Universidad Zaragoza
1542

# Trabajo Fin de Máster

## Localización de fuentes sonoras mediante agrupaciones de micrófonos

## Sound source localization with microphone arrays


Autor

David Díaz Guerra Aparicio

Director

José Ramón Beltrán Blázquez

ESCUELA DE INGENIERÍA Y ARQUITECTURA


2017

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

# DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

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

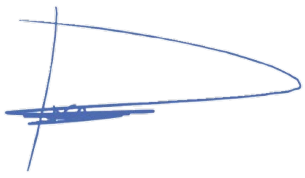
D./Dª. David Díaz-Guerra Aparicio,

con nº de DNI 73003712X en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Máster, (Título del Trabajo) Localización de fuentes sonoras mediante agrupaciones de micrófonos (Sound source localization with microphone arrays),

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

Zaragoza, 2 de febrero de 2017



Fdo: David Díaz-Guerra Aparicio

# Localización de fuentes sonoras mediante agrupaciones de micrófonos

## Resumen

El procesado de señal de agrupaciones de sensores es un tema ampliamente estudiado dentro del ámbito del procesado de señal, y que, pese a llevarse investigando casi medio siglo, todavía hoy sigue atrayendo el interés de la comunidad investigadora. Dentro de los distintos tipos de sensores resultan de especial interés las agrupaciones de micrófonos debido a sus diferentes ámbitos de aplicación, como la videoconferencia o el análisis acústico de entornos.

Este trabajo se plantea como una introducción a las técnicas de estimación de ángulo de llegada (DOA) en agrupaciones de micrófonos. En él, tras explicar la motivación y alcance del proyecto y una pequeña introducción al procesado de señal de agrupaciones, se estudian las distintas técnicas existentes a día de hoy para a continuación presentar 2 implementaciones en tiempo real. La primera, montada sobre un PC y usando micrófonos y equipos profesionales, sirve para demostrar las capacidades de este tipo de técnicas, mientras que la segunda, buscando reducir tamaños y costes, está más próxima a las necesarias en aplicaciones reales. Además, junto a la primera implementación se aborda el hecho de representar espacialmente, de manera conjunta, los resultados obtenidos mediante las técnicas de DOA con la imagen obtenida de una cámara de vídeo, y junto a la segunda, se presenta una técnica que permite combinar la información obtenida por diversas agrupaciones pequeñas para formar una red de agrupaciones de micrófonos.

# Índice general

**Índice de figuras** IX

**1. Introducción** **1**
1.1. Motivación y alcance del proyecto . . . . . 1
1.2. Introducción al procesado de señal de agrupaciones . . . . . 2
1.2.1. Modelo de señal . . . . . 2
1.2.2. Conformado de haz . . . . . 3
1.2.3. Agrupaciones de banda estrecha . . . . . 4
1.2.4. Agrupaciones circulares . . . . . 5

**2. Técnicas de estimación de ángulo de llegada** **7**
2.1. Técnicas basadas en *beamforming* . . . . . 7
2.1.1. SRP-D&S . . . . . 7
2.1.2. Método de Capon . . . . . 8
2.2. Técnicas basadas en técnicas de estimación espectral de alta resolución 9
2.3. Técnicas basadas en diferencias de tiempos de llegada . . . . . 10
2.4. El algoritmo SRP-PHAT . . . . . 10

**3. Implementación software** **13**
3.1. Representación del vídeo y el algoritmo SRP . . . . . 13
3.2. Hardware . . . . . 14
3.2.1. Micrófonos . . . . . 14
3.2.2. Conversión Analógico/Digital . . . . . 15
3.2.3. Vídeo . . . . . 15
3.3. Software . . . . . 15
3.3.1. Algoritmo . . . . . 16
3.3.1.1. Interfaz con la aplicación . . . . . 16
3.3.1.2. Correlaciones cruzadas . . . . . 17
3.3.1.3. Promediado del mapa de potencia . . . . . 17
3.3.1.4. Debug . . . . . 17
3.3.2. Aplicación . . . . . 17
3.3.2.1. Compensación del retardo entre tarjetas . . . . . 18
3.3.2.2. Llamada al algoritmo . . . . . 18
3.3.2.3. Gestión del vídeo . . . . . 18
3.3.2.4. Interfaz Gráfica de Usuario . . . . . 18
3.4. Resultado . . . . . 19

**4. Diseño de una red de sensores** **21**
4.1. Determinación de la posición de la fuente . . . . . 21
4.2. Hardware . . . . . 23
4.2.1. Micrófonos . . . . . 23
4.2.2. Acondicionado de las señales . . . . . 24
4.2.3. Microprocesador . . . . . 24
4.2.4. Otros . . . . . 25
4.3. Software . . . . . 25
4.3.1. Gestión del ADC . . . . . 25
4.3.2. Algoritmo . . . . . 26
4.4. Resultado . . . . . 27

**5. Conclusiones y lineas futuras** **29**

**Bibliografía** **31**

**A. Código de la implementación software** **I**

**B. Esquemático de la agrupación de 4 micrófonos** **XXIII**

**C. PCB de la agrupación de 4 micrófonos** **XXVII**

**D. Código del programa del microprocesador** **XXXIII**

# Índice de figuras

1.1. Sistema acústico diseñado para localizar aviones durante la Primera Guerra Mundial . . . . . 2
1.2. Geometría de una grupación circular . . . . . 5

2.1. Mapas de potencia estimados mediante SRP-D&S en entornos con y sin reverberación . . . . . 8
2.2. Mapas de densidad de potencia estimados mediante el método de Capon en entornos con y sin reverberación . . . . . 9
2.3. Mapas de potencia estimados mediante SRP-PHAT en entornos con y sin reverberación . . . . . 12

3.1. Agrupación de micrófonos utilizada para la implementación en tiempo real y para las grabaciones usadas para el estudio de los diversos algoritmos presentados en el capítulo 2. . . . . 14
3.2. Interfaz gráfica de usuario . . . . . 19

4.1. Placa de desarrollo SPW-2430 del micrófono utilizado . . . . . 23
4.2. Esquemático del circuito analógico para el acondicionado de la señal de un micrófono . . . . . 24
4.3. Placa de desarrollo CY8CKIT-142 del microprocesador PSoC 4 utilizado y tarjeta BLE Pionner Kit usada para programarlo. . . . . 25
4.4. Prototipo fabricado . . . . . 27

C.1. Cara superior de la PCB diseñada para la agrupación de 4 sensores . XXIX
C.2. Cara inferior de la PCB diseñada para la agrupación de 4 sensores . . XXXI

# 1

# Introducción

## 1.1. Motivación y alcance del proyecto

El procesado de señales de agrupaciones de sensores es un tema ampliamente estudiado dentro del ámbito del procesado de señal, y sobre el que existe una amplia y consolidada bibliografía de referencia [1], [2]. Esta disciplina incluye distintas técnicas, como la síntesis de diversos patrones de directividad, conocida como *beamforming*, o la estimación de dirección de llegada [3] (DOA por sus siglas en ingles), que es de la que trata este trabajo. Si bien muchas de estas técnicas se basan en modelos de señal teóricamente válidos para señales obtenidas por diferentes tipos de sensores, en muchos casos existen suficientes particularidades como para que un tipo de sensores constituya una extensa área de investigación, como es el caso de las agrupaciones de micrófonos [4].

Sin embargo, pese a que se trata de un tema sobre el que se lleva investigando casi medio siglo [5], hoy en día sigue atrayendo la atención de la comunidad investigadora debido a sus múltiples aplicaciones y los diversos problemas que aun quedan por resolver, y se siguen publicando artículos en revistas de impacto sobre él. Por ejemplo, en la edición de enero de 2017 de la revista *IEEE Signal Processing Letters* aparecen 3 artículos sobre estimación de DOA [6], [7], [8], y centrándonos en las agrupaciones de micrófonos, también podemos encontrar artículos en las principales publicaciones sobre acústica durante el año 2016, como *Acta Acustica united with Acustica* [9], [10], [11], [12], o en la conferencia de la *Audio Engineering Society* de septiembre de 2016 [13], [14], [15].

Este trabajo se centra en la estimación de dirección de llegada (DOA), en la que se busca estimar la dirección de la que proviene uno o varios frentes de onda que inciden sobre la agrupación. Esto tiene diversas aplicaciones, como la localización de fuentes sonoras (SSL por sus siglas en ingles), que resulta de interés, por ejemplo, en entornos de videoconferencia [17], o la caracterización de espacios acústicos [18], [19], dónde lo que se busca localizar no es la fuente original del sonido sino las reflexiones que genera una fuente conocida.

Con este trabajo se realiza una introducción a la estimación de dirección de llegada en agrupaciones de micrófonos. En concreto, se realiza un estudio de las diversas técnicas utilizadas (capítulo 2) y dos implementaciones en tiempo real, primero una sobre un PC y usando micrófonos y equipos profesionales en la que los recursos hardware no suponen, apenas, una limitación, y permite demostrar las posibilidades de esta tecnología (capítulo 3), y, a continuación, una implementación pensada para formar una red de sensores usando un microprocesador, micrófonos y

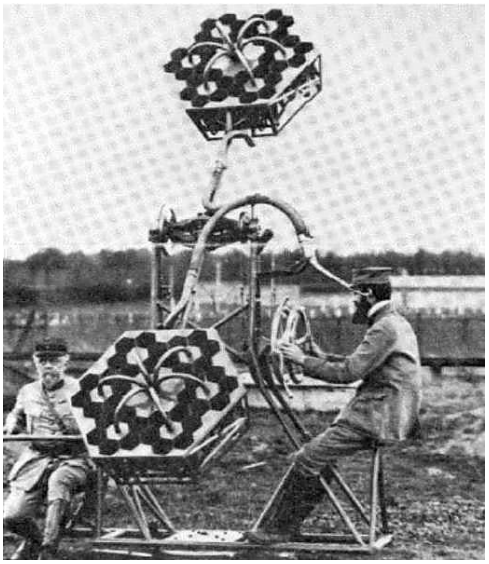


**Figura 1.1:** Sistema acústico diseñado por el premio nobel de física Jean Baptiste Perrin (Francia 1870–1942) para localizar aviones durante la Primera Guerra Mundial [16]

componentes electrónicos de bajo coste (capítulo 4). Por último se deducen algunas conclusiones del proyecto y se plantean los retos pendientes de estas tecnologías y posibles líneas futuras de investigación (capítulo 5). Además, en el capítulo 3, se aborda el problema de sincronizar espacialmente las mediciones de potencia acústica realizadas mediante la agrupación con la imagen obtenida de una cámara de vídeo y, en el capítulo 4, se presenta una técnica que permite combinar la estimación realizada por varias agrupaciones para lograr una estimación más exacta y completa.

## 1.2. Introducción al procesado de señal de agrupaciones

### 1.2.1. Modelo de señal

En el estudio de agrupaciones de sensores se supone que el efecto de la propagación de las ondas generadas por las distintas fuentes puede modelarse como un sistema lineal caracterizado por su respuesta impulsional. De esta forma, la señal captada por el sensor $n$ en un entorno con $M$ fuentes sonoras en posiciones $\theta_m$ será:

$$x_n(t) = \sum_{m=1}^{M} a_m(t) * h_n(\theta_m, t) + v_n(t) \tag{1.1}$$

donde $a_m(t)$ es la señal emitida por la fuente m-ésima y $v_n(t)$ es un ruido aditivo (generalmente modelado como blanco y gaussiano) incorrelado con las fuentes y con los ruidos de los otros sensores. La respuesta impulsional de la propagación $h_n(\theta_m, t)$ desde la fuente $m$ hasta el sensor $n$ puede descomponerse en un termino de atenuación $1/r_n(\theta_m)$, un retardo de propagación $\tau_n(\theta_m)$ y el efecto de la reverberación de la sala $h'_n(\theta_m, t)$:

$$x_n(t) = \sum_{m=1}^{M} \frac{1}{r_n(\theta_m)} a_m(t - \tau_n(\theta_m)) * h'_n(\theta_m, t) + v_n(t) \tag{1.2}$$

o expresado en el dominio frecuencial:

$$X_n(\omega) = \sum_{m=1}^{M} \frac{1}{r_n(\theta_m)} A_m(\omega) H'_n(\theta_m, \omega) e^{j\omega\tau_n(\theta_m)} + V_n(\omega) \tag{1.3}$$

Generalmente se supone que las fuentes están a una distancia muy superior al tamaño de la agrupación, por lo que se trabaja bajo modelos de frente de onda plano en cuyo caso todos los términos $1/r_n(\theta_m)$ son iguales y pueden despreciarse al no aportar información. También es muy común, sobre todo en agrupaciones de banda estrecha, despreciar el termino de reverberación, sin embargo, en el caso de las agrupaciones de micrófonos, los entornos suelen ser muy reverberantes, lo que degrada las prestaciones de la mayoría de algoritmos.

### 1.2.2. Conformado de haz

Si bien este trabajo se centra en la estimación de dirección de llegada (DOA), resulta conveniente presentar los fundamentos del conformado de haz (o *beamforming*), ya que muchas técnicas de DOA se basan en él. El *beamforming* supone un filtrado espacial, en el que la señal de cada sensor se filtra temporalmente (o se multiplica por una constante compleja en el caso de agrupaciones de banda estrecha) y se suman.

La técnica de *beamforming* más simple, que además es la que maximiza la relación señal a ruido en un escenario con una única fuente y ruido omnidireccional, es la denominada *delay-and-sum*, que consiste en compensar los retardos sufridos por la propagación para la dirección $\theta_0$ a la que se desea apuntar con la agrupación, de forma que las señales provenientes de ésta se sumen en fase:

$$Y(\omega) = \sum_{n=1}^{M} X_n(\omega) e^{-j\omega\tau_n(\theta_0)} \tag{1.4}$$

Esta técnica puede generalizarse a las *filter-and-sum* mediante el filtrado de las señales de cada sensor:

$$Y(\omega) = \sum_{n=1}^{M} G_n(\omega) X_n(\omega) e^{-j\omega\tau_n(\theta_0)} \tag{1.5}$$

donde los filtros $G_n(\omega)$ se diseñan siguiendo distintas técnicas en función del diagrama de directividad que se desee sintetizar. Cabe destacar que, además de técnicas deterministas, existen técnicas adaptativas que buscan maximizar o minimizar alguna propiedad de la señal de salida $Y(\omega)$ en función de las señales $X_n(\omega)$ que captura la agrupación.

### 1.2.3. Agrupaciones de banda estrecha

Aunque la señal de audio con la que trabajan las agrupaciones de micrófonos son de banda ancha, resulta conveniente conocer los principios de funcionamiento de las agrupaciones de banda estrecha, ya que muchas técnicas de banda ancha de basan en ellos.

Se considera como agrupaciones de banda estrecha aquellas en las que su ancho de banda es mucho menor que su frecuencia central, por lo que las variaciones temporales de su envolvente son despreciables frente a la fase de la portadora y por tanto se puede considerar que las diferencias en los tiempos de propagación de las señales hasta cada sensor solo afectan a la fase de la portadora. De esta forma, el equivalente paso bajo de la señal captada por el sensor $n$ será:

$$\hat{x}_n(t) = \sum_{m=1}^{M} \hat{a}_m(t) e^{j\omega\tau_n(\theta_m)} + \hat{v}_n(t) \tag{1.6}$$

donde $\hat{a}(t)$ es el equivalente paso bajo de la señal generada por la fuente y se ha despreciado el efecto de la reverberación del entorno.

En el procesado de señales de banda estrecha suele trabajarse con notación vectorial, ya que con ella se consiguen expresiones más compactas. Para esto, se define el vector $\mathbf{x}(t)$ a partir de las señales recibidas por cada sensor:

$$\mathbf{x}(t) = \begin{bmatrix} \hat{x}_1(t) \\ \hat{x}_2(t) \\ \vdots \\ \hat{x}_N(t) \end{bmatrix} = \mathbf{D}(\boldsymbol{\theta})\mathbf{a}(t) + \mathbf{v}(t) \tag{1.7}$$

donde los vectores columna $\mathbf{a}(t)$ y $\mathbf{v}(t)$ están compuestos a partir de las señales generadas por cada fuente y el ruido de cada sensor:

$$\mathbf{a}(t) = \begin{bmatrix} \hat{a}_1(t) & \hat{a}_2(t) & \dots & \hat{a}_M(t) \end{bmatrix}^T \tag{1.8}$$

$$\mathbf{v}(t) = \begin{bmatrix} \hat{v}_1(t) & \hat{v}_2(t) & \dots & \hat{v}_N(t) \end{bmatrix}^T \tag{1.9}$$

y la matriz $\mathbf{D}(\boldsymbol{\theta})$ modela los retardos sufridos por cada señal hasta cada sensor:

$$\mathbf{D}(\boldsymbol{\theta}) = \begin{bmatrix} e^{-j\omega\tau_1(\theta_1)} & e^{-j\omega\tau_1(\theta_2)} & \dots & e^{-j\omega\tau_1(\theta_M)} \\ e^{-j\omega\tau_2(\theta_1)} & e^{-j\omega\tau_2(\theta_2)} & \dots & e^{-j\omega\tau_2(\theta_M)} \\ \vdots & \vdots & \ddots & \vdots \\ e^{-j\omega\tau_N(\theta_1)} & e^{-j\omega\tau_N(\theta_2)} & \dots & e^{-j\omega\tau_N(\theta_M)} \end{bmatrix} \tag{1.10}$$

En estos casos los filtros de la agrupación de la expresión 1.5 se reducen simplemente a constantes complejas $\mathbf{w}$ en las que su modulo representa un factor de ganancia y su fase un retardo sobre la portadora:

$$y(t) = \sum_{n=1}^{N} w_n x_n(t) = \mathbf{w}^H \mathbf{x}(t) \tag{1.11}$$

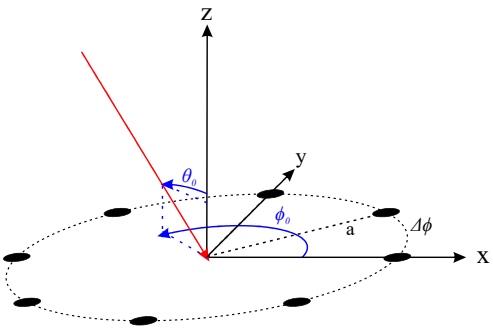



**Figura 1.2:** Geometría de una grupación circular

En el procesado señal de agrupaciones de banda estrecha conviene definir la matriz de correlación entre sensores, ya que en esta se encuentra toda la información espacial de las señales que inciden en la agrupación:

$$\mathbf{R} = E\{\mathbf{x}\mathbf{x}^H\} \tag{1.12}$$

Por último, cabe destacar que las técnicas de banda estrecha pueden aplicarse en agrupaciones de banda ancha mediante la descomposición de de la señal de cada sensor en $K$ señales de banda estrecha mediante la transformada discreta de Fourier, aplicando la técnica de banda estrecha a cada *bin* frecuencial, y combinando después el resultado mediante una transformada inversa de Fourier.

### 1.2.4. Agrupaciones circulares

Tanto en las dos sistemas implementados en tiempo real como en las simulaciones y pruebas realizadas para las distintas técnicas, se ha trabajado con agrupaciones circulares, por lo que conviene repasar antes su geometría y los retardos que sufren las señales para llegar a cada uno de los sensores.

En una agrupación circular con $N$ sensores equiespaciados en un circulo de radio $a$ situada en el plano $XY$ como la mostrada en la figura 1.2, una onda plana proveniente de la dirección $(\theta_0, \phi_0)$ llegará a los sensores $n$ y $m$ con una diferencia de tiempo:

$$\Delta\tau_{nm} = \frac{a}{c}\sin\theta_0 * [\cos(\phi_0 - \phi_n) - \cos(\phi_0 - \phi_m)] \tag{1.13}$$

donde $\phi_n$ y $\phi_m$ son las coordenadas $\phi$ de los sensores $n$ y $m$, y $c$ es la velocidad de propagación de la onda.

Puede observarse que éste sólo depende de la dirección de llegada del frente de onda y no de la distancia a la que se haya originado, por lo que bajo este modelo no será posible estimar la distancia de la fuente. Para evitar esta situación cabría suponer un modelo de propagación de ondas esféricas en lugar planas, sin embargo, excepto que la distancia de la fuente sea comparable al radio de la agrupación, apenas habrá diferencias respecto al modelo de ondas planas y la resolución en distancia será bajísima.

# 2

# Técnicas de estimación de ángulo de llegada

Tal y como se propone en [4], las técnicas de estimación de ángulo de llegada (DOA) pueden dividirse en tres grandes grupos: las basadas en buscar la dirección que maximiza la energía a la salida de un *beamformer*, técnicas basadas en conceptos de técnicas de estimación espectral de alta resolución y técnicas basadas en la estimación de diferencias de tiempos de llegada (TDOA).

## 2.1. Técnicas basadas en *beamforming*

Estas técnicas se basan en el uso de *beamforming* para filtrar las señales incidentes en la agrupación y quedarse exclusivamente con las de una dirección determinada para, a continuación, estimar su potencia. Barriendo el espacio con diferentes filtros de *beamforming* se puede obtener un mapa de potencias, de forma que la posición de las fuentes serán los máximos del mismo. A estas técnicas se las conoce como SRP (a partir de las siglas en ingles de *steered response power*).

### 2.1.1. SRP-D&S

La estrategia más básica es el uso de filtros *delay-and-sum* como los presentados en la sección 1.2.2. Estos, si bien son óptimos para un escenario con una única fuente y ruido omnidireccional, ven sus prestaciones altamente degradadas en escenarios con múltiples fuentes, ya que estas pueden incidir en la agrupación por los lóbulos secundarios de la misma. Además, en el caso de las agrupaciones de micrófonos, los entornos suelen ser altamente reverberantes, por lo que, aunque solo haya una fuente, las distintas reflexiones de la misma harán que el sistema pierda resolución. Este efecto puede observarse en la figura 2.1, donde, si bien el desapuntamiento en el resultado obtenido con la señal real probablemente sea debido a que el altavoz no estaba situado exactamente en frente de la agrupación, puede observarse cómo el nivel de ruido omnidireccional es mucho mayor (e incluso presenta máximos locales bastante pronunciados) que la de la señal simulada sin reverberaciones, a pesar de tener una relación señal a ruido muy superior.

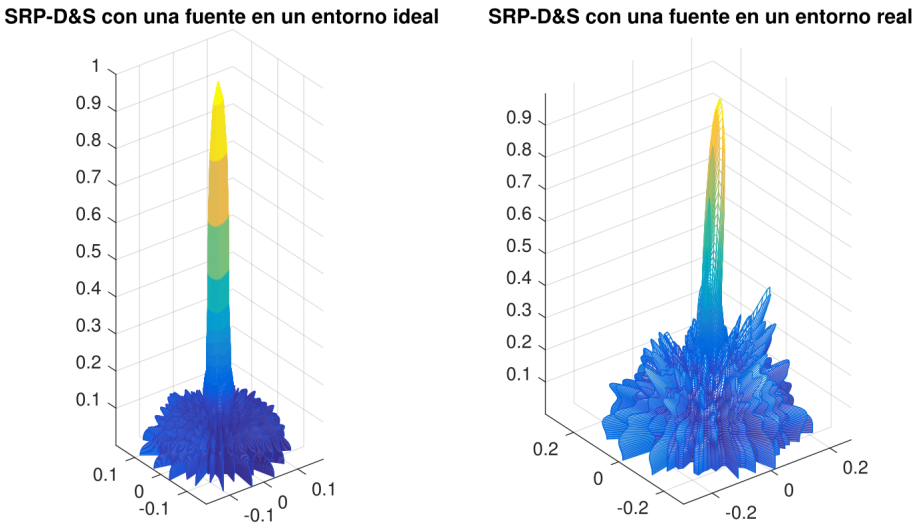



**Figura 2.1:** Mapas de potencia estimados mediante *beamforming* con filtros *delay-and-sum* en una agrupación como la descrita en el capítulo 3. El primer mapa se ha obtenido para una señal simulada de banda ancha supuesta justo en frente de la agrupación, sin tener ningún efecto de reverberación, pero con una relación señal a ruido de tan solo 0dB. El segundo mapa se ha obtenido grabando con dicha agrupación una fuente de banda ancha en una habitación sin apenas ruido pero con una fuerte reverberación.

### 2.1.2. Método de Capon

Para conseguir adaptarse al escenario presente en cada situación y mejorar el funcionamiento de estos sistemas en entornos con múltiples fuentes, existen diversos métodos estadísticos en los que los filtros o coeficientes de la agrupación se calculan en función de la señal incidente. Estos métodos suponen un nexo de unión entre las técnicas SRP basadas en *beamforming* y las basadas en estimación espectral de alta resolución.

Un ejemplo de esto es el método de Capon [20], en el que los coeficientes de una agrupación de banda estrecha se calculan de forma que se minimice la energía a la salida de la agrupación manteniendo a 1 la ganancia para la dirección a la que se desea apuntar. Con esto se consigue que los ceros del diagrama sintetizado se ubiquen en la posición del resto de fuentes, lo que permite aumentar la resolución del sistema.

Los coeficientes que cumplen con este criterio de optimización pueden expresarse en términos de la matriz de correlación (1.12) presentada en la sección 1.2.3:

$$\mathbf{w}(\theta_0) = \frac{\mathbf{R}^{-1}\mathbf{D}(\theta_0)}{\mathbf{D}^H(\theta_0)\mathbf{R}^{-1}\mathbf{D}(\theta_0)} \tag{2.1}$$

y, a partir de estos coeficientes, la densidad de potencia estimada para la dirección $\theta_0$ resulta:

$$S(\theta_0) = \frac{\mathbf{D}^H\mathbf{R}_{xx}^{-1}\mathbf{D}(\theta_0)}{\mathbf{D}^H(\theta_0)\mathbf{R}^{-2}\mathbf{D}(\theta_0)} \tag{2.2}$$

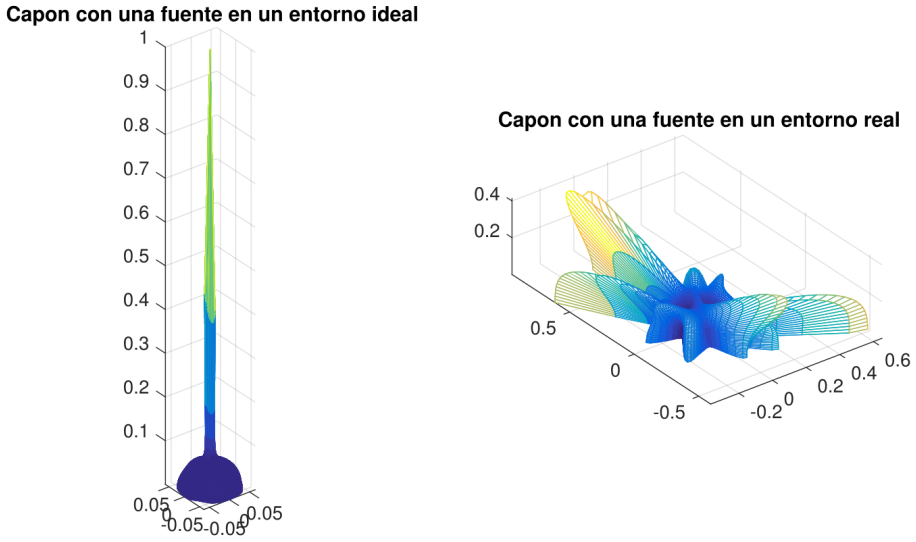



**Figura 2.2:** Mapas de densidad de potencia estimados mediante el método de Capon en una agrupación como la descrita en el capítulo 3. El primer mapa se ha obtenido para una señal simulada de banda ancha supuesta justo en frente de la agrupación, sin tener ningún efecto de reverberación, pero con una relación señal a ruido de tan solo 0dB. El segundo mapa se ha obtenido grabando con dicha agrupación una fuente de banda ancha en una habitación sin apenas ruido pero con una fuerte reverberación.

El problema de estas técnicas, además del elevado coste computacional de estimar la matriz de correlación para cada *bin* de la FFT de una señal de banda ancha y evaluar el anterior funcional para cada dirección en cada frecuencia, es que, como se aprecia en la figura 2.2, son muy poco robustas. En dicha figura se puede observar como, si bien en un entorno sin reverberación se consigue una mejor resolución y un menor nivel de ruido que con el SRP-D&S, en un entorno reverberante el resultado es completamente erróneo.

Esto hace que, si bien existen técnicas para aumentar su robustez y el método de Capon es ampliamente utilizado en agrupaciones de otro tipo de sensores, como antenas, esta técnica no suela utilizarse en agrupaciones de micrófonos.

## 2.2. Técnicas basadas en técnicas de estimación espectral de alta resolución

Al igual que el método de Capon, existen gran variedad de técnicas basadas en las propiedades estadísticas (generalmente en la matriz de autocorrelación) de las señales recibidas por la agrupación para realizar estimación de ángulo de llegada de alta resolución. Estas técnicas son, de nuevo, ampliamente utilizadas en otro tipo de agrupaciones (sobre todo las basadas en el algoritmo MUSIC [21]), pero todas tienen el problema de que son poco robustas en entornos reverberantes [22], por lo que no suelen dar buenos resultados en agrupaciones de micrófonos. Debido a la bibliografía consultada, y a la experiencia obtenida con las pruebas con el método de Capon, se optó por no implementar ninguna de estas técnicas.

## 2.3. Técnicas basadas en diferencias de tiempos de llegada

Por último, encontramos las técnicas basadas en la estimación de las diferencias de tiempos de llegada (TDOA por sus siglas en inglés). En este caso se siguen estrategias de dos pasos, primero se estiman estos tiempos y, después, se busca para qué posición se obtiene la máxima verosimilitud de los TDOA estimados.

Para la estimación temporal se suele trabajar con las funciones de correlación cruzada entre sensores, concretamente con la correlación cruzada generalizada (GCC):

$$R_{nm}(\tau) = \frac{1}{2\pi}\int_{-\infty}^{+\infty} \Psi_{nm}(\omega) X_n(\omega) X_m^*(\omega) e^{j\omega\tau} d\omega \tag{2.3}$$

En concreto, se suele usar la transformación de fase (PHAT) propuesta en [23], en la que se divide entre los modulos de las señales, de forma que estas quedan blanqueadas:

$$\Psi_{nm}(\omega) = \frac{1}{|X_n(\omega) X_m^*(\omega)|} \tag{2.4}$$

El interés de esta transformación reside en que la correlación cruzada estándar da mayor importancia a las frecuencias en las que hay un mayor contenido energético, sin embargo, para determinar el retardo entre dos señales, las zonas frecuenciales menos energéticas también aportan información relevante. Esta transformación, sin embargo, ha de usarse con precaución, ya que es inestable si las señales recibidas presentan ceros en su espectro.

Una vez estimadas las diferencias de tiempos es necesario estimar la posición de la fuente a partir de estas, lo cual supone un problema de optimización que, en función de la geometría de la fuente, no es lineal.

El principal problema de estas técnicas reside en el hecho de que en el paso de la estimación de tiempos al de posición se está perdiendo una gran cantidad de información, de forma que si el máximo de las GCC elegido no se corresponde con el verdadero TDOA, la estimación de posición será completamente incorrecta aunque las GCC tuvieran un máximo local en el TDOA correcto. A cambio, estas técnicas suelen ser las de menor coste computacional.

## 2.4. El algoritmo SRP-PHAT

En vista de las ventajas y desventajas de cada método, en [24] se propone una nueva técnica capaz de combinar la robustez de las técnicas SRP con la resolución que permite obtener la transformación PHAT y un (relativamente) bajo coste computacional.

Puede demostrase que la potencia estimada mediante las técnicas SRP puede escribirse en términos de GCCs:

$$P(\theta_0) = \int_{-\infty}^{+\infty} |Y(\omega)|^2 d\omega = \int_{-\infty}^{+\infty} |\sum_{n=1}^{M} G_n(\omega) X_n(\omega) e^{-j\omega\tau_n(\theta_0)}|^2 d\omega =$$
$$\sum_{n=1}^{N}\sum_{m=1}^{N} \int_{-\infty}^{+\infty} (G_n(\omega)X_n(\omega))(G_m(\omega)X_m(\omega))^* e^{-j\omega\tau_n(\theta_0)} d\omega =$$
$$2\pi \sum_{n=1}^{N}\sum_{m=1}^{N} R_{nm}(\Delta\tau_{nm}(\theta_0)) \quad (2.5)$$

de forma que la transformación utilizada en las GCCs es igual al producto de los filtros usados en el *beamforming* para cada señal: $\Psi(\omega) = G_n(\omega)G_m^*(\omega)$.

Este algoritmo puede implementarse con un coste computacional relativamente bajo realizando las FFTs de las señales capturadas por cada sensor, aplicando la transformación de fase normalizando el modulo de cada *bin* frecuencial, multiplicando todas con todas y realizando sus iFFTs para obtener las correlaciones cruzadas. Una vez realizados estos cálculos, para estimar la potencia de cada dirección, basta con sumar la componente adecuada de cada GCC. La componente a utilizar estará determinada por $\Delta\tau_{nm}(\theta_0)$, que está definido en tiempo continuo; algunos autores proponen realizar un interpolado lineal de las GCCs para obtener un valor más exacto [25] pero en las pruebas realizadas con agrupaciones circulares se han obtenido resultados satisfactorios quedándonos con la muestra más cercana sin realizar ningún tipo de interpolación.

En la figura 2.3 se puede observar como, tanto para el caso de señales simuladas sin reverberación como para el de señales reales, se obtiene una cierta mejora en resolución y ruido respecto al resultado obtenido con el algoritmo SRP-D&S de la figura 2.1 pese a que ambas se habían realizado usando ruido blanco como fuente, que no es el escenario en el que se saca mayor partido de la transformación de fase.

En cualquier caso, lo más interesante de este algoritmo es su implementación computacionalmente eficiente basada en el cálculo de GCCs mediante FFTs, y es el motivo por el que se ha elegido esta técnica para realizar la estimación de DOA en los capítulos 3 y 4.

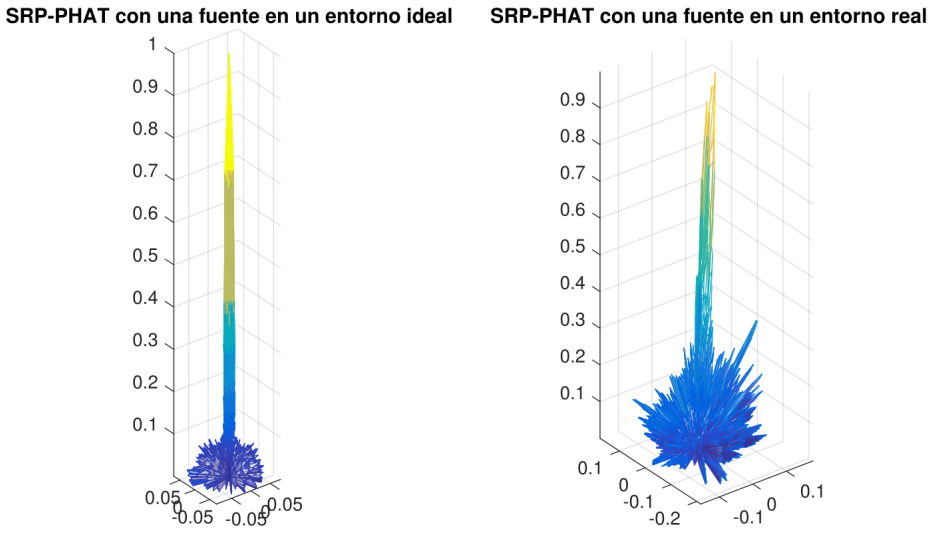



**Figura 2.3:** Mapas de potencia estimados mediante el algoritmo SRP-PHAT en una agrupación como la descrita en el capítulo 3. El primer mapa se ha obtenido para una señal simulada de banda ancha supuesta justo en frente de la agrupación, sin tener ningún efecto de reverberación, pero con una relación señal a ruido de tan solo 0dB. El segundo mapa se ha obtenido grabando con dicha agrupación una fuente de banda ancha en una habitación sin apenas ruido pero con una fuerte reverberación.

# 3

# Implementación software

Como aplicación con la que demostrar el funcionamiento y la potencia del algoritmo SRP-PHAT, se programó una aplicación que, en tiempo real, muestra la potencia estimada mediante una agrupación circular de 16 micrófonos y 50cm de diámetro, superpuesta a la imagen obtenida por una webcam.

## 3.1. Representación del vídeo y el algoritmo SRP

Para evitar generar la sensación de que el algoritmo SRP tiene peor resolución de la que realmente tiene, es imprescindible que su mapa de potencia se superponga correctamente al vídeo sin que haya ningún tipo de desplazamiento entre ambas. Para lograrlo, es necesario calcular el valor en coordenadas esféricas (con las que trabaja el algoritmo) de cada punto de la imagen.

La ecuación de proyección de una cámara *pinhole* nos permite transformar cualquier punto $(x_1, x_2, x_3)$ en el espacio a un punto $(y_1, y_2)$ en el plano capturado por una cámara situada en el punto $(0, 0, 0)$ apuntando en la dirección $\widehat{x}_3$:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \frac{f}{x_3} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{3.1}$$

donde $f$ es la distancia focal de la cámara.

Manteniendo el centro de la agrupación en el origen de coordenadas, pero desplazando la cámara al punto $(\Delta x_1, \Delta x_2, \Delta x_3)$, la ecuación de proyección se transforma en:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \frac{f}{x_3 - \Delta x_3} \begin{bmatrix} x_1 - \Delta x_1 \\ x_2 - \Delta x_2 \end{bmatrix} \tag{3.2}$$

Y transformado las coordenadas cartesianas de $\boldsymbol{x}$ a las coordenadas esféricas del algoritmo SRP llegamos a:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \frac{f}{\rho\cos(\theta) - \Delta x_3} \begin{bmatrix} \rho\sin(\theta)\cos(\phi) - \Delta x_1 \\ \rho\sin(\theta)\sin(\phi) - \Delta x_2 \end{bmatrix} \tag{3.3}$$

donde la distancia a la agrupación (dato que no podemos obtener en agrupaciones planas) solo se anula para el caso $(\Delta x_1, \Delta x_2, \Delta x_3) = (0, 0, 0)$. Por lo que concluimos que, excepto que dispongamos de otro medio con el que estimar la distancia a la que se encuentra la fuente, es necesario que la cámara se encuentre en el centro de la agrupación para lograr una correcta calibración.

**Figura 3.1:** Agrupación de micrófonos utilizada para la implementación en tiempo real y para las grabaciones usadas para el estudio de los diversos algoritmos presentados en el capítulo 2.

Conocido ésto, la forma más fácil de conseguir que el mapa de potencia se superponga correctamente con la imagen de vídeo es precalculando el valor en coordenadas esféricas de cada punto de la imagen para el que se quiere calcular la potencia y aplicar el algoritmo SRP únicamente para estos puntos. Cabe destacar que los puntos en los que se calcula la potencia no tienen por qué coincidir con los píxeles de la imagen.

## 3.2. Hardware

Tanto para probar las distintas técnicas de detección de ángulo de llegada como para la implementación del sistema en tiempo real, se ha construido la agrupación circular de 50cm de diámetro con 16 micrófonos sobre una plancha de poliestireno extruido mostrada en la figura 3.1.

### 3.2.1. Micrófonos

Como sensores se ha optado por el uso de micrófonos de condensador debido a su mayor sensibilidad que los micrófonos dinámicos. En cuanto a la directividad, se consideró que lo más conveniente era el uso de patrones cardioides que, si bien el hecho de no ser omnidireccionales genera (si no se corrige) cierto sesgo las medidas de potencia en función de la dirección de las fuentes, reduce el efecto de las reflexiones en la plancha usada como soporte de los micrófonos, lo que se valoró como prioritario para el correcto funcionamiento del sistema. En concreto, por cuestiones de disponibilidad, se ha trabajado con 8 micrófonos Behringer® B-5 y 8 micrófonos Rode® M5.

### 3.2.2. Conversión Analógico/Digital

Para la interfaz de los micrófonos con el ordenador se necesitaba disponer de un sistema capaz alimentarlos, acondicionar sus señales y digitalizar 16 canales de forma completamente síncrona. De nuevo por cuestiones principalmente de disponibilidad, se ha trabajado con dos tarjetas de sonido Motu® 8 pre. Cada una de estas tarjetas cuenta con 8 entradas de micrófono con sus amplificadores y la posibilidad de utilizar alimentación *phantom* (necesaria para el uso de micrófonos de condensador) y salida tanto óptica (mediante protocolo ADAT) como USB.

Debido a los *drivers* para Windows®, solo es posible acceder a una única tarjeta a la vez, por lo que se optó por interconectar ambas tarjetas mediante fibra ótica (usando protocolo ADAT) configurando la primera de ellas para que trabaje con su reloj interno y la segunda con el recibido de la primera. Así, conectando la primera tarjeta al PC vía USB, los *drivers* pueden acceder a los 16 micrófonos, 8 como canales analógicos y 8 como canales ADAT.

Pese a que con esta configuración ambas tarjetas comparten reloj, se observó que existía un retardo de 44 muestras en los canales de la segunda tarjeta. Desgraciadamente, no es posible lograr una sincronización a nivel de muestra sin hardware adicional, por lo que se optó por, aprovechando que el retardo entre las tarjetas es constante, corregir esta diferencia vía software.

### 3.2.3. Vídeo

Para la captación del vídeo, la aplicación desarrollada se ha programado de forma que puede usarse cualquier cámara cuyos *drivers* sean compatibles con openFrameworks[1]. En concreto se ha trabajado con una Logitech® C270. Sí que resulta imprescindible, como se indica en la sección 3.1, que ésta se sitúe en el centro exacto de la agrupación.

## 3.3. Software

Tanto el algoritmo como la aplicación final se programó en C++, ya que usando este lenguaje se pueden lograr implementaciones bastante eficientes y cercanas a las que podrían programarse en un microprocesador o DSP. El algoritmo se programó en un objeto (`srp_phat`) que proporciona a la aplicación las funciones necesarias para inicializar y ejecutar el algoritmo. De esta forma el algoritmo desarrollado es fácilmente trasladable a cualquier otra aplicación sin tener que cambiar su programación. La aplicación se programó usando las librerías de openFrameworks[1], lo que facilita la gestión de los micrófonos, la cámara y la interfaz gráfica.

[1]OpenFrameworks es un conjunto de librerías multiplataforma de código abierto diseñada para la "programación creativa". Combina múltiples librerías, como OpenGL o OpenCV, y añade otras propias para facilitar a programadores de C++ el desarrollo de interfaces gráficas o la gestión de dispositivos de audio y video. [26]

### 3.3.1. Algoritmo

El algoritmo se implementa en su totalidad en el objeto `srp_phat` pensado para poderse usar en cualquier aplicación y con cualquier agrupación de micrófonos circular (independientemente de su número de sensores y su tamaño). En primer lugar, es necesario configurar las siguientes variables en su fichero de cabecera `srp_phat.h`:

1. `static int const N`: Número de sensores de la agrupación.
2. `static int const K`: Número de muestras de las ventanas de trabajo.
3. `static int const resX`: Número de puntos en x para los que se calcula la potencia.
4. `static int const resY`: Número de puntos en y para los que se calcula la potencia.
5. `float const an`: Radio de la agrupación (en metros).
6. `float const alpha`: Máximo ángulo analizado en el plano XZ.
7. `float mu`: Coeficiente del promediado exponencial del mapa de potencia (ver sección 3.3.1.3).

En el anexo A está disponible el código completo, tanto del algoritmo como de la aplicación, y en el fichero de cabeceras `srp_phat.h` se encuentra una descripción más detallada de la función de cada método del algoritmo.

#### 3.3.1.1. Interfaz con la aplicación

El objeto cuenta con los métodos públicos para su control desde la aplicación principal:

1. `srp_phat(float* const channels, float* const y)` Mediante este constructor se indica la dirección en la que se dejarán las señales leídas de cada micrófono y en la que se guardará la matriz con las potencias estimadas para cada dirección durante la ejecución del algoritmo. La aplicación principal debe encargarse de reservar la memoria necesaria para almacenar las señales como una matriz `float x[N][K]` y la matriz de resultados como una matriz `float y[resX][resY]`.
2. `float execute()` Mediante esta función el objeto `srp_phta` calcula el mapa de potencia a partir de las señales guardadas en `x` y lo guarda en `y`; además, devuelve el valor del máximo de potencia para facilitar que la aplicación lo normalice a este valor si así se desea. La aplicación debe guardar las señales de los micrófonos antes de llamar a esta función y debe asegurarse de que no se modifican sus valores durante su ejecución.
3. `void switchPHAT()` Activa o desactiva la transformación de fase de las correlaciones cruzadas.
4. `bool getPHAT()` Devuelve si la transformación de fase está activada.

#### 3.3.1.2. Correlaciones cruzadas

Para conseguir una implementación computacionalmente eficiente, las correlaciones cruzadas de la ecuación 2.5 se calculan como la transformada inversa del producto de las transformadas de Fourier de las señales de cada micrófono. Concretamente se ha utilizado la librería FFTW [27] usando únicamente dos llamadas, una para realizar todas las transformadas directas de cada señal y otra para todas las inversas de los productos. Entre ambas, además del producto, se puede elegir mediante el método público `void switchPHAT()` si se realiza la transformación de fase PHAT para mejorar la resolución del algoritmo.

Los espectros se guardan en memoria como vectores de `float` de longitud `K` aprovechando su simetría hermítica tal y como especifica la librería FFTW, por lo que ha sido necesario implementar la multiplicación y la transformación de fase sobre este formato.

#### 3.3.1.3. Promediado del mapa de potencia

Para evitar que se pierda información en el caso de que no se representen todos los mapas de potencia, se ha implementado un promediado exponencial de dichos mapas de la forma:

$$\tilde{P}_n(x,y) = (1-\mu)\tilde{P}_{n-1}(x,y) + \mu P_n(x,y) \tag{3.4}$$

Esto equivale a un filtrado IIR de orden 1 que puede verse como un filtro *antialliasing* para el muestreo realizado al no representar el resultado de todas las ventanas de audio.

#### 3.3.1.4. Debug

Para la búsqueda y corrección de errores se ha hecho uso de la librería Matlab® Engine [28], la cual ha permitido enviar vectores de la implementación en C++ a Matlab® y así poder representarlos visualmente y comparar que los resultados obtenidos con esta implementación son equivalentes los obtenidos con la implementación en Matlab®.

### 3.3.2. Aplicación

Como ya se ha comentado anteriormente, para la programación de la aplicación se ha optado por usar el *framework* openFrameworks, que facilita la implementación de sistemas en tiempo real al proveer al desarrollador de funciones para el manejo de las interfaces de audio y vídeo y de la interfaz gráfica.

Las aplicaciones de openFramewroks se basan en la clase `ofApp` que cuenta, entre otras, con las siguientes funciones:

1. `void setup()` Se ejecuta una vez al comenzar la aplicación, y se encarga de realizar todas las inicializaciones necesarias para la aplicación.

2. `void draw()` Se encarga de dibujar la interfaz gráfica en pantalla, debe ser lo más ligera posible trasladando todo el procesado con carga computacional a la función `update`.

3. `void update()` Se llama justo antes de la función `draw` y en ella debe realizarse todo el procesado relacionado con la interfaz gráfica.

4. `void audioIn(float * input, int bufferSize, int nChannels)` Se ejecuta cuando se completa un buffer de audio (su tamaño se configura mediante la función `soundStream.setup` en la función `setup`).

#### 3.3.2.1. Compensación del retardo entre tarjetas

Como se ha comentado en la sección 3.2.2, existe un retraso de 44 muestras entre las dos tarjetas que debe ser compensado vía software. Esto se ha solucionado en la propia función `audioIn`, de forma que al objeto `srp_phat` se le pasan las ventanas correctamente sincronizadas.

#### 3.3.2.2. Llamada al algoritmo

Para llamar a la función `execute` del objeto `srp_phat` existen dos posibilidades distintas, cada una con sus ventajas e inconvenientes. Por un lado, se podría llamar al algoritmo en la función `audioIn`, de forma que éste se ejecutara para todas las ventanas obtenidas de los micrófonos, o en la función `update`, de forma que se ejecutara únicamente cuando su resultado va a mostrarse por pantalla. Se ha optado por la primera opción, ya que, con la segunda, si bien es más eficiente, se pierde la información de algunas ventanas, de forma que algunas fuentes que sólo emiten durante una ventana que no coincide con una llamada a las funciones `update` y `draw` se pierden. Ejecutando el algoritmo en la función `audioIn` para todas las ventanas y usando el parámetro `mu` del algoritmo a modo de filtro *antialliasing* se reduce la perdida de información.

Esta opción se ha comprobado que da buenos resultados cuando la aplicación se ejecuta en ordenadores con potencia suficiente como para ejecutarse en tiempo real; sin embargo, en ordenadores con menor potencia (o si se introducen modificaciones en el algoritmo que lo hagan más pesado) puede ser interesante llamar al algoritmo en la función `update` con un parámetro `mu=0` para que no se ralentice toda la aplicación.

#### 3.3.2.3. Gestión del vídeo

Para la gestión del vídeo adquirido de la webcam se ha usado el objeto `ofVideoGrabber` de las librerías de openFramework, de forma que se obtiene un *frame* cada vez que se llama a la función `update` que se guarda en un objeto de tipo `ofTexture` (de nuevo de las librerías de openFramework). Este objeto se imprime por pantalla en la función `draw` en la misma posición que después va a presentarse el mapa de potencia.

#### 3.3.2.4. Interfaz Gráfica de Usuario

La interfaz gráfica de usuario muestra a la izquierda de la pantalla las señales recibidas por los 16 micrófonos y a la derecha la imagen de la webcam con el mapa de

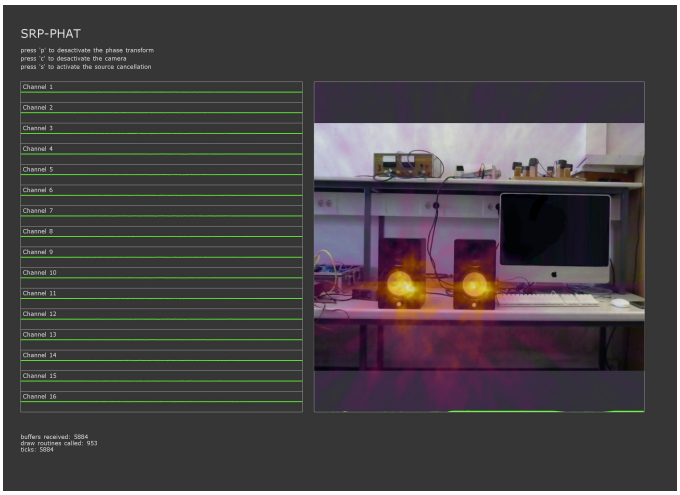



**Figura 3.2:** Interfaz gráfica de usuario

potencia superpuesto. Además, se permite desactivar la cámara y la transformación PHAT mediante comandos de teclado.

## 3.4. Resultado

En https://youtu.be/6-ieRDWeFWU [29] está disponible un vídeo que muestra el funcionamiento del sistema en tiempo real. Como puede apreciarse, éste es capaz de localizar, con bastante resolución, una o múltiples fuentes estáticas o dinámicas siempre y cuando estas emitan a un nivel de potencia similar. La principal limitación del algoritmo SRP que se observa es que, cuando hay una fuente que emite más potencia que el resto, ésta puede enmascararlas.

Otra limitación que se ha observado al sistema propuesto es lo sensible que es a la posición de la cámara. Pequeñas variaciones de su posición (o inclinación) respecto al centro de la agrupación genera una fuerte sensación de desapuntamiento en el algoritmo SRP, que no se debe al propio algoritmo sino a una incorrecta superposición del vídeo y el mapa de potencias.

# 4

# Diseño de una red de sensores

El sistema presentado en la sección anterior, si bien es interesante de cara a demostrar las posibilidades de los algoritmos SRP, tiene varios problemas de cara a su aplicación práctica:

1. Su elevado coste, al depender de un ordenador con un microprocesador con potencia suficiente para ejecutar el algoritmo, además de las tarjetas de sonido y los propios micrófonos.

2. Su elevado tamaño, al tratarse de una agrupación de 0,5 metros de diámetro. En este punto cabe destacar el hecho de que el tamaño es un factor bastante relativo a la aplicación concreta en la que vaya a utilizarse el sistema; por ejemplo, una agrupación con un tamaño correcto o incluso pequeño para instalarse en un televisión inteligente (*smartTV*) será a todas luces excesivamente grande para instalarla en un dispositivo móvil como un *smartphone*.

3. Su incapacidad para estimar la distancia de la fuente a la agrupación, pudiendo dar información únicamente del ángulo de llegada.

Los dos primeros inconvenientes pueden reducirse usando agrupaciones de menor tamaño y menor número de sensores, de forma que, además de reducirse el número de componentes necesarios, también se reduzca el coste computacional del algoritmo, por lo que se pueda usar microprocesadores menos potentes. A cambio de estas ventajas, se obtendrá una peor precisión a la hora de determinar la dirección en la que se encuentra la fuente.

Para poder determinar la posición exacta de la fuente (incluida la distancia) y compensar la perdida de precisión surgida al reducir el tamaño y número de componentes de la agrupación, en este capítulo se propone un sistema que hace uso, en lugar de una agrupación de gran tamaño, de varias agrupaciones de menor tamaño y coste, formando una red de sensores. Lógicamente, esta solución tampoco es ideal, y su conveniencia dependerá de la aplicación final, ya que no siempre varios sensores pequeños resultarán más interesantes que un único sensor de mayor tamaño.

## 4.1. Determinación de la posición de la fuente

Antes de realizar el diseño concreto e implementar la agrupación que conformará cada nodo de la red, es necesario disponer de una forma de combinar la información de cada uno de los nodos para determinar la posición exacta de la

fuente. Suponiendo que todos los sensores son iguales, el sensor i-ésimo quedará perfectamente caracterizado por su posición $\mathbf{x}_i = (x_i, y_i, z_i)^T$ y su orientación, que podemos caracterizar por su matriz de cosenos directores:

$$\mathbf{C}_i = \begin{bmatrix} \hat{u}_x\hat{u}_{xi} & \hat{u}_x\hat{u}_{yi} & \hat{u}_x\hat{u}_{zi} \\ \hat{u}_y\hat{u}_{xi} & \hat{u}_y\hat{u}_{yi} & \hat{u}_y\hat{u}_{zi} \\ \hat{u}_z\hat{u}_{xi} & \hat{u}_z\hat{u}_{yi} & \hat{u}_z\hat{u}_{zi} \end{bmatrix} \tag{4.1}$$

donde $\hat{\mathbf{u}} = (\hat{u}_x, \hat{u}_y, \hat{u}_z)$ es el vector unitario que conforma el sistema de referencia global del sistema y $\hat{\mathbf{u}}'_i = (\hat{u}'_{xi}, \hat{u}'_{yi}, \hat{u}'_{yi})$ los vectores unitarios que conforman el sistema de referencia local de la agrupación i-ésima.

La agrupación localizará la fuente según sus coordenadas esféricas $(\theta'_i, \phi'_i, r'_i)$, donde la distancia $r'_i$ al sensor es desconocida, que podrán transformarse a coordenadas cartesianas según:

$$\mathbf{x}'_i = \begin{bmatrix} x'_i \\ y'_i \\ z'_i \end{bmatrix} = r'_i \begin{bmatrix} \cos\phi'_i \sin\theta'_i \\ \sin\phi'_i \sin\theta'_i \\ \cos\theta'_i \end{bmatrix} \tag{4.2}$$

que en el sistema de referencia global del sistema serán:

$$\mathbf{x}' = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = r'_i\mathbf{C}_i \begin{bmatrix} \cos\phi'_i \sin\theta'_i \\ \sin\phi'_i \sin\theta'_i \\ \cos\theta'_i \end{bmatrix} + \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} + \begin{bmatrix} e_{xi} \\ e_{yi} \\ e_{zi} \end{bmatrix} = r'_i \begin{bmatrix} v'_{xi} \\ v'_{yi} \\ v'_{zi} \end{bmatrix} + \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} + \begin{bmatrix} e_{xi} \\ e_{yi} \\ e_{zi} \end{bmatrix} = r'_i\mathbf{v}'_i + \mathbf{x}_i + \mathbf{e}_i \tag{4.3}$$

donde se ha añadido un termino de error $\mathbf{e}$ debido a que la dirección $(\theta'_i, \phi'_i)$ es una estimación y no la dirección exacta.

Esto nos permite construir el sistema de ecuaciones:

$$\begin{bmatrix} 1 & 0 & 0 & -v'_{xi} \\ 0 & 1 & 0 & -v'_{yi} \\ 0 & 0 & 1 & -v'_{zi} \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \\ r'_i \end{bmatrix} = \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} + \begin{bmatrix} e_{xi} \\ e_{yi} \\ e_{zi} \end{bmatrix} \tag{4.4}$$

que esta indeterminado al tener 4 incógnitas y solamente 3 ecuaciones (realmente podría añadirse la ecuación $r'_i = \sqrt{x'_i + y'_i + z'_i}$, pero esta rompería la linealidad del sistema y además seguiría siendo indeterminado). Para poder tener una solución es necesario agrupar la información de $M$ sensores en un macro-sistema de ecuaciones:

$$\begin{bmatrix} \mathbf{I} & -\mathbf{v}'_1 & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{I} & \mathbf{0} & -\mathbf{v}'_2 & \dots & \mathbf{0} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{I} & \mathbf{0} & \mathbf{0} & \dots & -\mathbf{v}'_M \end{bmatrix} \begin{bmatrix} \mathbf{x}' \\ r'_1 \\ r'_2 \\ \vdots \\ r'_M \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_M \end{bmatrix} + \begin{bmatrix} \mathbf{e}_1 \\ \mathbf{e}_2 \\ \vdots \\ \mathbf{e}_M \end{bmatrix} \tag{4.5}$$

que tiene $3 + M$ incógnitas y $3M$ ecuaciones, por lo que estará sobredeterminado para $M \geqslant 2$ y cuya solución de máxima verosimilitud suponiendo que el error $\mathbf{e}_i$

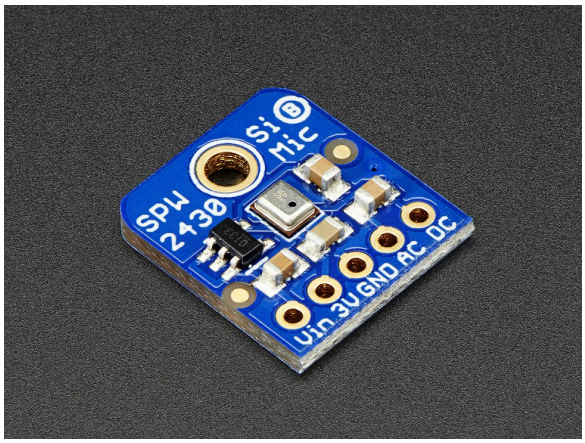



**Figura 4.1:** Placa de desarrollo SPW-2430 del micrófono utilizado

sigue una distribución gaussiana de media cero es:

$$\hat{\mathbf{x}}' = \begin{bmatrix} \mathbf{I} & -\mathbf{v}'_1 & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{I} & \mathbf{0} & -\mathbf{v}'_2 & \dots & \mathbf{0} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{I} & \mathbf{0} & \mathbf{0} & \dots & -\mathbf{v}'_M \end{bmatrix}^{+} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_M \end{bmatrix} \tag{4.6}$$

donde el operador $\bullet^{+}$ indica la matriz pseudoinversa: $\mathbf{A}^{+} = (\mathbf{A}^{H}\mathbf{A})^{-1}\mathbf{A}^{H}$. Cabe destacar que, dado que realmente sólo nos interesan las 3 primeras incógnitas, podemos usar métodos eficientes para calcular las tres primeras filas de la pseudoinversa mediante la descomposición SVD.

## 4.2. Hardware

Como compromiso entre resolución, tamaño y coste, se decidió usar como sensores de la red agrupaciones de 4 micrófonos formando un cuadrado de 20cm de lado. Estos sensores tienen una parte analógica, formada por los micrófonos y los componentes necesarios para acondicionar su señal, y un microprocesador para ejecutar el algoritmo SRP.

### 4.2.1. Micrófonos

Como micrófonos se optó por trabajar con micrófonos MEMSs (*Micro Electro-Mechanical System*) por su menor consumo y bajo nivel de ruido comparado con los micrófonos ECMs (*Electrets Condenser Microphones*). Concretamente, se ha trabajado con la *breakout board* SPW2430 de Adafruit®, que monta un micrófono SPW2430HR5H-B y todo lo necesario para alimentarlo a partir de una tensión de 5V y extraer la señal sin continua pudiéndola conectar fácilmente en una PCB mediante orificio pasante o con pines hembra de décima de pulgada.

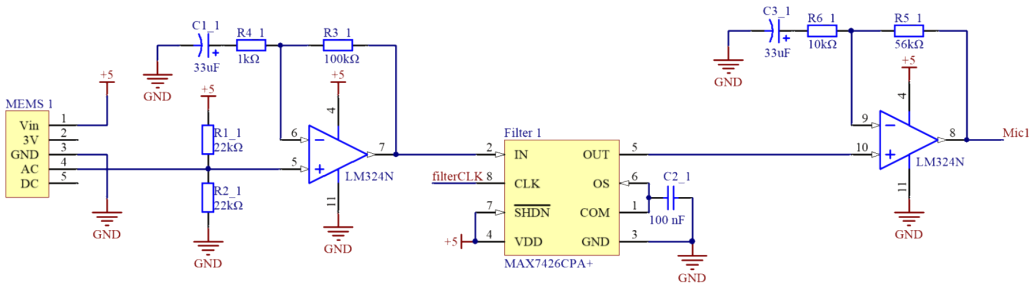



**Figura 4.2:** Esquemático del circuito analógico para el acondicionado de la señal de un micrófono

### 4.2.2. Acondicionado de las señales

Dado que el conversor trabaja en un rango de 0 a 5V, es necesario aplicar una gran ganancia a las señales de los micrófonos. Para ésto, se optó por utilizar dos etapas no inversoras con amplificador operacional. Debido a que en total se necesitaban 8 amplificadores (2 etapas por cada uno de los 4 micrófonos) y por cuestiones de disponibilidad, se optó por trabajar con dos integrados LM324, que incluyen 4 operacionales cada uno.

Dado que la frecuencia de trabajo elegida era de 8kHz, también resulta necesario filtrar las señales de los micrófonos. En concreto, se optó por trabajar con los filtros de capacidades conmutadas MAX7426. La frecuencia de corte de dichos filtros se controla mediante una señal de reloj conectada a uno de sus pines, lo que nos permite, generando dicha señal mediante un *timmer* de microprocesador, elegir su frecuencia de corte vía *software*.

En la figura 4.2 se muestra el circuito concreto utilizado para acondicionar la señal del primer micrófono.

### 4.2.3. Microprocesador

Como microprocesador, se ha optado por trabajar con PSoC® 4 BLE de Cypress®, que cuenta con un microprocesador Cortex® M0 y varios bloques periféricos tanto analógicos como digitales, entre los que destacan 4 amplificadores operacionales, un conversor analógico/digital de aproximaciones sucesivas de 4 canales y un módulo Bluetooth Low Energy (BLE). En concreto, por cuestiones de disponibilidad, se ha trabajado con un módulo CY8CKIT-142, que monta un CY8C4247LQI-BL483 con tan solo 128kB de memoria, en una *breakout board* fácilmente programable mediante la CY8CKIT-042-BLE Pioneer Kit.

Con estas limitaciones de memoria solo era posible trabajar con ventanas de 256 muestras por canal, por lo que se optó por trabajar con una frecuencia de 8kHz, lo que da como resultado una ventana de duración temporal similar a las utilizadas en el capítulo 3 para la agrupación de 16 micrófonos y ventanas de 1024 muestras a 44,1kHz. El microprocesador y el conversor se han configurado de forma que el algoritmo se interrumpe cada vez que se ha convertido una nueva muestra en los 4 canales. Cabe destacar que, si bien el modulo utilizado no los tiene, existe otro módulo, el CY8CKIT-143A, que es compatible con todo el diseño realizado (tanto *software* como *hardware*), que monta un CY8C4248LQI-BL583, que, además de tener

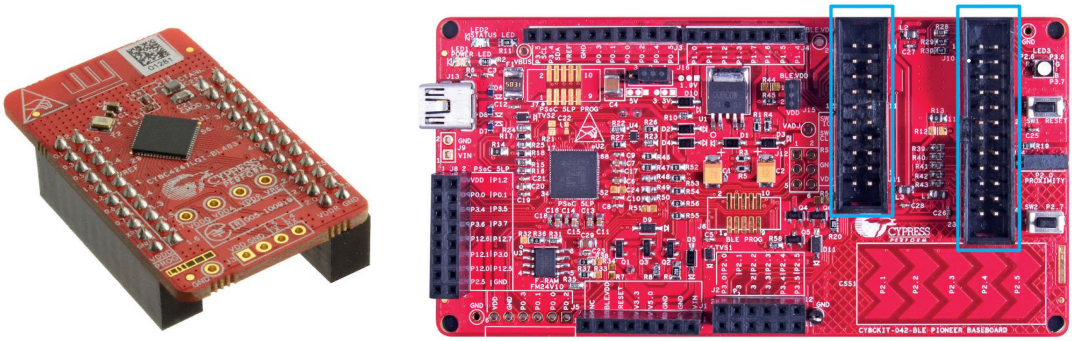


**Figura 4.3:** Placa de desarrollo CY8CKIT-142 del microprocesador PSoC 4 utilizado y tarjeta BLE Pionner Kit usada para programarlo.

el doble de memoria, incluye 8 canales DMA que podrían usarse para interrumpir al microprocesador únicamente cuando se haya convertido una ventana completa.

Además de ejecutar el algoritmo y de la conversión, el PSoC® se encarga de generar la señal de reloj de los filtros. También habría resultado interesante usar los cuatro amplificadores operacionales que integra para la última etapa de amplificación, sin embargo, debido a algunos problemas encontrados con el uso de éstos y a la falta de tiempo, finalmente se descartó esta opción.

Por último, hubiera sido interesante aprovechar que el PSoC 4 utilizado integra un módulo Bluetooth Low Energy (BLE), e incluso una antena, para enviar la posición estimada inalámbricamente. Sin embargo, por cuestiones de tiempo, no ha sido posible implementar esto a fecha de redacción de esta memoria y la información se envía por puerto serie.

### 4.2.4. Otros

Por último, para facilitar el *debug* y posibles interacciones con el usuario, se añadieron conexiones para conectar el puerto serie del microprocesador al puerto USB de un ordenador mediante cable FTDI y dos pulsadores y dos diodos LEDs. Para alimentar la placa se ha optado por usar el propio cable FTDI, aunque también se han añadido un segundo conector para otras fuentes de alimentación. Debido a que tanto el microprocesador como los filtro se alimentan a 5V y la tensión suministrada por dicho cable es también de 5V, se ha utilizado un regulador de bajo *dropout*, concretamente un LP2954.

En el anexo B se enceuntra el esquemático completo del sistema y en el anexo C la PCB diseñada y fabricada para el prototipo.

## 4.3. Software

### 4.3.1. Gestión del ADC

Debido a que el tiempo de cómputo del algoritmo es muy superior al de adquisición (más detalles en la sección 4.4) se ha optado por implementar un ciclo en el que primero se adquiere toda la ventana a procesar y después se procesa.

Podría haberse implementado una adquisición paralela a la ejecución del algoritmo, pero esto habría realentizado la ejecución del mismo (ya que no se disponía de canales DMA y había que interrumpirlo con cada nueva muestra) cuando el tiempo de adquisición es despreciable frente al de computo.

Además, hubiera sido necesario encontrar una forma de sincronizar ambos procesos para que terminaran a la vez, ya que si ambos comienzan en el mismo instante de tiempo y el de adquisición se para al terminar para esperar al de computo, se estaría introduciendo un retardo innecesario desde la adquisición hasta que se obtiene el resultado.

### 4.3.2. Algoritmo

Debido a que el microprocesador utilizado no es un DSP y no tiene hardware específico para realizar FFTs, ni una gran capacidad de cómputo, se optó por buscar una implementación del algoritmo distinta a la descrita en la sección 3.3. Dado que el algoritmo no necesita todos los valores de las correlaciones cruzadas, sino solamente los valores cercanos a $\tau = 0$, se decidió calcular estos valores en el dominio temporal. Concretamente se optó por usar un estimador de correlación sesgado del tipo:

$$r_{12}[k] \approx \frac{1}{N} \sum_{n=k}^{N-1} x_1[n] x_2[n-k] \tag{4.7}$$

El problema de esta implementación es que no permite aplicar (de forma sencilla y eficiente) la transformación PHAT, ya que ésta trabaja en el dominio frecuencial. Al igual que el el capítulo anterior, $r_{12}(t)$ se aproxima por el valor de $r_{12}[n]$ más cercano.

Una vez calculadas las correlaciones cruzadas se procede al calculo de las potencias para cada dirección (en concreto se ha trabajado con un mallado de 32 puntos tanto en $\theta$ como en $\phi$) guardando el valor máximo de potencia estimada y su posición para evitar tener que volver a recorrer la matriz en busca del máximo una vez calculados todos los valores. Se observó que, debido al uso de una frecuencia de muestreo bastante baja y no realizar ningún interpolado, muchos puntos tenían el mismo valor, y esta implementación de la búsqueda del máximo siempre se queda con primero (o el último) de ellos, por lo que tenía bastante sesgo. Para evitarlo, durante la estimación de potencias, se guardan las posiciones con mayor y menor $\theta$ y $\phi$ que tienen la máxima potencia estimada hasta el momento, de forma que al acabar de realizar las estimaciones se considera como posición del máximo el punto medio de ellos:

$$(\theta_0, \phi_0) = \left( \frac{\theta_{max} - \theta_{min}}{2}, \frac{\phi_{max} - \phi_{min}}{2} \right) \tag{4.8}$$

Una vez estimada la posición de la fuente, se envía su índice por puerto serie, de forma que esta puede ser leída por un PC a través de un cable FTDI. Faltaría, como ya se ha comentado anteriormente, enviar esta información vía Bluetooth Low Energy.

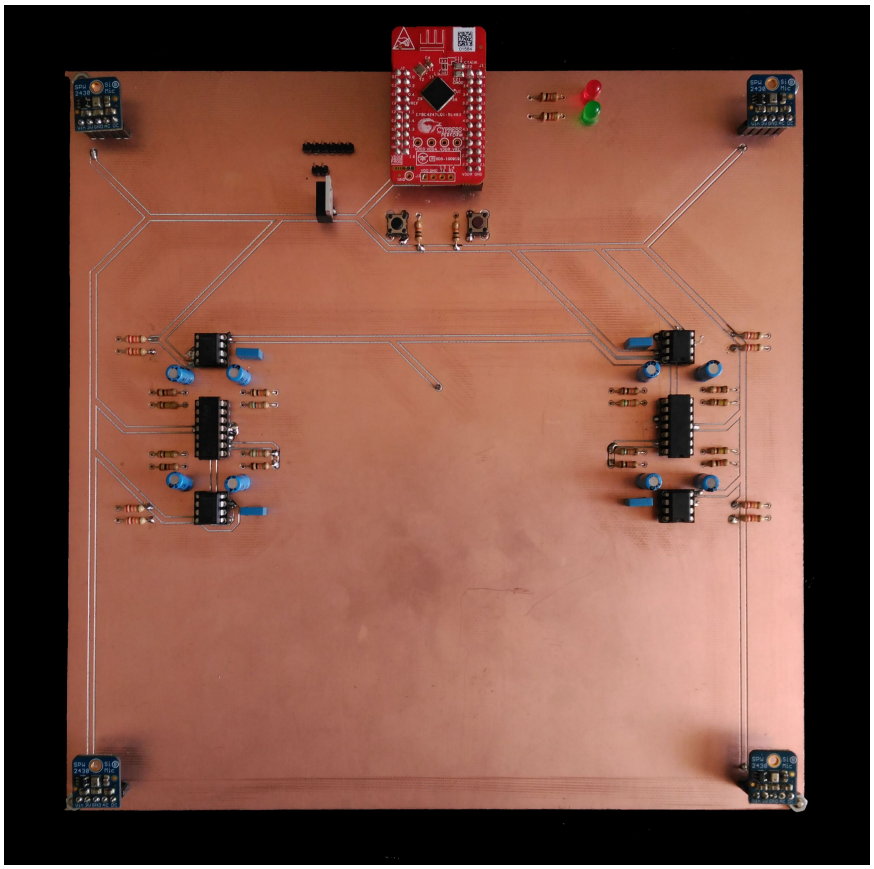


**Figura 4.4:** Prototipo fabricado

## 4.4. Resultado

Tras fabricar la PCB, se observó que su superficie resultaba demasiado reflectante para el sonido, lo que generaba una gran varianza en la estimación de la posición de la fuente, especialmente cuando ésta se situaba en valores bajos de $\theta$. Para evitar ésto, se ha cubierto la PCB (salvo el microprocesador, los pulsadores y los LEDs) con una lámina de gomaespuma.

Con la configuración descrita, cada ciclo del programa del microprocesador dura 220ms, de los cuales 17 son de adquisición y 203 de ejecución del algoritmo. Esto equivale a unas 4,5 mediciones por segundo.

Tal y como puede observarse en el vídeo demostrativo disponible en https://youtu.be/PSL_F7MSc6M [30], si bien se tiene menos resolución y sensibilidad que en el sistema presentado en el capítulo 3, el sistema tiene bastante buenas prestaciones; sobre todo para valores de $\theta$ inferiores a 45º. Cuando la fuente empieza a situarse demasiado lateral, la varianza de la estimación aumenta bastante apareciendo valores espurios, por lo que podría ser interesante el uso de un filtro de mediana sobre dicha estimación.

Desgraciadamente, por cuestiones de tiempo, solamente ha sido posible fabricar un único sensor, por lo que no se ha podido probar el funcionamiento del algoritmo presentado en la sección 4.1.

# 5

# Conclusiones y lineas futuras

A lo largo de este trabajo se ha realizado una introducción a las técnicas de estimación de dirección de llegada (DOA) en agrupaciones de micrófonos. En el capítulo 2 se han estudiado distintas técnicas existentes a día de hoy y se ha concluido, al igual que en la bibliografía existente [4], que el mejor compromiso entre prestaciones y coste computacional es el algoritmo SRP-PHAT.

El sistema implementado en el capítulo 3 nos ha permitido comprobar como, con una agrupación con tamaño y número de micrófonos suficientes, el algoritmo SRP-PHAT logra una gran resolución. También hemos comprobado que uno de los principales problemas de este algoritmo es que cuando hay diversas fuentes con distintas potencias, las más energéticas pueden enmascarar al resto. Para evitar esto pueden estudiarse diversas soluciones, por ejemplo, modificar las funciones de correlación cruzada para eliminar el efecto de una fuente (como suele hacerse con la matriz de correlación de agrupaciones de banda estrecha).

A partir de los mapas de potencia generados por el algoritmo SRP, normalmente se realizan búsquedas de máximos locales para localizar las fuentes. Los efectos generados por los lóbulos de difracción del conformado de haz del algoritmo SRP sobre el mapa de potencia se consideran un problema que puede dificultar encontrar segundas fuentes, pero podría resultar interesante aplicar técnicas de reconocimientos de patrones para intentar aprovechar éstos para localizar las fuentes.

Por último, en el capítulo 4, se ha realizado el diseño de una agrupación de bajo coste que podría utilizarse como nodo en una red de sensores. Se ha comprobado que, si bien se produce una fuerte perdida de prestaciones, el algoritmo SRP también es capaz de funcionar en agrupaciones de menor tamaño y con menor número de sensores. Por cuestiones de tiempo, sólo ha sido posible fabricar una agrupación, por lo que no ha sido posible estudiar la ganancia introducida por el uso de varios sensores mediante el algoritmo presentado en la sección 4.1, lo que puede resultar de gran interés.

En líneas generales, en los capítulos 3 y 4 hubiera resultado de interés realizar un análisis cuantitativo de la resolución y sensibilidad de los sistemas planteados. Sin embargo, llevar a cabo estas medias tiene una dificultad elevada, ya que en primer lugar sería necesario poder situar las fuentes con precisión en la posición deseada para después comprobar la desviación de la estimación. Además, al no disponer de una forma de automatizar estas medidas, un buen estudio estadístico del error acarrearía un elevado coste temporal.

# Bibliografía


[1] S. Haykin, Ed., *Array Signal Processing*, first edition edition ed. Englewood Cliffs, N.J: Prentice Hall, Jun. 1984.

[2] D. H. Johnson and D. E. Dudgeon, *Array Signal Processing: Concepts and Techniques*, 1st ed. Englewood Cliffs, NJ: Prentice Hall, Feb. 1993.

[3] S. Chandran, *Advances in Direction-of-Arrival Estimation.* Norwood: Artech House, 2005, oCLC: 437160320.

[4] M. Brandstein and D. Ward, *Microphone Arrays: Signal Processing Techniques and Applications.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, oCLC: 851392110.

[5] H. Krim and M. Viberg, "Two decades of array signal processing research: The parametric approach," *IEEE Signal Processing Magazine*, vol. 13, no. 4, pp. 67–94, Jul. 1996.

[6] J. Liu, X. Wu, W. J. Emery, L. Zhang, C. Li, and K. Ma, "Direction-of-Arrival Estimation and Sensor Array Error Calibration Based on Blind Signal Separation," *IEEE Signal Processing Letters*, vol. 24, no. 1, pp. 7–11, Jan. 2017.

[7] J. Dai, X. Bao, W. Xu, and C. Chang, "Root Sparse Bayesian Learning for Off-Grid DOA Estimation," *IEEE Signal Processing Letters*, vol. 24, no. 1, pp. 46–50, Jan. 2017.

[8] Q. Shen, W. Liu, W. Cui, S. Wu, Y. D. Zhang, and M. G. Amin, "Focused Compressive Sensing for Underdetermined Wideband DOA Estimation Exploiting High-Order Difference Coarrays," *IEEE Signal Processing Letters*, vol. 24, no. 1, pp. 86–90, Jan. 2017.

[9] Y. Wang, Y. Yang, Y. Ma, and Z. He, "High-Order Superdirectivity of Circular Sensor Arrays Mounted on Baffles," *Acta Acustica united with Acustica*, vol. 102, no. 1, pp. 80–93, Jan. 2016.

[10] H. Dong and N. R. Chapman, "Measurement of Ocean Bottom Reflection Loss with a Horizontal Line Array," *Acta Acustica united with Acustica*, vol. 102, no. 4, pp. 645–651, Jul. 2016.

[11] Y. Hwang, Y. Je, H. Lee, J. Lee, C. Lee, W. Kim, and W. Moon, "A Parametric Array Ultrasonic Ranging Sensor with Electrical Beam Steering Capability," *Acta Acustica united with Acustica*, vol. 102, no. 3, pp. 423–427, May 2016.

[12] A. Muscheites, D. Leckschat, and C. Epe, "Line Array Sound Reinforcement Systems using Air Motion Transformer," *Acta Acustica united with Acustica*, vol. 102, no. 3, pp. 592–599, May 2016.

[13] J. Xie, S. D. Yoo, and K. Jain, "A Low Computational Complexity Beamforming Scheme Concatenated with Noise Cancellation." Audio Engineering Society, Sep. 2016.

[14] K. Sunder and W. Woszczyk, "Investigation of Impulse Response Recording Techniques in Binaural Rendering of Virtual Acoustics." Audio Engineering Society, Sep. 2016.

[15] B. Martin, R. King, and W. Woszczyk, "Subjective Graphical Representation of Microphone Arrays for Vertical Imaging and Three-Dimensional Capture of Acoustic Instruments, Part I." Audio Engineering Society, Sep. 2016.

[16] D. Self, "Acoustic Radar." http://www.douglas-self.com/MUSEUM/COMMS/ear/ear.htm#chin.

[17] H. Wang and P. Chu, "Voice source localization for automatic camera pointing system in videoconferencing," in *1997 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 1, Apr. 1997, pp. 187–190 vol.1.

[18] A. Farina, A. Amendola, A. Capra, and C. Varani, "Spatial Analysis of Room Impulse Responses Captured with a 32-Capsule Microphone Array." Audio Engineering Society, May 2011.

[19] A. O'Donovan, R. Duraiswami, and D. Zotkin, "Imaging concert hall acoustics using visual and audio cameras," in *2008 IEEE International Conference on Acoustics, Speech and Signal Processing*, Mar. 2008, pp. 5284–5287.

[20] J. Capon, "High-resolution frequency-wavenumber spectrum analysis," *Proceedings of the IEEE*, vol. 57, no. 8, pp. 1408–1418, Aug. 1969.

[21] R. Schmidt, "Multiple emitter location and signal parameter estimation," *IEEE Transactions on Antennas and Propagation*, vol. 34, no. 3, pp. 276–280, Mar. 1986.

[22] A. Johansson, G. Cook, and S. Nordholm, "Acoustic direction of arrival estimation, a comparison between root-music and SRP-PHAT," in *TENCON 2004. 2004 IEEE Region 10 Conference*, vol. B, Nov. 2004, pp. 629–632.

[23] C. Knapp and G. Carter, "The generalized correlation method for estimation of time delay," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 24, no. 4, pp. 320–327, Aug. 1976.

[24] J. H. DiBiase, "A high-accuracy, low-latency technique for talker localization in reverberant environments using microphone arrays," Ph.D. dissertation, Brown University, 2000.

[25] H. F. Silverman, Y. Yu, J. M. Sachar, and W. R. Patterson, "Performance of real-time source-location estimators for a large-aperture microphone array," *IEEE Transactions on Speech and Audio Processing*, vol. 13, no. 4, pp. 593–606, Jul. 2005.

[26] openFrameworks Community, "openFrameworks," http://openframeworks.cc/.

[27] M. Frigo and S. G. Johnson, "The Design and Implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, Feb. 2005.

[28] I. The MathWorks, "Introducing MATLAB Engine API for C/C++ and Fortran - MATLAB & Simulink - MathWorks España," https://es.mathworks.com/help/matlab/matlab_external/introducing-matlab-engine.html.

[29] Díaz-Guerra Aparicio, David, "Localización de fuentes sonoras mediante agrupaciones de micrófonos: Implementación Software," Jan. 2017.

[30] ——, "Localización de fuentes sonoras mediante agrupaciones de micrófonos: Implementación Hardware," Jan. 2017.

# A

# Código de la implementación software

...\of_v0.9.3_vs_release\apps\myApps\srp-phat\src\main.cpp 1

```cpp
#include "ofMain.h"
#include "ofApp.h"


int main( ){

    ofSetupOpenGL(2448,1936, OF_WINDOW);     // Setup the GL context

    // this kicks off the running of my app
    // can be OF_WINDOW or OF_FULLSCREEN
    // pass in width and height too:
    ofRunApp( new ofApp() );

}
```

...of_v0.9.3_vs_release\apps\myApps\srp-phat\src\headers.h 1

```
#pragma once

#define FS 44100.0
#define BUFFER_SIZE 1024
#define R_SENSORS 0.25
#define N_SENSORS 16
#define RES_X 129
#define RES_Y 129
#define ALPHA 20
#define PI 3.14159265359
```

...s\of_v0.9.3_vs_release\apps\myApps\srp-phat\src\ofApp.h 1

```
#pragma once

#include "ofMain.h"

#include <vector>
#include <mutex>

#include "ofxTrueTypeFontUC.h"

#include "headers.h"
#include "srp_phat.h"

class ofApp : public ofBaseApp {

    public:

        bool camera = true;
        bool sourceCancelation = false;

        static const int resX = RES_X;  //Nuemro de puntos en X
        static const int resY = RES_Y;  //Numero de puntos en Y
        static const int bufferSize = BUFFER_SIZE;
        static const int nChannels = N_SENSORS;
        static const int lag = 44;      //Muestras de retardo de los canales pares

        void setup();
        void update();
        void draw();

        void keyPressed(int key);
        void keyReleased(int key);
        void mouseMoved(int x, int y );
        void mouseDragged(int x, int y, int button);
        void mousePressed(int x, int y, int button);
        void mouseReleased(int x, int y, int button);
        void mouseEntered(int x, int y);
        void mouseExited(int x, int y);
        void windowResized(int w, int h);
        void dragEvent(ofDragInfo dragInfo);
        void gotMessage(ofMessage msg);

        void audioIn(float * input, int bufferSize, int nChannels);

        ofxTrueTypeFontUC textSmall, textBig;

        float x[nChannels][bufferSize];
        float xpre[nChannels/2][lag];
        float P[resX][resY];
        float maxP;
        vector <float> volHistory;

        const int camWidth  = 640;
        const int camHeight = 480;
```

```
        ofVideoGrabber vidGrabber;
        ofTexture videoTexture;

        mutex mutexChannels;

        int     bufferCounter;
        int     drawCounter;

        float smoothedVol;
        float scaledVol;

        ofSoundStream soundStream;

        srp_phat srp;

        ofImage imageResult;

};
```

...of_v0.9.3_vs_release\apps\myApps\srp-phat\src\ofApp.cpp 1

```
#include "ofApp.h"
#include "palette.h"

//--------------------------------------------------------------
void ofApp::setup(){

    ofSetVerticalSync(true);
    ofSetCircleResolution(80);
    ofBackground(54, 54, 54);
    textSmall.loadFont("verdana.ttf", 16, true, true);
    textBig.loadFont("verdana.ttf", 32, true, true);

    srp.setup((float*)x, (float*)P);

    imageResult.allocate(resX, resY, OF_IMAGE_COLOR_ALPHA);
    imageResult.setColor(ofColor::black);

    // 0 output channels,
    // 2 input channels
    // 44100 samples per second
    // 256 samples per buffer
    // 4 num buffers (latency)

    soundStream.printDeviceList();
    soundStream.setDeviceID(1);

    volHistory.assign(400, 0.0);

    bufferCounter   = 0;
    drawCounter     = 0;
    smoothedVol     = 0.0;
    scaledVol       = 0.0;

    vector<ofVideoDevice> devices = vidGrabber.listDevices();

    for (int i = 0; i < devices.size(); i++) {
        if (devices[i].bAvailable) {
            ofLogNotice() << devices[i].id << ": " << devices[i].deviceName;
        }
        else {
            ofLogNotice() << devices[i].id << ": " << devices[i].deviceName << " - ↵
              unavailable ";
        }
    }

    vidGrabber.setDeviceID(2);
    vidGrabber.setDesiredFrameRate(60);
    vidGrabber.initGrabber(640, 480);

    soundStream.setup(this, 0, nChannels, 44100, bufferSize, 4);

}
```

...of_v0.9.3_vs_release\apps\myApps\srp-phat\src\ofApp.cpp 2

```
//--------------------------------------------------------------
void ofApp::update(){

    /*
       Ejecutar aquí el algoritmo SRP mejora el rendimiento del sistema al realizarse solo
       cuando el resultado se va a mostrar por pantalla, pero impide poder usar tecnicas de
       suavizado del resultado para que se localicen también fuentes que estuvieran en ventanas
       distintas a la última recibida antes de mostrar por pantalla el resultado.
       La alternativa es ejecutarlo en la función audioIn.
    */
    if (sourceCancelation) maxP = srp.execute();

    //lets scale the vol up to a 0-1 range
    scaledVol = ofMap(smoothedVol, 0.0, 0.17, 0.0, 1.0, true);

    //lets record the volume into an array
    volHistory.push_back( scaledVol );

    //if we are bigger the the size we want to record - lets drop the oldest value
    if( volHistory.size() >= 400 ){
        volHistory.erase(volHistory.begin(), volHistory.begin()+1);
    }

    //Actualiza la imágen con los últimos valores de SRP
    for (int x = 0; x < resX; x++) {
        for (int y = 0; y < resY; y++) {
            int level = maxP>0? (int) P[y][x] / maxP * 512 : 0;
            int alpha = camera ? level/2 : 255;
            imageResult.setColor(resX - x-1, y, ofColor(thermal_palette[level][0], thermal_palette[level][1], thermal_palette[level][2], alpha));
        }
    }
    imageResult.update();

    // Camara
    vidGrabber.update();
    videoTexture.loadData(vidGrabber.getPixels());
}

//--------------------------------------------------------------
void ofApp::draw(){

    ofSetColor(225);
    textBig.drawString("SRP-PHAT", 64, 120);
    if (srp.getPHAT()) textSmall.drawString("press 'p' to desactivate the phase transform", 64, 173);
    else textSmall.drawString("press 'p' to activate the phase transform", 64, 173);
    if (camera) textSmall.drawString("press 'c' to desactivate the camera", 64, 203);
```

```
    else textSmall.drawString("press 'c' to activate the camera", 64, 203);
    if (sourceCancelation) textSmall.drawString("press 's' to desactivate the
      source cancellation", 64, 233);
    else textSmall.drawString("press 's' to activate the source cancellation", 64,
      233);

    ofNoFill();


    // draw each channel:
    for (int i = 0; i < nChannels; i++) {
        ofPushStyle();
            ofPushMatrix();
            ofTranslate(64, 280+75*i, 0);

            ofSetColor(225);
            textSmall.drawString("Channel " + to_string(i+1), 8, 26);

            ofSetLineWidth(1);
            ofDrawRectangle(0, 0, 1024, 75);

            ofSetColor(0x66, 0xFF, 0x33);
            ofSetLineWidth(3);

                mutexChannels.lock();
                ofBeginShape();
                for (unsigned int j = 0; j < bufferSize; j++){
                    ofVertex(j*1, 37.5 - x[i][j]*60.0f);
                }
                ofEndShape(false);
                mutexChannels.unlock();

            ofPopMatrix();
        ofPopStyle();
    }


    ofPushStyle();
        ofPushMatrix();
        ofTranslate(1130, 280, 0);

        ofSetColor(225);
        ofDrawRectangle(0, 0, 1200, 1200);

        // Draw the camera and de SRP result
        videoTexture.draw(0, 150, 1200, 900);
        imageResult.draw(0, 0, 1200, 1200);

        //lets draw the volume history as a graph
        ofSetColor(0x66, 0xFF, 0x33);
        ofFill();
        ofBeginShape();
        for (unsigned int i = 0; i < volHistory.size(); i++){
```

...of_v0.9.3_vs_release\apps\myApps\srp-phat\src\ofApp.cpp 4

```
            if( i == 0 ) ofVertex(3*i, 1200);

            ofVertex(3*i, 1200 - volHistory[i] * 210);

            if( i == volHistory.size() -1 ) ofVertex(3*i, 1200);
        }
        ofEndShape(false);

        ofPopMatrix();
    ofPopStyle();

    drawCounter++;

    ofSetColor(225);
    string reportString = "buffers received: "+ofToString(bufferCounter)+"\ndraw
      routines called: "+ofToString(drawCounter)+"\nticks: " + ofToString
      (soundStream.getTickCount());
    textSmall.drawString(reportString, 64, 1578);

}

//--------------------------------------------------------------
void ofApp::audioIn(float * input, int bufferSize, int nChannels) {

    float curVol = 0.0;

    // samples are "interleaved"
    int numCounted = 0;

    //Con FFTW se podría calcular directamente la FFT sobre "input" jugando con los
       parametros "istride" e "idist"
    //pero si queremos representar los canales hace falta guardarlos en la variable
       "x". Además está el retardo...
    mutexChannels.lock();
    for (int i = 0; i < bufferSize; i++) {
        for (int j = 0; j < nChannels; j++) {
            if ((j+1)%2 && i<lag) x[j][i] = xpre[j/2][i]; //Primeras muestras de
              los canales adelantados
            else if ((j+1)%2) x[j][i] = input[(i-lag)*nChannels + j/2 + (j%2)*
              (nChannels/2)];
            else x[j][i] = input[i*nChannels + j/2 + (j%2)*(nChannels/2)];

            curVol += x[j][i] * x[j][i];
            numCounted++;
        }
    }


    for (int i = bufferSize; i < bufferSize+lag; i++) {
        for (int j = 0; j < nChannels; j+=2) {
            xpre[j/2][i-bufferSize] = input[(i-lag)*nChannels + j/2 + (j%2)*
              (nChannels/2)];
        }
```

```
    }

    /*
       Ejecutar aquí el algoritmo SRP empeora el rendimiento del sistema al
         calcularse en ventanas
       cuyo resultado no va a mostrarse por pantalla, pero permite poder usar
         tecnicas de
       suavizado del resultado para que se localicen también fuentes que estuvieran
          en ventanas
       distintas a la última recibida antes de mostrar por pantalla el resultado.
       La alternativa es ejecutarlo en la función update.
    */
    if (!sourceCancelation) maxP = srp.execute();

    mutexChannels.unlock();

    //this is how we get the mean of rms :)
    curVol /= (float)numCounted;

    // this is how we get the root of rms :)
    curVol = sqrt(curVol);

    smoothedVol *= 0.93;
    smoothedVol += 0.07 * curVol;

    bufferCounter++;

}

//--------------------------------------------------------------
void ofApp::keyPressed  (int key){
    if (key == 'p') {
        srp.switchPHAT();
    }

    if (key == 'c') {
        camera ^= 1;
    }

    if (key == 's') {
        sourceCancelation ^= 1;
        srp.switchSourceCancellation();
    }
}

//--------------------------------------------------------------
void ofApp::keyReleased(int key){

}

//--------------------------------------------------------------
void ofApp::mouseMoved(int x, int y ){
```

```
}

//--------------------------------------------------------------
void ofApp::mouseDragged(int x, int y, int button){

}

//--------------------------------------------------------------
void ofApp::mousePressed(int x, int y, int button){

}

//--------------------------------------------------------------
void ofApp::mouseReleased(int x, int y, int button){

}

//--------------------------------------------------------------
void ofApp::mouseEntered(int x, int y){

}

//--------------------------------------------------------------
void ofApp::mouseExited(int x, int y){

}

//--------------------------------------------------------------
void ofApp::windowResized(int w, int h){

}

//--------------------------------------------------------------
void ofApp::gotMessage(ofMessage msg){

}

//--------------------------------------------------------------
void ofApp::dragEvent(ofDragInfo dragInfo){

}
```

...f_v0.9.3_vs_release\apps\myApps\srp-phat\src\srp_phat.h 1

```
#pragma once

#include <vector>
#include <cmath>
#include <fstream>
#include <string>
#include <initializer_list>

#include "fftw3.h"

#include "headers.h"

#include "hanning1024.h"

#include "engine.h" //Matlab Engine

class srp_phat {

    private:

        bool DEBUG = false;
        bool PHAT = true;
        bool sourceCancellation = false;

        Engine *ep = nullptr;

        static int const N = N_SENSORS;     //Número de sensores de la agrupación
        static int const K = BUFFER_SIZE;   //Tamaño de las ventanas
        static int const resX = RES_X;      //Núemro de puntos en X
        static int const resY = RES_Y;      //Número de puntos en Y

        float const c  = 343.0;     //Velocidad del sonido
        float const an = R_SENSORS; //Radio de la agrupación [m]
        float const Fs = FS;        //Frecuencia de muestreo del sistema
        float const alpha = ALPHA;  //Máximo ángulo analizado en el plano XZ
        float mu = 0;               //Factor de inercia del resultado

        float(*x)[K];       //Puntero a la matriz con las señales de entrada (NxK)
        float(*P)[resX];    //Puntero a la matriz con la estimación realizada
          (resTxresP)

        float X[N][K];      //FFT de x (NxK)
        float r[N][N][K];   //Correlación entre canales (NxNxK)
        float R[N][N][K];   //FFT de r (NxNxK)
        float rp[N][N][K];  //Versión temporal de r modificada


        int tau[resX][resY][N][N];      //Muestra de r que hay que usar en cada
          dirección   (resX x resY x N x N)
        float tauF[resX][resY][N][N];   //Valor de tau con parte fraccional para
          interpolar (resX x resY x N x N)

        fftwf_plan fft;
```

```
        fftwf_plan ifft;

        void calculateAngles();
        void calculateTau();
        void saveTau();
        bool loadTau();

        //Evalúa el funcional de SRP-PHAT para la dirección correspondientes a los
          indices indicados.
        inline float functional(int x_ind, int y_ind);

        //Calcula las correlaciones entre todos los canales de x y las guarda en r.
        //Para esto usa las matrices X y R (ésta última puede quedar corrupta)
        inline void correlations();

        //Proyecta las correlaciones cruzadas en un espacio ortogonal al de una
          fuente en la dirección
        //indexada por (x,y) en la matriz P.
        void cancelSource(int x, int y);

        //Multiplica los vectores x e y* que representan vectores hermíticos de
          tamaño K
        //siguiendo el formato de FFTW (solo valido si K es par)
        inline void hermitian_conj_multiplication(float* const x, float* const y,
          float* z);

        //Divide entre el módulo de cada componente de la FFT para quedarse solo
          con la información
        //de fase siguiendo el formato de FFTW (solo valido si K es par)
        inline void module_division(float* x);

        //Devuelve el valor de la correlación cruzada entre los sensores i y j para
           la muestra
        //fraccíonal n calculada mediante interpolación lineal.
        inline float interpoledCorrelation(int i, int j, float n);

        void sendMemoryToMatlab();

    public:
        float theta[resX][resY];  //Valores de theta para los que se evalúa P
        float phi[resX][resY];  //Valores de phi para los que se evalua P

        srp_phat();
        srp_phat(float* const channels, float* const y);
        ~srp_phat();

        //Asigna todos los valores necesarios para comenzar a realizar estimaciones
           de DOA
        void setup(float* const channels,   //Matriz con las señales de entrada
          (nChannels x bufferSize)
                   float* const y           //Matriz en la que guardar la
                     estimación realizada
                  );
```

```
        //Activa o desactiva la transformación de fase
        void switchPHAT();
        //Devuelve si la transformación de fase está actica
        bool getPHAT();

        //Activa o desactiva la transformación de fase
        void switchSourceCancellation();
        //Devuelve si la transformación de fase está actica
        bool getSourceCancellation();

        //Realiza la estimación de DOA. Devuelve el máximo valor encontrado.
        float execute();
};
```

...v0.9.3_vs_release\apps\myApps\srp-phat\src\srp_phat.cpp 1

```
#include "srp_phat.h"

srp_phat::srp_phat() {
    calculateAngles();

    if (!loadTau()) {
        calculateTau();
        saveTau();
    }
}

srp_phat::srp_phat(float* const channels, float* const y) {
    calculateAngles();

    if (!loadTau()) {
        calculateTau();
        saveTau();
    }

    setup(channels, y);
}

srp_phat::~srp_phat() {
}

void srp_phat::setup(float* const channels, float* const y) {
    x = (float(*)[K]) channels;
    P = (float(*)[resX]) y;

    const int n[] = {K};
    const fftw_r2r_kind kind_fft[1]  = { FFTW_R2HC };
    const fftw_r2r_kind kind_ifft[1] = { FFTW_HC2R };
    fft  = fftwf_plan_many_r2r(1, n, N,   (float*)x, NULL, 1, K, (float*)X, NULL,
      1, K, kind_fft,  FFTW_MEASURE);
    ifft = fftwf_plan_many_r2r(1, n, N*N, (float*)R, NULL, 1, K, (float*)r, NULL,
      1, K, kind_ifft, FFTW_MEASURE);
    //TODO: En realidad bastaría con hacer la mitad de iffts, intentar optimizarlo

}

void srp_phat::switchPHAT() {
    PHAT ^= true;
}

bool srp_phat::getPHAT() {
    return PHAT;
}

void srp_phat::switchSourceCancellation() {
    sourceCancellation ^= true;
    mu = (!sourceCancellation) * 0.5;
}
```

```
bool srp_phat::getSourceCancellation() {
    return sourceCancellation;
}

float srp_phat::execute() {
    for (int i = 0; i < N; i++)
        for (int j = 0; j < K; j++)
            x[i][j] *= hanning[j];

    correlations();

    if (DEBUG && ep == nullptr) {
        if (!(ep = engOpen(NULL))) {
            exit(-1);
        }
    }

    float maxVal = 0;
    int maxPosI=0, maxPosJ=0;
    for (int i = 0; i < resX; i++) {
        for (int j = 0; j < resY; j++) {
            P[i][j] = mu*P[i][j] + (1-mu)*functional(i, j);
            if (P[i][j] > maxVal) {
                maxVal = P[i][j];
                maxPosI = i;
                maxPosJ = j;
            }
        }
    }

    if (DEBUG) {
        sendMemoryToMatlab();
    }

    if (sourceCancellation) {
        cancelSource(maxPosI, maxPosJ);

        maxVal = 0;
        for (int i = 0; i < resX; i++) {
            for (int j = 0; j < resY; j++) {
                P[i][j] = functional(i, j);
                maxVal = P[i][j] > maxVal ? P[i][j] : maxVal;
            }
        }
    }

    if (DEBUG) {
        sendMemoryToMatlab();
    }

    return maxVal;
}
```

...v0.9.3_vs_release\apps\myApps\srp-phat\src\srp_phat.cpp 3

```
void srp_phat::calculateAngles() {
    float x, y;
    float f = resX/2 / tan(alpha / 180.0*PI);

    for (int i = 0; i < resX; i++)  {
        x = - (i - resX/2);
        for (int j = 0; j < resY; j++) {
            y = - (j - resY/2);
            phi[i][j] = x!=0 ? atan(y/x) : PI/2;
            theta[i][j] = y!=0? y/(f*sin(phi[i][j])) : x/(f*cos(phi[i][j]));
        }
    }
}

void srp_phat::calculateTau() {
    float phin[N]; //Ángulo de cada micrófono
    for (int i = 0; i < N; i++) {
        phin[i] = - 2*PI/N * i;
    }

    for (int x = 0; x < resX; x++) {
        for (int y = 0; y < resY; y++) {
            for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++) {
                    float IMTDF = an / c * sin(theta[x][y]) * (cos(phi[x][y] - phin
                     [i]) - cos(phi[x][y] - phin[j]));
                    tauF[x][y][i][j] = IMTDF * Fs;
                    tau[x][y][i][j] = static_cast<int>( round(tauF[x][y][i][j]) );
                    tau[x][y][i][j] += 1024 * (tau[x][y][i][j] < 0);
                }
            }
        }
    }

}

void srp_phat::saveTau() {
    std::string filename = "tau_" + std::to_string(K) + "_" + std::to_string((int)
      Fs) + "_"
                           + std::to_string(resY) + "_" + std::to_string(resX) +
                     "_"
                           + std::to_string((int)alpha) + "_xy.dat";
    std::ofstream outfile (filename, std::ofstream::binary);
    outfile.write(reinterpret_cast<char*>(tau),   resY*resX*N*N * sizeof(int));
    outfile.write(reinterpret_cast<char*>(tauF), resY*resX*N*N * sizeof(float));
}

bool srp_phat::loadTau() {
    std::string filename = "tau_" + std::to_string(K) + "_" + std::to_string((int)
      Fs) + "_"
                           + std::to_string(resY) + "_" + std::to_string(resX) +
                     "_"
                           + std::to_string((int)alpha) + "_xy.dat";
```

...v0.9.3_vs_release\apps\myApps\srp-phat\src\srp_phat.cpp 4

```
    std::ifstream infile(filename, std::ofstream::binary);
    infile.read(reinterpret_cast<char*>(tau),   resY*resX*N*N * sizeof(int));
    infile.read(reinterpret_cast<char*>(tauF), resY*resX*N*N * sizeof(float));
    return (bool)infile;
}

inline float srp_phat::functional(int x_ind, int y_ind) {
    float p = 0;
    for (int i = 0; i < N; i++) {
        for (int j = i+1; j < N; j++)   {
            p += r[ i ][ j ][ tau[x_ind][y_ind][i][j] ];
        }
    }

    // Nunca debería ser negativo, pero hay veces que pasa... por coger la r más
      cercana??
    return p>0? p : 0;
}

inline void srp_phat::correlations() {
    fftwf_execute(fft);

    if (PHAT) {
        for (int i = 0; i < N; i++) module_division(X[i]);
    }

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            hermitian_conj_multiplication(X[i], X[j], R[i][j]);
        }
    }

    fftwf_execute(ifft);
}

void srp_phat::cancelSource(int x, int y) {


    for (int n=0; n<N; n++) {
        for (int m = 0; m<N; m++) {

            for (int t=0; t<K; t++) {
                rp[n][m][t] = (N-1)*(N-1) * r[n][m][t];
            }

            for (int k=0; k<N; k++) {
                if (k != n) {
                    for (int t=0; t<K; t++) {
                        rp[n][m][t] -= (N-1) * interpoledCorrelation(k, m, t-tauF
                     [x][y][n][k]);
                    }
                }
            }
```

```
            for (int l=0; l<N; l++) {
                if (l != m) {
                    for (int t=0; t<K; t++) {
                        rp[n][m][t] -= (N-1) * interpoledCorrelation(n, l, t-tauF
                     [x][y][l][m]);
                    }
                }
            }

            for (int k=0; k<N; k++) {
                for (int l=0; l<N; l++) {
                    if (k!=n && l!=m) {
                        for (int t=0; t<K; t++) {
                            rp[n][m][t] += interpoledCorrelation(k, l, t-tauF[x][y]
                     [n][k]-tauF[x][y][l][m]);
                        }
                    }
                }
            }

            //for (int t = 0; t<K; t++) {
            //  rp[n][m][t] = rp[n][m][t] / (N*N);
            //}
        }
    }

    memcpy(r, rp, N*N*K*sizeof(float));
}

inline void srp_phat::hermitian_conj_multiplication(float * const x, float * const
  y, float * z) {
    z[0] = x[0] * y[0];
    for (int i = 1; i < K/2; i++) {
        z[i] = x[i] * y[i] - x[K-i] * -y[K-i];
    }
    z[K/2] = x[K/2] * y[K/2];
    for (int i = K/2+1; i < K; i++) {
        z[i] = x[K-i] * y[i] + x[i] * -y[K-i];
    }
}

inline void srp_phat::module_division(float* x) {
    if (x[0]>0.01) x[0] /= abs(x[0]);
    else 0;

    float module;
    for (int i = 1; i < K/2; i++) {
        module = sqrt(x[i]*x[i] + x[K-i]*x[K-i]);
        if (module>0.01) {
            x[i] /= module;
            x[K - i] /= module;
        }
```

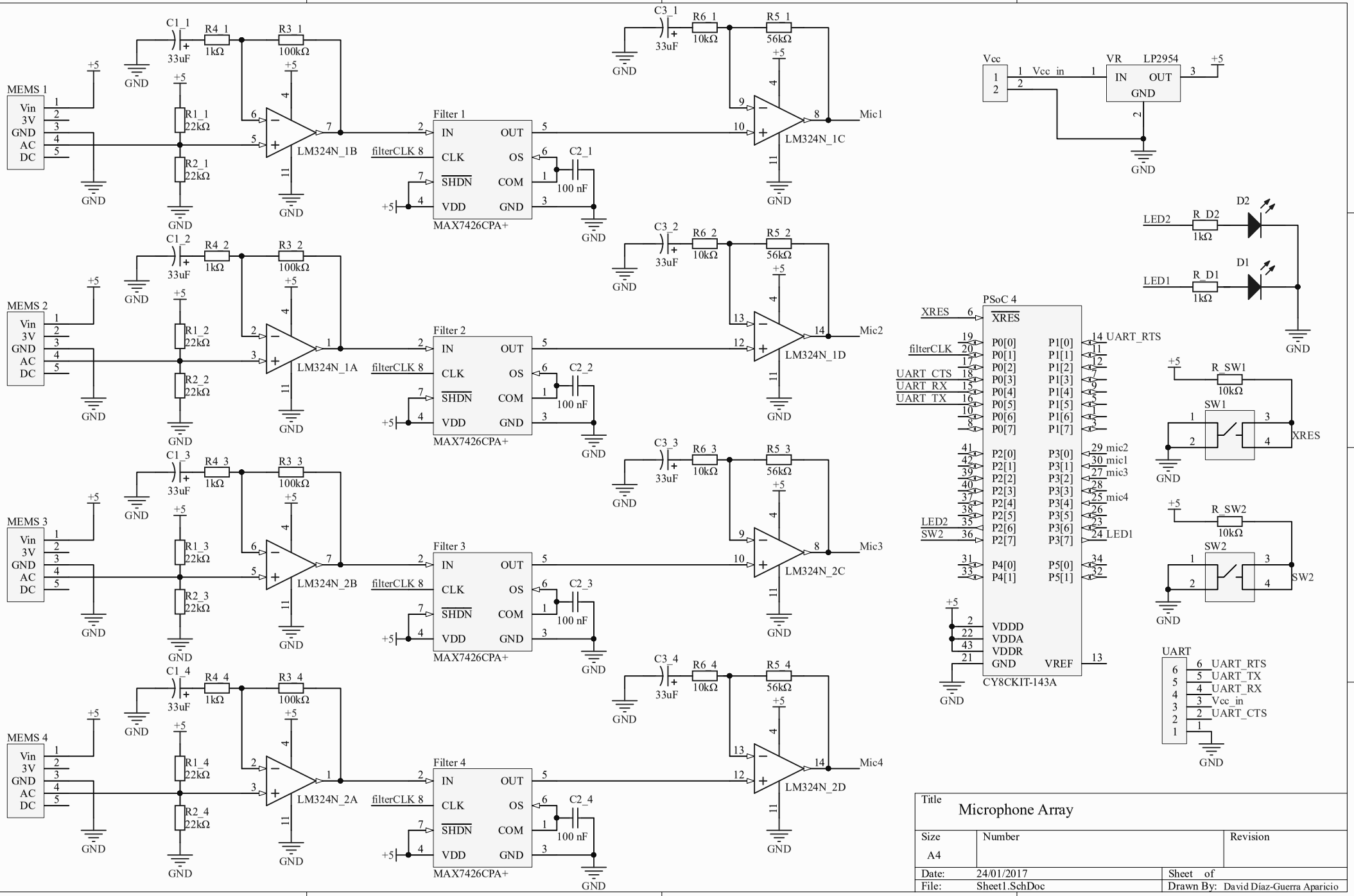
...v0.9.3_vs_release\apps\myApps\srp-phat\src\srp_phat.cpp 6

```
        else {
            x[i] = 0;
            x[K - i] = 0;
        }
    }

    if (x[K/2]>0.01) x[K/2] /= abs(x[K/2]);
    else 0;
}

inline float srp_phat::interpoledCorrelation(int i, int j, float n) {
    n += 1024 * (n<0);
    float out = (1-(n-int(n))) * r[i][j][int(n)] + (n-int(n)) * r[i][j][int(n)+1];
    return out;
}

void srp_phat::sendMemoryToMatlab() {
    size_t size_x[2] = { K, N }; // HAY QUE PONERLO AL REVÉS DE COMO SE HA DEFINIDO↵
       EN C !!!!!
    mxArray* mat_x = mxCreateNumericArray(2, size_x, mxSINGLE_CLASS, mxREAL);
    memcpy(mxGetPr(mat_x), &x[0][0], N*K * sizeof(float));
    int error = engPutVariable(ep, "x", mat_x);
    mxDestroyArray(mat_x);

    size_t size_X[2] = { K, N };
    mxArray* mat_X = mxCreateNumericArray(2, size_X, mxSINGLE_CLASS, mxREAL);
    memcpy(mxGetPr(mat_X), X, N*K * sizeof(float));
    engPutVariable(ep, "X", mat_X);
    mxDestroyArray(mat_X);

    size_t size_R[3] = { K, N, N };
    mxArray* mat_R = mxCreateNumericArray(3, size_R, mxSINGLE_CLASS, mxREAL);
    memcpy(mxGetPr(mat_R), R, N*N*K * sizeof(float));
    engPutVariable(ep, "R", mat_R);
    mxDestroyArray(mat_R);

    size_t size_r[3] = { K, N, N };
    mxArray* mat_r = mxCreateNumericArray(3, size_r, mxSINGLE_CLASS, mxREAL);
    memcpy(mxGetPr(mat_r), r, N*N*K * sizeof(float));
    engPutVariable(ep, "r", mat_r);
    mxDestroyArray(mat_r);

    size_t size_P[2] = { resX, resY };
    mxArray* mat_P = mxCreateNumericArray(2, size_P, mxSINGLE_CLASS, mxREAL);
    memcpy(mxGetPr(mat_P), P, resX*resY * sizeof(float));
    engPutVariable(ep, "P", mat_P);
    mxDestroyArray(mat_P);

    size_t size_tau[4] = { N, N, resX, resY };
    mxArray* mat_tau = mxCreateNumericArray(4, size_tau, mxINT32_CLASS, mxREAL);
    memcpy(mxGetPr(mat_tau), tau, resX*resY*N*N * sizeof(float));
    engPutVariable(ep, "tau", mat_tau);
    mxDestroyArray(mat_tau);
```

```
...v0.9.3_vs_release\apps\myApps\srp-phat\src\srp_phat.cpp                    7
}
```

# B

# Esquemático de la agrupación de 4 micrófonos


MEMS 1
MEMS 2
MEMS 3
MEMS 4
Vin
3V
GND
AC
DC
C1_1
C1_2
C1_3
C1_4
33uF
R4_1
R4_2
R4_3
R4_4
1kΩ
R3_1
R3_2
R3_3
R3_4
100kΩ
R1_1
R1_2
R1_3
R1_4
22kΩ
R2_1
R2_2
R2_3
R2_4
LM324N_1B
LM324N_1A
LM324N_2B
LM324N_2A
Filter 1
Filter 2
Filter 3
Filter 4
IN
OUT
CLK
OS
SHDN
COM
VDD
GND
filterCLK
MAX7426CPA+
C2_1
C2_2
C2_3
C2_4
100 nF
C3_1
C3_2
C3_3
C3_4
R6_1
R6_2
R6_3
R6_4
10kΩ
R5_1
R5_2
R5_3
R5_4
56kΩ
LM324N_1C
LM324N_1D
LM324N_2C
LM324N_2D
Mic1
Mic2
Mic3
Mic4
Vcc
Vcc_in
VR
LP2954
PSoC 4
XRES
P0[0]
P0[1]
P0[2]
P0[3]
P0[4]
P0[5]
P0[6]
P0[7]
P1[0]
P1[1]
P1[2]
P1[3]
P1[4]
P1[5]
P1[6]
P1[7]
P2[0]
P2[1]
P2[2]
P2[3]
P2[4]
P2[5]
P2[6]
P2[7]
P3[0]
P3[1]
P3[2]
P3[3]
P3[4]
P3[5]
P3[6]
P3[7]
P4[0]
P4[1]
P5[0]
P5[1]
VDDD
VDDA
VDDR
VREF
CY8CKIT-143A
UART_RTS
UART_CTS
UART_RX
UART_TX
mic2
mic1
mic3
mic4
LED1
LED2
SW2
D1
D2
R_D1
R_D2
R_SW1
R_SW2
SW1
UART
Vcc_in
Title
Microphone Array
Size
A4
Number
Revision
Date:
24/01/2017
Sheet of
File:
Sheet1.SchDoc
Drawn By:
David Díaz-Guerra Aparicio

# C

# PCB de la agrupación de 4 micrófonos

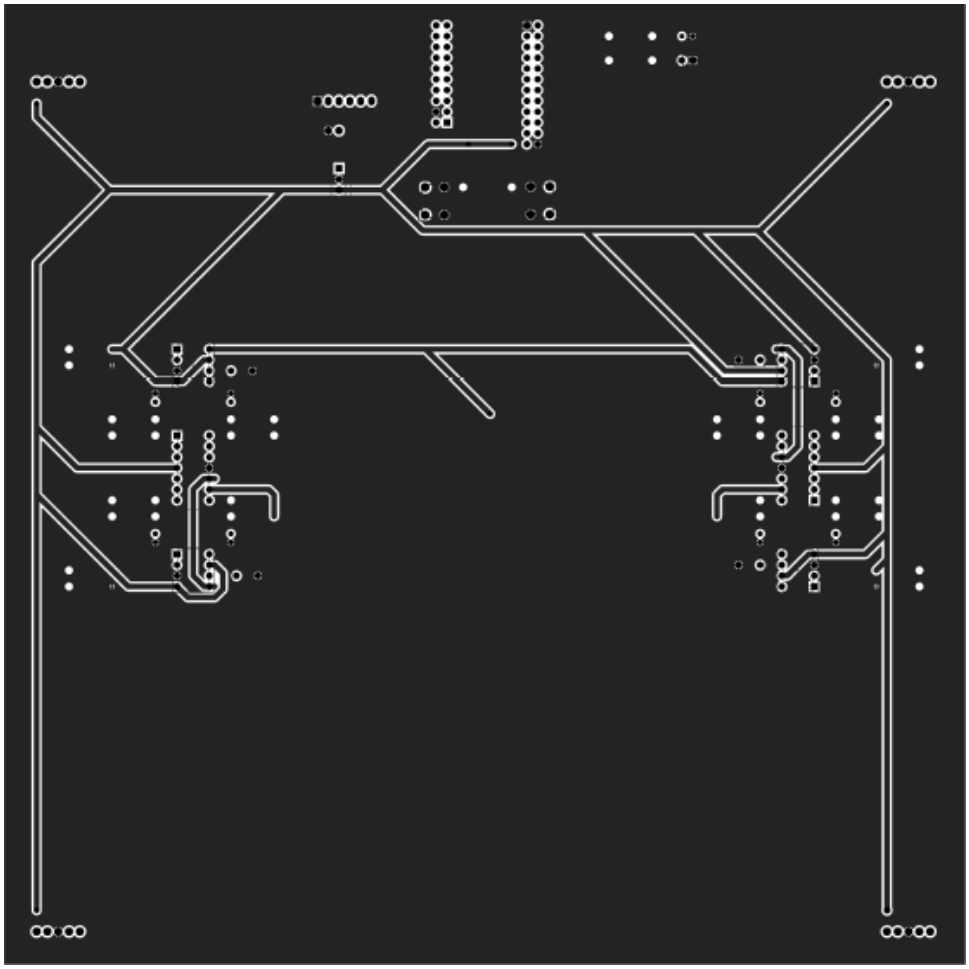


**Figura C.1:** Cara superior de la PCB diseñada para la agrupación de 4 sensores

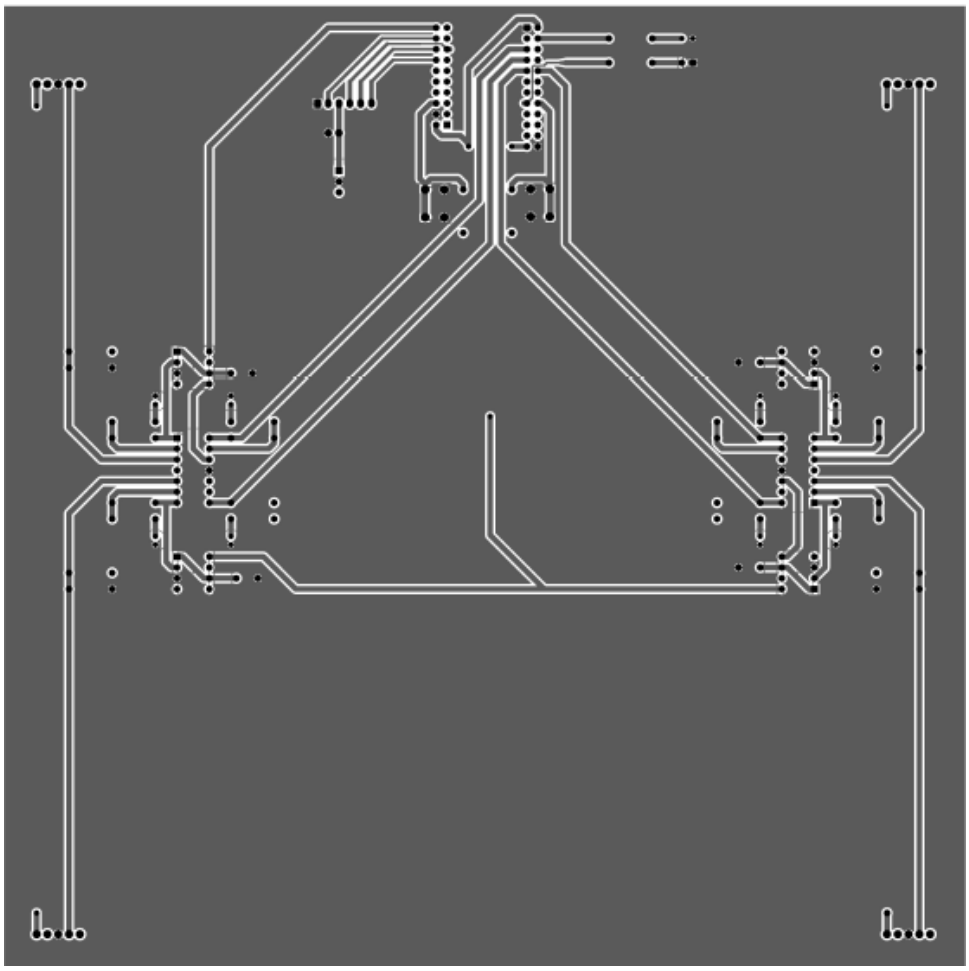


**Figura C.2:** Cara inferior de la PCB diseñada para la agrupación de 4 sensores

# D

# Código del programa del microprocesador

variables.h

```
/* ========================================
 *
 * Archivo: variables.h
 * Autor: David Díaz-Guerra Aparicio
 *
 * Definición de constantes y declaración de variables
 * del algoritmo SRP.
 *
 * ========================================
 */

#include <project.h>
#include "hanning.h"
#include "tau0.h"

#define N 4     //Número de sensores de la agrupación
#define K 256   //Tamaño de las ventanas
#define resT 32 //Núemro de puntos en theta
#define resP 32 //Número de puntos en phi

int16 x[N][K];          // Señal de entrada
int32 r[N][N][kLen];    // Correlación entre canales
int32 P[resT][resP];    // Estimación de potencia en cada dirección

int32 maxVal, maxPosT, maxPosT1, maxPosT2, maxPosP, maxPosP1, maxPosP2;

/* [] END OF FILE */
```

main.c

```
/* ========================================
 *
 * Archivo: main.c
 * Autor: David Díaz-Guerra Aparicio
 *
 * Programa del PSoC 4. Algoritmo SRP.
 *
 * ========================================
 */

#include "project.h"
#include "variables.h"

/* Macro definitions */
#define LOW                 (0u)
#define HIGH                (1u)
#define CHANNEL_1           (0u)
#define CHANNEL_2           (1u)
#define CHANNEL_3           (2u)
#define CHANNEL_4           (3u)
#define NO_OF_CHANNELS      (4u)
#define CLEAR_SCREEN        (0x0C)
#define CONVERT_TO_ASCII    (0x30u)

#define abs(x) ((x)<0 ? -(x) : (x))

/* Send the channel number and voltage to UART */
static void SendCoord(int32 maxPosT, int32 maxPosP);

/* Interrupt prototypes */
CY_ISR_PROTO(ADC_ISR_Handler);

/* Global variables */
volatile uint8 dataReady = 1u;
int16 buffSample = 0;

int16 n = 0, m = 0, ki = 0, l = 0, t = 0, p = 0, i = 0, j = 0;

int main()
{
    /* Start the Components */
    OpAmp_VreffFolower_Start();
    ADC_Start();
    UART_Start();
    LED_Write(0u);//LED_Write(1u);
    DEBUG_Write(0u);

    /* Start ISRs */
    ADC_IRQ_StartEx(ADC_ISR_Handler);

    /* Enable global interrupts */
```

main.c

```
    CyGlobalIntEnable;

    /* Start ADC conversion */
    ADC_StartConvert();

    for (;;) {

        /* Lectura del conversor */
        while(dataReady == 0u);
        x[CHANNEL_1][buffSample] = (ADC_GetResult16(CHANNEL_1) * hanning[buffSample]) >>13;
        x[CHANNEL_2][buffSample] = (ADC_GetResult16(CHANNEL_2) * hanning[buffSample]) >>13;
        x[CHANNEL_3][buffSample] = (ADC_GetResult16(CHANNEL_3) * hanning[buffSample]) >>13;
        x[CHANNEL_4][buffSample] = (ADC_GetResult16(CHANNEL_4) * hanning[buffSample]) >>13;
        buffSample = (buffSample+1)%K;
        dataReady = 0u;

        /* DOA */
        if (buffSample==0)
        {
            ADC_StopConvert();
            //DEBUG_Write(1u);

            /* Correlaciones cruzadas*/
            for (n = 0; n<N; n++) {
                for (m = 0; m<N; m++) {

                    for (ki = 0; ki <= kLen; ki++) {
                        r[n][m][ki] = 0;
                        for (l = 0; l<K; l++) {
                            r[n][m][ki] += x[n][l] * x[m][abs((l + k[ki]) % K)];

                        }

                    }
                }
            }

            /* SRP */
            maxVal = -2147483648;
            maxPosT1 = 0;
            maxPosT2 = 0;
            maxPosP1 = 0;
            maxPosP2 = 0;
            for (t = 0; t<resT; t++) {
                for (p = 0; p<resP; p++) {
                    P[t][p] = 0;
                    for (i = 0; i<N; i++) {
                        for (j = i + 1; j<N; j++) {
```

main.c

```
                                P[t][p] += r[i][j][tau0[t][p][i][j]];
                            }
                        }
                        if (P[t][p] > maxVal) {
                            maxVal = P[t][p];
                            maxPosT1 = t;
                            maxPosT2 = t;
                            maxPosP1 = p;
                            maxPosP2 = p;
                        } else if (P[t][p] == maxVal) {
                            if (t < maxPosT1) maxPosT1 = t;
                            else if (t > maxPosT2) maxPosT2 = t;
                            if (p < maxPosP1) maxPosP1 = p;
                            else if (p > maxPosP2) maxPosP2 = p;
                        }
                    }
                }

                maxPosT = (maxPosT1+maxPosT2)>>1;
                maxPosP = (maxPosP1+maxPosP2)>>1;
                SendCoord(maxPosT, maxPosP);

                //DEBUG_Write(0u);
                //LED_Write(LED_Read()^1);
                ADC_StartConvert();

            }

        }
    }

static void SendCoord(int32 maxPosT, int32 maxPosP)
{
    volatile int test = 0;

    /* Clear screen */
    UART_UartPutChar(CLEAR_SCREEN);

    /* Send Theta to UART */
    test = (maxPosT/10u) + CONVERT_TO_ASCII;
    UART_UartPutChar((maxPosT/10u) + CONVERT_TO_ASCII);
    maxPosT %= 10u;
    test = (maxPosT) + CONVERT_TO_ASCII;
    UART_UartPutChar((maxPosT) + CONVERT_TO_ASCII);
    UART_UartPutString(", ");

    /* Send Phi to Uart */
    test = (maxPosP/10u) + CONVERT_TO_ASCII;
    UART_UartPutChar((maxPosP/10u) + CONVERT_TO_ASCII);
    maxPosP %= 10u;
    test = maxPosP + CONVERT_TO_ASCII;
```

main.c

```
    UART_UartPutChar(maxPosP + CONVERT_TO_ASCII);
    UART_UartPutCRLF(0u);
}

CY_ISR(ADC_ISR_Handler)
{
    uint32 intr_status;

    /* Read interrupt status registers */
    intr_status = ADC_SAR_INTR_REG;
    /* Check for End of Scan interrupt */
    if((intr_status & ADC_EOS_MASK) != 0u)
    {
        dataReady = 1u;
    }
    /* Clear handled interrupt */
    ADC_SAR_INTR_REG = intr_status;
}
/* [] END OF FILE */
```