sin importar el número de estados, longitud de cadena y muestreo elegidos. No existen además limitaciones en cuanto al número de modelos que se pueden cargar desde un mismo archivo. Sin embargo, lo ideal es exportar sólo uno de cada clase.

Los tiempos de procesamiento obtenidos en la implementación final del sistema son de apenas 305 μs y 366 μs, al utilizar sólo el procesador ARM o la combinación ARM + DSP, respectivamente. Este es el tiempo que toma procesar un bloque completo de 256 muestras hasta la salida de la etapa de generación de símbolos. Para la etapa de clasificación el tiempo de procesamiento varía de acuerdo a los parámetros elegidos para cada HMM, donde se obtuvieron tiempos de 250 μs para evaluar modelos de 3 estados con longitud de cadena de 25, y 1 ms al evaluar modelos de 10 estados y longitud de cadena de 50.

Las tasas de reconocimiento obtenidas por el sistema embebido son las mismas que para la implementación en el computador de propósito general, donde las diferencias surgen a partir de que la primera permite evaluar la totalidad de la cadena de observación generada

a partir de los archivos de ejemplo al desplazar una ventana sobre las muestras, mientras que la última sólo considera las primeras T muestras adquiridas. Esto provoca que al evaluar en una ventana diferente a la inicial exista la posibilidad de que se den cambios en la clasificación en algunos casos.

5.2 Recomendaciones

Para el entrenamiento es crítica la selección adecuada de los parámetros del número de estados, muestreo y longitud de cadena, además de los archivos de audio utilizados para realizar la generación de símbolos. Todos estos factores influyen sobre las tasas de reconocimiento, por lo que deben realizarse pruebas exhaustivas hasta determinar aquellos valores que maximizan las tasas. Un procedimiento de prueba automatizado ayudaría con esta tarea.

Deben realizarse nuevamente pruebas con todas las variantes de entrenamiento, prestando especial atención a la de entrenamiento negativo, dado que en este trabajo no se obtuvieron tasas satisfactorias mientras que en [7] este tipo de entrenamiento resultó en tasas de reconocimiento altas, alcanzando inclusive un 100% de clasificación correcta. Por ello, deben seleccionarse con cuidado los parámetros y archivos usados para entrenar, hasta mejorar estas tasas.

El número de archivos de audio disponibles para entrenamiento y evaluación se ve reducido conforme se incrementa el muestreo y/o la longitud de cadena utilizada, dado que el tiempo de duración para cada uno de ellos es diferente. Esto provoca que estas cadenas deban ser descartadas al no cumplir con los parámetros introducidos, lo que afecta directamente las tasas de reconocimiento. Este problema es más notorio en los archivos para la clase de disparos, y en especial para aquellos con sonidos a distancias mayores a los 250m.

Por lo anterior, es necesario realizar una extensión de la duración temporal de los archivos, para lo que debe evaluarse la mejor alternativa para hacerlo sin afectar los resultados del entrenamiento. Una manera sería introducir sonidos del bosque al final, pero debe verificarse que hayan sido grabados a una distancia similar a aquella de dónde se realizó la grabación original. Otras alternativas serían introducir silencios, o realizar interpolación en las muestras. Sería bueno, además, realizar grabaciones adicionales para todas las clases, pero especialmente para las de disparo y motosierra, con el fin de contar con un mayor número de cadenas para entrenamiento y evaluación.

Un punto importante corresponde al hecho de que el servicio utilizado para la captura de muestras en el sistema embebido (JACK) opera a una frecuencia de 48 kHz, por lo que si se utilizan los archivos disponibles actualmente para realizar el entrenamiento se tendrán discrepancias en los resultados de la evaluación en el modo de operación en línea. Esto se debe a que las bandas de frecuencia en que se separa la señal abarcarían frecuencias diferentes que cuando se trabaja a 44,1 kHz (son alrededor de 9% más extensas), por lo

que la estimación de energía es diferente. Para solucionar esto deben remuestrearse los archivos disponibles a la nueva frecuencia, y realizar el entrenamiento en base a ellos. De esta forma los resultados sí concordarían en ambos casos.

Se recomienda realizar una revisión de las constantes para el bloque promediador y para el normalizador, pues estas son críticas para la estimación de energía en cada banda de frecuencia, y podrían existir valores diferentes a los seleccionados que permitirían mejorar las tasas de reconocimiento. Además, deben realizarse pruebas exhaustivas al entrenar con LDA y el algoritmo de k-medias, pues los valores determinados con cada uno de ellos influencian también la efectividad del reconocimiento realizado. Esto es un proceso iterativo lento, pues cada vez que se modifica una etapa, debe repetirse el entrenamiento de todas las que le siguen.

Finalmente, debe revisarse un error presente actualmente en la aplicación de entrenamiento, y que obliga a realizar el entrenamiento con LDA desde otra aplicación. El problema consiste en que los resultados generados a partir del entrenamiento con LDA, al ejecutarse este algoritmo desde la aplicación principal, genera resultados inconsistentes, que varían aún cuando se utilicen los mismos archivos para realizar el entrenamiento. Esto obliga a realizar este procedimiento de manera externa y a importar luego los resultados.

Este problema parece estar asociado a uno de los componentes de la biblioteca en que se basa el algoritmo de LDA. Se trata de la biblioteca lapack, que es utilizada para realizar operaciones de álgebra lineal, y se encuentra desarrollada en el lenguaje Fortran. Ésta es también utilizada por otra de las librerías bibliotecas en la aplicación de entrenamiento, llamada ghmm, que es la encargada de las operaciones relacionadas con el entrenamiento y evaluación de los HMM. Por características de la forma en que se implementa lapack, su uso desde estas dos secciones diferentes podría provocar interferencia mutua, dando paso a las variaciones mencionadas en los resultados.

Bibliografía

[1] L. Abrahams. Implementación de una metodología para lograr la integración física correcta por construcción (CBC) del Sistema de Reconocimiento de Patrones Acústicos (SiRPA). Tesis de Licenciatura, Escuela de Ingeniería Electrónica, ITCR, Enero 2012.

[2] J. P. Alegre, S. C. Pueyo, and B. C. López. *Automatic Gain Control Techniques and Architectures for RF Receivers*. Springer, 2011.

[3] P. Alvarado. Señales y Sistemas. Fundamentos Matemáticos, 2010. URL http://www.ie.itcr.ac.cr/palvarado/pmwiki/index.php/PabloAlvarado/ModelosDeSistemas?action=downloadman&upname=senales_y_sistemas.pdf.

[4] P. Alvarado, N. Hernández, M. Hernández, and J. Mora. D2ARS: Diseño y desarrollo de aplicaciones basadas en redes de sensores. *Escuela de Ingeniería Electrónica, ITCR*, Junio 2010.

[5] J. Axelson. Using eclipse to cross-compile applications for embedded systems [online]. 2011 [visitado el 3 de abril de 2013]. URL http://www.lvr.com/eclipse1.htm.

[6] beagleboard.org. BeagleBoard-xM system reference manual [online]. 2010 [visitado el 3 de abril de 2013]. URL http://beagleboard.org/static/BBxMSRM_latest.pdf.

[7] J. Cárdenas. Estrategias de entrenamiento de modelos ocultos de markov para reconocimiento de patrones acústicos. Tesis de Maestría, Escuela de Ingeniería en Computación, ITCR, Junio 2012.

[8] Safety Dynamics. The SENTRI solution: A new age in law enforcement [online]. 2013 [visitado el 3 de abril de 2013]. URL http://www.safetydynamics.net/prods.html.

[9] A. Eleyan and H. Demirel. PCA and LDA based neural networks for human face recognition. *InTech*, 2007.

[10] A. Schliep et. al. GHMM library [online]. 2013 [visitado el 3 de abril de 2013]. URL http://ghmm.org/.

[11] C. Cannam et. al. QM-DSP [online]. 2013 [visitado el 3 de abril de 2013]. URL http://code.soundsoftware.ac.uk/projects/qm-dsp/repository/entry/hmm/hmm.c.

[12] P. Alvarado et. al. LTI-Lib [online]. 2012 [visitado el 3 de abril de 2013]. URL http://ltilib.sourceforge.net/doc/homepage/index.shtml.

[13] P. Davis et. al. JACK documentation [online]. 2013 [visitado el 3 de abril de 2013]. URL http://jackaudio.org/documentation.

[14] The Eclipse Foundation. Eclipse IDE [online]. 2013 [visitado el 3 de abril de 2013]. URL http://www.eclipse.org/.

[15] Texas Instruments. TMS320C64x Technical Overview [online]. 2001 [visitado el 3 de abril de 2013]. URL http://www.ti.com/lit/ug/spru395b/spru395b.pdf.

[16] Texas Instruments. TMS320C64x/c64x+ DSP CPU and Instruction Set [online]. 2008 [visitado el 3 de abril de 2013]. URL http://www.ti.com/lit/ug/spru198k/spru198k.pdf.

[17] Texas Instruments. C6RunLib documentation [online]. 2011 [visitado el 3 de abril de 2013]. URL http://processors.wiki.ti.com/index.php?title=C6RunLib_Documentation.

[18] Texas Instruments. DM3730, DM3725 digital media processors [online]. 2011 [visitado el 3 de abril de 2013]. URL http://www.ti.com/lit/ds/symlink/dm3730.pdf.

[19] Texas Instruments. TMS320C6000 programmer's guide [online]. 2011 [visitado el 3 de abril de 2013]. URL http://www.ti.com/lit/ug/spru198k/spru198k.pdf.

[20] L. Logesparan, A. Casson, and E. Rodriguez. Assessing the impact of signal normalization [online]. 2007 [visitado el 3 de abril de 2013]. URL https://spiral.imperial.ac.uk/bitstream/10044/1/9006/1/paper.pdf.

[21] D. MacKay. *Information Theory, Inference and Learning Algorithms*. Cambridge, 2003.

[22] R. Maher. Summary of gun shot acoustics. *Montana State University*, Abril 2006.

[23] H. Mamitsuka. Supervised learning of hidden markov models for sequence discrimination. *Proceedings of the first annual international conference on Computational molecular biology*, 1997.

[24] T. P. Mann. Numerically stable Hidden Markov Model implementation. 2006.

[25] A. Moore. An intoductory tutorial on kd-trees. Tesis de Doctorado, Computer Laboratory, University of Cambridge, 1991.

[26] Nokia. Qt development frameworks [online]. 2013 [visitado el 3 de abril de 2013]. URL http://qt.digia.com/Product/Developer-Tools/.

[27] S. M. Omohundro. Efficient algorithms with neural network behavior. *University of Illinois at Urbana-Champaign*, 1987.

[28] A. Oppenheim, R. Schafer, and J. Buck. *Discrete-Time Signal Processing*. Prentice Hall, 1999.

[29] D. Phillips. Knowing JACK. *Linux Magazine*, 2006.

[30] K. Prabakar and S. Mallikarjun. Pattern recognition analysis of acoustic emission signals propagated through a waveguide. *Indian Journal of Pure & Applied Physics*, 2007.

[31] GNU Project. GCC, the GNU compiler collection [online]. 2013 [visitado el 3 de abril de 2013]. URL http://gcc.gnu.org/.

[32] M. Puckette. Envelope following [online]. 2006 [visitado el 3 de abril de 2013]. URL http://crca.ucsd.edu/~msp/techniques/v0.11/book-html/node153.html.

[33] L. Rabiner. A tutorial on Hidden Markov Models and selected applications. *IEEE*, 1990.

[34] E. Salas. Reconocimiento en tiempo real de patrones acústicos de motosierras y disparos por medio de una implementación en FPGA de Modelos Ocultos de Markov. Tesis de Licenciatura, Escuela de Ingeniería Electrónica, ITCR, Mayo 2010.

[35] T. Saramäki and R. Bregović. *Multirate Systems and Filter Banks*. Idea Group Publishing, 2010.

[36] M. Sequeira. Módulo de reducción de dimensiones espectrales en un sistema de reconocimiento de patrones acústicos de motosierras y disparos por medio de una implementación en FPGA. Tesis de Licenciatura, Escuela de Ingeniería Electrónica, ITCR, Junio 2011.

[37] D. Shaillay. Normalization [online]. 2013 [visitado el 3 de abril de 2013]. URL http://www.qsarworld.com/qsar-statistics-normalization.php.

[38] D. Shen. Some mathematics for HMM. *Massachusetts Institute of Technology*, 2008.

[39] SINAC. Políticas para las Áreas silvestres protegidas (ASP) [online]. 2011 [visitado el 3 de abril de 2013]. URL https://www.sinac.go.cr/documentacion/.

[40] SINAC. SINAC en números [online]. 2011 [visitado el 3 de abril de 2013]. URL https://www.sinac.go.cr/documentacion/.

[41] E. Smith. Reconocimiento digital en línea de patrones acústicos para la protección del ambiente por medio de HMM. Tesis de Licenciatura, Escuela de Ingeniería Electrónica, ITCR, Enero 2008.

[42] J. R. Smith. *Modern Communication Circuits*. McGraw Hill Electrical and Computer Engineering Series, 1998.

[43] F. G. Stremler. *Introducción a los sistemas de comunicación*. Addison Wesley Longman, 1993.

[44] M. Sáenz. Reconocimiento de patrones acústicos para la protección del ambiente utilizando wavelets y Modelos Ocultos de Markov. Tesis de Licenciatura, Escuela de Ingeniería Electrónica, ITCR, Noviembre 2006.

[45] S. Theodoridis and K. Koutroumbas. *Pattern Recognition*. Elsevier, 2009.

[46] R. F. Tobler. The rkd-tree. *Computer Graphics International*, 2011.

[47] W. Tomasi. *Sistemas de comunicaciones electrónicas*. Prentice Hall, 2003.

[48] J. Tsiombikas. kdtree [online]. 2012 [visitado el 3 de abril de 2013]. URL http://code.google.com/p/kdtree/.

[49] J. Valverde. Implementación de una metodología para lograr la integración física correcta por construcción (CBC) de un banco de filtros digitales descrito en alto nivel. Tesis de Licenciatura, Escuela de Ingeniería Electrónica, ITCR, 2011.

[50] O. Villalta. Circuito de control automático de ganancia de baja potencia para un sistema de reconocimiento de patrones acústicos. Tesis de Licenciatura, Escuela de Ingeniería Electrónica, ITCR, Diciembre 2012.

[51] T. Weaver. Installing Angstrom on the BeagleBoard-xM [online]. 2010 [visitado el 3 de abril de 2013]. URL http://treyweaver.blogspot.com/2010/10/installing-angstrom-on-beagleboard-xm.html.

[52] Q. Wu, F. Merchant, and K. Castleman. *Microscope Image Processing*. Elsevier, 2008.

Apéndice A

Resultados completos de pruebas de entrentamiento

Tabla A.1: Resultados entrenamiento múltiple con 3 estados.

			Sensitividad (%)			VPP (%)		
Muestreo	Long. Cadena		Bosque	Disparo	Motosierra	Bosque	Disparo	Motosierra
1	5		57,89	80,95	56,67	62,86	94,44	47,22
	10		81,58	71,43	56,67	73,81	88,24	56,67
	15		76,32	61,9	63,33	72,5	86,67	55,88
	20		73,68	66,67	80	77,78	87,5	64,86
	25		78,95	61,9	76,67	78,95	92,86	62,16
	20		84,21	85,71	70	78,05	94,74	72,41
5	5		78,95	76,19	76,67	83,33	88,89	65,71
	10		55,26	85,71	76,67	72,41	90	57,5
	15		81,58	89,47	70	77,5	94,44	72,41
	20		78,95	88,89	83,33	85,71	94,12	73,53
	25		71,05	87,5	83,33	84,38	100	65,79
	20		89,47	90	76,67	82,93	100	82,14
10	5		71,05	85,71	70	77,14	90	61,76
	10		57,89	94,44	73,33	73,33	94,44	57,89
	15		68,42	86,67	86,67	86,67	100	65
	20		73,68	84,62	76,67	77,78	100	67,65
	25		89,47	90	86,67	89,47	100	83,87
	20		86,84	0	90	91,67	0	79,41
20	5		55,26	94,44	70	70	94,44	55,26
	10		84,21	85,71	80	84,21	100	75
	15		89,47	88,89	66,67	77,27	100	80
	20		84,21	25	76,67	80	100	74,19
	25		84,21	0	86,67	88,89	0	76,47
	20		89,47	N.A.	68,97	79,07	N.A.	95,24

Tabla A.2: Resultados entrenamiento múltiple con 5 estados.

		Sensitividad (%)			VPP (%)		
Muestreo	Long. Cadena	Bosque	Disparo	Motosierra	Bosque	Disparo	Motosierra
1	5	52,63	80,95	63,33	66,67	94,44	46,34
	10	78,95	71,43	63,33	75	88,24	59,38
	15	71,05	61,9	66,67	75	86,67	52,63
	20	71,05	66,67	73,33	71,05	93,33	61,33
	25	68,42	66,67	80	78,79	93,33	58,54
	20	86,84	90,48	73,33	80,49	95	78,57
5	5	63,16	76,19	80	80	94,12	57,14
	10	73,68	85,71	76,67	80	100	63,89
	15	76,32	89,47	76,67	80,56	94,44	69,7
	20	84,21	88,89	76,67	82,05	94,12	76,67
	25	73,68	87,5	86,67	87,5	100	68,42
	20	78,95	90	90	90,91	100	75
10	5	60,53	80,95	80	79,31	94,44	57,14
	10	65,79	94,44	80	80,65	94,44	64,86
	15	76,32	86,67	73,33	78,38	100	66,67
	20	78,95	92,31	83,33	85,71	100	73,53
	25	86,84	80	80	82,5	100	80
	20	86,84	0	90	91,67	0	79,41
20	5	63,16	61,11	66,67	68,57	100	50
	10	76,32	78,57	73,33	74,36	100	68,75
	15	94,74	88,89	73,33	81,82	100	88
	20	84,21	25	80	82,05	100	75
	25	92,11	0	86,67	87,5	0	86,67
	20	94,74	N.A.	86,21	90	N.A.	96,15

Tabla A.3: Resultados entrenamiento múltiple con 10 estados.

		Sensitividad (%)			VPP (%)		
Muestreo	Long. Cadena	Bosque	Disparo	Motosierra	Bosque	Disparo	Motosierra
1	5	65,79	71,43	53,33	69,44	83,33	45,71
	10	73,68	71,43	46,67	63,64	93,75	48,28
	15	68,42	71,43	66,67	74,29	88,24	54,05
	20	71,05	80,95	70	71,05	89,47	65,62
	25	71,05	71,43	83,33	84,38	88,24	62,5
	20	84,21	85,71	70	78,05	94,74	72,41
5	5	63,16	85,71	80	80	94,74	60
	10	60,53	85,71	73,33	74,19	94,74	56,41
	15	78,95	84,21	73,33	76,92	100	68,75
	20	76,32	83,33	73,33	76,32	93,75	68,75
	25	84,21	75	80	80	100	75
	20	86,84	80	90	89,19	100	81,82
10	5	71,05	80,95	73,33	77,14	100	59,46
	10	73,68	83,33	83,33	82,35	100	67,57
	15	78,95	73,33	76,67	75	100	71,88
	20	78,95	92,31	83,33	85,71	100	73,53
	25	84,21	80	86,67	86,49	100	78,79
	20	94,74	0	93,33	94,74	0	87,5
20	5	81,58	66,67	70	72,09	100	67,74
	10	73,68	78,57	76,67	75,68	100	67,65
	15	92,11	66,67	70	74,47	100	87,5
	20	84,21	25	70	74,42	100	75
	25	94,74	0	73,33	80	0	88
	20	89,47	N.A.	82,76	87,18	N.A.	100

Tabla A.4: Resultados entrenamiento múltiple con 15 estados.

		Sensitividad (%)			VPP (%)		
Muestreo	Long. Cadena	Bosque	Disparo	Motosierra	Bosque	Disparo	Motosierra
1	5	65,79	61,9	73,33	78,12	92,86	51,16
	10	76,32	66,67	56,67	69,05	93,33	53,12
	15	71,05	76,19	63,33	72,97	94,12	54,29
	20	78,95	80,95	66,67	73,17	89,47	68,97
	25	86,84	66,67	76,67	82,5	93,33	67,65
	20	81,58	85,71	70	77,5	94,74	70
5	5	68,42	80,95	63,33	70,27	94,44	55,88
	10	81,58	76,19	76,67	77,5	94,12	71,88
	15	81,58	78,95	70	73,81	100	70
	20	39,47	72,22	86,67	68,18	92,86	52
	25	89,47	75	76,67	77,27	100	82,14
	20	86,84	70	83,33	84,62	100	78,12
10	5	71,05	76,19	80	79,41	100	61,54
	10	71,05	77,78	83,33	81,82	100	64,1
	15	94,74	66,67	73,33	75	100	88
	20	86,84	61,54	73,33	80,49	100	68,75
	25	76,32	30	66,67	72,5	100	57,14
	20	94,74	0	86,67	90	0	86,67
20	5	60,53	77,78	73,33	65,71	93,33	61,11
	10	81,58	71,43	76,67	73,81	90,91	79,31
	15	92,11	44,44	66,67	76,09	100	74,07
	20	94,74	25	63,33	73,47	100	86,36
	25	97,37	0	60	74	0	90
	20	92,11	N.A.	89,66	92,11	N.A.	100

Tabla A.5: Resultados entrenamiento negativo con 3 estados.

		Sensitividad (%)			VPP (%)		
Muestreo	Long. Cadena	Bosque	Disparo	Motosierra	Bosque	Disparo	Motosierra
1	5	97,37	0	13,33	47,44	0	66,67
	10	44,74	4,76	0	30,36	3,23	0
	15	13,16	4,76	6,67	12,82	2,38	25
	20	13,16	4,76	10	12,5	2,56	30
	25	26,32	0	3,33	20,41	0	16,67
	20	13,16	0	16,67	16,67	0	19,23
5	5	57,89	0	10	35,48	0	37,5
	10	15,79	0	20	14,29	0	54,55
	15	23,68	0	16,67	20,93	0	35,71
	20	21,05	0	20	25	0	24
	25	21,05	0	33,33	34,78	0	31,25
	20	31,58	0	43,33	46,15	0	39,39
10	5	47,37	0	20	31,58	0	60
	10	26,32	0	16,67	25,64	0	29,41
	15	28,95	0	33,33	40,74	0	31,25
	20	31,58	0	40	46,15	0	36,36
	25	28,95	0	40	42,31	0	40
	20	42,41	0	46,67	53,33	0	51,85
20	5	55,26	0	20	40,38	0	28,57
	10	42,11	0	33,33	50	0	33,33
	15	23,68	0	40	37,5	0	37,5
	20	23,68	0	43,33	37,5	0	46,43
	25	23,68	0	43,33	37,5	0	50
	20	18,42	N.A	58,62	36,84	N.A	48,57

Tabla A.6: Resultados entrenamiento negativo con 5 estados.

		Sensitividad (%)			VPP (%)		
Muestreo	Long. Cadena	Bosque	Disparo	Motosierra	Bosque	Disparo	Motosierra
1	5	97,37	0	13,33	50	0	36,36
	10	28,95	4,76	0	24,44	2,63	0
	15	21,05	4,76	10	18,18	2,56	50
	20	13,16	4,76	16,67	13,16	2,56	41,67
	25	10,53	0	6,67	9,3	0	25
	20	13,16	0	20	11,11	0	66,67
5	5	42,11	0	10	28,07	0	37,5
	10	10,53	0	20	10	0	46,15
	15	15,79	0	16,67	21,43	0	17,86
	20	15,79	0	20	22,22	0	20,69
	25	15,79	0	26,67	25	0	25,81
	20	21,05	0	43,33	34,78	0	48,15
10	5	42,11	0	20	28,57	0	54,55
	10	21,05	0	16,67	27,59	0	20
	15	23,68	0	23,33	32,14	0	24,14
	20	23,78	0	24,57	34,29	0	25,18
	25	21,05	0	46,67	36,36	0	50
	20	18,42	0	50	35	0	57,69
20	5	52,63	0	16,67	38,46	0	29,41
	10	39,47	0	30	48,39	0	34,62
	15	23,68	0	36,67	34,62	0	40,74
	20	18,42	0	43,33	31,82	0	50
	25	15,79	0	43,33	28,57	0	54,17
	20	10,53	N.A.	55,17	23,53	N.A.	61,54

Tabla A.7: Resultados entrenamiento negativo con 10 estados.

		Sensitividad (%)			VPP (%)		
Muestreo	Long. Cadena	Bosque	Disparo	Motosierra	Bosque	Disparo	Motosierra
1	5	97,37	0	6,67	46,84	0	50
	10	39,47	4,76	0	30	2,78	0
	15	26,32	4,76	10	23,26	2,63	37,5
	20	10,53	4,76	13,33	10,53	2,5	36,36
	25	10,53	0	6,67	13,33	0	8,7
	20	7,89	0	13,33	12	0	15,38
5	5	63,16	0	6,67	37,5	0	28,57
	10	13,16	0	16,67	12,5	0	41,67
	15	18,42	0	16,67	24,14	0	20,83
	20	18,42	0	20	25	0	20,69
	25	18,42	0	43,33	35	0	36,11
	20	18,42	0	43,33	31,82	0	41,94
10	5	42,11	0	16,67	28,57	0	50
	10	26,32	0	16,67	33,33	0	20
	15	31,58	0	23,33	38,71	0	25,93
	20	23,68	0	40	37,5	0	41,38
	25	21,05	0	46,67	38,1	0	46,67
	20	18,42	0	46,67	33,33	0	56
20	5	55,26	0	16,67	40,38	0	25
	10	44,74	0	33,33	51,52	0	37,04
	15	23,68	0	36,67	37,5	0	39,29
	20	21,05	0	43,33	34,78	0	48,15
	25	18,42	0	43,33	31,82	0	50
	20	18,42	N.A.	58,62	36,84	0	62,96

Apéndice B

Manuales de usuario

B.1 Manual de usuario para aplicación de entrenamiento

Se presenta a continuación una guía de uso para cada una de las secciones de entrenamiento. Se presentan sólo detalles de funcionamiento para la interfaz gráfica, pero debe notarse que en ocasiones se imprime información relevante en consola, tal como resultados de los diferentes algoritmos de entrenamiento, o notificación de fallos. Esto debe tomarse en cuenta al utilizar la aplicación.

B.1.1 Pantalla principal

La pantalla principal de la interfaz se muestra en la figura B.1. Esta se muestra cada vez que se inicia la aplicación.

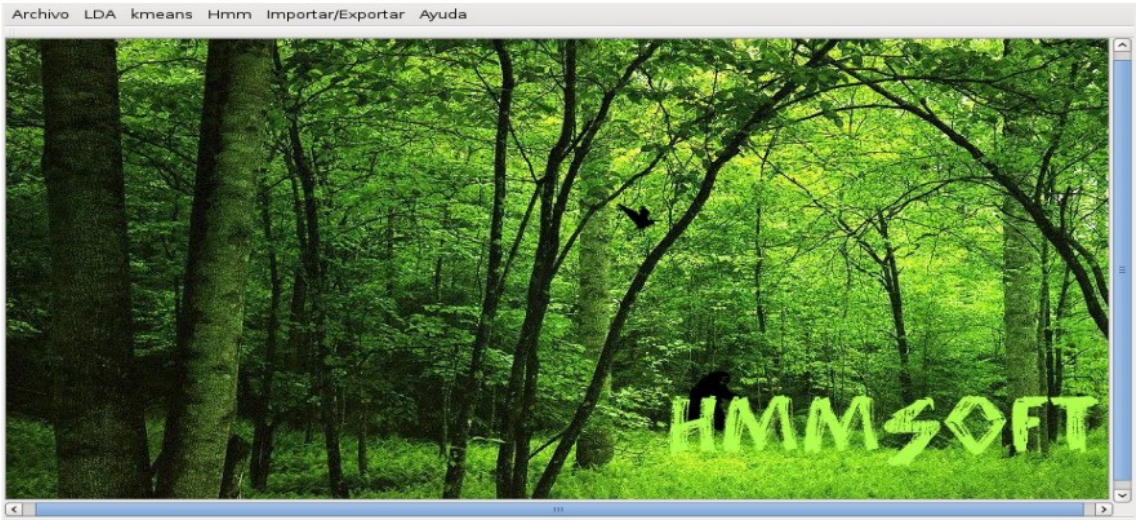


Figura B.1: Pantalla principal de aplicación de entrenamiento.

Se tienen en total 6 menús, donde cada uno se encarga de una sección específica del entrenamiento o de la administración de resultados. Se detallan a continuación cada uno de ellos.

Menú *Archivo*

- Salir: Cierra la aplicación.

Menú *LDA*

- Extraer Vectores: Permite extraer los vectores utilizados para el entrenamiento con LDA.
- Consultar Vectores: Permite consultar los vectores que han sido extraídos y almacenados en la base de datos. Además, pueden ser eliminados desde acá.
- Entrenar: Abre pantalla para realizar el entrenamiento con LDA.
- Consultar Matriz: Despliega en pantalla la matriz almacenada en la base de datos, de existir. Además, es posible exportarla o eliminarla desde acá.

Menú *kmeans*

- Extraer Vectores: Permite extraer los vectores utilizados para el entrenamiento con algoritmo de k-medias.
- Consultar Vectores: Permite consultar los vectores que han sido extraídos y almacenados en la base de datos. Además, pueden ser eliminados desde acá.
- Entrenar: Abre pantalla para realizar el entrenamiento con algoritmo de k-medias.
- Consultar Matriz: Despliega en pantalla la matriz almacenada en la base de datos, de existir. Además, es posible exportarla o eliminarla desde acá.

Menú *HMM*

- Clases: Usada para crear o eliminar clases usadas por los HMM, tal como pueden ser disparo, motosierra, etc.
- Cadenas de observación: Permite la consulta y extracción de los símbolos desde archivos de audio. Estos son utilizados para entrenamiento y evaluación. Se compone por diferentes submenú para realizar estas acciones.
- Modelos: Permite la creación, consulta, entrenamiento y evaluación de los HMM, usando diferentes algoritmos para cada acción. Cada submenú incluye funciones para administrar modelos, entrenarlos mediantes las diferentes variantes implementadas (entrenamiento simpe, múltiple y negativo), crear grupos de evaluación y llevar estas a cabo, consultar resultados y matrices de confusión, entre otras funciones adicionales.

Menú *Importar/Exportar*

- Exportar Embebido: Permite exportar los modelos deseados a archivos de texto que pueden ser posteriormente cargados por el sistema embebido donde se ejecuta el SiRPA. Exporta todos los otros elementos necesarios junto al archivo de modelos.
- Importar Reductor: Usado para importar resultados de entrenamiento con LDA desde archivos de texto, en caso de que entrenamiento sea realizado de manera

externa a la presente aplicación. Estos pueden ser almacenados en la base de datos y/o vinculados con el SiRPA directamente desde acá.

- Importar Centroides: Usado para importar resultados de entrenamiento con *k*-means desde archivos de texto, en caso de que entrenamiento sea realizado de manera externa a la presente aplicación. Estos pueden ser almacenados en la base de datos y/o vinculados con el SiRPA directamente desde acá.

Menú *Ayuda*

- Documentación: Por el momento no realiza acción alguna, pero en un futuro se usaría para desplegar ayuda y documentación de uso de la aplicación.
- Acerca de: Despliega información de la aplicación.

En este manual se muestra información de uso para los menús de LDA, kmeans e Importar/Exportar. Las otras secciones se explican de manera detallada en el manual de usuario en [7].

B.1.2 Menús LDA y kmeans

Extraer Vectores

La ventana desplegada al seleccionar este elemento se muestra en la figura B.2. Esta es igual tanto para el menú de LDA como el de kmeans.

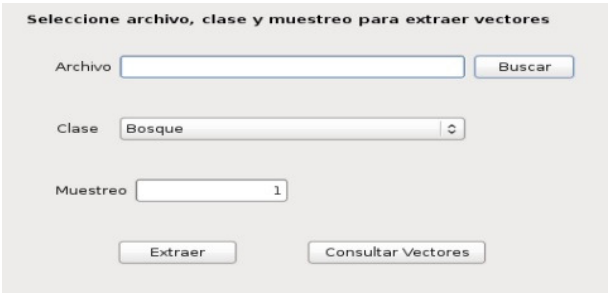


Figura B.2: Ventana para menú de extracción de vectores.

Al presionar en *buscar* se abre un explorador de archivos, por medio del cual pueden seleccionarse los archivos de audio a utilizar para realizar cada tipo de entrenamiento. Puede seleccionarse un sólo archivo, o varios a la vez. No deben mezclarse archivos de diferentes clases, pues para un entrenamiento correcto estos deben ser almacenados con identificadores diferentes para cada una. La clase debe ser seleccionada desde la lista desplegable mostrada bajo la ruta del archivo.

El muestreo permite reducir el número de muestras que son extraídas desde cada archivo, en caso que se considere necesario. Si se coloca en 1 (valor por defecto) se extraen *todos* los vectores de salida contenidos por el archivo. Valores mayores a uno dividen esta cantidad en el factor que se haya introducido. Para archivos con longitudes cortas, tal como algunos de disparos, es preferible dejarlo en 1 para contar con un mayor número de

muestras. Para archivos de extensión larga podría resultar beneficioso incrementar este valor para mejorar tasas de reconocimiento.

Al presionar en *extraer*, se realiza este procedimiento con todos los archivos seleccionados y se almacenan resultados en base de datos. Si se presiona en *consultar*, se abre ventana de consulta de vectores.

Consultar Vectores

La ventana para esta opción se muestra en la figura . El funcionamiento es el mismo en la sección de LDA y la de kmeans.

Figura B.3: Ventana para menú de consulta de vectores.

Acá pueden seleccionarse archivos en la lista de la izquierda, y al presionar en *consultar* de despliega el archivo de texto con los vectores respectivos. En caso de seleccionar varios archivos, se abre *solamente* el primero de ellos. Por otro lado, si se presiona en *eliminar* se borran los vectores seleccionados del sistema de archivos y de la base de datos.

En la lista de la izquierda, si se presiona *ninguno* o *todos* se deseleccionan/seleccionan todos los archivos. En el de la derecha se hace los mismo pero con las clases. En caso de modificar las clases a desplegar de manera manual, debe presionarse en *actualizar* para mostrar los archivos correctos a la izquierda.

Entrenar

El menú para este caso se muestra en la figura B.4. Igualmente es compartido para la sección de LDA y kmeans.

Figura B.4: Ventana para entrenamiento usado LDA o algoritmo k-means.

El funcionamiento es similar al de la ventana de la subsección previa. La única diferencia es que si se presiona en *entrenar*, se lleva a cabo el entrenamiento con los archivos seleccionados, usando LDA o el algoritmo k-means según sea el caso. Al presionar en *consultar matriz*, se abre la ventana donde se despliegan los resultados obtenidos con el entrenamiento.

Consultar Matriz LDA

En esta ventana se despliegan los resultados obtenidos con LDA, y que se encuentran almacenados en la base de datos. La figura B.5 muestra un ejemplo.

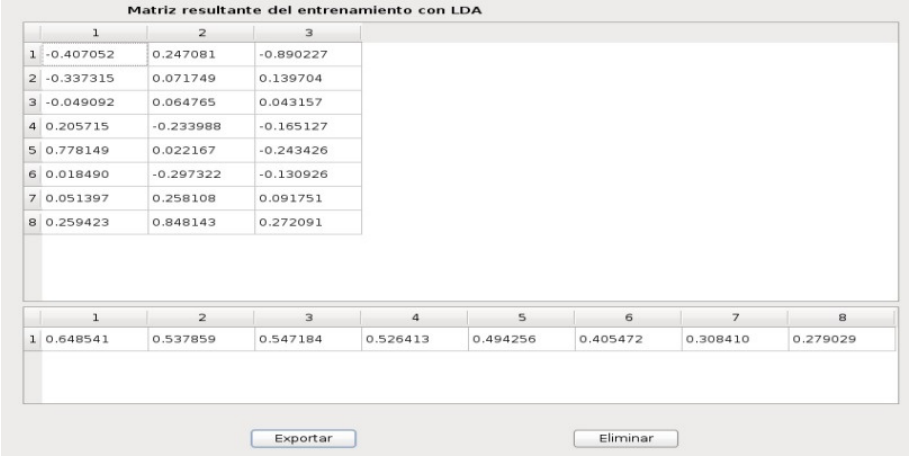


Figura B.5: Ventana para despliegue de resultados de LDA.

La matriz mostrada en la parte superior es la que se encuentra almacenada en la base de datos, resultante del último entrenamiento que se haya realizado. Esta corresponde a la matriz de transformación $\mathbf{W}$, de tamaño 8×3. En la parte inferior se muestra el vector de media $\underline{\mu}$, que es de 8 dimensiones.

En caso de que se presione *eliminar*, la matriz y el vector son borrados en pantalla y de la base de datos. Si se desea utilizar esta matriz con el SiRPA, ya sea en esta misma implementación o en el sistema embebido, debe presionarse *exportar*. Si esto no se hace, estos elementos **no** son vinculados al sistema, y por lo tanto etapas posteriores no darán los resultados esperados.

Consultar Matriz kmeans

Esta ventana se muestra en la figura B.6, y es la que despliega los resultados del entrenamiento con el algoritmo de k-medias. Aplican las consideraciones explicadas en la subsección previa, con la excepción de que en este caso lo que se muestra en pantalla son los 32 centroides obtenidos al aplicar el algoritmo al conjunto de datos usados para entrenar.

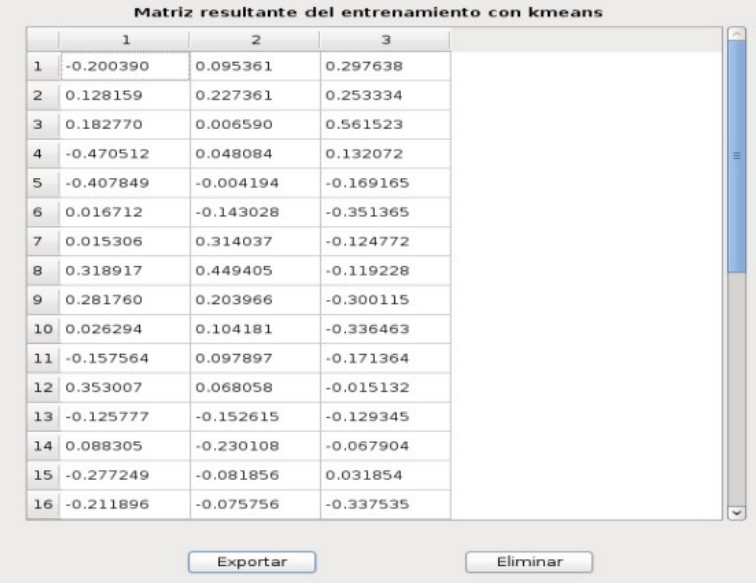


Figura B.6: Ventana para despliegue de resultados de algoritmo k-means.

B.1.3 Menú Importar/Exportar

Exportar Embebido

La ventana para esta opción se muestra en la figura B.7.

Figura B.7: Ventana para exportación de archivos al sistema embebido.

Desde esta ventana es posible exportar todos los archivos requeridos para el funcionamiento de la aplicación ejecutada en el sistema embebido. Estos son colocados en la dirección indicada por la ruta. Se exportan todos los modelos seleccionados en la lista de la derecha en un mismo archivo. Todo esto se hace cuando se presiona en *guardar*.

Antes de exportar, debe definirse la longitud de cadena y el muestreo utilizado por los modelos. Si el primero se coloca en 0, se utiliza la longitud de cadena con la que se entrenó

el modelo. El muestreo **sí** debe ser introducido, dado que los modelos no almacenan este valor en su entrada en la base de datos, por lo que debe ser definido previo a exportar.

Importar Reductor

Desde este elemento es posible importar la matriz y el vector generados del entrenamiento con LDA mediante una aplicación externa a la utilizada para entrenamiento. La ventana desplegada se muestra en la figura B.8.

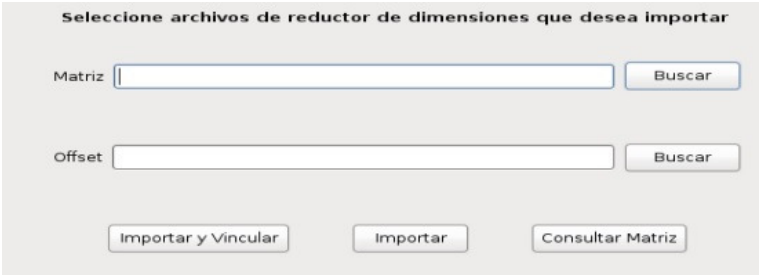


Figura B.8: Ventana para importación de archivos del reductor.

En este caso, deben indicarse primero la ruta de ambos archivos, para lo que se incluye la función de *buscar*. Una vez que estos han sido ubicados se puede proceder a importar los valores respectivos. Si se presiona en *importar y vincular*, los valores son almacenados en la base de datos para su administración, y también vinculados con el generador de símbolos del SiRPA. En caso de presionar en *importar*, solamente se almacenan en la base de datos.

Importar Centroides

La ventana desplegada en este caso se muestra en la figura B.9. El funcionamiento es igual al explicado en la subsección previa, con la diferencia de que sólo debe buscarse el archivo que contiene los centroides.

Figura B.9: Ventana para importación de centroides.

B.2 Manual de usuario para aplicación de sistema embebido

La aplicación para el sistema embebido es ejecutada completamente desde la terminal que se abre al acceder al sistema embebido. Existen varias maneras de hacer esto, entre las que se tiene el acceso mediante Ethernet utilizando *ssh* o el acceso usando un cable serial. Se explican ambas alternativas, usando para la última el programa *minicom*.

B.2.1 Compilación y configuración de la aplicación

Con el fin de hacer más flexible el uso de la aplicación, se tiene un gran número de parámetros configurables, almacenados todos en el archivo de encabezado SiRPA.h. Los valores almacenados permiten modificar número de muestras tomadas desde archivos, tamaño de bloque usado al procesar archivos, constantes utilizadas por algunos bloques, entre otros factores. Una configuración importante es la plataforma para la que se compila, que puede ser el uso de sólo el procesador ARM, o la combinación ARM + DSP; esto se deja como parámetro configurable para que con sólo modificar su valor se hagan todos los cambios necesarios entre implementaciones.

Por lo anterior, es necesario comprender cómo se compila la aplicación. Para ello, se supone que el entorno de desarrollo ya ha sido configurado (ver próximo apéndice), que el sistema anfitrión trabaja con la distribución de Linux Ubuntu y que se compila utilizando *make*. Así, es necesario que junto a todos los archivos fuente (.c) y los encabezados (.h) se tenga un *makefile* adecuadamente configurado; el que se incluye con el proyecto ya ha sido configurado a como se requiere.

Lo primero que debe realizarse es cargar al entorno una serie de banderas y variables requeridas para realizar la compilación y vinculación de la aplicación. Para ello se siguen los siguientes comandos

```
cd ruta_de_carpeta/C6Run_X_XX_XX_XX
source environment.sh
```

Puede verificarse la carga de las variables ejecutando alguno de los siguientes comandos antes y después de hacer el *source*

```
printenv | grep ARM
printenv | grep C6
printenv | grep FLAGS
```

En caso de cargarse correctamente, deberían desplegarse diversas rutas y variables en cada caso, las cuales no deberían ser visibles *antes* de cargar.

Una vez hecho esto, se puede compilar la aplicación buscando primero la ruta del makefile como

```
cd ruta_del_makefile
```

Una vez ubicado, puede compilarse la aplicación para el procesador ARM **o** para la combinación ARM + DSP, según se haya definido en el archivo de configuraciones. Los comandos serían, respectivamente

```
make gpp_exec
make dsp_exec
```

Al hacer esto, se generará un archivo ejecutable que puede ser transferido al sistema embebido. Esto puede hacerse por medio de Ethernet o conectando la tarjeta MicroSD del embebido en el computador anfitrión. Para el primer caso se haría como

```
scp Archivo_1 Archivo_2 ... Archivo_N usuario@ip.del.embebido:ruta_en_embebido
```

El siguiente ejemplo ilustra este comando

```
scp Exec_ARM Exec_DSP root@192.168.1.6:/home/root/SiRPA_Apps
```

donde se copian los dos ejecutables en el embebido con la IP 192.168.1.6, usando como usuario *root*, y en la carpeta SiRPA_Apps. Esto pediría la contraseña del usuario root, que por defecto es igual al nombre de usuario.

En caso de copiarlo usando la tarjeta MicroSD del embebido, el comando a utilizar es

```
cp Archivo_1 Archivo_2 ... Archivo_N ruta_en_tarjeta_MicroSD
```

Por ejemplo, suponiendo que la partición del sistema de archivos del embebido se llama *Angstrom*, se tendría

```
cp Exec_ARM Exec_DSP /media/Angstrom/home/root/SiRPA_Apps
```

B.2.2 Acceso al sistema embebido

Tal como se mencionó previamente, existen diferentes maneras de acceder al sistema embebido. Por ejemplo, al utilizar *ssh*, se ejecuta el comando

```
ssh usuario@ip.del.embebido
```

Un ejemplo de acceso sería

```
ssh root@192.168.1.6
```

Una vez que se introduce la contraseña de usuario, se estaría ejecutando una terminal *desde* el sistema embebido, por lo que ya pueden utilizarse las aplicaciones compiladas para este.

Para el acceso mediante cable serial primero debe realizarse la conexión entre el computador anfitrión y el sistema embebido, ya sea mediante uno de estos cables o un adaptador USB a serial. Una vez conectado, y teniendo configurado el programa *minicom*, basta con ejecutar

```
sudo minicom
```

Al ejecutar esto, se solicita la contraseña de administrador del computador anfitrión. Una vez introducida, se lleva a cabo la conexión con el sistema embebido.

Se recomienda utilizar siempre la conexión mediante *ssh*, pues es más flexible para la transferencia de archivos entre el computador y el sistema embebido, y permite también que varios equipos puedan acceder al sistema embebido de ser necesario. El uso de la conexión serial es importante en aquellos casos donde se desean ver los mensajes de arranque, mensajes desplegados al cargar módulos y otra información que no es mostrada al conectar mediante la red. Por ello, es útil en caso de que se desee identificar errores y depurar la aplicación.

B.2.3 Ejecución de la aplicación

Para ejecutar la aplicación se debe acceder primero al sistema embebido. Una vez hecho esto, se busca la ruta del archivo ejecutable como

```
cd ruta_del_ejecutable
```

Una vez que ha sido ubicado, puede iniciarse el reconocimiento mediante el siguiente comando

```
./nombre_exec matriz_reductor.txt vector_reductor.txt centroides.txt modelos.txt resultados.txt archivo_audio.wav
```

Los dos últimos parámetros son opcionales, y son requeridos si se desea aplicar el reconocimiento sobre archivos de audio. En caso de no incluirlos, se activa el modo en línea y el sistema embebido comienza a capturar muestras desde la entrada de audio. En este último caso, los resultados del reconocimiento son solamente impresos en consola. En la figura B.10 se muestra un ejemplo de ejecución operando sobre archivos, mientras

```
root@beagleboard:~/SiRPA_Apps# ./SiRPA_ARM LDA_TM_SG.txt LDA_Offset_SG.txt kmeans_Centroides_SG.txt HMM_3E.txt Resultados.txt
PRB_D030m_S12_090_a_mono_recortado_JC.wav
Cargando desde archivo entrenado LDA_TM_SG.txt...
Cargando desde archivo entrenado LDA_Offset_SG.txt...
Cargando desde archivo entrenado kmeans_Centroides_SG.txt...
Iniciando reconocimiento a partir del archivo: PRB_D030m_S12_090_a_mono_recortado_JC.wav
Frecuencia de muestreo del archivo: 44100
Numero de canales: 1
Numero de muestras (por canal) a procesar: 64665
Cargando codigo en el DSP...
Listo!
Estado actual: MULT_M10_25_Disparo_25_SIM_3_EST
Probabilidad:  -66.901595
```

Figura B.10: Ejemplo de reconocimiento sobre archivos.

```
root@beagleboard:~/SiRPA_Apps# ./SiRPA_ARM LDA_TM_SG.txt LDA_Offset_SG.txt kmeans_Centroides_SG.txt HMM_3E.txt Resultados.txt
Cargando desde archivo entrenado LDA_TM_SG.txt...
Cargando desde archivo entrenado LDA_Offset_SG.txt...
Cargando desde archivo entrenado kmeans_Centroides_SG.txt...
Iniciando reconocimiento desde entrada de audio
jackd 0.118.0
Copyright 2001-2009 Paul Davis, Stephane Letz, Jack O'Quinn, Torben Hohn and others.
jackd comes with ABSOLUTELY NO WARRANTY
This is free software, and you are welcome to redistribute it
under certain conditions; see the file COPYING for details

JACK compiled with System V SHM support.
loading driver ..
creating alsa driver ... hw:0|hw:0|256|4|48000|0|0|nomon|swmeter|-|32bit
control device hw:0
configuring for 48000Hz, period = 256 frames (5.3 ms), buffer = 4 periods
ALSA: final selected sample format for capture: 16bit little-endian
ALSA: use 4 periods for capture
ALSA: final selected sample format for playback: 16bit little-endian
ALSA: use 4 periods for playback
JACK server started
Client Name: SiRPA_Beagle
engine sample rate: 48000
Process Buffer Size: 256
Cargando codigo en el DSP...
Listo!
Estado actual: MULT_M10_25_Bosque_25_SIM_3_EST
Probabilidad:  -68.909139
```

Figura B.11: Ejemplo de reconocimiento desde entrada de línea.

que en la figura B.11 se muestra la ejecución usando la entrada de línea. Esta última opera de manera indefinida, hasta que sea detenida por el usuario; para hacer esto, debe introducirse en la terminal la combinación *Ctrl - C*.

El archivo de resultados es generado en caso de no existir y almacena información sobre el tiempo de procesamiento por bloque y probabilidades de salida de cada modelo. Al usar este modo se generan además una serie de archivos con nombre *SiRPA_NombreBloque.txt*, que almacenan todos los resultados de salida de cada bloque individualmente, lo que permite estudiar resultados intermedios. Todos estos son generados en la misma carpeta donde se ubica el ejecutable.

Los archivos con los valores para el reductor, los centroides y los modelos son generados por la aplicación de entrenamiento, y deben ser copiados en el sistema de archivos del sistema embebido. Se recomienda copiarlos en la misma carpeta donde se coloca el ejecutable, pues de lo contrario debe indicarse la ruta para cada uno de ellos al llamar la aplicación. Esto aplica también para el archivo de audio, pues si no está en la misma carpeta debe introducirse la ruta adecuada.

En ocasiones se indica que la aplicación no puede ser ejecutada, lo cual se debe usualmente a que no cuenta con los permisos (bandera) de ejecución. Para resolver esto basta con usar el siguiente comando, y luego reintentar la ejecución

```
chmod a+x nombre_exec
```

En caso de utilizar la aplicación compilada para usar el DSP, deben cargarse varios módulos para el *kernel* que se encargan de la comunicación con el DSP y de la administración de memoria compartida. De no hacer esto, la aplicación fallará al intentar ejecutarla.

En la sección de configuración del entorno se indica cuales son estos módulos, que deben ser copiados en la memoria del sistema embebido. Una vez hecho esto, deben cargarse como

```
cd ruta_de_modulos
./loadmodules.sh
```

Una vez hecho esto, la aplicación puede ser ejecutada tal a como se explicó antes, funcionando de la misma manera.

B.2.4 Profiling

En ocasiones se busca estudiar el comportamiento de la aplicación desarrollada en cuanto a llamadas a funciones y tiempos de ejecución para cada una. Para esto se hace un *profiling* del ejecutable, lo que requiere que ésta se compile y ejecute de una manera específica.

Se recomienda realizar este procedimiento sólo para estudiar el comportamiento de la aplicación, y no para las pruebas de reconocimiento dado que a la hora de compilar se introducen secciones de código adicionales que se encargan de la recolección de información de las funciones involucradas, lo que hace que la ejecución sea más lenta. Además, sólo puede ser utilizado en aplicaciones que tengan un final de ejecución con un *exit()*, o con un *return* desde la función principal (main), por lo que en este caso sólo puede realizarse con el reconocimiento sobre archivos.

El procedimiento para realizar el "profiling" es el siguiente: a la hora de compilar debe agregarse la bandera "-pg" a la hora de realizar la compilación y vinculación del ejecutable. Una vez hecho esto se genera de manera normal el archivo ejecutable, que luego se transfiere y ejecuta en el sistema embebido. Al hacer esto último se generará un archivo de nombre *gmon.out*; el archivo que contiene la información buscada se genera entonces ejecutando el comando

```
gprof nombre_exec gmon.out > nombre_archivo_analisis.txt
```

El archivo de texto será generado en la misma carpeta del ejecutable, y se recomienda transferirlo al computador de propósito general para facilitar su lectura. Este contiene información sobre número de veces que se llamó a una función determinada, tiempo de

ejecución acumulativo y por llamada individual y tiempo de ejecución total, entre otros parámetros. Mediante esto pueden ubicarse las secciones de código críticas, con el fin de optimizarlas.

Apéndice C

Configuración del entorno de desarrollo y del sistema embebido

Con el fin de poder compilar y ejecutar de manera adecuada la aplicación en el sistema embebido es necesario realizar ciertas configuraciones específicas en el sistema anfitrión y en el sistema embebido. Acá se presentan los detalles generales para esto, pero de realizar modificaciones en la implementación podrían requerirse paquetes o comandos adicionales.

Se asume que el computador anfitrión ejecuta la distribución de Linux *Ubuntu*, y que en el sistema embebido se ejecuta la distribución *Angstrom*. Comandos elementales son omitidos, mostrando sólo los más relevantes.

C.1 Configuración del entorno de desarrollo

Para llevar a cabo el desarrollo de la aplicación, así como su posterior compilación, se requiere instalar diversas aplicaciones y bibliotecas. Además se requiere configurarlas para que el funcionamiento en el sistema embebido sea el esperado.

C.1.1 Instalación de aplicaciones y bibliotecas

La instalación de aplicaciones en el computador de desarrollo o anfitrión se realiza mediante el comando

```
sudo apt-get install nombre_paquete
```

Lo cual requiere introducir la contraseña del administrador para realizar la instalación.

El sistema operativo incluye la mayor parte de las bibliotecas requeridas para realizar la compilación, y usualmente sólo se requiere instalar las siguientes: *gcc*, *g++*, *jack*, *jack-dev*, *jack-server* y *libjack*.

En cuanto a aplicaciones, deben instalarse las siguientes: *eclipse* y *minicom*. Puede incluirse también *audacity*, la cual permite modificar los archivos de audio para recortarlos, cambiar la razón de muestreo, entre otras acciones.

En ciertos casos podrían requerirse más bibliotecas y/o aplicaciones, por lo que debe verificarse cualquier mensaje que sea desplegado en pantalla.

C.1.2 Configuración de aplicaciones

Configuración de aplicación Eclipse

Lo primero que debe configurarse es el entorno de desarrollo *Eclipse*, lo cual se hace por medio de la detallada guía presentada en [5]. Sin embargo, deben realizarse ciertas modificaciones al procedimiento. Para la parte 1, se debe utilizar el *toolchain* descargado junto a los archivos del sistema operativo, para lo que debe verse primero la subsección de instalación de éste en el sistema embebido. Esto se requiere para garantizar que las aplicaciones sean compiladas con las bibliotecas correctas.

Además, los archivos contenidos en el *toolchain* deben ser colocados en una ruta conveniente y que facilite la posterior configuración de la herramienta C6Run, por lo que se recomienda leer primero la siguiente subsección. Así, debe modificarse la parte 3 para que se apunte a los directorios adecuados.

Finalmente, deben agregarse dos parámetros requeridos para que la aplicación sea vinculada correctamente. Estas son *-ljack* y *-lsoundfile*, que deben colocarse al final de la línea *Command line pattern* en la sección GCC C LINKER, la cual es configurada también en la parte 3.

Configuración de aplicación minicom

La configuración de la aplicación *minicom* se explica con detalle en el siguiente enlace

```
http://elinux.org/BeagleBoardBeginners#First_interaction_with_the_board
```

El único detalle es que debe determinarse el puerto con el que se vinculó el sistema embebido, lo que puede revisarse con los comandos

```
cd /dev
ls
```

Esto muestra los dispositivos de hardware asociados con el sistema operativo. En caso que la conexión sea directamente mediante un cable serial, el nombre del dispositivo sería *ttySX*, mientras que si se hace con un cable USB a serial debería ser *ttyUSBX*. Por ejemplo, el nombre del dispositivo podría ser *ttyUSB0*.

Si bien se indica en el sitio que la conexión puede realizarse utilizando *screen*, debe notarse que por cuestiones de compatibilidad este método no puede ser utilizado al realizar la conexión mediante el cable USB a serial; al menos en el sistema embebido utilizado. De intentarlo, no se lleva a cabo la conexión.

C.1.3 Configuración e instalación de C6Run

La herramienta C6Run permite compilar el ejecutable que se ejecuta utilizando el procesador ARM y el DSP, generando las bibliotecas y el código necesario para ejecutar funciones en este último. Para esto se requiere de la instalación de otras herramientas de Texas Instruments, además del propio C6Run. El siguiente procedimiento explica cómo instalar todos los elementos necesarios.

Se descarga el contenedor con los archivos usados por C6Run desde el enlace

```
https://gforge.ti.com/gf/project/dspeasy/frs/
```

Al momento de escribir esta guía la última versión era la 0.98.03.03, pero se utilizó la versión 0.97.03.03 dado que la primera no compilaba los archivos adecuadamente para el sistema embebido.

Además, debe descargarse la última versión de la herramienta de generación de código para el DSP desde el enlace

```
https://www-a.ti.com/downloads/sds_support/TICodegenerationTools/download.htm
```

Esto requiere generar una cuenta en el sitio web de Texas Instruments previamente.

Una vez descargados los archivos, deben ser extraídos en una ubicación adecuada; para ejemplo se utilizará la ruta */home/user/toolchains*. Para el contenedor de C6Run basta con realizar una extracción con el comando

```
tar -xzf C6Run_X_XX_XX_XX.tar.gz
```

Donde se extraen los archivos en la misma carpeta del contenedor, por lo que debe ubicarse en la ruta mencionada.

El otro archivo utiliza un instalador, por lo que debe establecerse la bandera de ejecución. Los comandos son, en este caso,

```
chmod +x ti_cgt_c6000_X.X.X_setup_linux_x86.bin
```

```
ti_cgt_c6000_X.X.X_setup_linux_x86.bin --mode silent --installto /home/user/toolchains
```

Una vez hecho esto, se ingresa a la carpeta generada al extraer los archivos de C6Run, y se busca el archivo de texto llamado *Rules.mak*. En este deben modificarse ciertas líneas con el fin de establecer las versiones de los paquetes a utilizar, y las rutas donde se ubicarán. Al editarlas deben quedar como sigue

```
LINUXUTILS_VERSION=2_25_05_11
CODEGEN_INSTALL_DIR     ?= $(HOME)/toolchains/TI_CGT_C6000_X.X.X
ARM_TOOLCHAIN_PATH      ?= $(HOME)/toolchains/arm-angstrom
ARM_TOOLCHAIN_PREFIX    ?= arm-angstrom-linux-gnueabi-
```

Donde se supone que las bibliotecas, o el kit de desarrollo, para el sistema embebido se han almacenado también en la ruta de trabajo (ver sección de configuración para Eclipse).

Una vez hecho esto, se accede a la carpeta de C6Run desde la terminar y se ejecuta el comando

```
make get_components
```

Éste se encarga de descargar todas las dependencias requeridas por la herramienta. Una vez que finaliza, se accede a la carpeta *platforms/beagleboard-xM*, se abre el archivo *platform.mak* y se comentan las líneas ARM_TOOLCHAIN_PATH y ARM_TOOLCHAIN_PREFIX, lo que se hace colocando un "#" antes de su declaración.

Con el fin de mejorar el rendimiento de la aplicación compilada para el DSP, se le indica al compilador que realice optimizaciones adicionales en el ejecutable, y que genere un reporte con las instrucciones de ensamblador utilizadas. Esto se hace agregando las banderas *-mw* y *-pm* en el parámetro DSP_CFLAGS del archivo *environment.sh*. Habiendo hecho esto, se regresa a la carpeta principal de C6Run y se finaliza la instalación ejecutando los siguientes comandos

```
make beagleboard-xM_config
make gpp_libs dsp_libs
```

Si se desea generar ejecutables que ejemplifican el uso de la herramienta, deben usarse los comandos

```
source environment.sh
make examples tests
```

Los archivos generados estarán en las carpetas *examples* y *test*. Estos pueden ser transferidos al sistema embebido para verificar que la herramienta funciona de manera correcta.

C.1.4 Makefile y scripts de compilación

Para realizar la compilación del ejecutable mediante el comando *make*, es necesario generar y configurar primero el *makefile* asociado. Junto con el proyecto realizado se incluye uno con todos los parámetros ya configurados tal a como se requiere, pero en caso de que éste no exista, en el siguiente vínculo se explica cómo puede descargarse un ejemplo y cual es el efecto de cada línea del archivo al momento de compilar

```
http://elinux.org/EBC_Exercise_09_Using_the_DSP_for_Audio_Processing
```

Con el fin de adaptar el archivo de ejemplo para el proyecto actual, usualmente lo que se debe hacer es colocar los nombres correctos para los archivos fuente (.c), verificando cuales de estos serán transferidos al DSP. Además, deben agregarse los parámetros *-ljack* y *-lsndfile* en ARM_LDFLAGS, por lo que esta línea quedaría como

```
ARM_LDFLAGS +=-lm -lpthread -ljack -lsndfile
```

También debe eliminarse el parámetros *–C6Run:replace_malloc*, dado que por la forma en que se realizó la implementación de la aplicación no se desea el cambio de todas las funciones de solicitud de memoria por las propias de C6Run. Finalmente, debe cambiarse el nombre de los ejecutables y bibliotecas generadas por nombres más convenientes.

Si se desea realizar el *profiling* de la aplicación, deben agregarse banderas en ARM_CFLAGS y ARM_LDFLAGS, lo que debe hacerse introduciendo los siguientes comandos *en la línea siguiente* a su declaración original

```
ARM_CFLAGS +=-pg
ARM_LDFLAGS +=-pg
```

C.2 Configuración del sistema embebido

El sistema embebido debe ser preparado con el fin de ejecutar las aplicaciones desarrolladas. Esto requiere la instalación del sistema operativo, *Angstrom* en este caso, y de bibliotecas y módulos para el kernel de Linux necesarios para ejecutarlas correctamente.

C.2.1 Partición de memoria e instalación del sistema operativo

El proceso de instalación del sistema operativo se puede consultar de manera detallada en [51], explicando el proceso paso a paso. Solo deben realizarse dos correcciones en los pasos mostrados. La primera es que debe sustituirse el comando

```
tar --wildcards -xjvf [YOUR-DOWNLOAD-FILE].tar.bz2 ./boot/*
```

por el comando

```
tar --wildcards -xzvf [YOUR-DOWNLOAD-FILE].tar.gz ./boot/*
```

Este comando es el que permite extraer los archivos de arranque del sistema operativo desde el contenedor o archivo comprimido, el cual es generado por el sitio web Narcissus. El cambio del comando se debe a este sitio utilizaba antes el formato bzip, mientras que ahora se utiliza gzip.

El segundo comando a cambiar es el que se utiliza para extraer los elementos que componen el sistema de archivos, dado por

```
sudo tar -xvj -C /media/Angstrom -f [YOUR-DOWNLOAD-FILE].tar.bz2
```

que debe ser sustituido por

```
sudo tar -xvz -C /media/Angstrom -f [YOUR-DOWNLOAD-FILE].tar.gz
```

lo cual se debe hacer debido a los cambios de formato previamente mencionados.

Si se sigue el tutorial en su totalidad, se generarán las particiones requeridas para el arranque y la del sistema de archivos, y se instalará además el sistema operativo. Así se está listo ya para utilizarlo en el sistema embebido.

C.2.2 Instalación de paquetes y bibliotecas

Al realizar la instalación del sistema operativo (SO) descrita en el paso previo ya se tienen la mayor parte de los paquetes y bibliotecas instaladas y actualizadas. Sin embargo, se recomienda realizar una actualización mediante los comandos

```
opkg update
mkdir /home/root/tmp
opkg -t /home/root/tmp upgrade
```

donde la carpeta creada puede tener cualquier otro nombre o ubicarse en otra ruta según convenga.

Se completan los paquetes faltantes usando el comando

```
opkg install nombre_paquete
```

En este caso, los paquetes que usualmente son requeridos por el sistema embebido son *jack*, *jack-dev*, *jack-server*, *libjack* y *alsa-utils*, pero podrían requerirse más de acuerdo a la configuración elegida al descargar el sistema operativo. Verificar cualquier mensaje desplegado al ejecutar la aplicación para verificar los paquetes faltantes.

Los paquetes actualmente instalados pueden ser consultados con el comando

```
opkg list-installed
```

Si se desea limitar los resultados a paquetes con un nombre específico, o con ciertos caracteres en su nombre, puede usarse

```
opkg list-installed | grep Nombre_o_caracteres
```

C.2.3 Configuración de jack y ALSA

Al iniciar el cliente de jack desde la aplicación del SiRPA, este configura sus parámetros en base a la configuración almacenada tanto para ALSA como para el servidor JACK, por lo que ambos deben ser configurados previamente.

Para el caso del ALSA, debe ejecutarse el siguiente comando

```
alsamixer
```

Lo que abre una ventana como la mostrada en la figura C.1. Se recomienda hacer esta configuración mediante *ssh*, pues si se hace mediante la conexión serial la ventana se muestra distorsionada.

Deben configurarse acá los siguientes parámetros, con el fin de que la captura de muestras desde la entrada de línea sea adecuada

```
Left Digital Loopback: Mute
Right Digital Loopback: Mute
DAC2 Analog: 6dB
DAC2 Digital Coarse: 6dB
DAC2 Digital Fine: -10dB
```

Una vez realizada la configuración, esta debe ser almacenada para hacerla permanente, pues de lo contrario debe reconfigurarse al reiniciar el sistema embebido. Para esto se ejecuta el comando

```
alsactl store
```

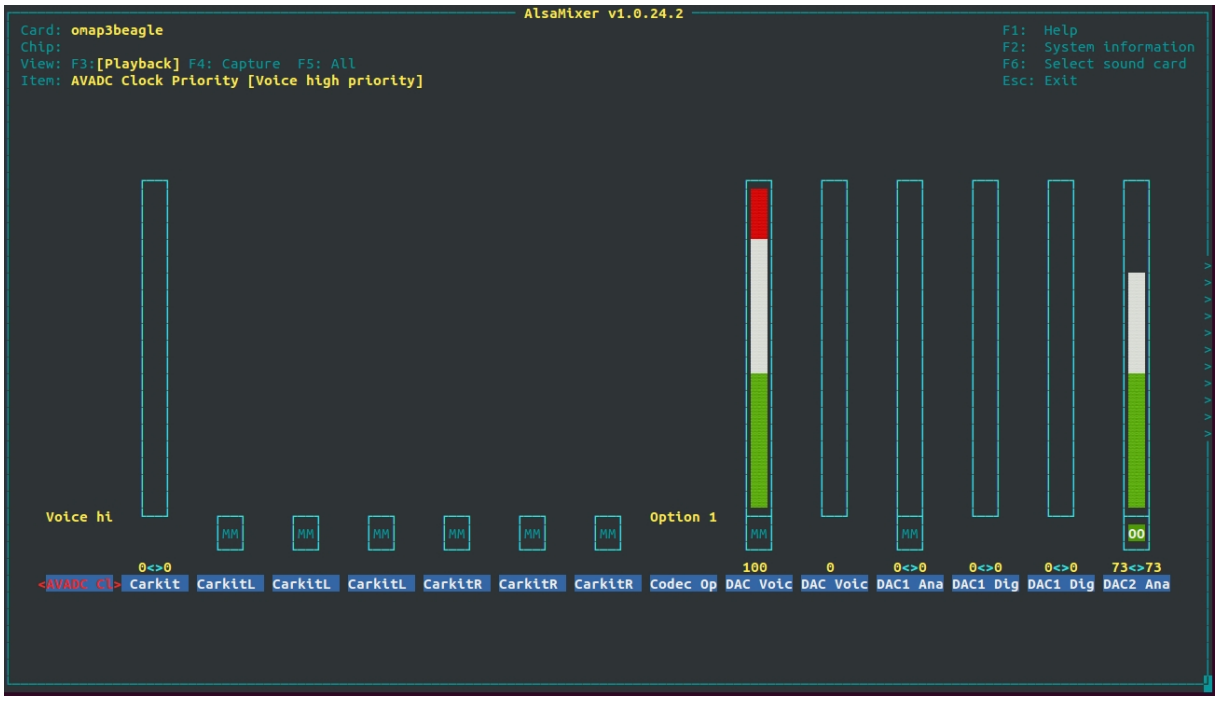


Figura C.1: Ventana de configuración para ALSA.

Para el caso de JACK, la configuración se hace ejecutando el siguiente comando

```
jackd -d alsa -p 256 -n 4 -P hw:0 -C hw:0 -S -r 48000 &amp;amp;
```

Este indica que se utiliza ALSA como *driver* de hardware, se trabaja con un tamaño de bloque de 256 muestras con 4 buffer diferentes, se utiliza el dispositivo hw:0 para captura de datos y reproducción de audio, y se utiliza una frecuencia de muestreo de 48 kHz. Las otras banderas modifican el comportamiento del cliente, y se puede consultar su significado en [29]. Una vez hecha esta configuración, la aplicación debe funcionar de manera correcta, aunque podría requerirse reiniciar el sistema.

C.2.4 Configuración para uso del DSP

Instalación de módulos

Con el fin de utilizar el DSP contenido en el SoC del sistema embebido (Beagleboard-xM) es necesario cargar módulos para el kernel encargados de la comunicación entre procesadores y de la conversión entre los punteros a memoria usados por cada uno de ellos. Este proceso requeriría compilar los módulos, escritos en C, utilizando una copia del último kernel disponible. Esto se hace por compatibilidad y para lograr una ejecución correcta.

Para evitar esto se utilizan módulos precompilados, los cuales son adecuados para la versión del sistema operativo *Angstrom* utilizado. Estos pueden ser descargados desde el

vínculo

```
http://elinux.org/images/f/fa/C6run_m_yoder.tgz
```

Una vez descargado este contenedor y extraídos los archivos en él, deben extraerse los archivos para el sistema embebido desde el archivo *c6run_target.tar.gz*. Los módulos necesarios estarán entonces en la carpeta

```
.../home/daniel/c6run_target
```

Específicamente, deben copiarse los archivos *cmemk.ko*, *dsplinkk.ko*, *lpm_omap3530.ko*, *loadmodules.sh* y *unloadmodules.sh*. Estos pueden ser colocados en cualquier lugar conveniente dentro del sistema de archivos del sistema embebido.

Argumentos de arranque y archivo uEnv.txt

Además de la instalación de los módulos, es necesario indicarle al sistema operativo a la hora del arranque sobre ciertos parámetros de uso de la memoria RAM disponible en el sistema embebido. Para hacer esto se utilizan los denominados *bootargs*, que corresponden a parámetros de configuración del SO que son cargados a la hora del arranque. Estos son leídos desde la memoria NAND del sistema embebido o desde un archivo ubicado en la partición de arranque, que debe nombrarse **uEnv.txt**.

Esto debe hacerse dado que la memoria compartida por el DSP y por el procesador ARM se ubica en un espacio específico del mapa de memoria, por lo que debe evitarse que esta sea administrada por el sistema operativo, y en su lugar es administrada por el módulo *cmem*. De lo contrario, los datos ubicados en este segmento de memoria podrían ser corrompidos al accederlos, o podrían pasarse punteros a datos inválidos.

En este caso se utiliza el método del archivo *uEnv.txt*, dado que el sistema embebido utilizado no tiene memoria NAND para almacenar los argumentos de arranque. El procedimiento consiste en generar un archivo de texto con este nombre en la partición *boot* de la tarjeta de memoria del sistema embebido. En este archivo debe colocarse la siguiente línea

```
optargs=mem=99M@0x80000000 mem=384M@0x88000000
```

La cual indica que el SO debe utilizar dos segmentos de memoria; uno de 99 MiB iniciando en la dirección 0x80000000, y otro de 384 MiB iniciando en 0x88000000. La memoria restante es utilizada para compartir datos entre el DSP y el procesador ARM. El espacio utilizado en este último caso inicia en la dirección 0x86300000 y tiene un tamaño de 16 MiB.