UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Análisis de Fourier Aplicado a la Convolución en Tiempo Real

TESIS
QUE PARA OBTENER EL TÍTULO DE
LICENCIADO
PRESENTA

Juan Pablo Martínez Fichtl

ASESOR

Caleb Antonio Rascón Estebané

Ciudad de México, 2023

Índice general

Capítulo 1

Introducción

Hablar es gratis.

El problema principal que abordaremos en la tesis es desarrollar la teoría necesaria para explicar un algoritmo existente de convolución que nos permite simular en tiempo real (con retrasos que equivalen a mover una fuente sonora un par de metros de distancia) un gabinete en el cual se encuentra montada una bocina. Tal explicación basada en términos matemáticos no está comúnmente disponible y es la aportación principal de esta tesis. El modelo de bloques y una excelente explicación de este y otros algoritmos de convolución se puede encontrar en (Wefers, 2015). El trabajo de Frank Wefers fue una inspiración importante para mi intento de llenar los agujeros que yo percibía en el texto. Con esta misma motivación es que trabajé en codificar el algoritmo en C. Aunque existe otra implementación abierta del mismo algoritmo llamada BruteFIR; está principalmente destinada a realizar corrección de cuarto para sistemas con un número de bocinas. Los mismos datos que proveen dan un uso del procesador de 30 % para un buffer de 128 muestras y una respuesta de impulso con 64 particiones. Esto es aproximadamente $170ms$. La implementación aquí presentada logra entre 10 % y 20 % de uso de procesador (un Core i3 3110M) para un buffer del mismo tamaño y de 188 particiones (aprox $500ms$), también tenía la meta de que fuera fácil de entender.

Para la parte de análisis de Fourier seguí muy de cerca a (Körner, 1989) y a (Hsu, Mehra, Velasco Coba y col., 1987) para verificar ciertas cuentas. Para la sección de procesamiento de señales consulté un número de textos, incluyendo textos propiamente de procesamiento de señales como (Proakis y Manolakis, 2006), que trata con bastante cuidado el tema del teorema del muestreo. El trabajo de (Orfanidis, 1995) fue una inspiración para la sección de cómo procesar código, que a su vez está escrita emulando un poco el estilo de Zed Shaw. Para un acercamiento

matemático el tema de las señales, consulté los textos de Shilov de análisis funcional (Shilov, 1974) y de álgebra lineal (Shilov, 1977) para consultar el teorema del muestreo y revisar ciertas propiedades de matrices y transformaciones lineales. Las notas de clase de (Rascón Estebané, s.f.) fueron absolutamente indispensables para implementar el código, al igual que su apoyo constante.

Comencemos propiamiente la introducción observando que hay una gran variedad de sistemas físicos que se pueden entender a través de la idea de reverberación. El problema de la propagación de ondas (y señales, en general) en un sistema capaz de reflejarlas surge naturalmente en campos tales como la acústica y los circuitos eléctricos. Por causas diferentes, cada uno de estos sistemas generará un número de copias retrasadas de la señal original. En una sala de conciertos, tales reflexiones se deben al tiempo que tarda un sonido en rebotar de una superficie y llegar a nuestros oídos. Un circuito eléctrico, por otro lado, tiene componentes con diferentes impedancias y estructuras internas que causan retrasos o incluso reflexiones cuando se aplica un voltaje. Los primeros pasos que tomamos para abordar la descripción matemática de estos fenómenos consisten de identificar a una función $\sigma(t)$ que se transformará en $\bar{\sigma}(t)$ bajo la acción del sistema. Tales funciones se conocerán como señales. Su dominio de definición pueden ser los reales (en el caso continuo/analógico) o los naturales (en el caso discreto/digital). Estos dominios están relacionados por el célebre teorema del muestreo, establecido por Claude Shannon y Harry Nyquist.

La convolución se puede pensar como una transformación que lleva $\sigma(t)$ a $\bar{\sigma}(t)$ de una forma que se puede considerar lineal. Entendemos por lineal cualquier transformación que consista de ajustes de volumen asociados a retrasos fijos. Para ilustrar este punto, consideremos que incrementar arbitrariamente la amplitud de una señal en sistemas reales generalmente tiene efectos destructivos. Una onda de presión con una amplitud suficientemente alta puede destruir (o por lo menos dañar) una sala de conciertos; mientras que un voltaje demasiado alto puede hacer que ciertos componentes de un circuito exploten o se derritan debido al calor. Para obtener las condiciones que mantienen a las amplitudes por abajo de este límite, imaginemos que estamos en un cuarto con paredes de madera delgada. Si reproducimos una señal $\sigma_1(t)$ con amplitud suficientemente grande, podremos generar una deformación de las paredes. [1] Esto causará una variación en las distancias recorridas por las reflexiones para llegar a nuestros oídos. por lo que el sistema actuará diferente sobre una segunda señal $\sigma_2(t)$ mientras se esté reproduciendo $\sigma_1(t)$. En

[1] Un ejemplo bastante dramático de este fenómeno ocurrió en 1985 en la ciudad de Gotenburgo durante un concierto de Bruce Springsteen. Una combinación de rock n roll a altos volúmenes y depósitos de arcilla debajo del estadio que se acoplaron con la E-Street Band causaron daños estructurales al estadio de Nya Ullevi. Las reparaciones y modificaciones evitaron que hubiera conciertos de rock hasta que Pink Floyd volviera a tocar ahí en 1994.

términos matemáticos, no se cumplirá que $\sigma_1(t) + \sigma_2(t) \to \bar{\sigma_1}(t) + \bar{\sigma_2}(t)$. Por otro lado, si conectamos un amplificador a una bocina que no es capaz de soportar la potencia que este produce, eventualmente el cono se romperá o algún componente electrónico fallará. En el mejor de los casos, la amplitud llegará a su máximo y el control de volumen solamente añadirá más distorsión. En este caso se deja de cumplir que $\lambda\sigma(t) \to \lambda\bar{\sigma}(t)$. La familia de funciones buscada se puede caracterizar con las condiciones:

(1) $\sigma_1(t) + \sigma_2(t) \to \bar{\sigma_1}(t) + \bar{\sigma_2}(t)$,

(2) $\lambda\sigma(t) \to \lambda\bar{\sigma}(t)$.

Por lo que, en términos matemáticos, el sistema tiene que actuar linealmente[2] sobre las señales. Vamos a discutir a detalle estos problemas a lo largo del texto.

En el capítulo 2 desarrollaremos de la teoría necesaria de análisis de Fourier que vamos a necesitar para el resto de la tesis. Usaremos principios geométricos nativos al álgebra lineal para organizar heurísticamente los resultados importantes. En el capítulo 3 exploraremos las propiedades del álgebra que resulta de añadir la operación de convolución al espacio de funciones integrables (L_1). Con esto, podemos definir el concepto de *unidades aproximadas*; esto efectivamente formaliza la delta de Dirac para series de Fourier. Se cubrirán los detalles teóricos necesarios para abordar los conceptos de procesamiento de señales en capítulos posteriores.

En el capítulo 4 discutiremos las ideas del procesamiento de señales discretas. Comenzaremos por estudiar el teorema del muestreo, luego introducimos la idea del *procesamiento por bloques* para entender mejor *cómo* es que una computadora procesa señales. Discutimos la forma matricial de la convolución y los problemas de distorsión que puede causar en ciertas implementaciones. Finalmente presentamos el algoritmo de UPOLS (por sus siglas en inglés, *uniformly partitioned overlap save*) para calcular en tiempo real la convolución de una respuesta de impulso finita con una señal arbitraria.

El capítulo 5 consiste de conclusiones y un breve panorama de trabajo futuro posible. Por último, el apéndice A contiene un breve tutorial para el lector interesado en tratar de implementar el código expuesto en un sistema que use [JACK/C] pero no sabe por dónde comenzar. Las notas en (Rascón Estebané, s.f.) son un excelente recurso para comenzar. El apéndice B contiene el código en C para la sonda de respuestas de impulso y la convolución en tiempo real de una respuesta de impulso finita con una señal arbitraria.

[2]Hay sistemas que presentan efectos no-lineales deseables. Por ejemplo, los amplificadores valvulares son altamente preciados por el sonido que producen al saturarse[3]; existen incluso una variedad de circuitos conocidos como *overdrives* que son diseñados específicamente para empujar amplificadores a su rango no-lineal. Por otro lado, los circuitos de *fuzz* o *distorsión* producen efectos no-lineales diferentes a través de la retroalimentación (y saturación) de un transistor o amplificador operacional.

1.1. Objetivos y métodos

El objetivo de este trabajo no es solamente dar una discusión teórica sobre los fenómenos previamente expuestos; queremos también describir adecuadamente cómo podemos llevar a cabo el cálculo en una computadora. Para hacer esto, abordaremos con detalle la manera óptima de llevar a cabo todas las operaciones necesarias y también la implementación en un lenguaje de programación específico. Favoreceremos el uso de [C/C++] sobre pseudocódigo; ya el argumento de presunta generalidad de este último pierde sentido cuando consideramos que la mayoría de las aplicaciones de procesamiento en tiempo real usan predominantemente [C/C++][4]. A grandes rasgos, necesitamos responder satisfactoriamente las siguientes preguntas:

(i) ¿Cómo se caracterizan las funciones $\sigma(t)$ que pueden ser señales?

(ii) ¿Cómo podemos calcular explícitamente la transformación que lleva $\sigma(t)$ a $\hat{\sigma}(t)$?

(iii) ¿Cómo se puede optimizar esta transformación?

(iv) ¿Cómo se implementa en tiempo real?

(v) ¿Cómo se obtiene esta transformación para un sistema físico?

Responder a las preguntas *(i)* y *(ii)* requiere desarrollar ciertos conceptos de análisis; la teoría necesaria se desarrollará en los capítulos 2 y 3. En el capítulo 2, apoyándonos en la teoría de espacios vectoriales con dimensión finita, introducimos a las series de Fourier como una *proyección* de una función sobre una *base ortonormal* del espacio. Usamos este método heurístico para llevar la analogía hasta la transformada de Fourier. En el capítulo 3, seguimos a (Körner, 1989) para obtener el teorema de Fejér, el resultado más importante para la convergencia de las series de Fourier.

Las preguntas *(iii)* y *(iv)* requieren establecer analogías discretas con resultados obtenidos en los capítulos anteriores; apoyándonos en la intuición detrás de la multiplicación de polinomios y matrices, desarrollamos la teoría necesaria para describir el algoritmo (Wefers, 2015) que usaremos para implementar la convolución en tiempo real. El código necesario para correr el algoritmo se discute brevemente en el capítulo 4; mientras que una introducción a JACK/C se puede

[4]La mayoría del desarrollo de audio en Linux usa clientes de Pipewire o Jack escritos en [C/C++]. Lo mismo aplica para el desarrollo de *plugins* usados en la producción de audio, como todos los que están basados en JUCE. Aunque hay algunos procesadores que usan lenguajes específicos, hay muchos otros (por ejemplo, daisy y bela) que corren código en [C/C++]

encontrar en el apéndice C. Por último, una solución satisfactoria a la pregunta v fue descrita en (Farina, 2000); la lectora interesada puede consultar el apéndice D.4 para ver la implementación específica a JACK. El resto del código usado para el procesamiento en tiempo real también se puede encontrar en el apéndice C.

1.2. Motivaciones

1.2.1. Reverberación

La percepción del sonido está ligada inseparablemente con el espacio en el que se está escuchando. Con la excepción de una cámara anecoica o el espacio exterior, todo espacio tendrá superficies sobre las cuales se puede reflejar una onda de sonido. En algunos casos -como el gran cañón o ciertos callejones estrechos- podemos distinguir ecos individuales que siguen a nuestras voces. Cuando la cantidad de ecos por segundo es suficientemente grande, dejamos de escuchar ecos individuales y, en cambio, percibimos un solo sonido continuo. Una forma de visualizar esto es imaginarnos una bombilla que parpadea periódicamente. Si hacemos que la bombilla parpadee cada vez más rápido, llegará un momento en el que dejaremos de percibir el prendido y apagado; veremos entonces una luz aparentemente continua. La densidad de ecos necesaria para que esto suceda depende de la frecuencia y fue calculada por M. R. Schroeder y B. F. Logan (Schroeder y Logan, 1961) en su artículo seminal "'Colorless" Artificial Reverberation', el primero en presentar reverberación creada por una computadora.

Podemos usar el concepto de reflexión para obtener la forma general de una reverberación que actúa linealmente[5] sobre una cierta señal $\sigma(t)$. Sabemos que las reflexiones sucesivas se podrán representar como $\sigma(t - t_k)$, donde t_k es el tiempo que tarda en llegar a nuestros oídos. Para cada retraso debemos incluir también un factor de reducción de volumen $H(t_k)$. Podemos entonces expresar a la transformación como

$$\sigma(t) \to \sum_{k=0}^{N} H(t_k)\sigma(t - t_k).$$

[5]Todas las personas aficionadas al pop de los 80s estarán familiarizadas con la reverberación no lineal. El famoso *gated reverb* consiste de cortar la cola de la reverberación justo a tiempo para evitar que se encime con sonidos posteriores. Esto permite mantener la reverberación a un volumen alto sin reducir la definición de los instrumentos. Otros ejemplos de no linealidad son el reverb en reversa -su no-causalidad evita que exista naturalmente- y ciertos tanques de resortes en amplificadores de guitarra de los 1960s que reducían el volumen de la reverberación cuando pasaba señal de la guitarra (Esto se conoce como *envleope reverb*). Otro ejemplo célebre son los reverbs modulados, que efectivamente crean el sonido de un cuarto que cambia continuamente de tamaño; usado con resultados espectaculares por Vangelis.

Es muy importante notar que los valores de $H(t_k)$ cambiarán dependiendo de la distancia entre la emisión y la escucha del sonido. Trivialmente, en todo espacio convexo habrá una línea recta que une a estas posiciones, por lo que cada función H estará asociada a una configuración específica de emisión y escucha. Existe, sin embargo, la posibilidad de tener posiciones diferentes con funciones H idénticas. Consideremos el caso de un cuarto cilíndrico donde el piso y el techo son círculos, si colocamos a las posiciones de emisión y escucha un puntos diametralmente opuestos, obtenemos que la función H sólo depende de la distancia a la que se colocan y de su posición relativa al piso. Claramente otras simetrías darán lugar a funciones similares. Tomando esto bajo consideración, llegamos a la conclusión de que necesitamos una variedad de funciones H para caracterizar el comportamiento de un sistema.

Volviendo al cálculo explícito de la función H, podemos formar la expresión general

$$\sigma(t) \to \int_{\mathbb{R}} H(t)\sigma(x-t)dt.$$

La función H que contiene a todos los factores de volumen[6] para una reflexión con un retraso dado se conoce como *respuesta de impulso*. La razón detrás de este nombre se explicará en el capítulo 3. En el siguiente capítulo usaremos técnicas de análisis de Fourier para estudiar la respuesta de impulso de ciertos sistemas simples de reverberación.

1.2.2. Circuitos Eléctricos

Esta sección presenta ciertos resultados básicos de análisis de circuitos, para evitar cometer errores, consulté los primeros capítulos de (Scherz y Monk, 2016). Todo circuito eléctrico está compuesto de ciertos componentes básicos. Para simplificar el estudio de circuitos simples, podemos considerar los efectos de cada componente sobre un voltaje sinusoidal de referencia. Las resistencias introducirán una reducción de amplitud independiente de la frecuencia, mientras que capacitores e inductores inducirán una reducción de amplitud y un retraso de la onda, ambos dependientes de la frecuencia. Analizar dichos efectos puramente con sinusoides reales es una tarea de tedio considerable. Por otro lado, trabajar con números complejos[7] simplifica maravillosamente el trabajo; permitiéndonos presentar todos estos efectos con fórmulas cerradas muy simples. Podemos entonces

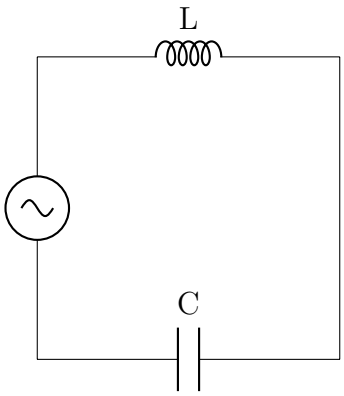
[6]En la práctica, calcular la convolución de H con una señal σ implica truncar a H. Esto se conoce como filtro de respuesta de impulso finita. Otros algoritmos de reverberación artificial (e.g. el de Schroeder-Logan) se pueden definir de forma recursiva; a esto se le conoce como filtro de respuesta de impulso infinita.

[7]Estos han sido criticados a lo largo de la historia por su falta de 'realidad física' pero uno haría bien en preguntarse si la expansión decimal infinita de π es más tangible que i.

representar nuestro voltaje sinusoidal como $V(t) = V_0 e^{i\omega t}$. Para los números complejos, la ley de Ohm se sigue expresando como $V = ZI$, con la diferencia de que la resistencia R se cambia por la *impedancia* Z y cada término potencialmente depende de el tiempo y la frecuencia. Conceptualmente, este análisis nos da el efecto de un componente sobre una fuente sinusoidal con frecuencia angular ω. La corriente que pasa por cada elemento sometido a un voltaje $V_0 e^{i\omega t}$ se puede usar para calcular las impedancias:

1. Resistencias: $I_R = \frac{V_0}{R} e^{i\omega t}$; $Z_R = R$.
2. Capacitores: $I_C = C\frac{dV}{dt} = i\omega C V_0 e^{i\omega t}$; $Z_C = -\frac{i}{\omega C}$.
3. Inductores: $I_L = \frac{1}{L}\int V dt = \frac{V_0}{i\omega L} e^{i\omega t} = -\frac{iV_0}{\omega L} e^{i\omega t}$; $Z_L = i\omega L$.

Consideremos el siguiente circuito:



Usando la regla para combinar impedancias de elementos en serie, obtenemos la impedancia equivalente del sistema

$$Z_{\text{tot}} = Z_L + Z_C = i\omega L - \frac{i}{\omega C} = i\left(\omega L - \frac{1}{\omega C}\right).$$

Para visualizar el efecto que tiene esta impedancia en un cierto voltaje de prueba, podemos graficar por separado la magnitud y fase de Z_{tot}. Podemos calcular la magnitud como la norma compleja

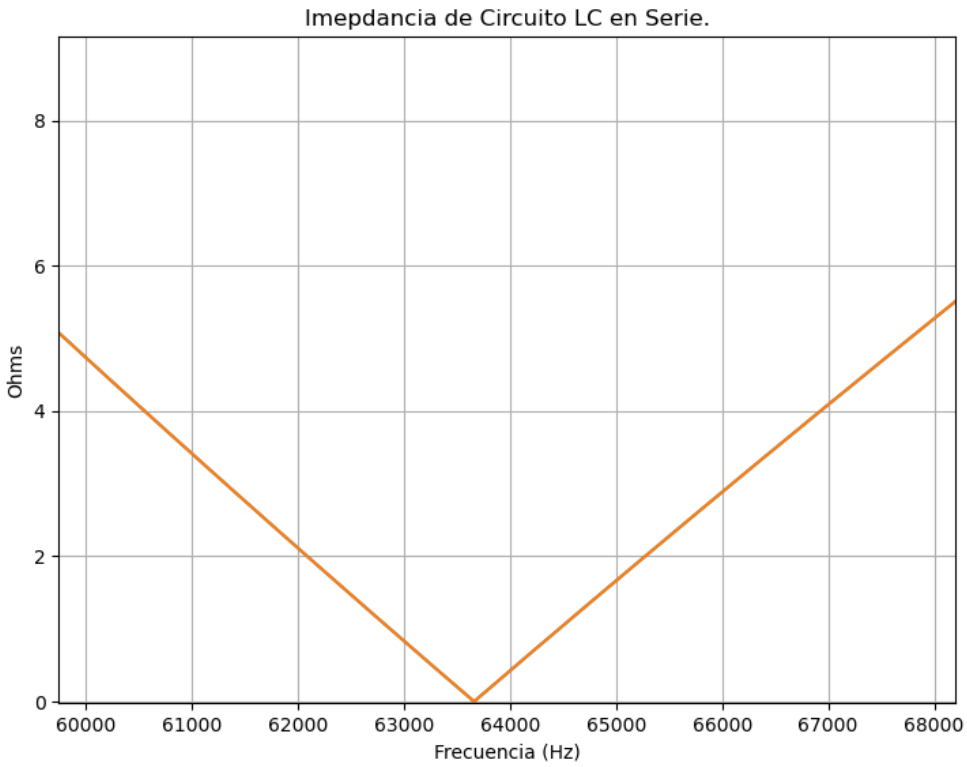
$$\|Z_{\text{tot}}\| = \left|\omega L - \frac{1}{\omega C}\right| = \left|\frac{\omega^2 LC - 1}{\omega C}\right|.$$

Claramente, el valor absoluto alcanzará su mínimo (0) cuando

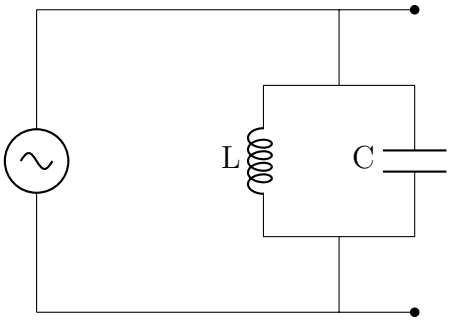
$$\omega = \frac{1}{\sqrt{LC}}, \quad f = \frac{1}{2\pi\sqrt{LC}}.$$

Para $L = 100\mu$H, $C = 62.5$nF, obtenemos

$$f \approxeq 63{,}662\text{Hz}.$$

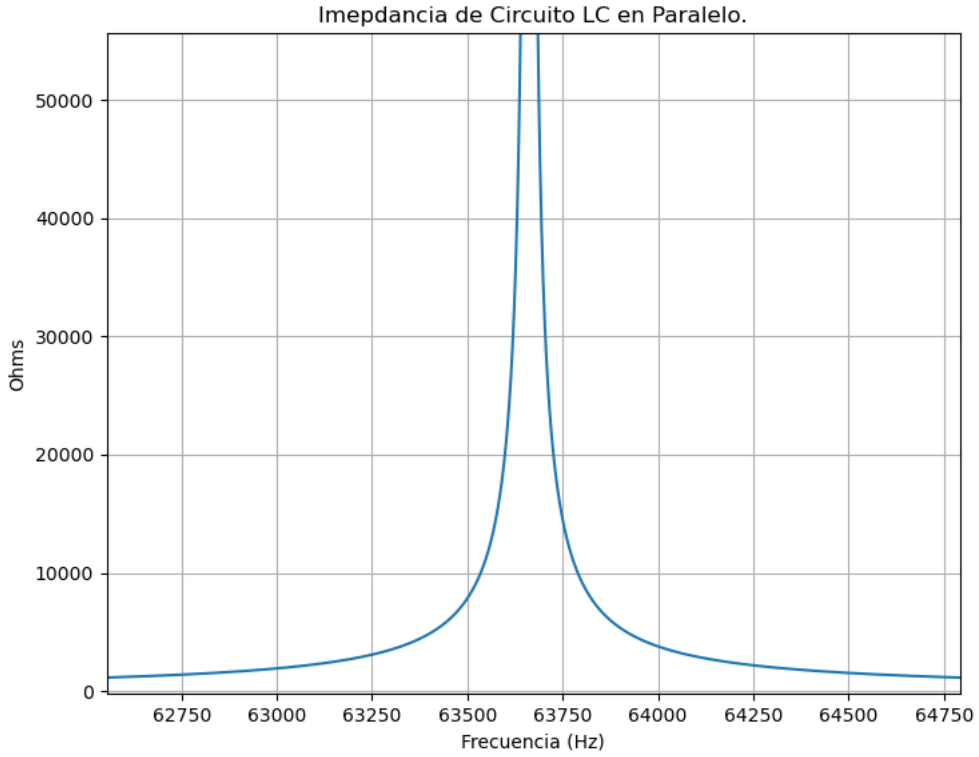


Por otro lado, en el siguiente circuito, la impedancia equivalente es:



$$Z_{\text{tot}} = \frac{Z_L Z_C}{Z_L + Z_C} = \frac{(L/C)}{i\omega L - \frac{i}{\omega C}} = \frac{(L/C)}{i(\omega L - \frac{1}{\omega C})} = \frac{-iL\omega}{\omega^2 LC - 1}.$$

Claramente la impedancia no estará acotada cuando $\omega \to 1/\sqrt{LC}$. Si tomamos a L y C como en el ejemplo anterior, obtenemos de nuevo la frecuencia $f \approxeq$ 63,662. El comportamiento de este circuito se puede entender considerando que para frecuencias no cercanas a f, la impedancia es muy pequeña y tiene el efecto de mandarlas a *tierra*; mientras que frecuencias cercanas a f pasarán de manera más o menos íntegra a las terminales.



Estas gráficas de respuesta nos dan una idea del efecto que tiene el sistema sobre fuentes sinusoidales, pero todavía queda la pregunta de cómo afectan a otro tipo de señales. Aquí es donde entra la idea de que toda función periódica se puede representar como una suma ponderada de senos y cosenos. Con este resultado podemos simplemente calcular el efecto sobre términos individuales de la suma y sumarlo para obtener el efecto sobre la señal. Veremos más de esto en los capítulos siguientes.

1.3. Sobre el código usado

A lo largo de este trabajo se usó una cantidad considerable de código para crear las gráficas e implementar ciertos algoritmos. El que aparece en el texto es el

que se ha considerado inseparable del trabajo y consiste de ejemplos sencillos de como comenzar a implementar algoritmos en JACK/C y el código necesario para obtener mediciones de la respuesta de impulso de un sistema acústico (o eléctrico, si se conecta la salida de las bocinas a algún procesador que acepte señales de línea) e implementar su cálculo en tiempo real. Cabe mencionar que todas las imágenes fueron producidas usando TikZ o python. El origen de cada una se puede encontrar directamente en el código fuente de Overleaf o en el repositorio JFichtl/jpmf_tesis. El código de procesamiento de audio se encuentra *aquí*, que es una carpeta de mi pequeño repositorio de efectos digitales JFichtl/noiseworks.

Capítulo 2

La geometría de las series de Fourier

2.1. Las series de Fourier como una proyección ortogonal

En el plano cartesiano $\mathbb{R}^2$, podemos obtener a las coordenadas de cualquier vector v sobre una base $\{e_1, e_2\}$ usando las proyecciones, via el producto interno usual en $\mathbb{R}^2$, i.e. $\langle \cdot, \cdot \rangle : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}^+$:

$$v_1 = \langle v, e_1 \rangle e_1, \quad v_2 = \langle v, e_2 \rangle e_2,$$

para representar al vector de la forma

$$v = v_1 e_1 + v_2 e_2.$$

Todo vector v en un espacio vectorial de dimensión finita n admite una representación de este tipo, considerando la base $\beta = \{e_1, e_2, ..., e_n\}$ i.e.:

$$v = \sum_{i=1}^{n} \lambda_i e_i,$$

donde los coeficientes λ_k se pueden obtener con la fórmula

$$\langle v, e_k \rangle = \lambda_k.$$

Sabemos, por otro lado, que todo espacio vectorial debe tener una base. La pregunta que surge naturalmente es si podemos llevar el concepto de proyección a espacios vectoriales de dimensión infinita. Consideremos al espacio vectorial $C[a, b]$

de funciones continuas definidas en $[a, b]$. El producto interno estándar en este espacio se puede definir como

$$\langle f, g\rangle = \int_a^b f(t)g(t)dt.$$

Mientras que la norma se define como

$$\|f\| = \sqrt{\langle f, f\rangle} = \left\{\int_a^b f^2(t)dt\right\}^{\frac{1}{2}}.$$

Más adelante lidiaremos con asuntos de convergencia. Por ahora podemos asociar a f una suma infinita que suplirá la representación en coordenadas. En concreto, tendremos

$$f \sim \sum_{n=0}^{\infty}\langle f, e_n\rangle e_n = \sum_{n=0}^{\infty}\left(\int_a^b f(t)e_n(t)dt\right)e_n(x) = \sum_{n=0}^{\infty}\int_a^b f(t)e_n(t)e_n(x)dt.$$

La posibilidad de representar a una función de esta forma se vuelve entonces un problema de convergencia que abordaremos más adelante. Por ahora, continuaremos nuestro programa geométrico usando la definición de producto interno en $C[a, b]$ para definir a un conjunto de funciones ortonormales $\beta = \{n \in \mathbb{N} \mid e_n\}$ en un espacio de funciones. Por analogía, una familia tal de funciones debe cumplir las condiciones

$$\int_a^b e_n(t)e_m(t)dt = 0 \quad si \quad n \neq m,$$

$$\int_a^b e_n^2(t)dt = 1 \quad \forall n \in \mathbf{N}.$$

Si escogemos $b = -a = \pi$, podemos usar el siguiente resultado elemental de cálculo

$$\int_{-\pi}^{\pi}\cos(nt)\,\mathrm{sen}(mt)dt = 0 \quad \forall\, n \neq m \in \mathbf{N},$$

para ver que la familia de funciones formada variando n y m en $\cos(nx), \mathrm{sen}(mx)$ es, de hecho, ortogonal. La normalización de las funciones se puede verificar usando las identidades

$$\cos^2(x) = \frac{1 + \cos(2x)}{2}, \quad \mathrm{sen}^2(x) = \frac{1 - \cos(2x)}{2},\ x \in \mathbb{R}.$$

Usando el hecho de que la integral del coseno es cero en el dominio considerado, podemos evitar calcular dos integrales. La norma al cuadrado de ambas funciones es entonces

$$\frac{1}{2}\int_{-\pi}^{\pi} 1+\lambda\cos(nt)\ dt = \pi + \frac{\lambda}{2}\int_{-\pi}^{\pi} t\cos t dt = \pi.$$

Como este resultado es independiente de λ, para todo $n > 0$ obtenemos

$$\int_{-\pi}^{\pi}\cos^2(nt)dt = \int_{-\pi}^{\pi}\mathrm{sen}^2(nt)dt = \pi.$$

Para el caso $n = 0$ tenemos una sola función constante e igual a uno. Tendrá entonces norma 2π. El sistema ortonormal buscado será

$$\beta = \left\{\frac{1}{\sqrt{2\pi}}, \frac{\cos t}{\sqrt{\pi}}, \frac{\mathrm{sen}\, t}{\sqrt{\pi}}, \frac{\cos(2t)}{\sqrt{\pi}}, \frac{\mathrm{sen}(2t)}{\sqrt{\pi}}, ...\right\}.$$

Ahora podemos calcular las proyecciones de una función $f \in C[-\pi,\pi]$ sobre el sistema ortonormal que obtuvimos. Separamos a las coordenadas en los términos a_0 y a_k y b_k asociados a β_0, β_c y β_s, obteniendo

$$a_0 = \langle f, \beta_0^0\rangle = \frac{1}{\sqrt{2\pi}}\int_{-\pi}^{\pi} f(t)dt,$$

$$a_n = \langle f, \beta_n^c\rangle = \frac{1}{\sqrt{\pi}}\int_{-\pi}^{\pi} f(t)\cos(nt)dt,$$

$$b_n = \langle f, \beta_n^s\rangle = \frac{1}{\sqrt{\pi}}\int_{-\pi}^{\pi} f(t)\,\mathrm{sen}(nt)dt.$$

La representación de f en este sistema de "coordenadas" ortonormales se conoce como polinomio trigonométrico, a saber:

$$f \sim a_0\beta_0^0 + \sum_{n=1}^{\infty}\left(a_n\beta_n^c + b_n\beta_n^s\right).$$

Esta expresión se puede expandir para obtener una expresión explícita. Nótese que el polinomio trigonométrico tiene variable x y la integral con la que se calculan los parámetros tiene variable de integración t.

$$P(x) = \frac{1}{2\pi}\int_{-\pi}^{\pi} f(t)dt + \frac{1}{\pi}\sum_{n=1}^{\infty}\int_{-\pi}^{\pi} f(t)(\cos(nt)\cos(nx) + \mathrm{sen}(nt)\,\mathrm{sen}(xt))dt$$

$$= \frac{1}{2\pi}\int_{-\pi}^{\pi} f(t)dt + \frac{1}{\pi}\sum_{n=1}^{\infty}\int_{-\pi}^{\pi} f(t)\cos(n(x-t))dt.$$

Usando la paridad de $\cos(nx)$ la expresión dentro de la integral en el segundo término de la suma se puede reescribir como

$$\int_{-\pi}^{\pi} f(t)\cos(\theta_0 - nt))dt = \int_{-\pi}^{\pi} f(t)\cos(nt - \theta_n))dt$$

con $\theta_n = nx$. Por otro lado, el primer término:

$$\frac{1}{2\pi}\int_{-\pi}^{\pi} f(t)dt$$

es simplemente el promedio de la función sobre el intervalo $(-\pi, \pi)$. En el contexto del procesamiento de señales recibe el nombre de componente directa, haciendo referencia a la corriente directa que puede estar presente en una señal eléctrica. En términos geométricos es la altura de la recta sobre la que ocurren el resto de las oscilaciones.

Podemos obtener otra expresión alternativa de los polinomios trigonométricos introduciendo la operación de convolución entre dos funciones $f : [a, b] \to \mathbb{R}$ y $g : [c, d] \to \mathbb{R}$. Definimos

$$(f * g)(t) = \int f(x)g(t - x)dx.$$

Para encontrar el dominio de $(f * g)$, observamos primero que x debe ir de a a b. Si $x = a$ y queremos que g tome como valor a c, necesitamos tomar $t = c + a$. Análogamente obtenemos $t = d + b$ cuando $x = b$. El intervalo de definición de una convolución tomada de ésta forma es $[a + c, b + d]$. Tal intervalo tiene longitud $(b - a) + (d - c)$. Tal cálculo equivale a tomar la integral sobre $\mathbb{R}$ y multiplicar por funciones características correspondientes. La convolución definida así se conoce como *lineal*. Una convolución con esta forma es inapropiada para resumir nuestros resultados sobre el cálculo de los coeficientes de Fourier. La razón de esto es que los coeficientes de Fourier se calcularon en el intervalo $[-\pi, \pi]$ y necesariamente todo término de la suma debe tener periodo 2π. El polinomio trigonométrico entonces solamente puede aproximar a una función f en un cierto intervalo. Por otro lado, si tomamos valores angulares arbitrarios obtenemos una extensión periódica de la función original. Para salvar esta situación, supongamos que esto es exactamente lo que queríamos hacer desde el principio y restrinjamos a las funciones bajo consideración a las periódicas dentro de un intervalo. Formalmente, a esto se le llama tomar funciones en $\mathbb{T} = \mathbb{R}/2\pi$. En la práctica y para un periodo T arbitrario, esto significa que podemos producir una función susceptible para el análisis de Fourier:

1. Tomando una función f continua (a lo menos, continua por pedazos) con un dominio de definición más grande que el periodo T que vamos a tomar.

2. Restringiendo esta función a un subconjunto del dominio con la forma $[a, b]$ y tal que cumpla $b - a = T$.

3. Definiendo una nueva función $\overline{f}$ como la extensión periódica de f restringida a $[a, b]$.

La convolución con estas restricciones adicionales se conoce como circular. Una diferencia importante entre las dos es que la convolución lineal es invariante bajo traslaciones pero la circular no.

Para ilustrar este hecho, tomemos $f(x) = |x|$ restringida a $[-\pi, \pi]$. Luego:

$$f(x) = \begin{cases} -x & -\pi < x \leq 0, \\ x & 0 < x \leq \pi. \end{cases}$$

Como $-\pi < x$, $0 < x + \pi$, la siguiente traslación será exclusivamente positiva:

$$f(x + \pi) = x + \pi \quad -\pi < x \leq \pi.$$

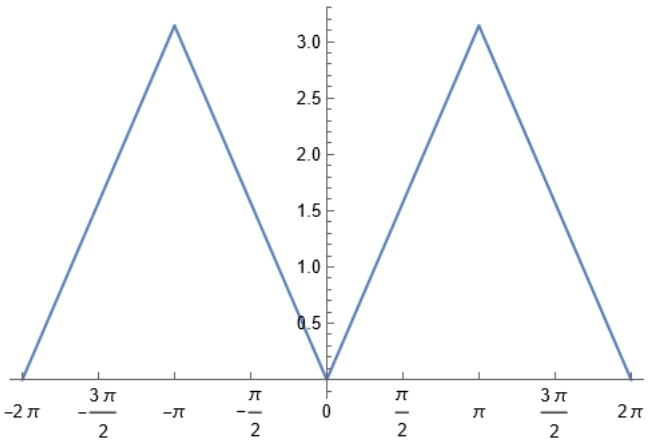


Figura 2.1: $f(x)$

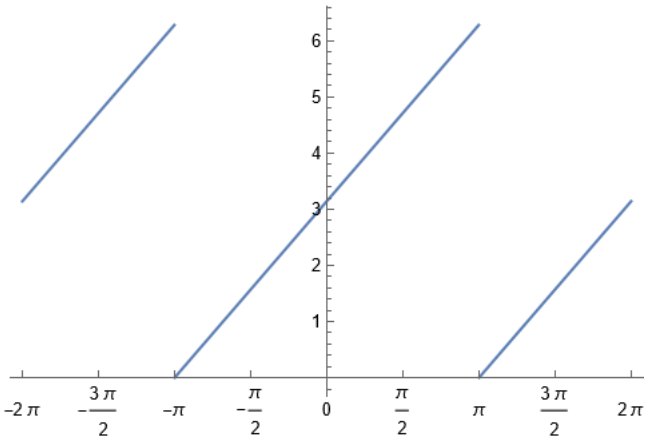


Figura 2.2: $f(x + \pi)$

Puesto que $f(-\pi) = f(\pi) = 0$, la función $f(x)$ alcanza el mismo valor en los extremos del intervalo $[-\pi, \pi]$, por lo que su extensión periódica debe ser continua. Por otro lado, los valores de la función $f(x+\pi)$ en los extremos del intervalo serán $f(-\pi + \pi) = 0$ y $f(\pi + \pi) = 2\pi$, como esta función alcanza valores diferentes en los extremos, su extensión periódica será discontinua. El polinomio trigonométrico asociado a cada función será obviamente diferente.

Habiendo aclarado este punto, continuamos con la simplificación del cálculo de coeficientes. La posibilidad de intercambiar la integral con el límite depende de la convergencia uniforme de la serie. Tales problemas serán abordados con más cuidado en el siguiente capítulo. Por ahora, tomemos

$$\sum_{k=1}^{n} \int_{-\pi}^{\pi} f(t) \frac{\cos(k[x-t])}{\pi} dt = \int_{-\pi}^{\pi} f(t) \sum_{k=1}^{n} \frac{\cos(k[x-t])}{\pi} dt.$$

Podemos entonces definir al núcleo de Dirichlet D_n como

$$D_n(x) = \frac{1}{\pi} \left(\frac{1}{2} + \sum_{k=1}^{n} \cos(kx) \right).$$

Reagrupando y usando la definición de convolución obtenemos

$$\frac{1}{2\pi} \int_{-\pi}^{\pi} f(t) dt + \frac{1}{\pi} \sum_{k=1}^{n} \int_{\pi}^{\pi} f(t) \cos(x-t) dt$$

$$= \int_{-\pi}^{\pi} f(t) \left(\frac{1}{2\pi} + \frac{1}{\pi} \sum_{k=1}^{n} \cos(k(x-t)) \right) dt = \int_{-\pi}^{\pi} f(t) D_n(x-t) dt = (f \star D_n)(x).$$

El polinomio trigonométrico de una función con un dominio de la forma $[a, b]$ se puede obtener con un cambio de variable sencillo. Tal cambio de variable es obvio cuando consideramos que, si las funciones $cos(nx)$ completan n oscilaciones completas en el intervalo $[-\pi, \pi]$, entonces el mapeo lineal definido por

$$t(x) = \frac{2\pi}{b-a}(x-a) - \pi,$$

nos asegurará n oscilaciones completas en el dominio $[a, b]$. Nótese que la *frecuencia fundamental* que induce la transformación $t(x)$ se puede obtener expandiendo

$$\frac{2\pi}{b-a}(x-a) - \pi = \left(\frac{2\pi}{b-a} \right) x - \pi \left(\frac{a+b}{b-a} \right).$$

Podemos definir

$$f = \frac{2\pi}{b-a},$$

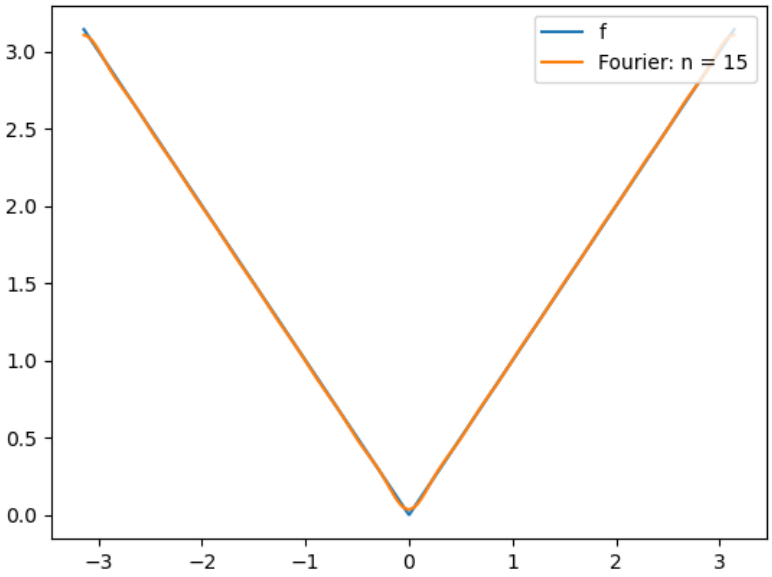


Figura 2.3: $f(x)$

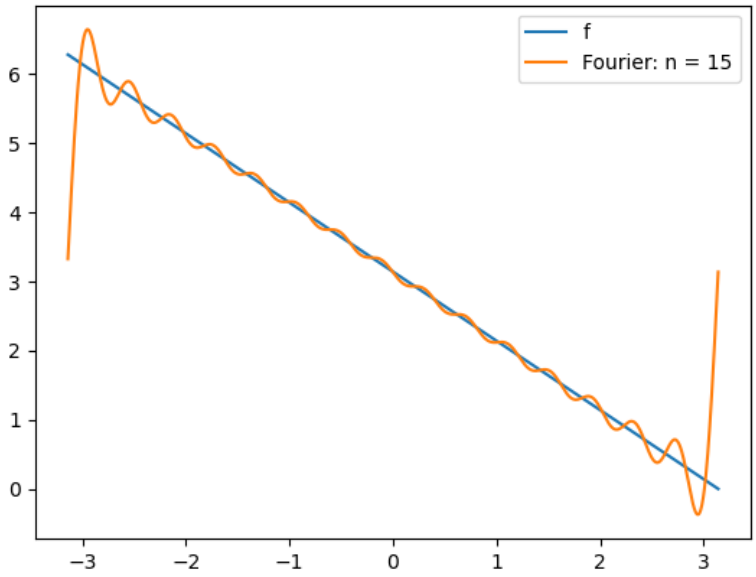


Figura 2.4: $f(x - \pi)$

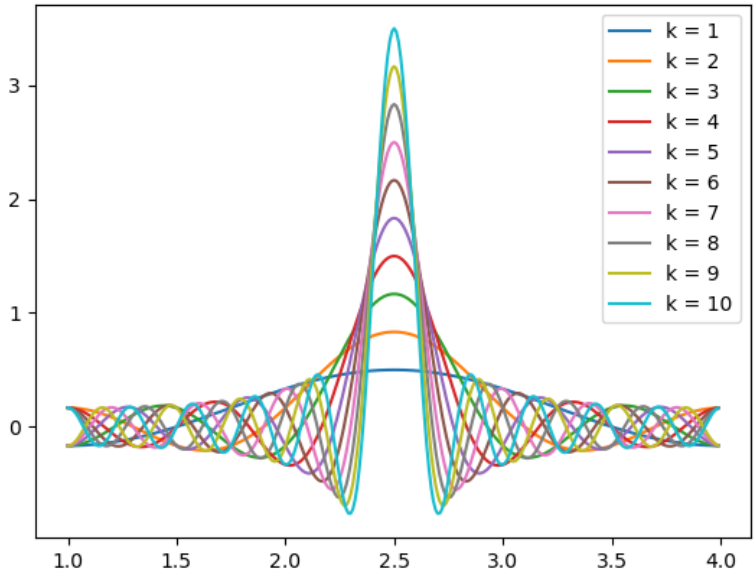


Figura 2.5: Núcleo de Dirichlet para diferentes valores de n.

$$\phi = \pi \left(\frac{a+b}{b-a} \right).$$

El núcleo de Dirichlet toma entonces la forma

$$D_k(x) = \frac{1}{b-a} \left(1 + 2 \sum_{n=1}^{k} \cos(n(fx - \phi)) \right).$$

La convolución circular $f * D_k$ nos permite encontrar los coeficientes del polinomio trigonométrico, también se puede usar para hacer una aproximación numérica eficiente de la aproximación de Fourier a una función. En el siguiente capítulo veremos cómo encontrar una expresión cerrada para este y otros núcleos.

Como último ejemplo del cálculo de coeficientes de Fourier, aplicaremos el mismo método a funciones con valores en los complejos. Tal entorno presenta ciertas facilidades para tratar con el cálculo de los coeficientes de Fourier. Comenzamos tomando a dos funciones $f(t), g(t)$ de la forma:

$$f(t) = f_1(t) + i f_2(t),$$
$$g(t) = g_1(t) + i g_2(t).$$

Para encontrar los coeficientes de Fourier con el mismo método que hemos estado usando, necesitamos usar el producto interno usual de los complejos; definido como:

$$\langle f, g\rangle = \int_{\mathbb{R}} f(t)\overline{g(t)}dt.$$

Con la propiedad de que

$$\int_{-\pi}^{\pi} e^{ikx}dx = \begin{cases} 2\pi & \text{cuando } k = 0, \\ 0 & \text{cuando } k \neq 0. \end{cases}$$

Si hacemos $k = n - m$, podemos obtener fácilmente un conjunto ortogonal de funciones cuyo término enésimo viene dado por $\beta_n = e^{inx}/\sqrt{2\pi}$. Verificamos:

$$\langle \beta_n, \beta_m\rangle = \frac{1}{2\pi}\int_{-\pi}^{\pi} e^{i(n-m)t}dt = \begin{cases} 0 & \text{cuando } n \neq m, \\ 1 & \text{cuando } n = m. \end{cases}$$

Normalizando, los coeficientes del polinomio de Fourier complejo toman la forma

$$c_n = \langle f, \beta_n\rangle = \frac{1}{\sqrt{2\pi}}\int_{-\pi}^{\pi} f(t)e^{-int}dt.$$

Existe una simetría entre los coeficientes que nos permite simplificar el cálculo del núcleo correspondiente. Si sumamos las proyecciones *reflejadas* $c_n\beta_n$ y $c_{-n}\beta_{-n}$ obtenemos

$$c_n\beta_n + c_{-n}\beta_{-n} = \frac{1}{2\pi}\int_{-\pi}^{\pi} f(t)e^{-in(t-x)}dt + \frac{1}{2\pi}\int_{-\pi}^{\pi} f(t)e^{in(t-x)}dt$$

$$= \frac{1}{\pi}\int_{-\pi}^{\pi} f(t)\left(\frac{e^{-in(t-x)} + e^{in(t-x)}}{2}\right)dt = \frac{1}{\pi}\int_{-\pi}^{\pi} f(t)\cos(n(x-t))dt.$$

Por otro lado, el término cero será exactamente igual que en el caso real. Esto demuestra que en el caso complejo volvemos a obtener el núcleo de Dirichlet. En este punto, es provechoso dar un paso atrás y explotar la forma compleja para obtener una fórmula cerrada para D_k. Primero, nótese que de la suma $2\pi D_n(t) = \sum_{k=-n}^{n} e^{int}$ obtenemos directamente

$$D_n(0) = \frac{1+2n}{2\pi}.$$

Expandiendo D_k y multiplicando por e^{int} obtenemos

$$2\pi e^{int}D_n(t) = e^{int}(e^{-int} + e^{i(-n+1)t} + \ ... \ + e^{i(n-1)t} + e^{int})$$

$$= 1 + e^{i2t} + \ ... \ + e^{i(2n-1)t} + e^{i2nt} = \sum_{k=0}^{2n} (e^{it})^k.$$

La última expresión claramente es una suma geométrica igual a

$$\frac{1 - e^{i(2n+1)t}}{1 - e^{it}} = \frac{e^{i(n+\frac{1}{2})t}}{e^{i\frac{t}{2}}} \frac{e^{-i(n+\frac{1}{2})t} - e^{i(n+\frac{1}{2})t}}{e^{\frac{-it}{2}} - e^{\frac{it}{2}}} = e^{int} \frac{e^{-i(n+\frac{1}{2})t} - e^{i(n+\frac{1}{2})t}}{e^{-i\frac{t}{2}} - e^{i\frac{t}{2}}}.$$

Por lo que:

$$2\pi D_n(x) = \frac{e^{-i(n+\frac{1}{2})t} - e^{i(n+\frac{1}{2})t}}{e^{-\frac{it}{2}} - e^{\frac{it}{2}}} = \frac{e^{i(n+\frac{1}{2})t} - e^{-i(n+\frac{1}{2})t}}{e^{\frac{it}{2}} - e^{-\frac{it}{2}}} = \frac{\operatorname{sen}((n+\frac{1}{2})t)}{\operatorname{sen}(\frac{t}{2})}.$$

En resumen, el núcleo de Dirichlet se puede expresar de forma cerrada como

$$D_k(t) = \begin{cases} \dfrac{\operatorname{sen}((n+\frac{1}{2})t)}{2\pi \operatorname{sen}(\frac{t}{2})} & t \neq 0, \\ \dfrac{1+2n}{2\pi} & t = 0. \end{cases}$$

La pregunta de la convergencia de una serie de Fourier se reduce a verificar

$$\hat{f}_n(x) = (f * D_n)(x).$$

Aunque sabemos que esta serie converge para algunas funciones sencillas, es natural preguntarnos si podría existir alguna f tal que la secuencia $\hat{f}_n$ no sea convergente, o bien, si existe algún otro núcleo K_n tal que la secuencia $f * K_n$ es convergente para toda f razonablemente *bien portada* (integrable, por ejemplo). Tales son las preguntas que se tratan en el siguiente capítulo. En general, las expresiones de la forma $(f * K_n)(x)$ son de gran interés en áreas tales como las ecuaciones diferenciales. Problemas como la ecuación de calor $u_t = u_{xx}$ se pueden resolver mediante la convolución con un núcleo apropiado. Las funciones de Green tienen un principio de funcionamiento similar. Una exposición al tema de las ecuaciones diferenciales parciales de la física puede encontrarse en (Sobolev, 1964).

2.2. Ejemplos del cálculo de coeficientes

En la práctica sólo es posible calcular explícitamente los coeficientes de Fourier para funciones muy sencillas. A continuación mostramos un par de formas cerradas. Las figuras 2.6 y 2.7 presentan datos del índice S&P 500, nótese que la serie de

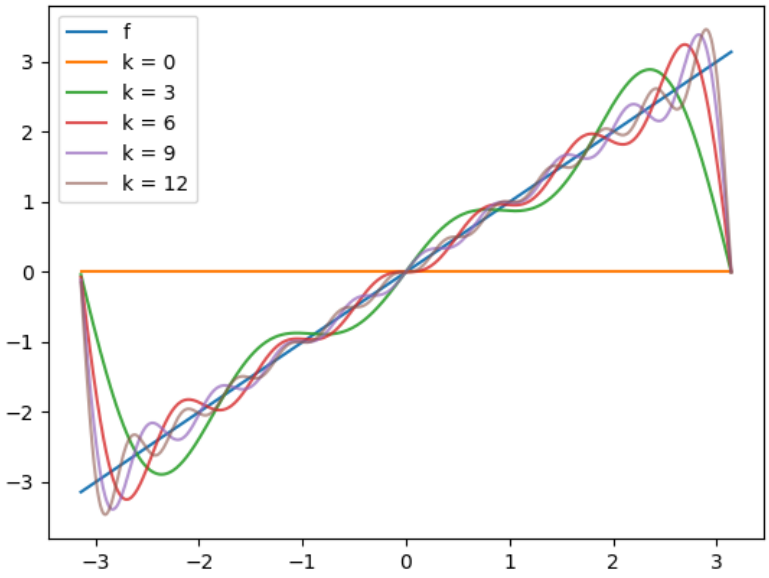


Figura 2.6: Serie de Fourier de $f(x) = x$.

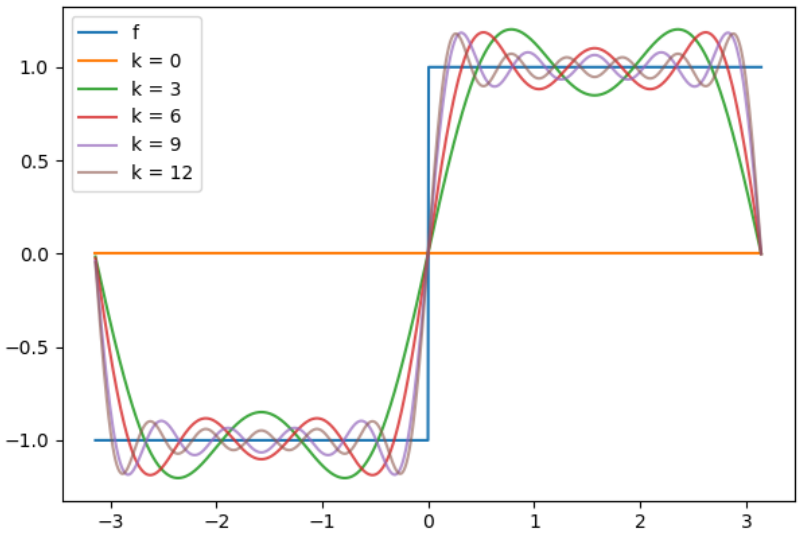


Figura 2.7: Suma de Fourier de una onda cuadrada.

Fourier tarda mucho en converger y presenta imprecisiones importantes en ambos extremos.

1. $f(x) = x$.

$$A_0 = \frac{1}{2\pi}\int_{-\pi}^{\pi} x dx = 0.$$

$$A_k = \frac{1}{\pi}\int_{-\pi}^{\pi} x\cos(kx)dx = \frac{1}{\pi}\left\{-xk^{-1}\operatorname{sen}(kx)\mid_{-\pi}^{\pi} + \int_{-\pi}^{\pi}\operatorname{sen}(kx)dx\right\} = 0$$

$$B_k = \frac{1}{\pi}\int_{-\pi}^{\pi} x\operatorname{sen}(kx)dx = \frac{1}{\pi}\left\{-xk^{-1}\cos(kx)\mid_{-\pi}^{\pi} + \int_{-\pi}^{\pi}\cos(kx)dx\right\}$$

$$= (\pi k)^{-1}(-\cos(kx))x\mid_{-\pi}^{\pi} = -2k^{-1}\cos(k\pi) = 2k^{-1}(-1)^{k+1}.$$

La serie de Fourier toma la forma

$$\hat{f}_n(x) = \sum_{k=1}^{n} 2k^{-1}(-1)^{k+1}\sin(kx).$$

2. $f(x) = \begin{cases} -1 & x \leq 0, \\ 1 & 0 < x. \end{cases}$

$$A_0 = \frac{1}{2\pi}\left\{\int_0^{\pi} dx - \int_{-\pi}^{0} dx\right\} = \frac{1}{2\pi}(\pi - \pi) = 0.$$

$$A_k = \frac{1}{\pi}\left\{\int_0^{\pi}\cos(kx)dx - \int_0^{\pi}\cos(-kx)dx\right\} = 0.$$

$$B_k = \frac{1}{\pi}\left\{\int_0^{\pi}\operatorname{sen}(kx)dx - \int_0^{\pi}\operatorname{sen}(-kx)dx\right\} = -\frac{2}{k\pi}\cos(kx)\mid_0^{\pi},$$

por lo que

$$B_k = \begin{cases} 0 & \text{si } k = 2n, \\ \frac{4}{k\pi} & \text{si } k = 2n+1. \end{cases}$$

En conclusión

$$\hat{f}_n(x) = \frac{4}{\pi}\sum_{k=0}^{n}\frac{\operatorname{sen}((2k+1)x)}{2k+1}.$$

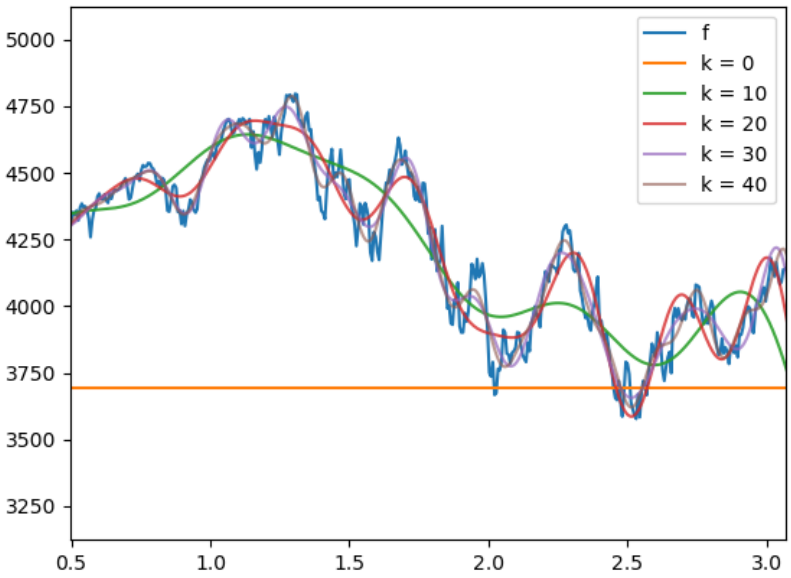


Figura 2.8: Series de Fourier del S&P 500. Fechas normalizadas a $[-\pi, \pi]$.

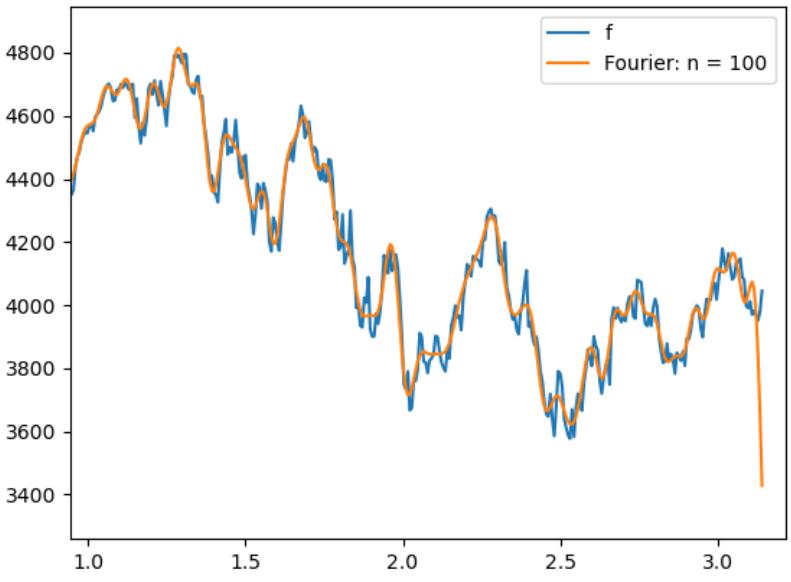


Figura 2.9: Serie de Fourier de orden 100 del S&P 500.

2.3. Reverberación artificial

Como una primera aproximación al tema del procesamiento de audio, consideremos el problema de recrear artificialmente la reverberación producida dentro de un espacio físico. En la introducción vimos que la transformación H asociada a la acción de un sistema lineal sobre una señal $\sigma(t)$ se puede expresar como $\sigma(t) \to (\sigma \star H)(t)$. El problema de crear reverberación artificial entonces se puede abordar tratando de encontrar ciertas condiciones sobre H para poder calcular tal función explícitamente. En (Schroeder y Logan, 1961) se dio por vez primera una solución al problema. A continuación damos esencialmente una exposición detallada del artículo.

Si reducimos el fenómeno a principios básicos, podemos imaginar que los términos de la suma de convolución son causados por una primera excitación que rebota en un juego de paredes paralelas. Se puede entonces definir a cada término de forma recursiva. Supongamos que cada vez que el sonido rebota, pierde volumen de acuerdo con un factor g que se mantiene constante durante el proceso. Escribimos explícitamente:

$$\sigma_0(t) = \sigma(t),$$
$$\sigma_1(t) = g\sigma_0(t-\tau) = g\sigma(t-\tau),$$
$$\sigma_2(t) = g\sigma_1(t-\tau) = g^2\sigma(t-2\tau),$$
$$\cdots\cdots\cdots\cdots$$
$$\sigma_n(t) = g\sigma_{n-1}(t-\tau) = g^n\sigma(t-n\tau).$$

Los términos sucesivos solamente convergen a cero cuando $g < 1$. Para $g = 1$, la señal se repite con exactamente el mismo volumen. Debido a inexactitudes computacionales, valores de g cercanos a 1 pueden evaluarse a 1 y subsecuentemente diverger. Podemos pensar en el valor $g = 1$ como un equilibrio inestable del proceso. Discutiremos otras condiciones explícitas en el capítulo siguiente. Supongamos que el tiempo que tarda la señal en salir de la fuente, chocar con la pared y regresat es τ. Tenemos la suma

$$\sigma(t) \to \sum_{k=1}^{n} g^k \sigma(t - k\tau).$$

Tomando su transformada de Fourier, obtenemos:

$$\sum_{k=1}^{n} g^k \sigma(t-k\tau) \to \sum_{k=1}^{n} g^k e^{-i\omega k\tau}\hat{\sigma}(\omega) = \hat{\sigma}(\omega)\sum_{k=1}^{n}(ge^{-i\omega\tau})^k.$$

Tomando el límite cuando $n \to \infty$ y usando la fórmula para el límite de una serie geométrica:

$$\lim_{n\to\infty} \hat{\sigma}(\omega) \sum_{k=1}^{n} (ge^{-i\omega\tau})^k = \hat{\sigma}(\omega)\, \frac{ge^{-i\omega\tau}}{1 - ge^{-i\omega\tau}}.$$

Usando el teorema de la convolución, podemos concluir que la función H (llamada *respuesta del sistema*) debe tener una transformada de Fourier igual a:

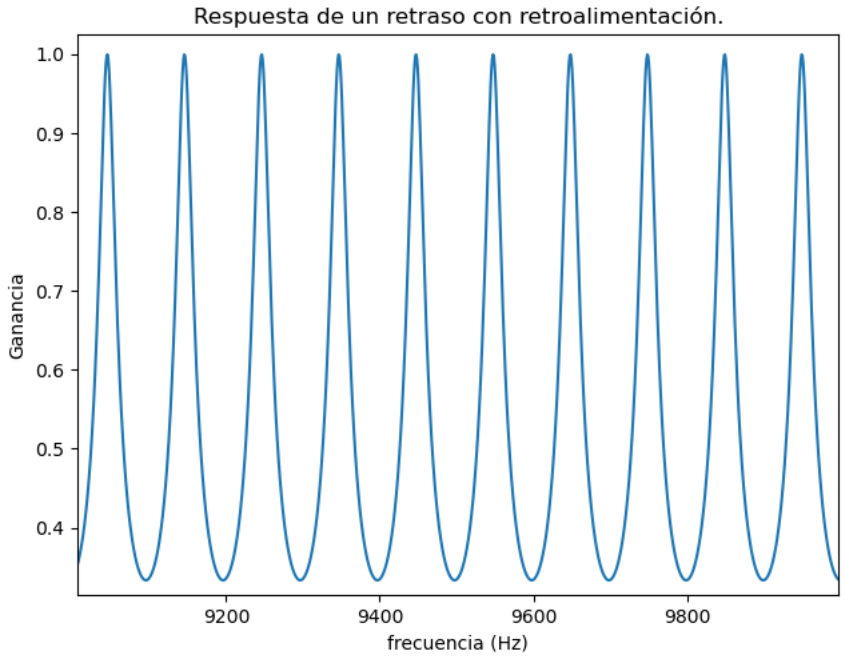
$$\hat{H}(\omega) = \frac{ge^{-i\omega\tau}}{1 - ge^{-i\omega\tau}}.$$

Usamos $|z|^2 = z\overline{z}$ para calcular la magnitud de H:

$$|\hat{H}(\omega)|^2 = \left(\frac{ge^{-i\omega\tau}}{1 - ge^{-i\omega\tau}}\right)\left(\frac{ge^{i\omega\tau}}{1 - ge^{i\omega\tau}}\right) = \frac{g^2}{1 - 2g\cos(\tau\omega) + g^2}.$$

$$|\hat{H}(\omega)| = \frac{g}{\sqrt{1 - 2g\cos(\tau\omega) + g^2}}.$$

Para $\tau = 0.99$, $g = 0.5$ obtenemos la siguiente gráfica de $|H|$:



Evaluando la magnitud en los puntos $\omega = \frac{k\pi}{\tau}$ podemos obtener el mínimo y máximo de la magnitud:

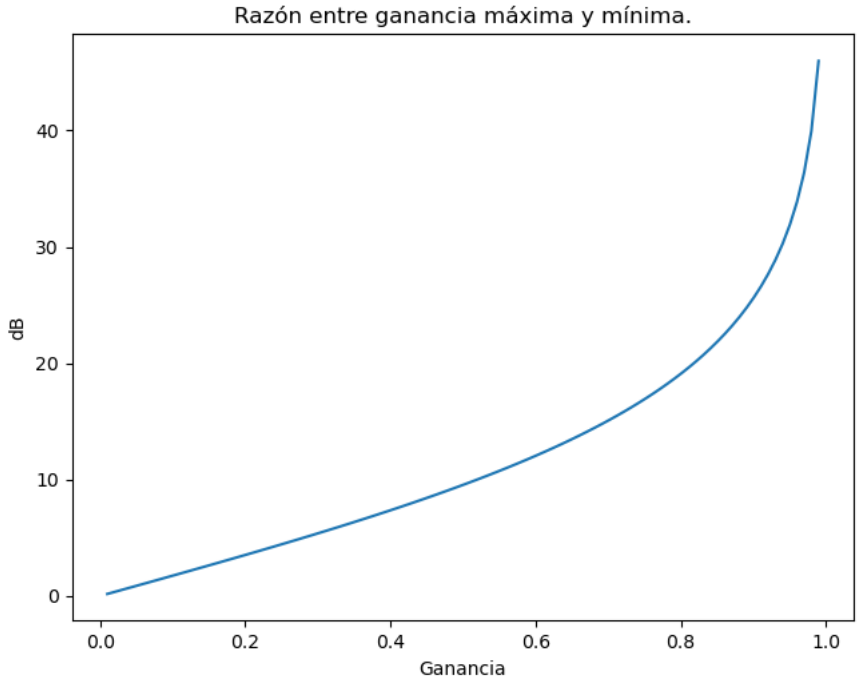


Figura 2.10

$$|\hat{H}(\omega)|_{\min} = \frac{g}{\sqrt{1+2g+g^2}} = \frac{g}{1+g},$$

$$|\hat{H}(\omega)|_{\max} = \frac{g}{\sqrt{1-2g+g^2}} = \frac{g}{1-g}.$$

La razón entre ambas cantidades es:

$$G = \frac{|\hat{H}(\omega)|_{\max}}{|\hat{H}(\omega)|_{\min}} = \frac{1+g}{1-g}.$$

En decibeles[1]; la ganancia es:

$$G_{\text{dB}} = 20\log_{10}\left(\frac{1+g}{1-g}\right).$$

[1]A través de la experimentación, se he llegado a una covención para representar la magnitud relativa entre dos señales de una manera psicoacústicamente apropiada a través del concepto de decibeles (o dB). Si σ_1 y σ_2 son magnitudes de dos señales y $g = \sigma_1/\sigma_2$ es la ganancia relativa; su ganancia en decibeles se calcula como $G_{\text{dB}} = 20\log_{10}(g)$. Véase (Scherz y Monk, 2016).

Este tipo de respuesta presenta una periodicidad que corresponde con la reverberación presente en cuartos muy grandes con paredes paralelas, pero no coincide con lo que esperaríamos escuchar en un cuarto real. Se conoce en inglés como *flutter echo*, que a falta de una traducción oficial al español le podríamos llamar eco ondulante. En términos de filtros de conoce como filtro de peine. Habiendo identificado la fuente del problema, no es complicado idear una manera de eliminar el eco ondulante. Para lograr que todas las frecuencias se conserven en la reverberación, podemos formar la combinación lineal y resolver para λ y η:

$$H_{\tau,g}(\omega) = \lambda + \eta\left(\frac{e^{-i\omega\tau}}{1-ge^{-i\omega\tau}}\right) = \frac{\lambda + (\eta - \lambda g)e^{-i\omega\tau}}{1-ge^{-i\omega\tau}} = \frac{\alpha + \beta e^{-i\omega\tau}}{1-ge^{-i\omega\tau}}.$$

El cuadrado de la magnitud es:

$$|H_{\tau,g}(\omega)|^2 = \left(\frac{\alpha + \beta e^{-i\omega\tau}}{1-ge^{-i\omega\tau}}\right)\left(\frac{\alpha + \beta e^{i\omega\tau}}{1-e^{i\omega\tau}}\right) = \frac{\alpha^2 + 2\alpha\beta\cos(\omega\tau) + \beta^2}{1 - 2g\cos(\omega\tau) + g^2}.$$

Para que sea igual a 1, necesitamos que

$$\alpha^2 + \beta^2 = 1 + g^2 = \lambda^2 + (\eta - \lambda g)^2$$

y

$$\alpha\beta = -g = \lambda(\eta - \lambda g).$$

Sustituyendo y simplificando obtenemos

$$\frac{\eta^2}{1-\lambda^2} = 1 - g^2.$$

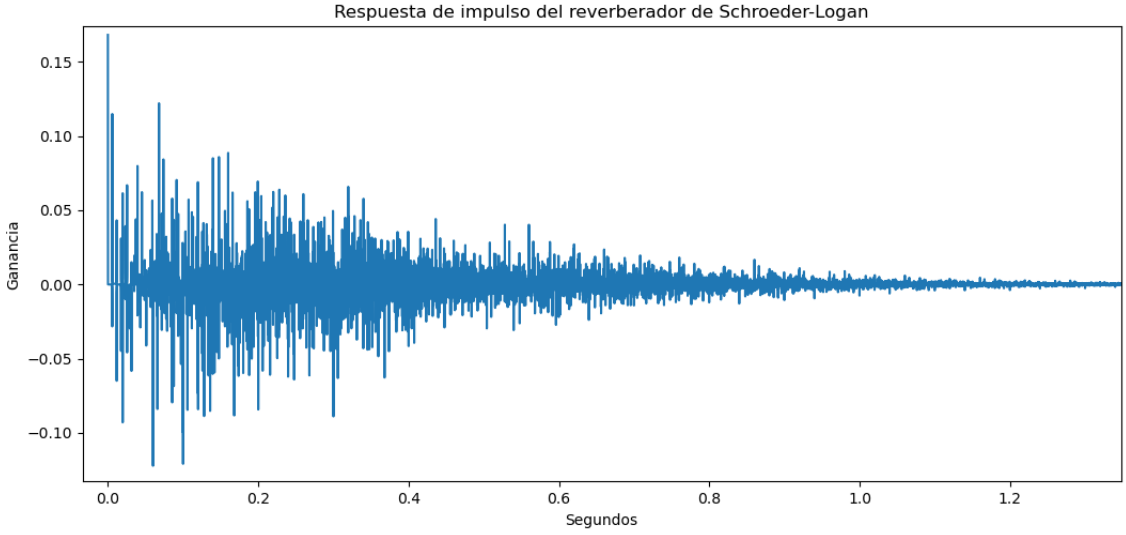
Claramente este problema no tiene una solución única. Sin embargo, podemos tomar $\lambda = -g$ y $\eta = (1 - g^2)$. Con ls constantes escogidas de esta forma, la reverberación modificada tiene una respuesta constante y no presenta ningún tipo de ecos ondulantes. Es en esta forma que, con el uso de las computadoras, se pudo simular aplicar una reverberación artificial. Tal algoritmo es el ancestro común[2] de todas las reverberaciones algorítmicas actualmente disponibles. La forma en la que se calculó originalmente fue usando cinco de estas reverberaciones en serie. Los valores de g y τ usados originalmente son $\tau = 100\text{ms}, 68\text{ms}, 60\text{ms}, 19.7\text{ms}, 5.85\text{ms}$

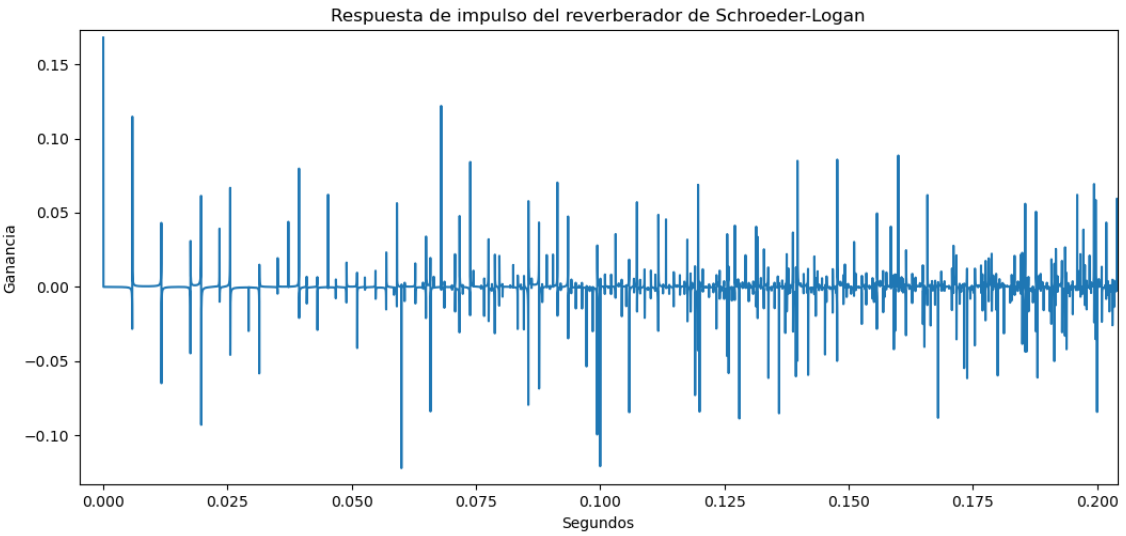
[2]Sean Costello, virtuoso de las reverberaciones y fundador de Valhalla DSP, presenta un montón de información sobre la historia de la reverberación digital en su blog https://valhalladsp.com

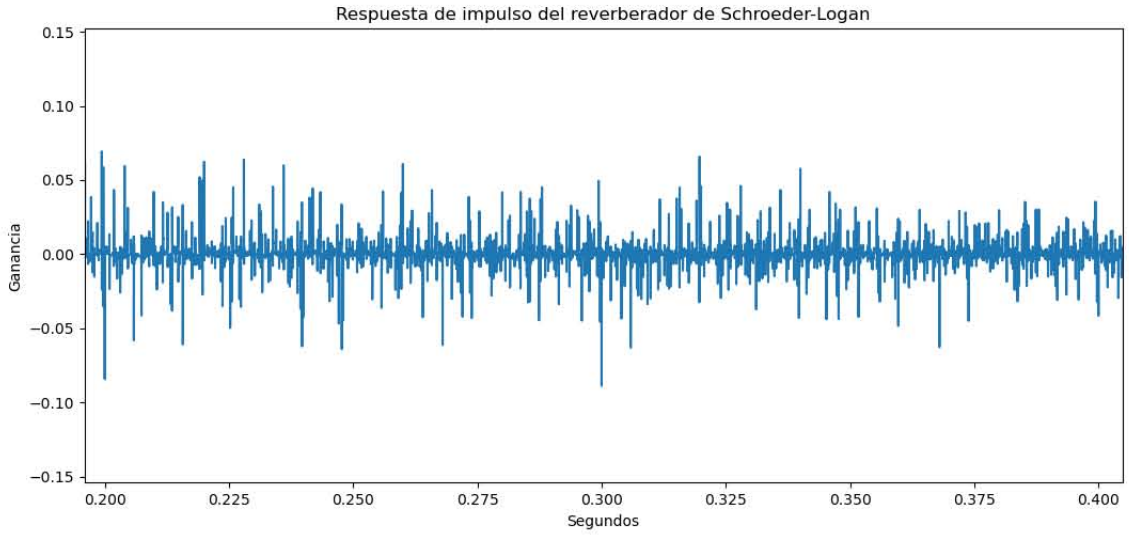
y $g = 0.7, -0.7, 0.7, 0.7, 0.7$. La respuesta de impulso asociada a estos parámetros se puede obtener formando el producto de funciones complejas:

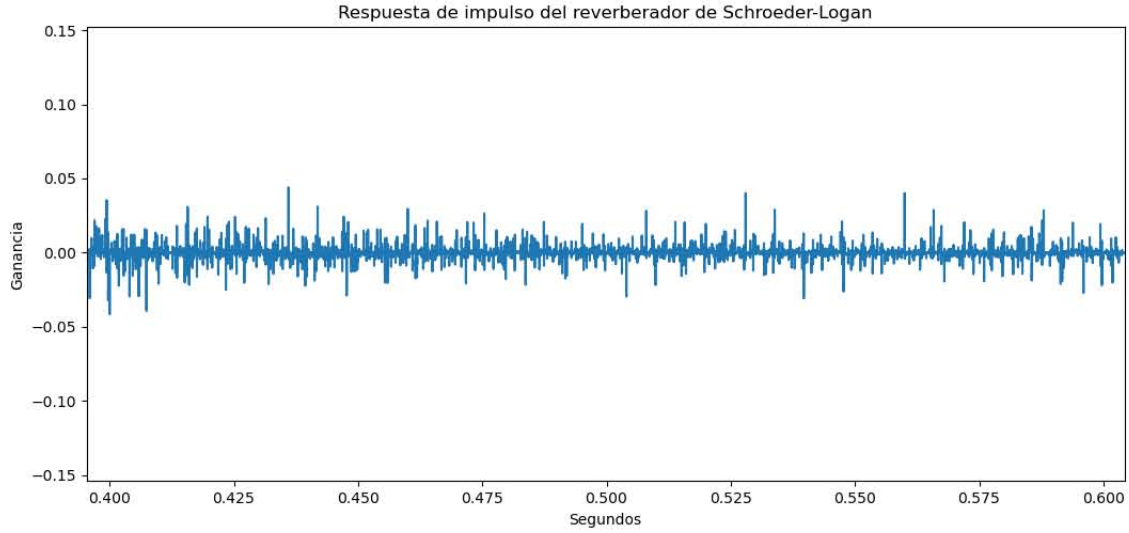
$$H(\omega) = (H_{100\text{ms},0.7} H_{68\text{ms},-0.7} H_{60\text{ms},0.7} H_{19.7\text{ms},0.7} H_{5.85\text{ms},0.7})(\omega).$$

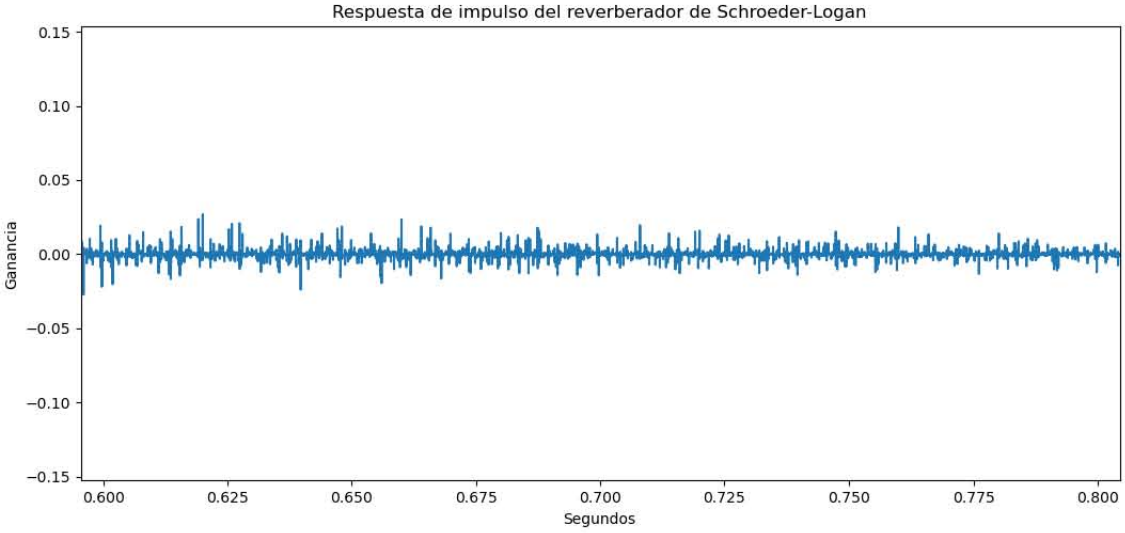
Para luego tomar la transformada inversa de H. El resultado se puede observar en las siguientes figuras, que son (casi) exactamente iguales a las del artículo original.






Respuesta de impulso del reverberador de Schroeder-Logan
Ganancia
Segundos
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
0.200
0.225
0.250
0.275
0.300
0.325
0.350
0.375
0.400


Respuesta de impulso del reverberador de Schroeder-Logan
Ganancia
Segundos
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
0.400
0.425
0.450
0.475
0.500
0.525
0.550
0.575
0.600


Respuesta de impulso del reverberador de Schroeder-Logan
Ganancia
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
0.600
0.625
0.650
0.675
0.700
0.725
0.750
0.775
0.800
Segundos

f

Capítulo 3

Un vistazo a las álgebras de convolución

Este capítulo presenta un estudio más detallado de las propiedades que debe cumplir una señal σ y una función de respuesta H de un sistema. Recordemos que la reverberación, como fenómeno lineal, puede ser representada completamente usando la convolución de una señal σ con la función de un sistema H. La restricción de linealidad nos lleva a buscar una familia de funciones cerradas esta operación, si incorporamos a la convolución a un espacio vectorial de funciones, llegamos al concepto de álgebra de convolución. Este contexto es excelente para formalizar las ideas que discutimos en la introducción.

3.1. Un álgebra de convolución

Supongamos que tenemos una señal $\sigma(t)$ que conocemos perfectamente (por ejemplo, $\sigma(t) = \text{sen}(t)$). Reproducimos a dicha señal en una sala de conciertos y registramos el resultado en la señal $\bar{\sigma}(t)$. Siempre que la sala de conciertos no se deforme en respuesta a $\sigma(t)$, la transformación se podrá escribir de forma explícita como $\bar{\sigma}(t) = (H * \sigma)(t)$.

El primer problema importante en este capítulo es obtener $H(t)$. Un primer paso es determinar la existencia de una función H tal que $u = H * u$ para toda función u en algún espacio apropiado. Desafortunadamente, una función así no existe. Tenemos que conformarnos con buscar una serie de funciones u_n con la propiedad de que

$$\lim_{n\to\infty} (H * u_n)(t) = H(t).$$

Para plantear rigurosamente este problema, necesitamos delimitar nuestro campo de estudio a un espacio de funciones cerrado bajo la convolución. Como veremos en breve, los espacios L_p son candidatos ideales para explorar este problema. A continuación exponemos algunos resultados relevantes. Por estar comúnmente presentes en textos de análisis, omitimos su demostración.

La primera definición importante es la de un espacio L_p. Estos espacios capturan y generalizan la idea de funciones integrables. Una función en un espacio tal es en realidad una clase de equivalencia de funciones que son iguales excepto en una cantidad numerable de puntos. En concreto:

Definición 1. (Igualdad en casi todo punto) Sean f, g dos funciones con dominio D. Consideremos al conjunto $A \subset D$ de puntos en los que f y g son iguales. Tendremos

- $A := \{x \in D \mid f(x) = g(x)\}$,
- $A^c := \{x \in D \mid f(x) \neq g(x)\}$.

Decimos que $f = g$ casi en todo punto si A^c es a lo más numerable. Esta relación es obviamente reflexiva y simétrica, por lo que solamente necesitamos verificar su transitividad para ver que es una relación de equivalencia. Supongamos que $f = g$ y $g = h$ casi en todo punto. Sean

- $A := \{x \in D \mid f(x) = g(x)\}$,
- $B := \{x \in D \mid g(x) = h(x)\}$,
- $C := \{x \in D \mid f(x) = h(x)\}$.

Para caracterizar al conjunto C^c en términos de A^c y B^c, basta considerar por casos cómo es que puede fallar $f = h$. La primera es notando que en $A \cap B$ se cumple trivialmente la transitividad, por lo que en $(A \cap B)^c = A^c \cup B^c$ tendremos $f \neq h$. Por otro lado, si $f(x) = g(x)$ y $g(x) \neq h(x)$, la transitividad fallará. Tomando en cuenta el caso simétrico, podemos escribir este caso como $(A \cap B^c) \cup (B \cap A^c) = A \Delta B$, donde Δ representa la diferencia simétrica. Puesto que A^c y B^c son a lo más numerables, obviamente las intersecciones $A \cap B^c$ y $B \cap A^c$ serán a lo más numerables también. Concluimos que C debe ser numerable y la relación transitiva. Usando las clases de equivalencia obtenidas a partir de esta relación, podemos definir a los espacios L_p.

Definición 2. (Espacios L_p) Sea f un representante de su clase de funciones iguales casi en todo punto con dominio D. Decimos que f se encuentra en un espacio L_p si

$$\left(\int_D |f|^p \right)^{\frac{1}{p}} < \infty.$$

Lema 3.1.1. *(Desigualdad Generalizada de Hölder) Sean* $\{f_k\}_{1\leq k\leq n}$, $\{p_k\}_{1\leq k\leq n}$, $\{\alpha_k\}_{1\leq k\leq n}$ *secuencias finitas de funciones y enteros -respectivamente- tales que* $f_k \in L_{\alpha_k}$, $\sum_{k=1}^{n} p_k^{-1} = 1$ *para todo* $1 \leq k \leq n$. *Tendremos que:*

$$\|\prod_{k=1}^{n} f_k\|_1 \leq \prod_{k=1}^{n} \|f_k\|_{p_k}.$$

El siguiente resultado general nos da una vía de comprobar cuándo la convolución de dos funciones sigue estando en un cierto espacio L_p:

Lema 3.1.2. *(Desigualdad de Young para la Convolución) Sean* p, q, r *enteros tales que* $1 \leq p, q \leq r < \infty$ *y tomemos a dos funciones* f, g *en* L_p *y* L_q, *respectivamente. Luego:*

$$\|f * g\|_r \leq \|f\|_p \|g\|_q.$$

Corolario 3.1.2.1. *Para todo par de funciones* $f, g \in L_1$ *se cumplirá que* $f * g \in L_1$*:*

Demostración:
Aplicando el teorema 3.1.2 con $q = p = r = 1$ obtenemos:

$$\|f * g\|_1 \leq \|f\|_1 \|g\|_1 < \infty,$$

$$f * g \in L_1.$$

Estamos listos para verificar que $(L_2, *)$ es un álgebra:

Teorema 3.1.3. *El espacio vectorial* L_1 *equipado con la convolución es un álgebra conmutativa. Explícitamente, la operación* $* : L_1 \times L_1 \to L_1$ *cumple:*

(i) $\alpha(f * g) = (\alpha f) * g = f * (\alpha g)$ para $f, g \in L_2$ y todo $\alpha \in \mathbb{R}$.

(ii) $(f * g) * h = f * (g * h)$ para $f, g \in L_2$ y todo $\alpha \in \mathbb{R}$.

(iii) $(f + g) * h = (f * h) + (g * h)$ para $f, g, h \in L_2$.

(iv) $f * g = g * f$ para $f, g \in L_2$.

Demostración:
Las propiedades (i) y (iii) se obtienen trivialmente de la linealidad de la integral. Empecemos por la conmutatividad de la convolución:

(iv) Consideremos la integral $\int_{-a}^{a} f(t)g(x-t)dt$; al hacer el cambio de variable $y = x - t$ obtenemos:

$$\int_{-a}^{a} f(t)g(x-t)dt = -\int_{x+a}^{x-a} f(x-y)g(y)dy = \int_{x-a}^{x+a} f(x-y)g(y)dy.$$

Tomando el límite cuando $a \to \infty$ claramente obtenemos:

$$f * g = \int_{\mathbb{R}} f(t)g(x-t)dt = \int_{\mathbb{R}} g(t)f(x-t)dt = g * f.$$

(iii) Usamos la conmutatividad de la convolución y el teorema de Fubini para obtener:

$$f * (g * h) = f * \left(\int_{\mathbb{R}} g(t)h(x-t)dt \right) = \int_{\mathbb{R}} f(s) \left(\int_{\mathbb{R}} g(t)h(s-t)dt \right) ds$$

$$= \int_{\mathbb{R}} \int_{\mathbb{R}} f(s)g(t)h(s-t)dtds = \int_{\mathbb{R}} \int_{\mathbb{R}} f(s)g(s-t)h(t)dtds$$

$$= \int_{\mathbb{R}} h(t) \left(\int_{\mathbb{R}} f(s)g(s-t)ds \right) dt = h * (f * g) = (f * g) * h.$$

El siguiente lema nos permitirá decidir la existencia de una unidad para esta álgebra de convolución. La demostración fue tomada de (Körner, 1989, p.260) con algunos pasos adicionales.

Lema 3.1.4. *(Riemann-Lebesgue) Sea f una función continua de $\mathbb{R}/\{2\pi\}$, entonces*

$$\lim_{n\to\infty} \hat{f}(n) = \lim_{n\to\infty} \frac{1}{2\pi} \int f(t)e^{-int}dt = 0.$$

Demostración:

Por ser f continua, para todo ε existe un polinomio trigonométrico $P_m(x)$ tal que

$$|f(x) - P_m(x)| < 2\pi\varepsilon.$$

Nótese que

$$\hat{P}(n) = \frac{1}{2\pi} \int \sum_{k=-m}^{m} a_k e^{ikt} e^{-int} dt = \sum_{k=-m}^{m} \frac{1}{2\pi} \int a_k e^{i(k-n)t} dt.$$

Claramente $\hat{P}(n) = 0$ cuando $n > m$. Si tomamos a n de esta forma, tenemos

$$|\hat{f}(n)| = |\hat{f}(n) - \hat{P}_k(n)| = \left| \frac{1}{2\pi} \int (f(t) - P_k(t)) e^{-int} dt \right|$$
$$\leq \frac{1}{2\pi} \int |(f(t) - P_k(t)) e^{-int}| dt \leq \frac{1}{2\pi} \int |f(t) - P_k(t)| dt < \varepsilon.$$

Para obtener el teorema de la convolución circular, necesitamos el siguiente lema cuya demostración puede encontrarse en el primer capítulo de (Hsu, Mehra, Velasco Coba y col., 1987):

Lema 3.1.5. *Sea $f(x)$ una función periódica con periodo $2T$ y a un número real. La integral entonces debe cumplir*

$$\int_{-T}^{T} f(x)dx = \int_{-T+a}^{T+a} f(x)dx.$$

Teorema 3.1.6. *(de la convolución circular) Para todo par de funciones $f, g \in L_1$ se cumplirá:*

$$\widehat{f \circledast g} = \hat{f}\hat{g}.$$

Demostración:

$$\widehat{f \circledast g} = \int_{-\pi}^{\pi} \left(\int_{-\pi}^{\pi} f(x)g(t-x)dx \right) e^{-i\omega t} dt$$
$$= \int_{-\pi}^{\pi} \left(\int_{-\pi}^{\pi} f(x)g(t-x)e^{-i\omega t} e^{-i\omega(x-x)} dx \right) dt$$
$$= \int_{-\pi}^{\pi} \left(\int_{\mathbb{R}} (f(x)e^{-i\omega x})(g(t-x)e^{-i\omega(t-x)}) dt \right) dx$$
$$= \left(\int_{-\pi}^{\pi} f(x)e^{-i\omega x} dx \right) \left(\int_{-\pi}^{\pi} g(t-x)e^{-i\omega(t-x)} dt \right)$$
$$= \left(\int_{-\pi}^{\pi} f(x)e^{-i\omega x} dx \right) \left(\int_{-\pi-x}^{\pi-x} g(u)e^{-i\omega(u)} du \right)$$
$$= \hat{f}\hat{g}.$$

Corolario 3.1.6.1. *El álgebra $(L_1, *)$ no tiene una unidad.*

Demostración:
Supongamos que existe un $\delta \in L_1$ tal que $f * \delta = f$. Luego, por el teorema 3.1.6:

$$\hat{f} = \widehat{(f * \delta)} = \hat{f}\hat{\delta}.$$

Por lo que $\hat{\delta}(\omega) = 1$ para todo $\omega \in \mathbb{C}$. Obtenemos una contradicción del lema de Riemann-Lebesgue:

$$\lim_{|\omega|\to\infty} \hat{\delta}(\omega) = 1.$$

Existen, sin embargo, secuencias de funciones que tienen un comportamiento de unidad en el límite. Consideremos a la *delta de Dirac* $\delta(t)$ definida como el límite de una secuencia $\{\delta_k\}_{k\in\mathbb{N}}$ definida como:

$$\delta_k(t) = \begin{cases} \frac{k}{2} & \text{si } |t| < \frac{1}{k}, \\ 0 & \text{si } |t| \geq \frac{1}{k}. \end{cases}$$

Sea $f \in L_1$ una función continua, formemos la secuencia $\sigma_n(f,t) = (f * \delta_n)(t)$ y calculemos

$$\sigma_n(f,t) = \int_{\mathbb{R}} f(x)\delta_n(t-x)dx = \int_{t-1/n}^{t+1/n} \frac{nf(x)}{2}dx = \left(\frac{nf(c_n)}{2}\right)\left(\frac{2}{n}\right) = f(c_n).$$

Para algún $c_n \in (t - \frac{1}{n}, t + \frac{1}{n})$ por el teorema del valor medio para integrales. Usando la continuidad de f, vemos que

$$\lim_{n\to\infty} \sigma_n(f,t) = \lim_{n\to\infty} f(c_n) = f(t).$$

Esta secuencia es un modelo para algo que se conoce como *identidades aproximadas*. En esencia, cualquier otra secuencia que comparta ciertas propiedades importante de δ_n también convergerá a $f(t)$ bajo la convolución. Dichas propiedades son:

(i) $\delta_k(t) \geq 0$ para todo k.

(ii) $\delta_k(t) \to 0$ uniformemente afuera de $[-\alpha, \alpha]$ para todo $\alpha > 0$.

(iii) $\int_{\mathbb{R}} \delta_k(t)dt = 1$.

La propiedad (i) se obtiene directamente de la definición. La propiedad (ii) es una consecuencia del principio de Arquímides. Si tomamos un $\alpha > 0$ arbitratio, obviamente podemos encontrar un k tal que $\alpha > 1/N$. Esto implica que $\delta_k(t) = 0$ afuera de $[-\alpha, \alpha]$ para todo $k > N$. Finalmente, la propiedad (iii) se obtiene de un cálculo directo.

Podemos recolectar los resultados anteriores en el siguiente teorema que garantiza que toda función con las mismas propiedades que nuestro modelo δ_k también debe ser una unidad aproximada. Un detalle que es clave para esta demostración es que la función f sea continua y acotada. Para esto bastaría tomar un dominio

acotado, ¡pero estamos tomando funciones periódicas en todos los reales! Aquí es donde entra en juego el dominio de definición $\mathbb{T} = \mathbb{R}/2\pi$. Tal definición nos permite resolver el problema de la compacidad. Por un lado, nos permite dar una extensión periódica a $\mathbb{R}$ de cualquier función definida en $[-\pi, \pi]$; por el otro, los valores de f en toda la recta real se encuentran completamente determinados por lo que sucede en $[-\pi, \pi]$. Esto significa que un cierto sentido de clases de equivalencia f puede ser tomado como compacto en $\mathbb{T}$. Ahora podemos dar la demostración del teorema de Fejér:

Teorema 3.1.7. *Sean f y $\{g_n(t)\}_{n\in\mathbb{N}}$ una función continua y una secuencia de funciones en $\mathbb{T} = \mathbb{R}/2\pi$, respectivamente. Si ambas son integrables y se cumple que*

(i) $g_n(t) \geq 0$ *para todo* k,

(ii) $g_n(t) \to 0$ *uniformemente afuera de* $[-\alpha, \alpha]$ *para todo* $\alpha > 0$,

(iii) $\int_{\mathbb{R}} g_n(t)dt = 1$.

La secuencia $\sigma_n(f,t) = (f * g_n)(t)$ convergerá puntualmente a f. Simbólicamente

$$\lim_{n\to\infty} \sigma_n(f,t) = f(t).$$

La secuencia $\{g_n(t)\}_{n\in\mathbb{N}}$ se conoce como unidad aproximada de $(L_1, *)$. En algunos libros también se puede encontrar como secuencia tipo delta (Shilov, 1974).

Demostración:

La siguiente demostración se puede encontrar casi verbatim en el segundo capítulo de (Körner, 1989). Sea $f \in \mathbb{T}$ una función continua e integrable. Todo valor de f se encuentra completamente determinado por su comportamiento en el intervalo $[-\pi, \pi]$ y podemos considerar a f como definida en un compacto. Entonces existirá un M tal que

$$(a)\ \ |f(x)| \leq M \quad \text{para todo } \ x \in \mathbb{T}.$$

Además, como f es continua; para todo $\varepsilon > 0$ podemos encontrar un δ tal que:

$$(b)\ \ |f(t) - f(x)| < \frac{\varepsilon}{2} \quad \text{para} \quad |t - x| < \delta.$$

Esto es equivalente a:

$$|f(t - x) - f(t)| < \frac{\varepsilon}{2} \quad \text{para} \quad |x| < \delta.$$

Una vez fijado $\delta(t,\varepsilon)$, usando la propiedad (ii) podemos obtener un N tal que $n > N$ implica:

$$(c)\ \ |g_n(x)| < \frac{\varepsilon}{4M}\ \text{ para todo }\ x \notin [-\delta,\delta]\ \text{ y }\ n \geq N(t,\varepsilon).$$

Sea ahora $\{g_n\}$ una identidad aproximada. Por la propiedad (iii) tendremos:

$$f(t) = f(t)\int_{\mathbb{R}} g_n(x)dx = \int_{\mathbb{R}} f(t)g_n(x)dx.$$

Calculamos:

$$|\sigma_n(f,t) - f(t)| = \left|\int_{\mathbb{R}} g_n(x)f(t-x)dx - \int_{\mathbb{R}} g_n(x)f(t)dx\right|$$

$$= \left|\int_{\mathbb{R}} g_n(x)(f(t-x) - f(t))dx\right|$$

$$= \left|\int_{x\in[-\delta,\delta]} g_n(x)(f(t-x) - f(t))dx\right| + \left|\int_{x\notin[-\delta,\delta]} g_n(x)(f(t-x) - f(t))dx\right|$$

$$< \frac{\varepsilon}{2}\int_{x\in[-\delta,\delta]} |g_n(x)|dx + 2M\int_{x\notin[-\delta,\delta]} |g_n(x)|dx$$

$$< \frac{\varepsilon}{2}\int_{x\in\mathbb{R}} g_n(x)dx + 2M\frac{\varepsilon}{4M}$$

$$= \frac{\varepsilon}{2} + \frac{\varepsilon}{2} = \varepsilon.$$

La compacidad de $\mathbb{T}$ nos da el siguiente sencillo corolario.

Corolario 3.1.7.1. *La convergencia de* $\lim_{n\to\infty}\sigma_n(f,t) = f(t)$ *es uniforme.*

Este resultado nos da un criterio bajo el cual podemos garantizar que una secuencia de funciones f_k pueda actuar como una base en el espacio. Si $\beta_n(x-t) = \beta_n(x)\overline{\beta_n(t)}$ y $K_n(x) = \sum \beta_n(x)$ actúa como una unidad aproximada, podemos estar justificados en tratar a esta familia de funciones como una base. La teoría de los polinomios (y en general, funciones) ortogonales explora este tema con más profundidad[1].

[1] Ver la tercera parte de (Körner, 1989).

3.2. Ejemplos de unidades aproximadas

Habiendo discutido las propiedades de las unidades aproximadas a partir de nuestro modelo δ, es provechoso mirar hacia atrás y evaluar nuestra discusión previa a través de los resultados de la sección anterior. No es difícil ver que el núcleo de Dirichlet no cumple las propiedades necesarias para ser una unidad aproximada. Omitiendo el 2π, el núcleo toma la forma:

$$D_n(x) = \begin{cases} \dfrac{\operatorname{sen}((n+1/2)x)}{\operatorname{sen}(x/2)} & x \neq 0, \\ 1+2n & x = 0. \end{cases}$$

Tomemos un $x = \pi/2$. Tendremos

$$D_n\left(\frac{\pi}{2}\right) = \frac{\operatorname{sen}\left(\frac{n\pi}{2} + \frac{\pi}{4}\right)}{\operatorname{sen}\left(\frac{\pi}{4}\right)}.$$

Si n es par, $n = 2k$ y

$$D_{2k}\left(\frac{\pi}{2}\right) = \frac{\operatorname{sen}\left(k\pi + \frac{\pi}{4}\right)}{\operatorname{sen}\left(\frac{\pi}{4}\right)} = 1.$$

Por otro lado, si n es impar, $n = 2k-1$ y

$$D_{2k-1}\left(\frac{\pi}{2}\right) = \frac{\operatorname{sen}\left(k\pi + \frac{\pi}{4} - \frac{\pi}{2}\right)}{\operatorname{sen}\left(\frac{\pi}{4}\right)} = -1.$$

Por lo que $D_k(x)$ no converge puntualmente a cero (mucho menos uniformemente). Sin embargo, encontrar un ejemplo de función continua f cuyo polinomio trigonométrico diverja en un punto (o en todo punto) es una tarea bastante complicada. Por fortuna, hay una manera relativamente sencilla de mejorar la convergencia de ciertas series.

3.2.1. El núcleo de Fejér

Fejér se dió cuenta de que la situación se podía remediar tomando promedios de D_n que de hecho sí logran ser unidades aproximadas. Dichos promedios habían sido estudiados previamente por Césaro, que había caído en cuenta de que hay series divergentes cuyos promedios sí convergen[2]. Escribimos entonces:

$$K_n(t) = \frac{1}{1+n} D_n = \frac{1}{1+n} \sum_{k=-n}^{n} \frac{e^{ikt}}{2\pi}.$$

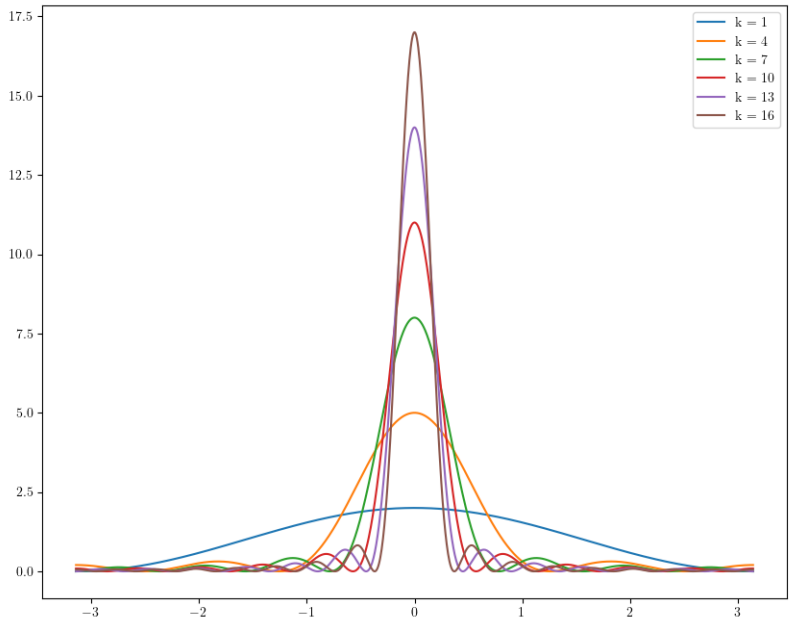


Figura 3.1: Núcleo de Fejér para distintos valores de n.

Definamos $g_k(t) = e^{ikt}$. Podemos reagrupar a la suma como

$$\begin{array}{ccccccccc} & & & & g_0(t)+ & & & & \\ & & & g_{-1}(t)+ & g_0(t)+ & g_1(t)+ & & & \\ & & g_{-2}(t)+ & g_{-1}(t)+ & g_0(t)+ & g_1(t)+ & g_2(t)+ & & \\ & ... & ... & ... & ... & ... & ... & ... & \\ g_{-n}(t)+ & ... & g_{-2}(t)+ & g_{-1}(t)+ & g_0(t)+ & g_1(t)+ & g_2(t)+ & ... & g_n(t). \end{array}$$

Nótese que la columna correpondiente a g_0 tiene $n+1$ elementos, las correspondientes a g_1 y g_{-1} tienen n y la última, que corresponde a g_n y g_{-n} tiene 1. Si agrupamos a las columnas en C_k, tendremos

$$C_k = (n+1-|k|)g_k$$

y

$$\frac{1}{n+1}\sum_{k=-n}^{n} g_k(t) = \sum_{k=-n}^{n} \frac{n+1-|k|}{n+1}\, g_k.$$

Podemos entonces reescribir el promedio como:

$$2\pi K_n(t) = \sum_{k=-n}^{n} \frac{n+1-|k|}{n+1} g_k(t) = \sum_{k=-n}^{n} \frac{n+1-|k|}{n+1} e^{ikt}$$

$$2\pi(n+1)K_n(t) = \sum_{k=-n}^{n} (n+1-|k|)e^{ikt}$$

$$= e^{-int} + 2e^{-i(n-1)t} + ... + 2e^{i(n-1)t} + e^{int}$$

$$= e^{-int}(1 + 2e^{it} + ... + 2e^{i(2n-1)t} + e^{i4\pi nt})$$

$$= e^{-int}\left(\sum_{k=0}^{n-1}(k+1)(e^{ikt} + e^{i(n-k)t}) + (n+1)e^{-int}\right)$$

$$= e^{-int}\left(\sum_{k=0}^{n} e^{ikt}\right)^2 = \left(e^{-i\frac{n}{2}t}\right)^2 \left(\sum_{k=0}^{n} e^{ikt}\right)^2.$$

De aquí podemos obtener que:

$$2\pi(n+1)K_n(0) = \left(\sum_{k=0}^{n} 1\right)^2 = (n+1)^2$$

[2]Por ejemplo, $s(n) = (-1)^n$.

$$K_n(0) = \frac{n+1}{2\pi}.$$

Supongamos ahora que $t \neq 0$. La expresión $\sum_{k=0}^{n} e^{ikt}$ es una serie geométrica con razón e^{it}. Podemos entonces reescribir la última igualdad como:

$$2\pi(n+1)K_n(t) = \left(e^{-i\frac{n}{2}t}\frac{1-e^{i(n+1)t}}{1-e^{it}}\right)^2$$

$$= \left(e^{-i\frac{n}{2}t}\frac{(e^{i(\frac{n+1}{2})t})(e^{-i(\frac{n+1}{2})t} - e^{i(\frac{n+1}{2})t})}{(e^{i(\frac{1}{2})t})((e^{\frac{-it}{2}}) - e^{\frac{it}{2}})}\right)^2 = \left(\frac{\text{sen } \frac{(n+1)t}{2}}{\text{sen } \frac{t}{2}}\right)^2.$$

Concluímos:

$$K_n(t) = \frac{1}{2\pi(n+1)}\left(\frac{\text{sen } \frac{(n+1)t}{2}}{\text{sen } \frac{t}{2}}\right)^2.$$

La demostración de que $K_n(t)$ es de hecho una unidad aproximada puede encontrarse al comienzo de (Körner, 1989). Mostraremos el cálculo equivalente para el núceo de Poisson en la siguiente sección.

3.2.2. El núcleo de Poisson

Si tomamos al núcleo de Fejér y lo multiplicamos por potencias de un número $0 \leq r < 1$; obtenemos la serie de funciones ortogonales definida por:

$$\beta_k = \sqrt{\frac{r^{|k|}}{2\pi}}\ e^{ikt}.$$

Tendremos:

$$\langle \beta_n, \beta_m \rangle = \frac{r^{\frac{|n|}{2}+\frac{|m|}{2}}}{2\pi}\int_{-\pi}^{\pi} e^{i(n-m)t}dt = \begin{cases} r^{|n|} & \text{si } n = m, \\ 0 & \text{si } n \neq m. \end{cases}$$

La idea detrás del núcleo de Poisson es explorar cómo las combinaciones lineales infinitas de elementos de la base pueden formar una identidad aproximada variando r. Como tal, el índice del núcleo de Poisson debe entonces correr sobre r (o más precisamente, sobre una sucesión r_n que converja a 1). Definamos $\xi_{r,n}(t) = \sum_{k=-n}^{n} \frac{r^{|k|}}{2\pi}e^{ikt}$ y, por el momento, supongamos que converge uniformemente a una función $P_r(\theta)$, luego:

$$\sigma_r(f,t) = \lim_{n\to\infty}\sum_{k=-n}^{n}\langle \beta_k, f\rangle\beta_k$$

$$= \lim_{n\to\infty} \sum_{k=-n}^{n} \left(\int_{-\pi}^{\pi} \frac{r^{\frac{|k|}{2}} f(x) e^{-ikx} dx}{\sqrt{2\pi}} \right) \frac{r^{\frac{|k|}{2}} e^{ikt}}{\sqrt{2\pi}}.$$

$$= \int_{-\pi}^{\pi} f(x) \left(\lim_{n\to\infty} \sum_{k=-n}^{n} \frac{r^{|k|}}{2\pi} e^{ik(t-x)} \right) dt = \int_{-\pi}^{\pi} f(x) \left(\lim_{n\to\infty} \xi_{r,n}(t-x) \right) dt.$$

En resumen:

$$\sigma_r(f,t) = (f * P_r)(t).$$

Para mostrar que $\sigma_r(f,t) \to f(t)$ cuando $r \to 1$ basta entonces verificar que $\xi_{r,n}(t) \to P_r(t)$ uniformemente y que P_r es una unidad aproximada cuando $r \to 1$. El primer paso es obtener una expresión cerrada para el núcleo de Poisson. Para lograr esto basta observar que podemos reescribir a P_k agrupando cada término negativo con su correspondiente positivo:

$$\xi_{r,n}(t) = \frac{1}{2\pi} \left(\sum_{k=0}^{n} r^k (e^{ikt} + e^{-ikt}) - 1 \right)$$

$$= \frac{1}{2\pi} \left(\sum_{k=0}^{n} 2\mathrm{Re}(r^k e^{ikt}) - 1 \right) = \frac{1}{2\pi} \mathrm{Re} \left(\sum_{k=0}^{n} 2r^k e^{ikt} - 1 \right).$$

Podemos obtener directamente el límite de esta serie geométrica:

$$P_r(\theta) = \lim_{n\to\infty} \xi_{r,n} = \frac{1}{2\pi} \mathrm{Re} \left(\frac{2}{1 - re^{i\theta}} - 1 \right)$$

$$= \frac{1}{2\pi} \mathrm{Re} \left(\frac{2 - 1 + re^{i\theta}}{1 - re^{i\theta}} \right) = \frac{1}{2\pi} \mathrm{Re} \left(\frac{1 + re^{i\theta}}{1 - re^{i\theta}} \right).$$

Luego:

$$\mathrm{Re} \left(\frac{1 + re^{i\theta}}{1 - re^{i\theta}} \right) = \frac{1 + re^{i\theta}}{2(1 - re^{i\theta})} + \frac{1 + re^{-i\theta}}{2(1 - re^{-i\theta})} = \frac{1 - r^2}{1 - 2r\cos(\theta) + r^2}$$

Podemos usar esta fórmula para mostrar que la convergencia es uniforme:

$$2\pi \mid \xi_{r,n}(\theta) - P_r(\theta) \mid = \left| \mathrm{Re} \left(\frac{2 - 2r^{n+1} e^{i(n+1)\theta}}{1 - re^{i\theta}} - 1 \right) - 2\pi P_r(\theta) \right|.$$

$$= \left| \mathrm{Re} \left(\frac{1 - 2r^{n+1} e^{i(n+1)\theta} + re^{it}}{1 - re^{i\theta}} \right) - 2\pi P_r(\theta) \right|$$

$$= \left| \mathrm{Re} \left(\frac{1 + re^{i\theta}}{1 - re^{i\theta}} - \frac{2r^{n+1} e^{i(n+1)\theta}}{1 - re^{i\theta}} \right) - 2\pi P_r(\theta) \right|$$

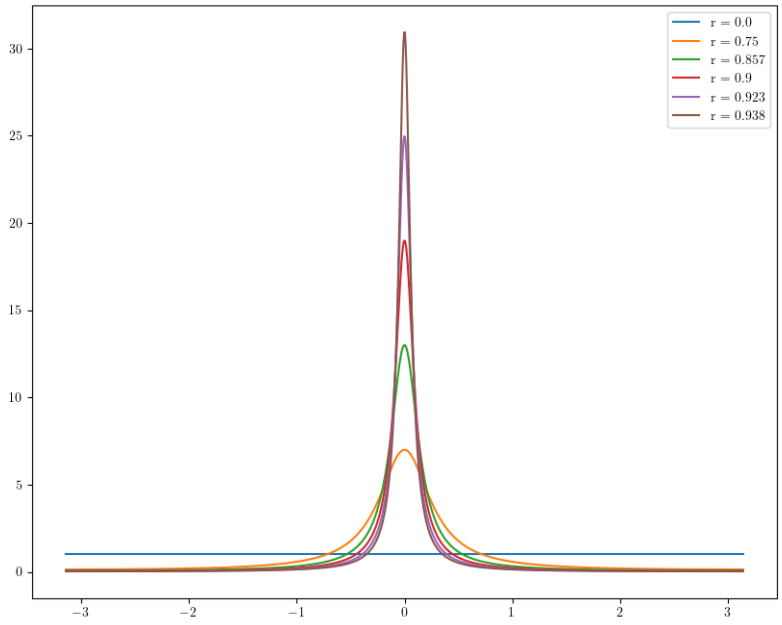


Figura 3.2: Núcleo de Poisson para distintos valores de n.

$$= \left| \mathrm{Re}\left(\frac{-2r^{n+1}e^{i(n+1)\theta}}{1-re^{i\theta}} \right) \right| = \left| \frac{r^{n+1}e^{i(n+1)\theta}}{1-re^{i\theta}} + \frac{r^{n+1}e^{-i(n+1)\theta}}{1-re^{-i\theta}} \right|$$

$$= \left| \frac{r^{n+1}e^{i(n+1)\theta} - r^{n+2}e^{in\theta} + r^{n+1}e^{-i(n+1)\theta} - r^{n+2}e^{-in\theta}}{1-re^{i\theta}-re^{-i\theta}+r^2} \right|$$

$$= 2r^{n+1} \left| \frac{\cos((n+1)\theta) - r\cos(\theta)}{1-2r\cos(\theta)+r^2} \right| \leq 2r^{n+1} \frac{|\cos((n+1)\theta)| - |r\cos(\theta)|}{1-2r\cos(\theta)+r^2}$$

$$\leq 2r^{n+1} \left(\frac{1+r}{1-2r\cos(\theta)+r^2} \right).$$

Como $\cos(\theta)$ tiene como puntos extremos a -1 y 1; el término $1-2r\cos(\theta)+r^2$ obviamente alcanza su mínimo cuando $\cos(\theta) = 1$. Usando que $r < 1$ obtenemos entonces que:

$$2r^{n+1} \left(\frac{1+r}{1-2r\cos(\theta)+r^2} \right) < 2r^{n+1} \left(\frac{1+r}{1-2r+r^2} \right) = \frac{2r^{n+1}(1+r)}{(1-r)^2}.$$

La cota anterior es independiente de θ, concluimos que $\xi_{r,n}(\theta)$ debe converger uniformemente:

$$2\pi \mid \xi_{r,n}(\theta) - P_r(\theta) \mid < \frac{2r^{n+1}(1+r)}{(1-r)^2} \to 0 \text{ cuando } n \to \infty.$$

El núcleo de Poisson está indizado por r. Formalmente, podemos cambiar el índice a un entero $n \in \mathbb{N}$ reemplazando a r por cualquier secuencia $\{r_n\}_{\mathbb{N}}$ que converja a 1. Con esto, podemos verificar que el núcleo de Poisson forma una unidad aproximada en el álgebra de convolución. Para verificar (i), podemos usar los valores extremales de $\cos(\theta)$ y dar las siguientes cotas:

$$(1-r)^2 \leq 1 - 2r\cos(\theta) + r^2 \leq (1+r)^2$$

Por lo que:

$$\frac{1-r^2}{(1+r)^2} \leq \frac{1-r^2}{1-2r\cos(\theta)+r^2} \leq \frac{1-r^2}{(1-r)^2}$$

y

$$\frac{1-r}{1+r} \leq 2\pi P_r(\theta) \leq \frac{1+r}{1-r}.$$

Esto es *casi* la desigualdad de Harnack[3] Además, P_r cumple que

$$0 \leq P_r(\theta),$$

$$2\pi P_r(0) = \frac{1+r}{1-r},$$

$$2\pi P_r(-\pi) = 2\pi P_r(\pi) = \frac{1-r}{1+r}.$$

La propiedad (ii) se puede obtener usando que $\cos(\theta) < \cos(\delta)$ siempre que $|\theta| < |\delta| \leq \pi$. Escribimos:

$$2\pi P_r(\theta) = \frac{1-r^2}{1-2r\cos(\theta)+r^2} < \frac{1-r^2}{1-2r\cos(\delta)+r^2}$$

$$= \frac{1-r^2}{(1-r)^2+2r(1-\cos(\delta))}.$$

Definiendo $r_n = \sum_{k=1}^{n} 2^{-n}$ y usando que $1-\cos(\delta) \neq 0$; obtenemos $P_r(\theta) \to 0$ cuando $r \to 1$ o equivalentemente $P_{r_n}(\theta) \to 0$ cuando $n \to \infty$. Con lo que obtenemos la convergencia uniforme de $P_r(\theta)$. Finalmente, la propiedad (iii) se verifica de la definición y usando la convergencia uniforme de $\xi_{r,n}$:

$$\int_{-\pi}^{\pi} \lim_{n\to\infty} \xi_{r,n}(t) = \lim_{n\to\infty} \int_{-\pi}^{\pi} \frac{1}{2\pi} \left(\sum_{k=-n}^{n} r^{|k|} e^{ikt} \right).$$

$$= \lim_{n\to\infty} \sum_{k=0}^{n} \frac{r^{|k|}}{2\pi} \int_{-\pi}^{\pi} e^{ikt} dt = \lim_{n\to\infty} \frac{1}{2\pi} \int_{-\pi}^{\pi} 1 dt = 1.$$

Concluimos que el núcleo de Poisson forma una unidad aproximada en el álgebra de convolución.

3.3. Interpolación de funciones en L_1

Más adelante presentaremos un método para interpolar mediciones acústicas para crear una aproximación de alguna medición intermedia faltante. Antes de presentar el método, presentaremos una discusión breve de convexidad.

[3]Para obtener dicha desigualdad, basta tomar la convolución del núcleo con una función armónica.

Definición 3. (Combinaciones Lineales Convexas) Decimos que una combinación lineal (finita) es convexa si todos sus coeficientes suman 1. Explícitamente:

$$\sum_{k=1}^{n} \lambda_k v_k.$$

es convexa si:

$$\sum_{k=1}^{n} \lambda_k = 1.$$

Es muy sencillo mostrar que toda combinación lineal es una función continua de sus coeficientes. Sea $l(\lambda_1, \lambda_2, ..., \lambda_n) = \sum_{k=1}^{n} \lambda_k v_k$. Sin pérdida de generalidad, podemos variar el coeficiente λ_1 mientras mantenemos todos los demás fijos. Sea $\varepsilon > 0$ y $\delta = \varepsilon / \|v_1\|$ tal que $|x - y| < \delta$:

$$\begin{aligned} \|l(x, \lambda_2, ..., \lambda_n) - l(y, \lambda_2, ..., \lambda_n)\| &= \|(x - y)v_1\| \\ &< \delta \|v_1\| = \frac{\varepsilon}{\|v_1\|} \|v_1\| = \varepsilon.. \end{aligned}$$

Un caso especial de particular interés aparece en las combinaciones lineales convexas de dos coeficientes. Sean $f, g \in L_1$ dos funciones arbitrarias (pueden ser discontinuas) con $\|f\|_1 \|g\|_1 > 0$ y formemos la siguiente interpolación:

$$l(t, x) = tf(x) + (1 - t)g(x).$$

Sea $\varepsilon > 0$ y $\delta = \varepsilon / (\|f\|_1 + \|g\|_1)$ tal que $|(t + h) - t| = |h| < \delta$.

$$\begin{aligned} \|l(t + h, x) - l(t, x)|_1 &= \|h(f(x) - g(x))\|_1 \\ = |h| \|f(x) - g(x)\|_1 &\leq |h|(\|f\|_1 + \|g\|_1) < \varepsilon. \end{aligned}$$

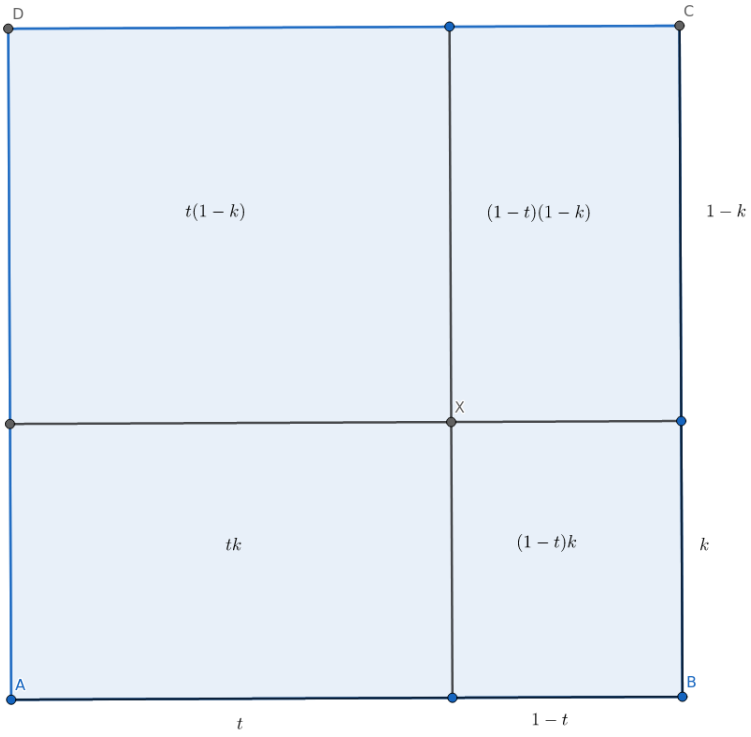
Por lo que l es continua en t (aunque posiblemente no en x). Esto significa que cualesquiera dos interpolaciones de f y g se podrán hacer arbitrariamente cercanas (en la métrica de L_1) incluso si una de las dos funciones *no es continua en* x.

El sistema que nos interesa interpolar consiste de una fuente sonora fija y de un micrófono que puede tener varias configuraciones en el espacio; dadas por el vector $(x, y, z, \theta, \omega)$; donde las primeras tres coordenadas x, y, z describen la posición del centro de masa de la cápsula del micrófono y θ, ω describen la dirección que tiene el eje de simetría de la cápsula. Para abordar el problema, comenzaremos con el caso más sencillo que consiste de dos vectores A, B. Podemos interpretar esto como una interpolación de dimensión (u orden) 1. Por lo que los coeficientes de interpolación de un punto intermedio X deben ser proporcionales a las distancias del punto X a los vectores A y B. De la figura, podemos inducir que cuando X se

acerca a B; el coeficiente t toma prioridad sobre $1-t$ y vice-versa. Podemos ver que debemos asignar los coeficientes usando el segmento opuesto al vector. i.e. t corresponde a B y $1-t$ a A. Asignamos el símbolo de λ_1 a esta interpolación de orden 1 y escribimos:

$$\lambda^1_{A,B}(t) = (1-t)A + tB.$$

Para el caso de dimensión 2; tendremos:



Aplicando el método anterior, asignamos las areas de los cuadrados *opuestos* a los vértices como coeficientes. Obtenemos, por analogía con el caso anterior:

$$\lambda^2_{A,B,C,D}(t,k) = (1-t)(1-k)A + t(1-k)B + tkC + (1-t)kD$$

$$\begin{aligned} &= (1-k)((1-t)A + tB) + k((1-t)D + tC) \\ &= (1-k)\lambda^1_{A,B}(t) + k\lambda^1_{D,C}(t) \\ &= \lambda^1_{\lambda^1_{A,B},\lambda^1_{D,C}}(k). \end{aligned}$$

Inmediatamente podemos ver que este método es aplicable para reducir cualquier interpolación de dimensión n a una interpolación de dimensión 1. En la práctica simplemente debemos crear un n-cubo con 2^n mediciones por vértices y aplicar la interpolación arista por arista; recordando que las aristas son direcciones en las que una de las variables de posición no cambia. Por ejemplo, en el caso de dimensión dos, podemos asignar las coordenadas imaginando que tenemos un micrófono restringido a moverse sobre el eje de simetría de la bocina, y a rotar sobre el plano transversal del gabinete donde está montada dicha bocina. Así, las aristas se pueden escribir como $A = (x_1, \phi_1)$, $B = (x_1, \phi_2)$, $C = (x_2, \phi_2)$, $D = (x_2, \phi_1)$; por lo que las aristas (o segmentos) verticales corresponden a rotar el micrófono sin cambiar la distancia de la cápsula a la bocina y las aristas (o segmentos) horizontales corresponden a cambiar la distancia de la cápsula a la bocina sin cambiar el ángulo de la cápsula.

En conclusión, siempre y cuando tengamos mediciones con un contenido suficiente de frecuencias, este método de interpolaciones iteradas nos permitirá simular satisfactoriamente mediciones intermedias.

Capítulo 4

Procesamiento digital de señales

4.1. El teorema del muestreo

En la práctica, podemos pasar de señales continuas a discretas usando un procedimiento llamado *muestreo*. Este consiste de discretizar, de forma similar a una suma de Riemann, el dominio de una señal continua $\sigma(t)$ para formar una secuencia σ_n. En la práctica, tanto el dominio como la imagen de la función son discretos. Siguiendo la idea de las sumas de Riemann, partamos al intervalo $[a, b]$ en N puntos con coordenadas

$$t_k = a + \frac{k}{N}(b - a).$$

Al número $f_s = N/(b-a)$ se le llama frecuencia de muestreo relativa al dominio. Si las unidades de t son segundos, la frecuencia de muestreo se puede expresar en unidades de Hertz. En el capìtulo 2, vimos que la frecuencia angular asociada a cualquier intervalo $[a, b]$ es $f_0 = 2\pi/(b - a)$. Obtenemos la siguiente relación entre frecuencia de muestreo relativa al dominio y frecuencia angular fundamental:

$$2\pi f_s = N f_0.$$

Al muestreo en el eje de las ordenadas se le denomina *cuantización* y se puede expresar en términos de *profundidad de bits*. Esto significa que, si el rango admisible de valores para nuestra señal es $-1 \leq \sigma \leq 1$, las computadoras partirán el intervalo en 2^k y le asignarán a σ el valor más cercano. En la práctica, el primer bit se usa para representar el signo del número. Esto aumenta la precisión de la cuantización. Como hay un número de maneras distintas de cuantizar señales y no todos los convertidores y procesadores usan el mismo sistema, esto es todo lo que diremos del tema.

Ahora, es natural preguntarnos qué sucede si tratamos de representar oscilaciones con frecuencias arbitrariamente pequeñas dada una frecuencia de muestreo fija. Nótese que si aumentamos la longitud del intervalo manteniendo constante el número de puntos, perdemos información de lo que se pueda encontrar entre los puntos del muestreo. Si, por ejemplo, existe una sinusoidal con nodos en t_k, t_{k+1} y t_{k+2}, tal sinusoidal será completamente invisible al proceso de muestreo. Una sinusoidal así tiene frecuencia fundamental

$$\frac{2\pi}{t_{k+2} - t_k} = 2\pi \, \frac{f_s}{2}.$$

El proceso de muestreo arroja un resultado idéntico al de la función constante cero, por lo que formalmente podemos decir que deja de ser inyectivo para frecuencias relativas al dominio que sean múltiplos enteros de$f_s/2$. Esto a razón de que tomar múltiplos de esta manera no altera los nodos de la sinusoidal. Este comportamiento se puede llevar al caso lineal. En general, la cota estricta para las frecuencias que se pueden representar únicamente con un proceso de muestreo es de $f < f_s/2$. Tal número lleva el nombre de frecuencia (o límite) de Nyquist-Shannon. La prioridad del teorema es un asunto contencioso; los rusos podrían decir que V.A.Kotelnikov llegó primero al resultado, los ingleses que T.E.Whittaker llegó incluso antes que Kotelnivov y alguna otra persona podría decir que a Borel (sí, ese Borel) se le ocurrió primero. En concordancia con el principio de Arnol'd[1] tendríamos que decir que todas estas personas probablemente están equivocadas. Una historia breve de las ideas y técnicas de interpolación se puede encontrar en (Meijering, 2002).

Siempre que dos señales presenten un muestreo idéntico se dice que una es un *pseudónimo* de la otra. A falta de una palabra apropiada en español usaremos el término de *aliasing* para referirnos a este fenómeno. Los aliases se vuelven muy importantes al trabajar con polinomios trigonométricos. Podemos dar una descripción del comportamiento del muestreo cuando las funciones se acercan al límite de Nyquist-Shannon estudiando la función

$$\sigma(t_k) = \operatorname{sen}(2\pi f t_k) = \operatorname{sen}\left(\frac{2\pi f k}{f_s}\right).$$

Si $f = f_s/2$, obtenemos $\operatorname{sen}(\pi k) = 0$ como vimos antes. Ahora, tomemos $f_1 = f_s/2 + l$ y $f_2 = f_s/2 - l$. Sustituyendo:

$$\operatorname{sen}(2\pi f_1 t_k) = \operatorname{sen}\left(\frac{2\pi f_1 k}{f_s}\right) = \operatorname{sen}\left(\pi k + \frac{2\pi l}{f_s}\, k\right) = (-1)^k \operatorname{sen}\left(\frac{2\pi l}{f_s}\, k\right).$$

[1]Principio de Arnol'd: Ningún descubrimiento científico que lleve el nombre de una persona fue descubierto por esa persona.
Principio de Barry: El principio de Arnol'd es aplicable a sí mismo.

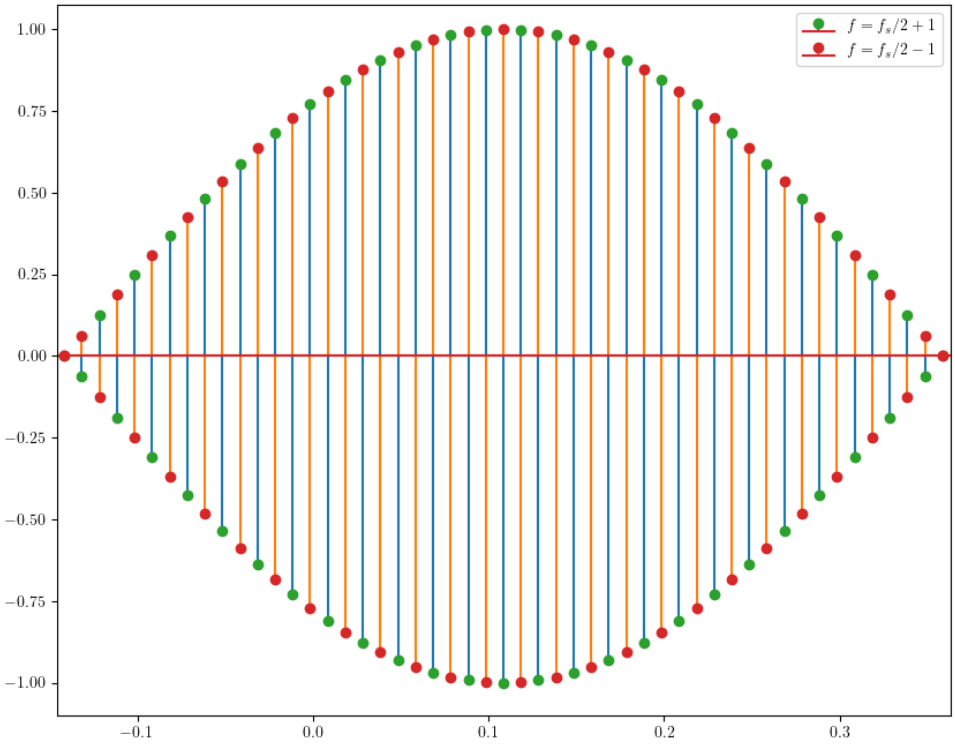


Figura 4.1: Sinusoides cerca de $f_s/2$.

$$\operatorname{sen}(2\pi f_2 t_k) = \operatorname{sen}\left(\frac{2\pi f_2 k}{f_s}\right) = \operatorname{sen}\left(\pi k - \frac{2\pi l}{f_s}\, k\right) = (-1)^{k+1} \operatorname{sen}\left(\frac{2\pi l}{f_s}\, k\right).$$

El efecto de acercarse a la frecuencia de Nyquist-Shannon es que comienzan a aparecer ciertas envolventes de frecuencia asociada a l.

Para introducir la otra parte de la intuición necesaria para comprender el teorema del muestreo, hay que formular los resultados anteriores en términos de la transformada de Fourier. El hecho de que la máxima frecuencia que se puede representar únicamente a través del muestreo es $f_s/2$ se puede aplicar a los polinomios trigonométricos directamente. Esto significa que solamente vamos a poder muestrear de forma única funciones cuyas series de Fourier tengan coeficientes cero para todas las frecuencias mayores o iguales a $f_s/2$. En términos de frecuencia angular, la cota es πf_s por la discusión anterior. Concrtamente, esto significa que la trans-

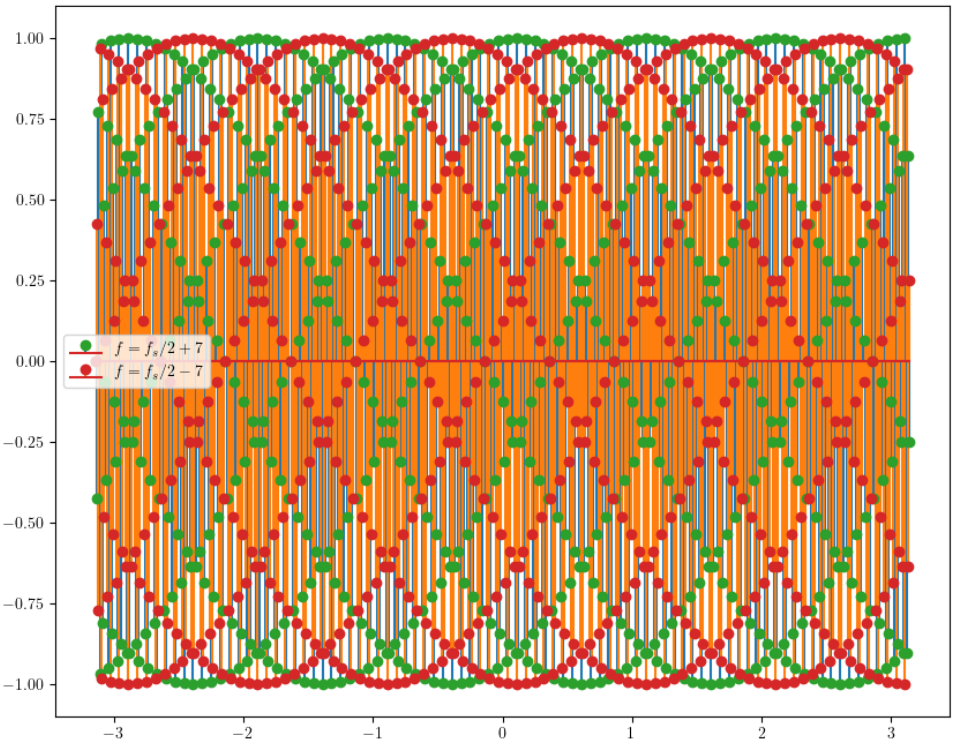


Figura 4.2: Sinusoides no tan cerca de $f_s/2$.

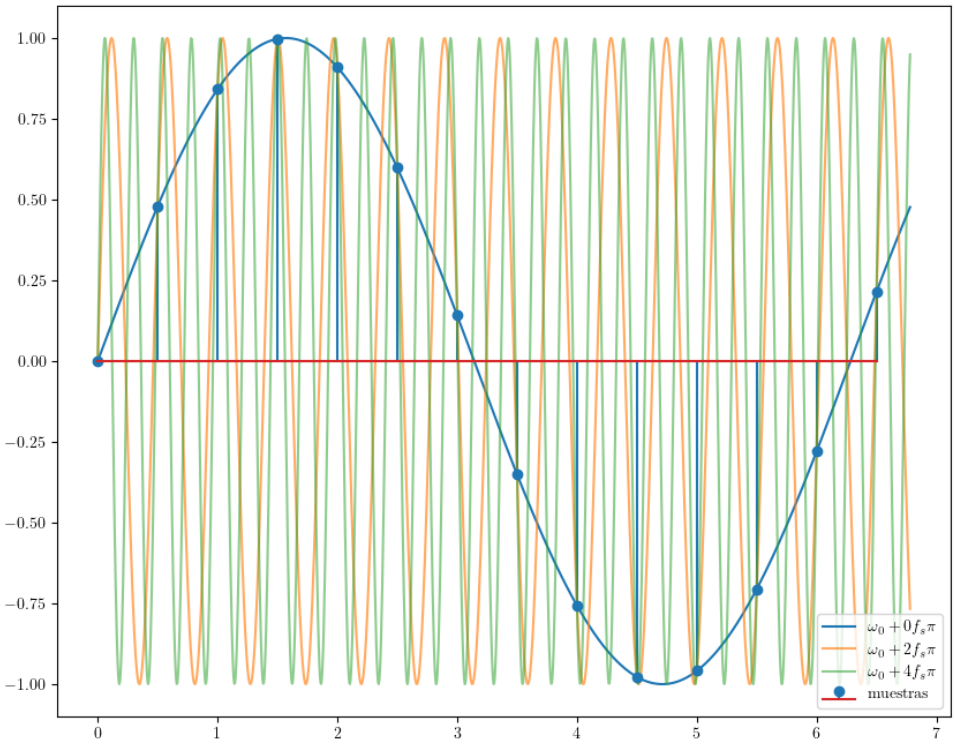


Figura 4.3: Pseudónimos de la función sen(x) con $f_s = 2$

formada de Fourier $\mathbf{F}(f)(\omega)$ debe ser cero ciempre que $\pi f_s \leq |\omega|$.

Lo único que necesitamos para completar nuestro delineamiento del teorema del muestreo es dar explícitamente una manera de invertir el proceso de muestreo siempre que se cumplan las propiedades antes expuestas. Usando el teorema de la convolución es relativamente sencillo encontrar tal función. Si nos restringimos al intervalo $T = [-\pi f_s, \pi fs]$ y usamos la función característica $\chi_T(x)$, podemos expresar formalmente los requisitos que hemos discutido. Escribimos:

$$\hat{\sigma}(\omega) = \chi_T(\omega)\hat{\sigma}(\omega).$$

Por la segunda versión del teorema de la convolución, esto será equivalente a la transformada de Fourier de:

$$(\beta * \sigma)(t).$$

Para una β cuya transformada sea χ_T. Podemos calcular esto explícitamente:

$$\beta(t) = \frac{1}{2a}\int \chi_T(\omega)e^{i\omega t}d\omega = \frac{1}{2a}\int_{-\pi f_s}^{\pi f_s} e^{i\omega t}d\omega = \frac{e^{i\omega t}}{i2\pi f_s t}\Bigg|_{-\pi f_s}^{\pi f_s}$$
$$= \frac{e^{i\pi f_s t} - e^{-i\pi f_s t}}{i2\pi f_s t} = \frac{\operatorname{sen}(\pi f_s t)}{\pi f_s t}.$$

A esta función se le da el nombre especial de $\operatorname{senc}_{\pi f_s}(t)$ y a veces se le llama el seno cardinal. En concreto, la inversión del proceso de muestreo se puede calcular explícitamente con la formula

$$\sigma(t) = (\sigma * \operatorname{senc}_{\pi f_s}(x))(t) = \int \operatorname{senc}_{\pi f_s}(x)\frac{\operatorname{sen}(\pi f_s(t-x))}{\pi f_s(t-x)}dx.$$

Para evitar una tangente que nos llevaría lejos de nuestra meta actual, omitiremos una demostración detallada del teorema del muestreo. Por ahora, nótese que si tomamos una suma parcial de la convolución con $x_k = k/f_s$ y $\Delta x = 1/f_s$, obtendremos:

$$\sigma(t) = \sum_{k=-\infty}^{\infty} \sigma(x_k)\frac{\operatorname{sen}(\pi f_s(t-x_k))}{\pi f_s(t-x_k)}\Delta x.$$

El resultado verdaderamente sorprendente del teorema del muestreo es que esta expresión se puede intercambiar por la integral cuando se cumplen las condiciones del teorema. Geométricamente, esto significa que las muestras de σ junto con Δx forman cuadriláteros que representan exactamente el área entre x_k y x_{k+1}, lo que nos permite intercambiar la integral por una suma. Para convencernos de que esto

es verdad, recordemos que toda periodicidad que pudiera existir entre muestras necesitaría tener una frecuencia de oscilación mayor a $f_s/2$; lo que está descartado por hipótesis. Esta condición nos garantiza que no hay puntos problemáticos intermedios y, por lo tanto, podemos reconstruir satisfactoriamente la función original a través de sus muestras.

Lema 4.1.1. *(Parseval) Si* $A(x) = \sum_{n=-\infty}^{\infty} a_n e^{inx}$ *y* $B(x) = \sum_{m=-\infty}^{\infty} b_m e^{imx}$ *son dos funciones de cuadrado integrable, con series de Fourier convergentes y con periodo* 2π*, se cumplirá que*

$$\langle a_n, b_n \rangle = \langle A, B \rangle.$$

Demostración:

Como la convergencia de las series es uniforme, podemos intercambiar la suma con la integral y escribir

$$\langle A, B \rangle = \left\langle \sum_n a_n e^{int}, \sum_m b_m e^{imt} \right\rangle = \sum_n \sum_m \langle a_n e^{int}, b_m e^{imt} \rangle$$

Nótese que:

$$\langle a_n e^{int}, b_m e^{imt} \rangle = a_n \overline{b_m} \int_{-\pi}^{\pi} \frac{e^{i(n-m)x}}{2\pi} dx = \begin{cases} a_n \overline{b_m} & \text{si } n = m, \\ 0 & \text{si } n \neq m. \end{cases}$$

Por lo que la suma solo consiste de términos $n = m$. En resumen

$$\langle A, B \rangle = \sum_n \langle a_n, b_n \rangle.$$

Explícitamente

$$\sum_n a_n \overline{b_n} = \int_{-\pi}^{\pi} A(x) \overline{B(x)} dx.$$

Un corolario interesante del teorema de Parseval es la siguiente relación entre los cuadrados de los coeficientes y la norma de la función

Corolario 4.1.1.1. *Si* f *es una función con periodo* 2π *y con serie de Fourier* $\hat{f}(\omega) = \sum_{n=-\infty}^{\infty} c_k e^{ikt}$*; se cumplirá:*

$$\sum_{k=-\infty}^{\infty} |c_k|^2 = \frac{1}{2\pi} \int_{-\pi}^{\pi} |\hat{f}(\omega)|^2 d\omega$$

También podemos usarlo para recuperar los coeficientes de la serie de Fourier. En efecto, si tomamos $A(x) = f(x)$, $B(x) = e^{-ikt}$ en el teorema de Parseval, obtenemos

$$c_k = \frac{1}{2\pi}\int_{-\pi}^{\pi} f(x)e^{-ikx}dx.$$

4.2. Procesamiento por bloques

Supongamos que tenemos una señal discreta $\sigma : \mathbb{N} \to \mathbb{R}$ cuya imagen no necesariamente es finita. Las computadoras procesan señales tomando un bloque de N muestras a la vez. Esto se puede expresar formalmente mediante la suma:

$$\sigma(t_n) = \sum_{k\in\mathbb{N}} \sigma(t_n)\chi_{B_N^k}(t) \quad B_N^k = [kN, (k+1)N].$$

Donde χ_A es la función característica de un conjunto A definida como:

$$\chi_A(t) = \begin{cases} 1 & \text{cuando } t \in A, \\ 0 & \text{cuando } t \notin A. \end{cases}$$

Cada bloque individual se puede indizar a partir del cero trasladándolo al origen. Tomamos $t_n - kN = t_N^\star$ de modo que $0 \leq t_N^\star \leq N-1$ para escribir:

$$\sigma_N^k(t_n^\star) = \sigma(t_n - kN)\chi_{B_N^k}(t_n).$$

La descomposición por bloques nos permite reducir el procesamiento de una señal de duración arbitraria al de una serie de señales con la misma duración finita. Como veremos más adelante, la descomposición por bloques nos permitirá reducir el tiempo de cómputo de operaciones como la convolución. Si el procesamiento se puede expresar como una función lineal L, tendremos que:

$$L(\sigma(t_n)\chi_{B_N^k}(t_n)) = \begin{cases} L(\sigma(t_n)) & \text{si } t_n \in B_N^k, \\ 0 & \text{si } t_n \notin B_N^k. \end{cases}$$

Obtenemos la descomposición por bloques de un procesamiento lineal:

$$L(\sigma(t_n)) = \sum_{k\in\mathbb{N}} L(\sigma(t_n)\chi_{B_N^k}(t_n)) = \sum_{k\in\mathbb{N}} L(\sigma(t_n))\chi_{B_N^k}(t_n).$$

El caso en el que $N = 1$ se conoce como *procesamiento muestra por muestra* y generalmente sólo se puede encontrar en hardware especializado[2]. Todo procesamiento en tiempo real está caracterizado por tener un límite de tiempo; todas

[2]Como los procesadores SHARC.

las operaciones que se tengan que hacer sobre un bloque deben terminar antes de que podamos empezar a procesar el bloque siguiente. El máximo tiempo $\tau(N, f_s)$ permitido depende de la frecuencia de muestreo f_s y el tamaño de bloque N que escojamos. Podemos obtenerlo fácilmente observando que si una muestra equivale a $1/f_s$ segundos; N muestras equivaldrán a $\tau = N/f_s$ segundos. Para las siguientes frecuencias de muestreo comunes, tendremos los límites de tiempo aproximados de:

$$4.8 \times 10^5 hz \rightarrow \tau_1 = 2.1 \times 10^{-6} s,$$

$$9.6 \times 10^5 hz \rightarrow \tau_2 = 1.0 \times 10^{-6} s,$$

$$19.2 \times 10^5 hz \rightarrow \tau_3 = 5.2 \times 10^{7} s.$$

Los tiempos para los casos $N > 1$ se pueden obtener trivialmente como múltiplos de τ_1, τ_2 y τ_3. Como generalmente N es una potencia de 2, podemos hacer una tabla estimada de tiempos de cálculo para diferentes valores de $N = 2^n$:

N	τ_1	τ_2	τ_2
1	$2.08\mu s$	$1.04\mu s$	$520ns$
2	$4.17\mu s$	$2.08\mu s$	$1.04\mu s$
4	$8.33\mu s$	$4.17\mu s$	$2.08\mu s$
8	$16.67\mu s$	$8.33\mu s$	$4.17\mu s$
16	$33.33\mu s$	$16.67\mu s$	$8.33\mu s$
32	$66.67\mu s$	$33.33\mu s$	$16.67\mu s$
64	$1.33ms$	$66.67\mu s$	$33.33\mu s$
128	$2.67ms$	$1.33ms$	$66.67\mu s$
256	$5.33ms$	$2.67ms$	$1.33ms$
512	$10.67ms$	$5.33ms$	$2.67ms$
1024	$21.33ms$	$10.67ms$	$5.33ms$
2048	$42.67ms$	$21.33ms$	$10.67ms$
4096	$85.33ms$	$42.67ms$	$21.33ms$

En lo que concierne al tiempo disponible para el procesamiento, aumentar el número de muestras por un factor de k equivale a reducir por un factor de $1/k$ la frecuencia de muestreo. Ambas operaciones resultan en un tiempo de $k\tau$ segundos:

$$f(kN) < \tau(kN, f_s) = \frac{kN}{f_s} = \frac{N}{f_s/k} = \tau(N, f_s/k).$$

En términos de f_s:

$$f(N) < N\tau(1, f_s) = \tau_{f_s} N.$$

Si tenemos $f(N) = g(N)N$ con $g(N)$ divergente, existirá un N_0 tal que $n > N_0 \rightarrow g(n) > \tau_{f_s}$, por lo que $f(n) > \tau_{f_s} n$. Esto significa que incluso si tenemos una

eficiencia de la forma $f(N) = Nlog_2(N)$, aumentar indefinidamente el número de muestras inevitablemente hará que superemos el tiempo límite, *independientemente de la velocidad de nuestro procesador*. Es entonces de vital importancia optimizar los algoritmos que pretendemos usar para el procesamiento en tiempo real.[3]

4.2.1. Una descomposición diferente en bloques

La función característica no es el único instrumento que nos permite descomponer a σ en bloques. Podemos modificar a los conjuntos B para incluir a un *tamaño de salto* s junto con un tamaño de ventana L que permite sobrelape entre los intervalos. Formamos los intervalos haciendo $B^k_{s,L} = [ks, ks + L]$. El sobrelape que habrá entre intervalos sucesivos es de $ks + L - (k + 1)s = L - s$. Por ejemplo, si tomamos $s = N$, $L = 2N$, tendremos $B^k_{N,2N} = [kN, (k + 2)N]$. Cada intervalo sucesivo tiene un sobrelape de N ($s/L \to 50\,\%$). Podemos entonces escribir:

$$\sigma(t_n) = \sum_{k\in\mathbb{N}} \frac{1}{2}\sigma(t_n)\chi_{B^{2k}_{N,2N}}(t) + \frac{1}{2}\sigma(t_n)\chi_{B^{2k+1}_{N,2N}}(t).$$

Con una función de descomposición en bloques $w^k_{s,L}(t) = w^k_{N,2N}(t) = \frac{1}{2}\chi_{B^k_{N,2N}}(t)$. La primera propiedad que deben observar las funciones de descomposición es que sean cero para cualquier $t \notin B^k_{s,L}$. Formalmente:

$$w^k_{s,L}(t) > 0 \quad \text{cuando} \;\; t \in A,$$

$$w^k_{s,L}(t) = 0 \quad \text{cuando} \;\; t \notin A.$$

En general, si existe un número natural l tal que $ls = P$; tendremos sobrelapes de s/P por ciento que requerirán de p sumas para devolvernos la señal original. Explícitamente:

$$\sigma(t_n) = \sum_k \sigma(t_n)w^{pk}_{s,P}(t_n) + \sigma(t_n)w^{pk+1}_{s,P}(t_n) + ... + \sigma(t_n)w^{pk+(p-1)}_{s,P}(t_n)$$

La segunda propiedad que debe cumplir una función de descomposición es:

$$1 = \sum_k w^{lk}_{s,L}(t_n) + w^{lk+1}_{s,L}(t_n) + ... + w^{lk+(l-1)}_{s,L}(t_n)$$

Sólo habrá un bloque en el cual se sobrelapen todas las $w^{lk}_{s,L}(t_n)$ por cada k, podemos reducir entonces la condición a:

$$1 = w^l_{s,L}(t) + w^{l+1}_{s,L}(t) + ... + w^{lk+(l-2)}_{s,L}(t) + w^{lk+(l-1)}_{s,L}(t) \quad t \in [sk + (l-1)s, sk + ls]$$

[3] Un ejemplo de esto es el legendario *hack* para calcular $\frac{1}{\sqrt{x}}$ que usaron los desarroladores de Quake 3.

4.3. Transformada discreta de Fourier

Para discretizar la transformada, recordemos que cada coeficiente se puede calcular como

$$c_k = \frac{1}{2\pi}\int_{-\pi}^{\pi} f(x)e^{-ikx}dx.$$

Haciendo el cambio de variable $t = -\pi + x$, obtenemos

$$c_k = \frac{1}{2\pi}\int_{0}^{2\pi} f(-\pi + x)e^{-ik(-\pi+x)}dx.$$

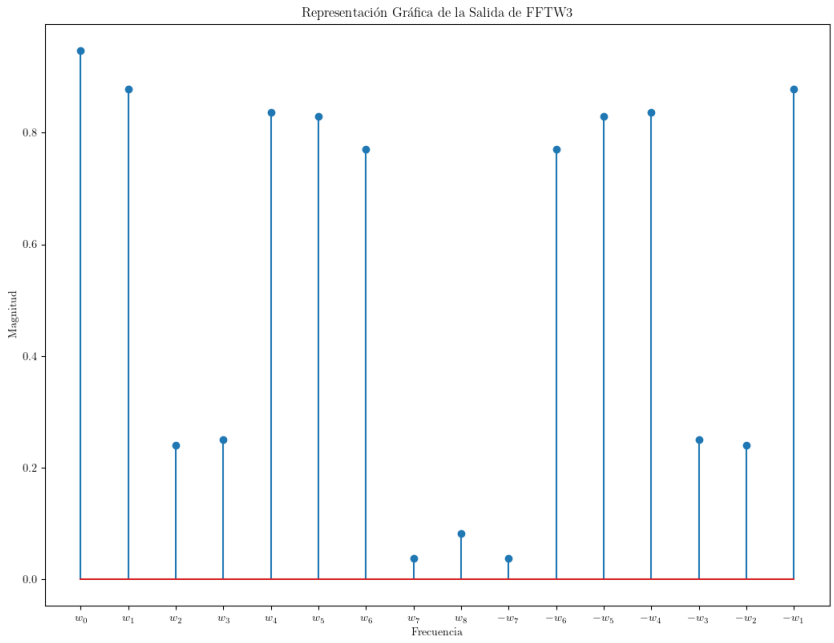
Podemos tomar el muestreo $t_k = 2\pi k/N = -\pi + x_k$ y aproximar la integral con una suma para obtener

$$c_k = \frac{1}{2\pi}\sum_{n=0}^{N-1} f(t_k)e^{-i2\pi nk/N}\Delta t_k.$$

Como $\Delta t_k = 2\pi/N$, obtenemos:

$$c_k = \frac{1}{N}\sum_{n=0}^{N-1} f_k e^{-i2\pi nk/N}.$$

Como todas las señales que estaremos procesando son reales, los coeficientes presentarán una simetría dada por $\hat{f}(\omega_{-k}) = \hat{f}(\omega_k)$. La librería FFTW3 que usaremos para implementar los resultados de ésta sección en la práctica organiza el resultado como sigue:



Otras librerías de cálculo numérico como numpy y scipy tienen una distribución diferente de componentes, por lo que es importante siempre revisar la documentación del código que estemos usando. Fijémonos que si hacemos $\omega_0 = 2\pi/N$, podemos definir a los coeficientes de Fourier como una transformación lineal definiendo

$$\Phi_{n,k} = e^{\frac{-i2\pi nk}{N}} = e^{-i\omega_0 nk}.$$

Podemos entonces reescribir la fórmula como:

$$c_k = \frac{1}{N}\sum_{k=0}^{N-1} \Phi_{n,k} f_k.$$

Si agrupamos a los términos f_k en un vector σ, obtenemos la forma matricial de los coeficientes de Fourier

$$\frac{1}{N}\begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & e^{-i2\omega_0} & \dots & e^{-i(N-1)\omega_0} \\ \dots & \dots & \dots & \dots \\ 1 & e^{-i2(N-1)\omega_0} & \dots & e^{-i(N-1)^2\omega_0} \end{bmatrix} \begin{bmatrix} \sigma_1 \\ \sigma_2 \\ \dots \\ \sigma_n \end{bmatrix} = \Phi\sigma.$$

La matriz de la transformación es una matriz de Vandermonde. Una propiedad

importante[4] de estas matrices es que su determinante se puede calcular como:

$$\det W(x_1, x_2, ..., x_n) = \prod_{1 \leq i < j \leq n} (x_j - x_i).$$

Lo que nos da inmediatamente la invertibilidad de la matriz. La simetría de Φ se puede aprovechar para optimizar el cálculo de la transformada. En 1965, James Cooley y John Tukey publicaron un artículo describiendo cómo usar una computadora para implementar una optimización de esta multiplicación de matrices que depende de la paridad del número de renglones. Observemos que si tenemos un número par de renglones en una matriz indizada a partir del 0, n será impar. El artículo original de Cooley y Tukey llega a la transformada rápida de Fourier de una forma distinta pero el principio básico del algoritmo es descomponer a la matriz como $\Phi = \Phi_1 + \Phi_2$ donde

$$\Phi_1 = \begin{bmatrix} 1 & 0 & 1 & ... & 0 & 1 & 0 \\ 1 & 0 & \omega^2 & ... & 0 & \omega^{n-1} & 0 \\ 1 & 0 & \omega^4 & ... & 0 & \omega^{2(n-1)} & 0 \\ ... & ... & ... & ... & 0 & ... & ... \\ 1 & 0 & ... & ... & 0 & \omega^{(n-1)^2} & 0 \\ 1 & 0 & \omega^{2n} & & 0 & \omega^{n(n-1)} & 0 \end{bmatrix},$$

$$\Phi_2 = \begin{bmatrix} 0 & 1 & 0 & ... & 1 & 0 & 1 \\ 0 & \omega^1 & 0 & ... & \omega^{n-2} & 0 & \omega^n \\ 0 & \omega^2 & 0 & ... & \omega^{2(n-2)} & 0 & \omega^{2n)} \\ ... & ... & ... & ... & ... & ... & ... \\ 0 & \omega^{n-1} & 0 & ... & \omega^{(n-1)(n-2)} & 0 & \omega^{n(n-1)} \\ 0 & \omega^n & 0 & ... & \omega^{n(n-2)} & 0 & \omega^{n^2} \end{bmatrix}.$$

Podemos reescribir a la transformada de Fourier discreta como unainterpolación de dos series geométricas de la siguiente forma:

$$[\Phi^n f]_l = \frac{1}{n} \sum_{m=0}^{(n-1)/2} \omega^{2ml} f_{2m} e_{2m} + \frac{1}{n} \sum_{m=0}^{(n-1)/2} \omega^{(2m+1)l} f_{2m+1} e_{2m+1}$$

$$= \frac{1}{n} \sum_{m=0}^{(n-1)/2} \omega^{2ml} f_{2m} e_{2m} + \omega^l \frac{1}{n} \sum_{m=0}^{(n-1)/2} \omega^{2ml} f_{2m+1} e_{2m+1}$$

$$= \Phi_{2\omega}^{(n-1)/2} f_{\text{par}} + \omega^l \Phi_{2\omega}^{(n-1)/2} f_{\text{impar}}.$$

[4] Véase Shilov, 1977

Donde f_{par}, f_{impar} representan los índices pares e impares del vector $f = (f_0, f_1, ..., f_n)$. Notemos que si extendemos la condición de paridad para considerar únicamente a potencias de dos, podemos iterar el algoritmo para reducir aún más el costo computacional. La implementación y optimización de este algoritmo está fuera de las miras de este trabajo pero la lectora interesada puede revisar la documentación de FFTW3[5].

4.4. Convolución Lineal Discreta

Sean $P_1(x) = \sum_{i=0}^{n} a_k x^k$, $P_2(x) = \sum_{j=0}^{m} b_k x^k$ dos polinomios en los reales con grados n, m, respectivamente. Su multiplicación claramente sigue siendo un polinomio de grado $n+m$, por lo que debe existir un $P_3(x) = \sum_{k=0}^{n+m} c_k x^k$ tal que:

$$P_3(x) = P_1(x)P_2(x.)$$

$$\sum_{k=0}^{n+m} c_k x^k = \left(\sum_{i=0}^{n} a_i x^i\right)\left(\sum_{j=0}^{m} b_j x^j\right) = \left(\sum_{i+j=0}^{n+m} a_i b_j x^{i+j}\right).$$

Formalmente, el k-ésimo término de la suma incluye a todos los índices tales que $i + j = k$. Usando $i = k - j$ y las relaciones:

$$j \leq k \leq m + j, \quad i \leq k \leq n + i.$$

Podemos desarrollar la suma de las siguientes maneras:

$$P_3(x) = \sum_{i=0}^{n}\sum_{k=i}^{m+i} a_i b_{k-i} x^k = \sum_{j=0}^{m}\sum_{k=j}^{n+j} b_j a_{k-j} x^k.$$

Si usamos las desigualdades:

$$0 \leq k - i \leq m, \quad 0 \leq k - j \leq n.$$

Podemos llegar a las expresiones alternativas:

$$P_3(x) = \sum_{k=0}^{n+m}\left(\sum_{i=\max(0,k-m)}^{\min(k,n)} a_i b_{k-i}\right) x^k = \sum_{k=0}^{n+m}\left(\sum_{j=\max(0,k-n)}^{\min(k,m)} b_j a_{k-j}\right) x \cdot k$$

El número de términos en la suma está dado por la diferencia $\max(0, k-n) - \min(k, m)$ y alcanzará su máximo cuando $n \leq k \leq m$. Nótese además que la multiplicación de dos polinomios de grados n, m con $n+1$ y $m+1$ términos nos da

[5]https://www.fftw.org/

un polinomio de grado $n+m$ con $(n+1)+(m+1)-1 = n+m+1$ términos. Para ilustrar el significado de estos resultados, consideremos la convolución discreta de polinomios con coeficientes a_0, a_1, a_2 y b_0, b_1. Si formamos las combinaciones teniendo en mente que la suma de los índices tiene que ser igual a la potencia de x^{n+m}, obtenemos:

$$A(x)B(x) = (a_0b_0)x^0 + (a_1b_0 + a_0b_1)x^1 + (a_1b_1 + a_2b_0)x^2 + (a_2b_1)x^3.$$

Nótese que los últimos coeficientes van descartando a índices cuya suma no es igual a la potencia requerida. La primera expresión que obtuvimos nos ayuda a entender cómo agrupar términos pero la segunda es más práctica si queremos calcular explícitamente los coeficientes. Podemos transferir estos resultados a espacios vectoriales de dimensión finita usando el mapeo que identifica coeficientes de un polinomio con coordenadas de un vector

$$c_0 + c_1x^1 + c_2x^2 + ...c_{n-1}x^{n-1} \to (c_0, c_1, c_2, ..., c_{n-1}).$$

Definido formalmente como

$$\varphi\left(\sum_{k=0}^{n-1} \lambda_k x^k\right) = \sum_{k=0}^{n-1} \lambda_k e_k.$$

Usando este mapeo podemos obtener una definición para la convolución de dos vectores. A diferencia de operaciones binarias como la suma o multiplicación, esta operación es un mapeo $* : V^n \times V^m \to V^{n+m-1}$ que no será cerrado para ningún espacio de dimensión mayor a 1 (en cuyo caso la convolución coincide con la multiplicación de escalares). Podemos comenzar con nuestra definición tomando a dos polinomios P_1, P_2 con grados $n-1$ y $m-1$, respectivamente. Tomamos:

$$P_1(x) = \sum_{i=0}^{n-1} a_i x^i, \quad P_2(x) = \sum_{j=0}^{m-1} b_i x^i.$$

Si mapeamos su multiplicación al espacio vectorial correspondiente, obtenemos:

$$\varphi(P_1P_2) = \varphi(P_1) \star \varphi(P_2) = u \star v := \sum_{i=0}^{n-1} \sum_{k=i}^{m-1+i} a_i b_{k-i} e_k.$$

Como el término b_{k-i} depende de dos índices, podemos expresar la convolución como un producto de una matriz por un vector. Podemos definir explícitamente a la matriz como:

$$[\Upsilon_m^n(v)]_{i,j} = \begin{cases} b_{j-i+1} & \text{si } \ (1 \leq j-i+1) \ \text{ ó } \ (j-i+1 \leq m)\,, \\ 0. & \text{en otro caso} \end{cases}$$

Las columnas de la matriz $\Upsilon_m^n(v)$ n contienen permutaciones circulares del vector $v^\star = (b_0, b_1, ..., b_{m-1}, \underbrace{0, ..., 0}_{n-1})$ que coincide con v en sus primeras m coordenadas y tiene n ceros en el resto de sus coordenadas.

$$u \star v = \sum_{i=0}^{n} \sum_{k=i}^{m+i} a_i b_{k-i} e_k.$$

$$= \begin{bmatrix} b_0 & 0 & 0 & 0 & ... & 0 & 0 & 0 \\ b_1 & b_0 & 0 & 0 & ... & 0 & 0 & 0 \\ b_2 & b_1 & b_0 & 0 & ... & 0 & 0 & 0 \\ ... & ... & ... & ... & ... & ... & ... & ... \\ b_{n-2} & b_{n-3} & b_{n-4} & b_{n-5} & ... & b_1 & b_0 & 0 \\ b_{n-1} & b_{n-2} & b_{n-3} & b_{n-4} & ... & b_2 & b_1 & b_0 \\ ... & ... & ... & ... & ... & ... & ... & ... \\ b_{m-1} & b_{m-2} & b_{m-3} & b_{m-4} & ... & b_{m-n+2} & b_{m-n+1} & b_{m-n} \\ 0 & b_{m-1} & b_{m-2} & b_{m-3} & ... & b_{m-n+3} & b_{m-n+2} & b_{m-n+1} \\ ... & ... & ... & ... & ... & ... & ... & ... \\ 0 & 0 & 0 & 0 & ... & b_{m-1} & b_{m-2} & b_{m-3} \\ 0 & 0 & 0 & 0 & ... & 0 & b_{m-1} & b_{m-2} \\ 0 & 0 & 0 & 0 & ... & 0 & 0 & b_{m-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ ... \\ a_{n-5} \\ a_{n-4} \\ a_{n-3} \\ a_{n-2} \\ a_{n-1} \end{bmatrix}$$

$$= \Upsilon_m^n(v)u.$$

La expresión matricial nos permite verificar que hay un número máximo de términos en la suma cuando $n < i < m$. Podemos también obtener trivialmente que:

$$\Upsilon_m^n(v)u = \Upsilon_n^m(u)v.$$

4.5. Convolución circular discreta

Como vimos en la sección anterior, las columnas de la matriz de convolución $\Upsilon_m^n(v)$ contienen a permutaciones circulares de un vector $v^\star = (b_0, b_1, ..., b_{m-1}, \underbrace{0, ..., 0}_{n-1})$. Si en vez de rellenar al vector con ceros asumiéramos una periodicidad de los índices, obtendríamos lo que se conoce como convolución circular o periódica. Podemos calcularla como:

$$u \circledast v = \sum_{i+j=0}^{n+m-1} a_i b_j e_{(i+j) \bmod (n)} = \sum_{i=0}^{n-1} \sum_{k=i}^{m-1+i} a_i b_{k-i} e_{k \bmod (n)}.$$

De modo que la convolución circular se sobrelapa consigo misma a partir de el índice n. Nos referiremos a este fenómeno como *distorsión circular*[6] y se vuelve bastante severa con funciones periódicas restringidas a dominios finitos. La matriz que representa a la convolución circular se puede obtener de la matriz de la convolución lineal eliminando el renglón $m-1+k$ y sumándolo al $k-1$:

$$u \circledast v = \sum_{i=0}^{n-1} \sum_{k=i}^{m-1+i} a_i b_{k-i} e_{k \bmod(n)}.$$

$$= \begin{bmatrix} b_0 & b_{m-1} & b_{m-2} & b_{m-3} & ... & b_{n-3} & b_{n-2} & b_{n-1} \\ b_1 & b_0 & b_{m-1} & b_{m-2} & ... & b_{n-2} & b_{n-3} & b_{n-2} \\ b_2 & b_1 & b_0 & b_{m-1} & ... & b_{n-1} & b_{n-4} & b_{n-3} \\ ... & ... & ... & ... & ... & ... & ... & ... \\ b_{n-2} & b_{n-3} & b_{n-4} & b_{n-5} & ... & b_1 & b_0 & b_{m-1} \\ b_{n-1} & b_{n-2} & b_{n-3} & b_{n-4} & ... & b_2 & b_1 & b_0 \\ ... & ... & ... & ... & ... & ... & ... & ... \\ b_{m-1} & b_{m-2} & b_{m-3} & b_{m-4} & ... & b_{n-4} & b_{n-3} & b_{n-2} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ ... \\ a_n \\ a_{n-1} \end{bmatrix}$$

$$= \Theta_v^u u.$$

Con:

$$[\Theta_v^u]_{i,j} = b_{(i-j) \bmod(m)}.$$

De donde que obtenemos otra fórmula para la convolución circular:

$$[\Theta_v^u u]_i = \sum_{j=0}^{n-1} a_j \; b_{(i-j) \bmod(m)}.$$

Podemos identificar a los términos de *distorsión* circular expresándo la suma como una convolución lineal sumada con unos términos adicionales. Claramente toda combinación de índices tal que $i-j>0$ y $b_{(m+i-j) \bmod(m)} \neq 0$ no se encontrará en la convolución lineal; por lo que podemos separar a la suma como:

$$\sum_{j=0}^{n-1} a_j \; b_{(i-j) \bmod(m)} = \sum_{j=0}^{i} a_j \; b_{i-j} + \sum_{j=i+1}^{n-1} a_j \; b_{m+i-j}.$$

La convolución circular coincidirá con la lineal cuando:

$$\sum_{j=i+1}^{n-1} a_j \; b_{m+i-j} = 0,$$

$$b_{m+i-j} = 0 \quad \text{para} \;\; m-1 \leq m+i-j \leq n-1.$$

[6] Conocido en la literatura como *time-aliasing*.

Este resultado nos da una estrategia para eliminar la distorsión circular en la convolución. Para aprovechar la optimización de FFTW3, consideraremos únicamente a vectores con 2^n entradas. Sea b un vector así y definamos a:

$$b^\star = (b_0, b_1, ..., b_{2^n-1}, \underbrace{0, ..., 0}_{2_n}).$$

Si formamos nuestra matriz de convolución claramente tendremos:

$$= \begin{bmatrix} b_0 & b_{2^{2n}-1} & b_{2^{2n}-2} & ... & b_{2^n+2} & b_{2^n+1} & b_{2^n} \\ b_1 & b_0 & b_{2^{2n}-3} & ... & b_{2^n+3} & b_{2^n+2} & b_{2^n+1} \\ b_2 & b_1 & b_0 & ... & b_{2^n+4} & b_{2^n+3} & b_{2^n+2} \\ ... & ... & ... & ... & ... & ... & ... \\ b_{2^n-3} & b_{2^n-4} & b_{2^n-5} & ... & b_0 & b_{2^{2n}-1} & b_{2^{2n}-2} \\ b_{2^n-2} & b_{2^n-3} & b_{2^n-4} & ... & b_1 & b_0 & b_{2^{2n}-1} \\ b_{2^n-1} & b_{2^n-2} & b_{2^n-3} & ... & b_2 & b_1 & b_0 \\ ... & ... & ... & ... & ... & ... & ... \\ b_{2^{2n}-3} & b_{2^{2n}-4} & b_{2^{2n}-5} & ... & b_{2^n-1} & b_{2^n-2} & b_{2^n-3} \\ b_{2^{2n}-2} & b_{2^{2n}-3} & b_{2^{2n}-4} & ... & b_{2^n} & b_{2^n-1} & b_{2^n-2} \\ b_{2^{2n}-1} & b_{2^{2n}-2} & b_{2^{2n}-3} & ... & b_{2^n+1} & b_{2^n} & b_{2^n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ ... \\ a_{2^n-1} \end{bmatrix}$$

$$= \begin{bmatrix} b_0 & 0 & 0 & ... & 0 & 0 & 0 \\ b_1 & b_0 & 0 & ... & 0 & 0 & 0 \\ b_2 & b_1 & b_0 & ... & 0 & 0 & 0 \\ ... & ... & ... & ... & ... & ... & ... \\ b_{2^n-3} & b_{2^n-4} & b_{2^n-5} & ... & b_0 & 0 & 0 \\ b_{2^n-2} & b_{2^n-3} & b_{2^n-4} & ... & b_1 & b_0 & 0 \\ b_{2^n-1} & b_{2^n-2} & b_{2^n-3} & ... & b_2 & b_1 & b_0 \\ ... & ... & ... & ... & ... & ... & ... \\ 0 & 0 & 0 & ... & b_{2^n-1} & b_{2^n-2} & b_{2^n-3} \\ 0 & 0 & 0 & ... & 0 & b_{2^n-1} & b_{2^n-2} \\ 0 & 0 & 0 & ... & 0 & 0 & b_{2^n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ ... \\ a_{2^n-1} \end{bmatrix}.$$

4.6. Convolución particionada uniforme con sobrelape

La multiplicación de una matriz de mn por un vector de n coordenadas involucra n sumas y multiplicaciones por cada renglón, resultando en $2mn$ operaciones en total (sin considerar el costo de acceder a la memoria del sistema). Una convolución lineal calculada en forma matricial requerirá entonces $2(n+m-1)n = 2n^2+2mn-n$ operaciones. Sin pérdida de generalidad podemos suponer que $n > m$. Obtenemos

entonces que $2n^2+2mn-n < 4n^2$ y podemos escribir $2(n+m-1)n = \mathcal{O}(n^2)$. Para ilustrar los obstáculos técnicos que representa una complejidad computacional así, consideremos a un par de vectores u, v con n y m entradas, respectivamente. Supongamos que la frecuencia promedio de operaciones de punto flotante es una proporción λ de la velocidad de reloj ν de nuestro procesador (comúnmente al rededor de $3 \times 10^9 s^{-1}$). Tenemos entonces la relación estimada:

$$\frac{2(n^2+mn-1)}{\lambda\nu} < n/f_s.$$

Fijando n, despejamos para m:

$$m+n < \frac{\lambda\nu}{2f_s} + \frac{1}{n.}$$

Como $n < 1$, el término $\frac{1}{n}$ será menor a 1 y por lo tanto no afectará significativamente nuestro estimado de m; por lo que podemos usar:

$$m+n < \frac{\lambda\nu}{2f_s}.$$

Para una eficiencia de $\lambda = 1$ (100 %), una velocidad de reloj de procesador de 3×10^9 y una frecuencia de muestreo estándar de $4.8\times10^5 s^{-1}$; obtenemos un límite de $n+m < 3125$. Por lo que la cota para la duración combinada de dos señales cuya convolución podemos calcular en tiempo real será aproximadamente 65.1ms.

Podemos usar el teorema de la convolución para reducir la complejidad que supone calcula una multiplicación de matrices. El plan de ataque a grandes rasgos es aplicar la transformada de Fourier de dos señales, multiplicar sus transformadas punto a punto y luego aplicar la transformada inversa para obtener la convolución de las dos señales. Como ya existen librerías altamente optimizadas para realizar la transformada rápida de Fourier de dos señales[7], el resto de esta sección está dedicada a describir la multiplicación punto a punto de las descomposiciones por bloques de un par de señales. Si además lo hacemos de manera que obtengamos una señal sin distorsión, obtenemos el denominado algoritmo de convolución particionada uniforme (Wefers, 2015) UPOLS[8].

Para empezar, sea x una señal y r una respuesta de impulso. Supongamos que ambas tienen las descomposiciones en bloques de $2N$ muestras:

$$x = \sum_{i\in\mathbb{N}} x_i,\ r = \sum_{j\in\mathbb{N}} r_j.$$

[7]Aquí usamos FFTW3, la transformada de Fourier más rápida del Oeste. http://www.fftw.org/

[8]Por sus siglas en inglés: *Uniformly Partitioned Overlap Save.*

La convolución de estas dos señales, por la propiedad de linealidad, se puede escribir como:

$$x \star r = \sum_{i,j \in \mathbb{N}} x_i \star r_j.$$

La transformada de Fourier $\mathcal{F}(x * r$ de tal convolución se puede calcular como:

$$\mathcal{F}(x \star r) = \sum_{i,j \in \mathbb{N}} \mathcal{F}(x_i)\mathcal{F}(r_j) = \sum_{i,j \in \mathbb{N}} u_i v_j.$$

Supongamos que la respuesta de impulso es finita. Para encontrar los términos que tienen que sonar hasta el bloque $i = N$, basta organizar a la suma como:

$$\mathcal{F}(x \star r) = \sum_{i,j \in \mathbb{N}} u_i \rho_j =$$
$$= u_1\rho_1 + u_1\rho_2 + ... + u_1\rho_n$$
$$+u_2\rho_1 + u_2\rho_2 + ... + u_2\rho_n$$
$$+ \quad ... \quad +$$
$$+u_n\rho_1 + u_n\rho_2 + ... + u_n\rho_n + ...$$

Si u_n es un bloque más reciente que u_{n-1}, podemos organizar a la suma en las diagonales:

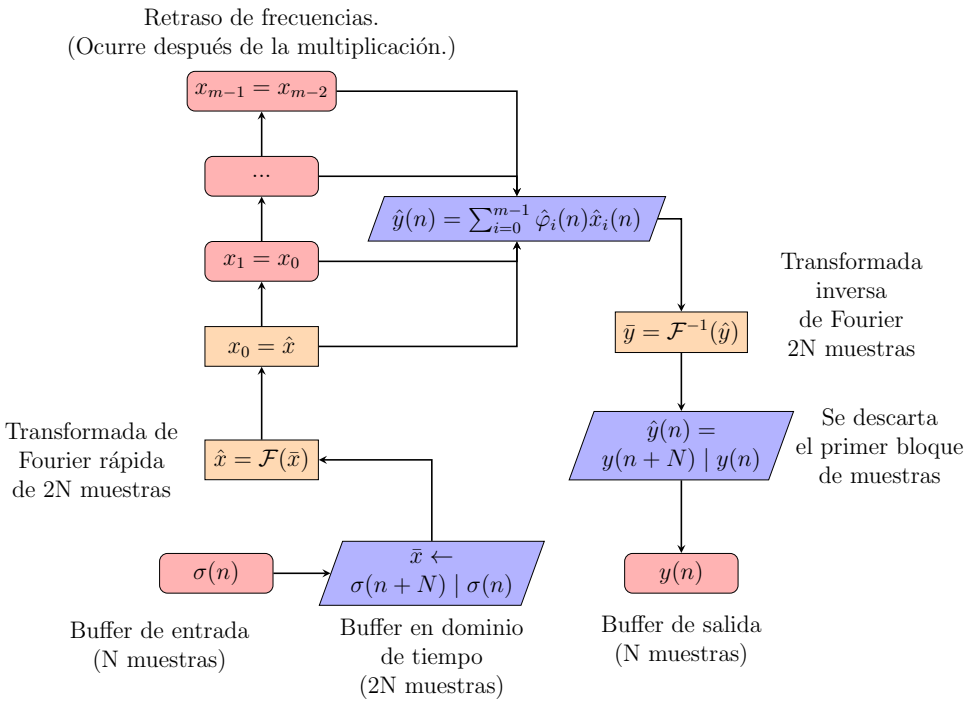
$$\mathcal{F}(x \star r) = (u_1\rho_1) + (u_2\rho_1 + u_1\rho_2) + (u_3\rho_1 + u_2\rho_2 + u_1\rho 3)$$
$$+... + (u_n\rho_1 + u_{n-1}\rho_2 + ... + u_1\rho_n).$$

Si tomamos $u_k = 0$ para $k < 1$, podemos expresar a cada paréntesis en la forma:

$$\sum_{i=1}^{n} u_{k-i}\rho_i.$$

Como la respuesta de impulso aparece en orden, cada paréntesis corresponderá a la salida correcta para el bloque n. Aplicando los resultados de la sección anterior para eliminar la distorsión circular, veremos que cuando entra el i-ésimo grupo de $2N$ muestras, sólo necesitamos calcular $\sum_{j=0}^{m-1} x_{i+j}\varphi_{j \bmod(m)}$ y descartar las primeras N muestras. En total, cada i-ésimo término de la suma contiene $m-1$ sumas y m multiplicaciones, lo que nos da un total de $2m-1$ operaciones aritméticas[9] por cada iteración del algoritmo. Sorprendentemente y en contraste con la multiplicación de matrices, UPOLS nos da eficiencia lineal a cambio de un pequeño retraso de N muestras. El algoritmo se puede representar con el siguiente diagrama:

[9]Sin contar el costo de acceder a la memoria, que puede exceder el costo de hacer aritmética en algunos casos.



El algoritmo se puede implementar usando el siguiente bloque de procesamiento:

```
1  int jack_callback (jack_nframes_t nframes, void *arg){
2    jack_default_audio_sample_t *in, *out;
3    int i, j, k;
4    in  = (jack_default_audio_sample_t *) jack_port_get_buffer (input_port,
5                                                                nframes);
6    out = (jack_default_audio_sample_t *) jack_port_get_buffer (output_port,
7                                                                nframes);
8    for (i = 0; i < nframes; i++){
9      buffer[nframes + i] = in[i];
10     i_time[i] = buffer[i];
11     i_time[nframes+i] = buffer[nframes+i];
12   }
13   fftw_execute(i_forward);
14   for (i = 0; i < two_nframes; i++){
15     for (k = partitions - 1; k > 0; k--){
16       fdl[k][i] = fdl[k-1][i];
17     }
18   }
```

```
19    for (i = 0; i <= nframes; i++){
20      fdl[0][i] = i_fft[i];
21      o_fft[i] = 0.0 + I*0.0;
22    }
23    for (i = 0; i < two_nframes; i++){
24      for (k = 0; k < partitions; k++){
25        o_fft[i] += fdl[k][i] * fir[k][i];
26      }
27    }
28    fftw_execute(o_inverse);
29    for (i = 0; i < nframes; i++){
30      out[i] = vol*creal(o_time[nframes+i])/two_nframes;
31      buffer[i] = in[i];
32      }
33    return 0;
34    }
```

Las partes específicas a `C` son mínimas, por lo que no debería presentar mucha dificultad adaptar este algoritmo a otros lenguajes de programación. Hay un número de pasos de preprocesamiento adicional que se deben realizar antes de empezar la función de procesamiento en tiempo real que se pueden consultar en en apéndice. Veamos el código por partes:

`jack_callback.c`**:1-7** Código específico de `C` que establece las muestras de audio que entran y las que salen.

`jack_callback.c`**:8-12** Las muestras de entrada se escriben en las últimas n entradas del buffer.

`jack_callback.c`**:13** Aplicamos la transformada rápida de Fourier al buffer de entrada.

`jack_callback.c`**:14-18** Aquí actualizamos la línea de retraso de frecuencias. Al reemplazar el k-ésimo renglón por el $(k-1)$-ésimo renglón, nos aseguramos de dejar intactos a todos los renglones con índice $n < k$. Nótese que se modificarán todos los renglones *menos el primero*.

Esta función se ha dejado así para aumentar la claridad de la exposición. La versión final ha sido optimizada para evitar copiar los arreglos muestra por muestra. El código se puede encontrar en el apéndice C.

`jack_callback.c`**:19-22** Escribimos la transformada de Fourier del buffer actual al primer renglón (¡que dejamos sin modificar!) de la línea de retraso de

frecuencias. Como en la siguiente etapa estamos acumulando valores sobre `o_fft`, este arreglo contendrá información del buffer anterior. Necesitamos entonces restablecerlo a cero para evitar que crezca indefinidamente.

`jack_callback.c`**:23** Regresamos al dominio del tiempo.

`jack_callback.c`**:30** Normalizamos la salida de FFTW3, multiplicamos por el control de volumen y descartamos las primeras n muestras.

`jack_callback.c`**:31** Reemplazamos las primeras n muestras del buffer con las últimas n muestras.

Capítulo 5

Epílogo

A farewell to the reader
We are now, reader, arrived at the last stage of our long journey. As we have, therefore, travelled together through so many pages let us behave to one another like fellow-travelers in a stage coach, who have passed several days in the company of each other; and who, notwithstanding any bickerings or little animosities which may have occured on the road, generally make all up at last, and mount, for the last time, into their vehicle with chearfulness and good humour; since after this one stage, it may possibly happen to us, as it commonly happens to them, never to meet no more.

Henry Fielding, *The History of Tom Jones.*

Habiendo presentado una introducción previa a los métodos del estado del arte de la convolución en tiempo real, solamente nos queda por mencionar qué otras direcciones han quedado sin explorar en este trabajo.

5.1. La Transformada de Fourier

En nuestro desarrollo de la teoría básica de análisis de Fourier realmente no llegamos al concepto general de *transformada de Fourier*. Para obtener todos los resultados discretos fue suficiente trabajar exclusivamente con los coeficientes de las series de Fourier, con lo que vemos que las versiones discretas (relevantes al problema que discutimos) de la transformada realmente no son propiamente *transformadas de Fourier* en el sentido de que no necesitan a la versión continua para tener sentido. La segunda parte de (Körner, 1989) está dedicada a extender los resultados de series a transformadas integrales. Otro tema que no se abordó fue la generalización de la teoría de las series de Fourier y el muestreo a dimensiones más grandes. Una función de interpolación para muestreos de dimensiones más grandes nos podría dar una manera de recolectar mediciones más complicadas que puntos. Por ejemplo, si colocamos un micrófono en frente de una bocina y la vamos deslizando a la derecha mientras tomamos mediciones de la respuesta de impulso en cada punto, podemos ver a cada respuesta de impulso como una *rebanada* de una función en un espacio de dimensión mayor. Es natural preguntarnos entonces la forma que toma la función de interpolación en estos casos y cómo es que se traduce la frecuencia de Nyquist-Shannon.

Por otro lado, el concepto de la *respuesta de impulso* de un sistema de puede explotar para obtener resultados espectaculares en el campo de las ecuaciones diferenciales parciales, como se puede ver en (Sobolev, 1964). En ciertos casos, una ecuación diferencial parcial se puede resolver encontrando un núcleo apropiando y calculando una convolución (a veces, una generalización de la convolución) con las condiciones iniciales. Para ciertos sistemas físicos dados por EDPs cuyo comportamiento es lineal y no varía con el tiempo, las soluciones pueden ser vistas directamente como una convolución. La relación entre núcleos y unidades aproximadas nos dan, por lo menos en el caso lineal, una equivalencia entre la técnica de núcleos con la técnica de series para resolver EDPs. Una aplicación sorprendente de esto son las ecuaciones parabólicas que se pueden resolver a través del núcleo del calor. Aplicando la técnica de la convolución podemos calcular la propagación de calor en una dimensión bastante más rápido que lo que es permitido por los métodos de elementos finitos. Una generalización del teorema de la convolución y el algoritmo UPOLS podría ser directamente aplicable para el estudio de la propagación de calor en dimensiones mayores a uno. Mi conjetura es que esta técnica debería mantener ventajas de eficiencia sobre métodos actuales de elementos finitos.

Por último, vale la pena mencionar que el análisis de Fourier se puede formular de manera muy general en grupos abelianos y grupos de Lie. En las primeras páginas de (Helgason, 1968) podemos leer cómo es que la segunda interpretación nos da el hecho sorprendente de que el nucleo de Poisson en el disco es simplemente

una versión no Euclideana (en geometría hiperbólica, de hecho) de la transformada de Fourier clásica!

5.2. Algoritmos Más Eficientes

En Gardner, 1994, Bill Gardner describe un algoritmo de convolución particionada no-uniforme[1] que -en teoría- podría ser usado para calcular la convolución entre una señal de entrada y una respuesta de impulso con un retraso prácticamente nulo[2]. Por otro lado, en Garcia, 2002 se usó el algoritmo de Viterbi para determinar el tamaño óptimo de las particiones. Nuestra implementación se podría beneficiar del concepto de Gardner (el de García está patentado, al parecer). El problema principal que representan las particiones no-uniformes es la de repartir equitativamente las operaciones en cada buffer. Para ilustrar este problema, supongamos que el bloque de entrada consiste de N muestras y consideremos una implementación donde tenemos a un *buffer* de MN muestras (donde M es un entero mayor a 1) que acumula bloques de la señal entrante. El algoritmo de NUPOLS se puede comprender como dos algoritmos UPOLS con buffers de tamaño diferente. Como estamos esperando a que se acumule el buffer más grande *antes* de realizar la transformada de Fourier, en la práctica el procesador verá un salto en su carga de cómputo cada M bloques, lo que nos presenta con un cuello de botella computacional. Una manera de resolver este problema podría consistir en asignar cada instancia de UPOLS a núcleos diferentes; lo que limitaría la cantidad de particiones que podemos usar. Otro método para homogeneizar la carga del procesador consiste en calcular transformadas de Fourier parciales para las instancias de NUPOLS con buffers mayores a N; el problema que esto presenta está en aprovechar la optimización de FFTW3 -que no admite transformadas parciales- para no desarrollar una implementación optimizada específica a nuestras necesidades. La paralelización y las transformadas parciales son problemas que requieren un desarrollo cuidadoso de las ideas presentadas en este trabajo. Una implementación sencilla de código abierto sin duda sería una aportación importante.

Otra manera completamente diferente de resolver el problema de las respuestas de impulso de tamaños grandes está presente en Schroeder y Logan, 1961; donde se discute el concepto de densidad de modos de vibración en un cuarto. Aquí se argumenta que la densidad de modos aumenta muy rápido con el tiempo y la frecuencia, por lo que la respuesta de impulso empieza a volverse *estadística* después de un tiempo δt que depende de las dimensiones del cuarto. Podríamos

[1] NUPOLS por sus siglas en inglés

[2] En la práctica no puede ser cero, pero un retraso que se encuentre en el orden del tiempo que tarda a una computadora realizar unas cuantas operaciones aritméticas seguiría estando por debajo de nuestra capacidad de percibirlo como un retraso.

entonces estimar el tiempo de reverberación de una respuesta de impulso, restringir la respuesta a δt y usar algún algoritmo de reverberación -al estilo de Schroeder- propiamente afinado para extender el tiempo de reverberación.

Apéndice A

Respuestas de Impulso con Python

Las librerías de cálculo numérico de python están altamente optimizadas para realizar ciertos cálculos rutinarios; por lo que todo el procesamiento que no requiriera de acceso al *hardware* o no tuviera una prioridad de ser completado en tiempo real se realizó usando scripts de python. Esto incluye alineación de fase y eliminación de espacios en blanco debidos al retraso de entrada-salida.

```
1  from scipy.io.wavfile import read, write
2  from scipy.stats.stats import pearsonr
3  import matplotlib.pyplot as plt
4  import numpy as np
5  import sys
6
7  # setup_irs: revisar que las frecuencias de muestreo sean iguales,
8  # igualar longitudes amortiguando con ceros si es necesario,
9  # normalizar y alinear en fase las respuestas de impulso.
10 def setup_irs(ir_filenames, L2_normalize = True, allign = True):
11         # sample rates
12         sr_list = []
13         # irs
14         ir_list = []
15         # sample count
16         sample_list = []
17         # number of files
18         file_count = len(ir_filenames)
19
20         for file in ir_filenames:
```

```
21                tmp_sr, tmp_ir = read(file)
22                sr_list.append(tmp_sr)
23                ir_list.append(tmp_ir)
24                sample_list.append(len(tmp_ir))
25
26        if len(ir_list) != file_count:
27                print("error reading files")
28                return np.zeros(file_count)
29
30        if len(set(sr_list)) > 1:
31                print("sample rates do not match!")
32                return np.zeros(file_count)
33
34        max_samples = max(sample_list)
35        min_samples = min(sample_list)
36        if len(set(sample_list)) > 1:
37                pad_size = max_samples - min_samples
38
39                # Esto es una necesidad para incluir el texto en el
40                # libro. En la práctica no se necesita definir mensaje
41                # como un string multilínea.
42
43                mensaje = """Las respuestas de impulso tienen
44                tamaños distintos, extendiendo
45                con ceros para igualar cuenta de caractéres..."""
46                print(mensaje.replace('\n', ' '))
47                for ir in ir_list:
48                        if len(item) > max_samples:
49                        np.pad(ir, (0, pad_size), 'constant')
50
51        first_ir = ir_list[0]
52
53        if L2_normalize:
54                ir_list_normalized = []
55                for ir in ir_list:
56                        ir = ir/np.linalg.norm(ir)
57                        ir_list_normalized.append(ir)
58                ir_list = ir_list_normalized
59
60        if allign:
```

```
61                  return allign_irs(ir_list), sr_list[0]
62          else:
63                  return ir_list, sr_list[0]
64
65  def allign_irs(ir_list):
66          shifted_ir_list = []
67          if len(ir_list) > 1:
68                  first_ir = ir_list[0]
69                  for ir in ir_list:
70                          crel, olag = optimal_lag(first_ir, ir)
71                          ir = np.roll(ir, olag)
72                          shifted_ir_list.append(ir)
73                  return shifted_ir_list
74          else:
75                  print("cant allign lists with less than two elements")
76                  return ir_list
77
78  def interpolate(ir0, ir1, t):
79          if (t < 0) or (1 < t):
80                  print("t must be inside (0, 1) interval!\n")
81                  return [-1]
82          ir_t = (1-t)*ir0 + t*ir1
83          return ir_t
84
85  # find best interpolation parameter for a measured IR that
86  # is taken between two IRs
87  def optimal_t(measured_ir, ir0, ir1, precision = 0.01):
88          interpolation_range = np.arange(0, 1+precision, precision)
89          corr_coef = -1
90          best_t = -1
91          for t in interpolation_range:
92                  ir_t    = interpolate(ir0, ir1, t)
93                  coef_t  = pearsonr(ir_t, measured_ir)[0]
94                  if coef_t > corr_coef:
95                          best_t = t
96                          corr_coef  = coef_t
97          return best_t, corr_coef
98
99  def optimal_lag(ir0, ir1):
100         len_ir0 = len(ir0)
```

```
101         len_ir1 = len(ir1)
102 
103         ccrel_array   = np.correlate(ir0, ir1, mode = 'full')
104         max_ccrel     = max(ccrel_array)
105         optimal_delay = np.where(ccrel_array == max_ccrel)[0][0]
106                       - len_ir0 + 1
107         return max_ccrel, optimal_delay
108 
109 def plot_ccorr(file_0, file_1, save = False):
110         ir0, ir1, sr = setup_irs(file_0, file_1, L2_normalize = True)
111         len_ir0 = len(ir0)
112         len_ir1 = len(ir1)
113 
114         ccrel_array = np.correlate(ir0, ir1, mode = 'full')
115         lag_array   = np.linspace(-len_ir0,
116                                   len_ir1,
117                                   len_ir0 + len_ir1 - 1)
118 
119         max_correlation = max(ccrel_array)
120         max_crel_index  = np.where(ccrel_array == max_correlation)[0][0]
121                                       - len_ir0 + 1
122 
123         tag1 = file_0.split('.')[0]
124         tag2 = file_1.split('.')[0]
125         plt.plot(lag_array, ccrel_array)
126         plt.xlabel('samples')
127         plt.ylabel('apmlitude')
128         plt.suptitle(f"cross correlation of {tag1} and {tag2}")
129         plt.title(f"max corr: {np.round(max_correlation, 4)},
130                   lag index: {max_crel_index}")
131         if save:
132                 plt.savefig(f"{tag1}_ccorr_{tag2}.jpg")
133         plt.show()
134 
135 def plot_wav(file, save = False):
136         samplerate, audio = read(file)
137         print(f"File samplerate: {samplerate}\n")
138 
139         L2_norm = np.linalg.norm(audio)
140         audio = audio/L2_norm
```

```
141         #print(f"L2 norm of signal: {L2_norm}\n")
142
143         title = file.split('.')[0]
144         plt.plot(audio)
145         plt.xlabel('samples')
146         plt.ylabel('amplitude')
147         plt.title(title)
148         if save:
149                 plt.savefig(f"{title}.jpg")
150                 plt.show()
151
152 def crop_ir(og_ir, target_file, cut_sample):
153         cropped_ir = og_ir[cut_sample:]
154         write(target_file, sr, cropped_ir)
155         print("done")
```

Apéndice B

Cómo Usar Código Para Procesar Señales

Hay un número de plataformas en las cuales podemos implementar los resultados que veremos en este trabajo. Se pueden agrupar, a grandes rasgos, en procesadores dedicados y computadoras de propósito general. Cada uno admite una variedad de lenguajes de programación; pero muchos tienen en común C/C++. Con algunas excepciones; la mayoría de los lenguajes de programación siguen convenciones de C; y es por esta razón que lo usaremos para presentar los algoritmos.

Como el procesamiento presentado en esta tesis está basado en Linux, necesitamos dar una introducción a JACK Audio Connection Kit. El código necesario para accesar a los puertos de hardware se puede aislar en una subrutina independiente de los algoritmos. Habiendo separado el código así, lo que queda es una abstracción que se puede trasladar fácilmente a otro ambiente. *Por ejemplo, en Windows/MacOS existe JUCE Framework; un entorno de desarrollo basado en C/C++ que podemos usar para crear plugins.* La ventaja de hacer las cosas de esta manera es que sólo necesitamos modificar la subrutina de acceso al hardware y no el algoritmo. El archivo de cabecera que inicializa el servidor, declara las variables globales y se encarga de conectar todos los puertos se llama `autojackio.h` y se puede encontrar en el apéndice C. Incluirlo nos permite hacer una simplificación importante del código. En general, la parte de procesamiento en tiempo real irá en `jack_callback` y la parte de pre-procesamiento irá en `main`.

B.1. Entra por un lado y sale por el otro

Lo más sencillo que podemos hacer en procesamiento de señales es *no hacer nada*. El siguiente cliente de JACK ilustra este principio copiando la entrada a la salida sin alterar nada:

jack_in_to_out.c

```
#include "autojackio.h"

int jack_callback (jack_nframes_t nframes, void *arg){
  for (int i = 0; i < num_channels; i++){
    // obtenemos las muestras del irán o vienen del hardware
    in[i]  = (jack_default_audio_sample_t*) jack_port_get_buffer ( input_port[i],
    nframes);
    out[i] = (jack_default_audio_sample_t*) jack_port_get_buffer (output_port[i],
    nframes);

      // bloque de procesamiento
      for (int j = 0; j < nframes; j++){
          out[i][j] = in[i][j];
      }
  }
  return 0;
}

int main (int argc, char *argv[]) {
  const char *client_name = "in_to_out";
  num_channels = 2;
  working = 1;

  initialize_client(client_name, num_channels)
  connect_ports();
  signal(SIGINT, INThandler);

  do{usleep (10);} while (working == 1);

  jack_client_close (client);
  printf("\nGoodbye!\n");
  exit (0);
}
```

El código anterior es posiblemente lo más sencillo que podemos hacer con C y

JACK. Necesitamos separar la parte en la que inicializamos el cliente de la parte en la que empezamos a procesar audio. La razón de hacer las cosas de esta forma es que al separar estas dos cosas podemos accesar a las variables de frecuencia de muestreo y número de muestras por bloque *antes* de empezar a procesar audio. Veamos cómo funciona por partes:

`jack_in_to_out.c:1` El archivo de cabecera `autojackio.h` incluye todo lo que necesitamos para automatizar la conexión de puertos físicos.

`jack_in_to_out.c:3` El nombre estándar de una función de JACK. El único argumento que realmente necesita es el número de muestras que tiene que procesar por bloque. Esto se establece en la configuración del servidor de JACK.

`jack_in_to_out.c:4-7` Para cada canal de audio usamos `jack_port_get_buffer` para definir las muestras que vienen de -o van a- el hardware.

`jack_in_to_out.c:10-12` Aquí podemos operar directamente sobre las muestras. En este ejemplo sólo copiamos el canal de entrada `in[i]` al canal de salida `out[i]`.

`jack_in_to_out.c:18-20` Definimos el nombre que tendrá el cliente, el número de canales que usaremos y declaramos que estamos listos para empezar a trabajar.

`jack_in_to_out.c:22` Esta función está definida en `autojackio.h`. Su propósito es inicializar el cliente de jack junto con las variables que ocuparán los puertos.

`jack_in_to_out.c:23` Connect ports conecta los puertos de hardware e inicia el procesamiento.

`jack_in_to_out.c:24` Aquí estamos diciéndole a C que esperamos una entrada de teclado del usuario. Cuando se presiona CTRL+C, se llama a la función `INThandler` (definida también en `autojackio.h`) para cambiar el valor de `working` a 0.

`jack_in_to_out.c:26` Esto evita que el programa llegue a su fin y mantiene corriendo a `jack_callback`. Cuando llamemos CTRL+C se romperá el bucle pero no saldremos del programa. Esto será útil después cuando necesitemos usar la optimización máxima del compilador porque necesitaremos cerrar el servidor de JACK y *después* liberar la memoria que hayamos reservado.

`jack_in_to_out.c:28-31` Cerramos el cliente de JACK y nos salimos del programa.

El tipo `jack_default_audio_sample_t` existe para tomar el tipo de muestra de audio por defecto del sistema sin necesidad de declararlo explícitamente. Generalmente se toma como un `float`. Aún así, habrá situaciones en las que necesitamos convertir esto a `double` para evitar errores de precisión.[1]

Si definimos una variable global `volumen` de tipo `float` y cambiamos la línea `11` por `out[i][j] = volumen*in[i][j];` obtenemos una aplicación de JACK que funciona como un control de volumen.

[1]Un ejemplo de esto es la sonda de respuestas de impulso que se puede encontrar en el apéndice.

B.2. Matrices de Mezcla

Si tenemos n entradas y m salidas; podemos mezclar las entradas para generar un número de salidas apropiado usando una multiplicación de matrices. Sea $\sigma(k) = (\sigma_1(k), \sigma_2(k), ..., \sigma_n(k))$ el vector que contiene a la muestra k-ésima de las señales que tenemos. Si M es una matriz de $m \times n$; obviamente $M\sigma(k)$ será un vector con m muestras.

En la práctica, M debe cumplir ciertas condiciones. Consideremos que si las entradas de la matriz fueran arbitrariamente grandes, podríamos destruir nuestro equipo o peor, nuestros oídos. Para encontrar una condición sencilla; representamos al i-ésimo renglón de M como c_i, la k-ésima muestra de la i-ésima salida $o_i(k)$ tendrá la forma:

$$\sum_{j=0}^{n} c_{i,j}\sigma_j(k) = o_i(k)$$

Como $\sigma_j(k) \leq 1$, una condición suficiente para controlar el volumen de salida es que $\sum_{j=0}^{n} c_{i,j} \leq G$, donde G es el factor de amplificación[2] máximo que queremos admitir. En particular, podemos tomar $c_{i,j} \leq G/n$ para cualesquiera índices i, j.

Si tuviéramos un sistema con n fuentes y m bocinas y quisiéramos crear la ilusión de las fuentes se están desplazando a diferentes bocinas; simplemente usamos la matriz de mezcla:

$$M(t) = \begin{bmatrix} t & 1-t & 0 & 0 & ... & 0 & 0 \\ 0 & t & 1-t & 0 & ... & 0 & 0 \\ 0 & 0 & t & 1-t & ... & 0 & 0 \\ ... & ... & ... & ... & ... & ... & ... \\ 0 & 0 & ... & ... & ... & t & 1-t \\ 1-t & 0 & ... & ... & ... & 0 & t \\ t & 1-t & ... & ... & ... & 0 & 0 \\ 0 & t & ... & ... & ... & 0 & 0 \end{bmatrix}$$

Que podemos definir como:

$$M(t) = \begin{cases} t & \text{si } \ i \bmod m = j \bmod n \\ 1-t & \text{si } \ i \bmod m = (j-1) \bmod n \\ 0 & \text{en otro caso.} \end{cases}$$

Si la cantidad de fuentes es a lo más el número de bocinas; vemos que iterar la mezcla tiene el efecto de *relevar* micrófonos adyacentes.

[2]También conocido como *ganancia*. Simplemente es la razón entre las amplitudes máximas de dos señales. Se puede expresar en decibeles.

Un ejemplo no-trivial de una matriz de mezcla es la matriz de rotación de dimensión n. Si $n = 2$, esta matriz simplemente mezcla dos señales y las manda a dos bocinas para crear la ilusión de que las fuentes no vienen directamente de las bocinas. Para $n > 2$, si fijamos $n - 2$ ejes podemos obtener una mezcla de dos fuentes que es análoga al caso $n - 2$. Aplicaciones sucesivas de matrices de mezcla con diferentes ejes fijos creará la ilusión de fuentes intermedias. A continuación vemos un ejemplo sencillo con dos señales:

simple_mixer.c

```
#include "autojackio.h"
#include <math.h>
double **mixing_matrix;
double theta;

int jack_callback (jack_nframes_t nframes, void *arg){
  for (int i = 0; i < num_channels; i++){
    in[i] =  (jack_default_audio_sample_t*) jack_port_get_buffer ( input_port[i],
    nframes);
    out[i] = (jack_default_audio_sample_t*) jack_port_get_buffer (output_port[i],
    nframes);
  }
  for (int j = 0; j < nframes; j++){
    out[0][j] = in[0][j]*mixing_matrix[0][0] + in[1][j]*mixing_matrix[0][1];
    out[1][j] = in[0][j]*mixing_matrix[1][0] + in[1][j]*mixing_matrix[1][1];
  }
  return 0;
}

int main (int argc, char *argv[]) {
  const char *client_name = "stereo_mixer";
  num_channels = 2;
  working = 1;

  if (argc != 2) {
    printf("Syntax is:\n");
    printf("\t ./simple_mixer.c [ANGLE]\n");
    exit(1);
  }

  theta = atof(argv[1]);
  if (theta < 0 || theta > 90){
    printf("Only angles between 0 and 90 are supported.\n");
    exit(1);
  }
  theta = (2*M_PI/360) * theta;
  printf("Angle: %.3f\n", theta);

  mixing_matrix = (double**) malloc(num_channels * sizeof(double*));
  for (int i = 0; i < num_channels; i++){
```

```
41        mixing_matrix[i] = (double*) malloc(num_channels * sizeof(double));
42      }
43      mixing_matrix[0][0] =  cos(theta); mixing_matrix[0][1] = sin(theta);
44      mixing_matrix[1][0] = -sin(theta); mixing_matrix[1][1] = cos(theta);
45
46      initialize_client(client_name, num_channels);
47      connect_ports();
48      signal(SIGINT, INThandler);
49
50      do{usleep (10);} while (working == 1);
51
52      jack_client_close (client);
53      printf("\nGoodbye!\n");
54      exit (0);
55    }
```

Ahora tenemos un paso extra de pre-procesamiento. Veamos cómo funciona el código nuevo:

`simple_mixer.c`:3-4 Declaramos un arreglo como un apuntador doble para que podamos crearlo dinámicamente a partir del número de canales. Declaramos al ángulo de rotación como una variable global.

`simple_mixer.c`:11-14 Aquí multiplicamos a las señales por la matriz de mezcla.

`simple_mixer.c`:24-28 Le pedimos al usuario el ángulo de rotación y cerramos el programa si no se encuentra dentro de los argumentos.

`simple_mixer.c`:30-36 Usamos `atof(argv[1])` para convertir el argumento a un tipo de punto flotante. Revisamos que no tenga algún valor absurdo o redundante y lo convertimos a radianes. Sacamos a la terminal el valor para confirmar que haya sido calculado con éxito.

`simple_mixer.c`:38 Declaramos a la matriz como un doble apuntador que contiene dos apuntadores que contendrán variables de tipo `double`. Efectivamente aquí declaramos el número de renglones que tendrá nuestra matriz.

`simple_mixer.c`:39-41 Podemos iterar sobre el número de renglones para ahora definir el número de columnas.

`simple_mixer.c`:41-42 Definimos explícitamente la matriz de mezcla.

¡Podemos incluso actualizar en tiempo real el ángulo de rotación para simular una rotación de las bocinas!

roto_mixer.c

```c
#include "autojackio.h"
#include <math.h>

double theta, step, c, s;

int jack_callback (jack_nframes_t nframes, void *arg){
  for (int i = 0; i < num_channels; i++){
    in[i] =  (jack_default_audio_sample_t*) jack_port_get_buffer ( input_port[i],
    nframes);
    out[i] = (jack_default_audio_sample_t*) jack_port_get_buffer (output_port[i],
    nframes);
  }
  for (int j = 0; j < nframes; j++){
    c = cos(theta);
    s = sin(theta);
    out[0][j] =  in[0][j]*c + in[1][j]*s;
    out[1][j] = -in[0][j]*s + in[1][j]*c;
    theta += step;
    if (theta > 2*M_PI){
            theta -= 2*M_PI;
    }

  }
  return 0;
}

int main (int argc, char *argv[]) {
  const char *client_name = "roto_mixer";
  num_channels = 2;
  working = 1;
  theta = 0;

  if (argc != 2) {
    printf("Syntax is:\n");
    printf("\t ./simple_mixer.c [STEP]\n");
    exit(1);
  }

```

```
39    step = atof(argv[1]);
40    if (step < -0.5 || step > 0.5){
41      printf("Step too large!.\n");
42      exit(1);
43    }
44    initialize_client(client_name, num_channels);
45    connect_ports();
46    signal(SIGINT, INThandler);
47
48    do{usleep (10);} while (working == 1);
49
50    jack_client_close (client);
51    printf("\nGoodbye!\n");
52    exit (0);
53  }
```

Tenemos que hacer algunos cambios mínimos:

`roto_mixer.c:4` Declaramos a una variable `step` que será sumada a `theta` cada ciclo de audio.

`roto_mixer.c:7-8` Calculamos coseno y seno una sola vez durante cada ciclo de audio. Podemos optimizar esto aún más creando una tabla de valores para evitar hacer cálculos innecesarios.

`roto_mixer.c:17-20` Actualizamos el valor del ángulo sumándole el valor de `step`. Si `theta` excede a 2π; le restamos exactamente 2π para que `theta` no crezca indefinidamente.

Un pequeño ejercicio que podemos hacer a partir de este ejemplo es crear una tabla de valores de seno y coseno en la etapa de pre-procesamiento para evitar hacer cálculos adicionales durante el procesamiento.

B.3. Retrasos

La capacidad de retrasar una señal es esencial para casi todo el procesamiento posterior que haremos. Hay un número de maneras de abordar este problema. La más sencilla es crear un buffer circular muy sencillo. La idea de un buffer circular es que tendremos un par de punteros; uno que lee y otro que escribe. Supongamos que nuestro buffer circular tiene N muestras. Si escribimos primero y luego movemos ambos punteros, el puntero de lectura leerá el principio del buffer *exactamente después de N muestras*. Esto nos permite hacer retrasos con una precisión de $1/f_s$ segundos. Si además movemos el puntero de lectura a una velocidad diferente al puntero de escritura, podemos obtener un cambio en la velocidad de reproducción del sonido. Esto nos permite demostrar cómo se puede hacer un trasponedor de tono muy sencillo. Desfortunadamente, al leer más rápido (o lento) tendremos saltos en el buffer que se podrán oir como *pops* en las bocinas. Escuchar estos sonidos por un periodo prolongado y/o a un volumen alto no es recomendable porque contienen una cantidad muy alta de energía y pueden dañar bocinas u oídos.

simple_delay.c

```
#include "autojackio.h"
#include <math.h>
double **buffer;
int     *read_index, *write_index;
int      delay_samples, read_skip, write_skip;

int jack_callback (jack_nframes_t nframes, void *arg){
  int i, j;
  for (i = 0; i < num_channels; i++){
    in[i] =  (jack_default_audio_sample_t*) jack_port_get_buffer ( input_port[i],
    nframes);
    out[i] = (jack_default_audio_sample_t*) jack_port_get_buffer (output_port[i],
    nframes);

    for (j = 0; j < nframes; j++){
      out[i][j]                  = buffer[i][read_index[i]];
      buffer[i][write_index[i]] = in[i][j];

      read_index[i]  = (read_skip  + read_index[i] + delay_samples)%delay_samples;
      write_index[i] = (write_skip + write_index[i]+ delay_samples)%delay_samples;

    }
  }
```

```
24    return 0;
25  }
26
27
28  int main (int argc, char *argv[]) {
29    const char *client_name = "simple_delay";
30    num_channels = 2;
31    working      = 1;
32
33    if (argc != 4) {
34      printf("Syntax is:\n");
35      printf("\t ./simple_delay.c [MILISECONDS] [READ_SKIP] [WRITE_SKIP]\n");
36      exit(1);
37    }
38
39    float time = atof(argv[1]);
40    read_skip  = atoi(argv[2]);
41    write_skip = atoi(argv[3]);
42
43    if (time < 0){
44      printf("C'mon, this isn't a time travel machine!\n.\n");
45      exit(1);
46    } else if (time > 1){
47      printf("C'mon, I don't have all day!.\n");
48      exit(1);
49    } else if (write_skip < -10 ||
50               read_skip < -10  ||
51               write_skip > 10  ||
52               read_skip > 10) {
53      printf("Invalid read-write skips: %i-%i\n", read_skip, write_skip);
54      exit(1);
55    }
56
57    initialize_client(client_name, num_channels);
58
59    // dynamic buffer allocation, setting indexes to zero:
60    delay_samples = (1000*time)*SAMPLE_RATE/1000;
61    buffer       = (double**) malloc(num_channels*sizeof(double*));
62    read_index   = (int*)     calloc(num_channels,sizeof(int));
63    write_index = (int*)     calloc(num_channels,sizeof(int));
64    for (int i = 0; i < num_channels; i++){
```

```
65      buffer[i]       = (double*) calloc(delay_samples,
66                                         sizeof(double));
67    }
68    /* automated client setup
69    * signal waits for CTRL+C to fire up a
70    * shutdown script that can free a number
71    * of variables for extreme optimization
72    */
73    connect_ports();
74    signal(SIGINT, INThandler);
75
76    /* keep running until stopped by the user
77    * usleep works in microseconds. 10 seems
78    * to be a good choice.
79    */
80    do{usleep (10);} while (working == 1);
81
82    jack_client_close (client);
83    printf("\nGoodbye!\n");
84    exit (0);
85  }
```

El código de preprocesamiento incluye un cierto manejo de punteros que es útil para crear arreglos de forma dinámica. Examinemos las partes nuevas del código:

`simple_delay.c`:14 Leemos del buffer.

`simple_delay.c`:15 Escribimos al buffer.

`simple_delay.c`:17-18 Movemos los punteros de lectura y escritura.

`simple_delay.c`:55 `C` redondea las operaciones de punto flotante cuando las asignamos a un entero. Como `time` es un número de punto flotante que puede ser menor a 1, necesitamos multiplicarlo por 1000 antes de multiplicar por la frecuencia de muestreo para evitar que se redondee a cero. Esto también fija la precisión del delay en milisegundos.

`simple_delay.c`:56-61 Inicializamos los buffers que usaremos para los retrasos.

Al leer primero y escribir después, estamos causando que el puntero de lectura siempre esté atrás del puntero de escritura. Los índices se juntarán solamente cuando el índice de lectura sobrepase el tamaño de `delay_samples`. Podemos obtener algunos efectos de transposición de tono cambiando el cociente entre `read_skip` y

`read_skip`. Si leemos el doble de rápido de lo que escribimos, transpondremos el sonido original una octava arriba; si la velocidad de escritura es el doble de rápido que la de lectura, transpondremos el sonido original una octava abajo. Las relaciones $n : m$ nos darán transposiciones por intervalos pitagóricos ($3 : 2$ es una quinta, $2 : 3$ una cuarta &c.). Este efecto de transposición se incluye para propósitos ilustrativos pero no se recomienda su uso en situaciones reales. Desfasar los punteros de lectura/escritura nos dará un corte abrupto de la señal y por consiguiente una serie de *pops* y *clics* que no son placenteros a los oídos ni seguros para las bocinas.

Un retraso a nivel muestras tiene un límite de $1/f_s$ segundos. Para aplicaciones de filtrado esto no presenta un problema (ya que podemos volver a calcular los coeficientes para diferentes Δt), pero es deseable tener algoritmos que no dependan de la frecuencia de muestreo. Una solución para el caso del retraso es usar la transformada de Fourier.

B.4. Reverberador de Schroeder-Logan.

Este es el código para implementar la reverberación artificial que vimos en el capítulo 2.

simple_delay.c

```
1
2  #include "autojackio.h"
3  #include <math.h>
4  double ***i_buffer;
5  double ***o_buffer;
6  int     **read_index, **write_index;
7  int      *delay_samples;
8  double    g, g2;
9
10 int stages = 5;
11 float delay_times[] = {0.100, 0.068, 0.060, 0.0197, 0.00585};
12 float loop_gains[]  = {0.7  , -0.7, 0.7 , 0.7   , 0.7   };
13
14
15 int jack_callback (jack_nframes_t nframes, void *arg){
16   int i, j, k;
17     for (i = 0; i < num_channels; i++){
18       in[i] =  (jack_default_audio_sample_t*) jack_port_get_buffer (input_port[i],
19                                                                     nframes);
20       out[i] = (jack_default_audio_sample_t*) jack_port_get_buffer (output_port[i],
21                                                                     nframes);
22
23       // All pass stages:
24       for (k = 0; k < stages; k++){
25         for (j = 0; j < nframes; j++){
26           g  = loop_gains[k];
27           g2 = 1-g*g;
28
29           o_buffer[k][i][read_index[k][i]] = i_buffer[k][i][read_index[k][i]];
30           i_buffer[k][i][read_index[k][i]] = in[i][j] + g*o_buffer[k][i][read_index[k][i]];
31
32           out[i][j] = -g*in[i][j] + g2*o_buffer[k][i][read_index[k][i]];
33           in[i][j] = out[i][j];
34
35           read_index[k][i]  = (1  + read_index[k][i])%delay_samples[k];
36       }
```

```
37      }
38    }
39    return 0;
40  }
41
42
43  int main (int argc, char *argv[]) {
44          const char *client_name = "schroeder_logan_reverb";
45          num_channels = 2;
46          working      = 1;
47          if (argc != 2) {
48                  printf("Syntax is:\n");
49                  printf("\t ./schroeder_logan_reverb.c [TIME_MULTIPLIER]\n");
50                  exit(1);
51          }
52
53
54          float time_multiplier = atof(argv[1]);
55
56
57          if (time_multiplier < 0){
58                  printf("C'mon, this isn't a time travel machine!\n.\n");
59                  exit(1);
60          } else if (time_multiplier > 10){
61                  printf("C'mon, I don't have all day!.\n");
62                  exit(1);
63          }
64
65          initialize_client(client_name, num_channels);
66
67          // dynamic buffer allocation, setting indexes to zero:
68          int time_to_samples = (time_multiplier * 1000*SAMPLE_RATE)/1000;
69          read_index  = (int**) malloc(stages*sizeof(int*));
70          delay_samples = (int*) malloc(stages * sizeof(int));
71          i_buffer      = (double***)malloc(stages * sizeof(double**));
72          o_buffer      = (double***)malloc(stages * sizeof(double**));
73          for (int i= 0; i < stages; i++){
74                  read_index[i] = (int*) calloc(num_channels, sizeof(int));
75                  delay_samples[i] = time_to_samples * delay_times[i];
76                  i_buffer[i] = (double**) malloc(num_channels*sizeof(double*));
77                  o_buffer[i] = (double**) malloc(num_channels*sizeof(double*));
```

```
78                  for (int j = 0; j < num_channels; j++){
79                          i_buffer[i][j] = (double*) calloc(delay_samples[i],
80                                                  sizeof(double));
81                          o_buffer[i][j] = (double*) calloc(delay_samples[i],
82                                                  sizeof(double));
83                  }
84          }
85
86
87          /* automated client setup
88          * signal waits for CTRL+C to fire up a
89          * shutdown script that can free a number
90          * of variables for extreme optimization
91          */
92          connect_ports();
93          signal(SIGINT, INThandler);
94
95          /* keep running until stopped by the user
96          * usleep works in microseconds. 10 seems
97          * to be a good choice.
98          */
99          do{usleep (10);} while (working == 1);
100
101         jack_client_close (client);
102         printf("\nGoodbye!\n");
103         exit (0);
104 }
```

schroeder_logan_reverb.c:11-12 Definimos a los coeficientes del reverberador. Usamos exactamente los mismos que en Schroeder y Logan, 1961.

schroeder_logan_reverb.c:22 El bucle `for` se usa para iterar sobre las etapas de los *all-pass*.

schroeder_logan_reverb.c:23-33 Ejecutamos el retraso exactamente como se expuso anteriormente y añadimos una sección de mezcla para obtener el *all-pass*.

schroeder_logan_reverb.c:66-78 Inicializamos los buffers para cada etapa individual. Esto es una extensión del código del retraso; con la diferencia de que usamos tres punteros para hacer un arreglo 3-dimensional que corresponde

a los canales, las etapas de filtros y las muestras individuales que contendrá el buffer.

Apéndice C

Código de C/JACK

C.1. Makefile Básico

Para compilar un código de C que use la librería `autojackio.h`; necesitamos tener los siguientes tres archivos:

`autojackio.h` Librería de conexión automática de JACK.

`mi_codigo.c` El código de procesamiento que queremos compilar.

`Makefile` Un archivo que nos permite compilar `micodigo.c` y pasar opciones personalizadas al compilador. Puede llevar cualquier otro nombre siempre y cuando tenga la extensión `.mak`.

En mi caso particular, necesito incluir las librerías de JACK que proporciona PipeWire. Excluyendo éste detalle, esto debería funcionar simplemente copiando y pegando en cualquier sistema basado en UNIX:

Makefile

```
1  CC       = gcc
2  // Omitir -L en un sistema con JACK nativo.
3  CFLAGS   = -L/usr/lib64/pipewire-0.3/jack  -Wall
4  LDLIBS   = -ljack -lm
5  CXXFLAGS = -ljack -lm -lstdc++
6  TARGET   = mi_codigo
7
8  OBJFILE   = $(patsubst %, %.o, $(TARGET))
9
10 all: $(TARGET)
11
12 $(TARGET): $(OBJFILE)
```

```
13 $(CC) $(CFLAGS) -o $(TARGET) $(OBJFILE) $(LDLIBS)
14 clean:
15 rm -f *.o *~
```

C.2. Conexiones Automáticas con `autojackio.h`.

El siguiente archivo de cabecera .h contiene a todas las variables globales y definiciones de funciones que usaremos al implementar los algoritmos. Incluir una definición de una función en un archivo de cabecera no es una "*buena práctica*" pero funciona.

autojackio.h

```
1  #ifndef AUTOJACKIO_H_INCLUDED
2  #define AUTOJACKIO_H_INCLUDED
3
4  #include <stdio.h>
5  #include <unistd.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <signal.h>
9  #include <jack/jack.h>
10
11 jack_default_audio_sample_t **in, **out;
12 jack_port_t                 **input_port, **output_port;
13 jack_client_t *client;
14
15 int num_channels, working, SAMPLE_RATE;
16
17 void initialize_client(const char *client_name, const int channels);
18 void connect_ports();
19 void jack_shutdown (void *arg);
20 int jack_callback (jack_nframes_t nframes, void *arg);
21
22 void initialize_client(const char *client_name, const int channels){
23   jack_options_t options = JackNoStartServer;
24   jack_status_t status;
25   client = jack_client_open (client_name, options, &status);
26
27   if (client == NULL){
28     /* if connection failed, say why */
29     printf ("jack_client_open() failed, status = 0x%2.0x\n", status);
```

```
30      if (status & JackServerFailed) {
31        printf ("Unable to connect to JACK server.\n");
32      }
33      exit (1);
34    }
35
36    /* if connection was successful, check if the name we proposed is not in use */
37    if (status & JackNameNotUnique){
38      client_name = jack_get_client_name(client);
39      printf ("Warning: other agent with our name is running,");
40      printf("`%s' has been assigned to us.\n", client_name);
41    }
42
43    /* tell the JACK server to call 'jack_callback()'
44    whenever there is work to be done. */
45    in  = malloc(num_channels * sizeof(jack_default_audio_sample_t*));
46    out = malloc(num_channels * sizeof(jack_default_audio_sample_t*));
47    jack_set_process_callback (client, jack_callback, 0);
48
49    /* tell the JACK server to call 'jack_shutdown()' if it ever shuts down,
50    either entirely, or if it just decides to stop calling us. */
51    jack_on_shutdown (client, jack_shutdown, 0);
52
53    /* display the current sample rate. */
54    SAMPLE_RATE = jack_get_sample_rate(client);
55    printf ("Engine sample rate: %d hz\n", SAMPLE_RATE);
56
57    printf("Initializing I/O channels... ");
58
59    input_port  = malloc(num_channels * sizeof(jack_port_t*));
60    output_port = malloc(num_channels * sizeof(jack_port_t*));
61    for (int i = 0; i < channels; i++){
62      char  input_name[20];
63      char output_name[20];
64      sprintf( input_name,  "input_%d", i+1);
65      sprintf(output_name, "output_%d", i+1);
66
67      input_port[i]  = jack_port_register (client,
68                                           input_name,
69                                           JACK_DEFAULT_AUDIO_TYPE,
70                                           JackPortIsInput,
```

```
71                                          0);
72      output_port[i] = jack_port_register (client,
73                                           output_name,
74                                           JACK_DEFAULT_AUDIO_TYPE,
75                                           JackPortIsOutput,
76                                           0);
77
78      /* check that both ports were created succesfully */
79      if ( (input_port[i] == NULL) || (output_port[i] == NULL) ) {
80        printf( "\nCould not create agent ports.");
81        printf("Have we reached the maximum amount of JACK agent ports?\n");
82        exit (1);
83      }
84    }
85    printf("Done.\n");
86  }
87
88  void connect_ports(){
89
90    /* Tell the JACK server that we are ready to roll.
91    Our jack_callback() callback will start running now. */
92    if (jack_activate (client)) {
93      printf ("Cannot activate client.");
94      exit (1);
95    }
96
97    printf ("Agent activated.\n");
98    printf ("Connecting ports... ");
99
100   /* Assign our input port to a server output port*/
101   const char **serverports_names;
102   serverports_names = jack_get_ports (client,
103                                       NULL,
104                                       NULL,
105                                       JackPortIsPhysical|JackPortIsOutput);
106
107   if (serverports_names == NULL) {
108     printf("No available physical capture (server output) ports.\n");
109     exit (1);
110   }
111   // Connect the first available to our input port
```

```
112   for (int i = 0; i < num_channels; i++){
113     if (jack_connect (client,
114                       serverports_names[i],
115                       jack_port_name (input_port[i]) )) {
116       printf("Cannot connect input port %d.\n", i+1);
117       exit (1);
118     }
119   }
120   // free ports_names variable for reuse in next part of the code
121   free (serverports_names);
122   serverports_names = jack_get_ports (client,
123                                       NULL,
124                                       NULL,
125                                       JackPortIsPhysical|JackPortIsInput);
126   if (serverports_names == NULL) {
127     printf("No available physical playback (server input) ports.\n");
128     exit (1);
129   }
130   // Connect the first available to our output port
131   for (int i = 0; i < num_channels; i++){
132     if (jack_connect (client,
133                       jack_port_name (output_port[i]),
134                       serverports_names[i] )) {
135       printf("Cannot connect output port %d.\n", i+1);
136       exit (1);
137     }
138   }
139   // free serverports_names variable, we're not going to use it again
140   free (serverports_names);
141
142
143   printf ("done.\n");
144   /* keep running until stopped by the user */
145 }
146
147 void  INThandler(int sig)
148 {
149   signal(sig, SIG_IGN);
150   working = 0;
151 }
152
```

```
153 void jack_shutdown (void *arg){
154   exit (1);
155 }
156
157 #endif  /* AUTOJACKIO_H_INCLUDED */
```

C.3. UPOLS

```
1  /**
2  * Convolución particionada en tiempo real
3  */
4
5  #include <stdio.h>
6  #include <unistd.h>
7  #include <stdlib.h>
8  #include <string.h>
9  #include <math.h>
10 #include <signal.h>
11
12 #include <jack/jack.h>
13
14 // Include FFTW header
15 //needs to be included before fftw3.h for compatibility
16 #include <complex.h>
17 #include <fftw3.h>
18
19 //To read audio files
20 #include <sndfile.h>
21 SNDFILE * audio_file;
22 SF_INFO audio_info;
23 unsigned int channels = 1;
24
25
26 double complex *i_fft ,
27                *ir_fft,
28                *o_fft ;
29
30 double  *i_time,
31         *ir_time,
32         *o_time;
```

```
33
34 double complex **fdl, **fir, *tmp;
35
36 jack_default_audio_sample_t *buffer;
37
38 float *ir, *conv;
39 double vol;
40 int ir_len, two_nframes;
41 fftw_plan i_forward, ir_forward, o_inverse;
42
43 int partitions;
44 int working = 1;
45
46 jack_port_t *input_port;
47 jack_port_t *output_port;
48 jack_client_t *client;
49
50 float sample_rate;
51
52 /*
53 * The process callback for this JACK application is called in a
54 * special realtime thread once for each audio cycle.
55 */
56
57 /* Para permutar circularmente el arreglo de frecuencias */
58 void swap_array(double complex *matrix_1, double complex *matrix_2){
59     tmp = matrix_1;
60     matrix_1 = matrix_2;
61     matrix_2 = tmp;
62 }
63
64 /* Ésta función sirve como amortiguador entre presionar CTRL+C
65 *  y salir del programa. Necesitamos hacer esto para liberar la
66 *  memoria manualmente antes de cerrar el programa si queremos
67 *  usar la máxima optimización del compilador. Podemos verificar
68 *  que funciona usando valgrind
69 */
70 void  INThandler(int sig)
71 {
72     signal(sig, SIG_IGN);
```

```
73      working = 0;
74  }
75
76  int jack_callback (jack_nframes_t nframes, void *arg){
77    jack_default_audio_sample_t *in, *out;
78    int i, j, k;
79
80    in  = (jack_default_audio_sample_t *) jack_port_get_buffer (input_port,
81                                                                nframes);
82    out = (jack_default_audio_sample_t *) jack_port_get_buffer (output_port,
83                                                                nframes);
84
85    for (i = 0; i < nframes; i++){
86      // entran nframes y se colocan
87      // en la parte derecha del buffer
88      buffer[nframes + i] = in[i];
89      i_time[i] = buffer[i];
90      i_time[nframes+i] = buffer[nframes+i];
91    }
92
93    // aplicamos fftw3:
94    fftw_execute(i_forward);
95
96    // movemos la línea de retraso de frecuencias:
97    for (k = 0; k < partitions; k++){
98      swap_array(fdl[k], fdl[k-1]);
99    }
100
101   // escribimos ifft al primer renglón de la fdl:
102   for (i = 0; i <= nframes; i++){
103     fdl[0][i] = i_fft[i];
104     o_fft[i] = 0.0 + I*0.0;
105   }
106
107   // Bloque de procesamiento.
108   for (i = 0; i < two_nframes; i++){
109     for (k = 0; k < partitions; k++){
110       o_fft[i] += fdl[k][i] * fir[k][i];
111     }
112   }
```

```
113
114   // Regresando al dominio del tiempo.
115   fftw_execute(o_inverse);
116   for (i = 0; i < nframes; i++){
117     out[i] = vol*creal(o_time[nframes+i])/two_nframes;
118     buffer[i] = in[i];
119   }
120
121   return 0;
122 }
123
124
125 /**
126 * JACK calls this shutdown_callback if the server ever shuts down or
127 * decides to disconnect the client.
128 * quizás podríamos meter la liberación de memoria aquí.
129 */
130 void jack_shutdown (void *arg){
131   exit (1);
132 }
133
134
135 int main (int argc, char *argv[]) {
136   const char *client_name = "rt_convolver";
137   jack_options_t options = JackNoStartServer;
138   jack_status_t status;
139
140   if (argc != 3) {
141     printf("Syntax is:\n");
142     printf("\t ./rt_convolution.c [IMPULSE_RESPONSE_FILE] [VOLUME]\n");
143     exit(1);
144   }
145
146   // file stuff
147   char audio_file_path[100];
148   strcpy(audio_file_path, argv[1]);
149   printf("trying to open impulse response file: %s\n", audio_file_path);
150   audio_file = sf_open(audio_file_path, SFM_READ, &audio_info);
151
152   // IR volume
```

```
153   vol = atof(argv[2]);
154
155   if (audio_file == NULL){
156     printf("%s\n", sf_strerror(NULL));
157     exit(1);
158   } else {
159     ir_len = audio_info.frames;
160     printf("IR file info:\n");
161     printf("\tSample Rate: %d\n", audio_info.samplerate);
162     printf("\tChannels: %d\n", audio_info.channels);
163     printf("\tFrames: %d\n", ir_len);
164     sf_command (audio_file, SFC_SET_NORM_FLOAT, NULL, SF_TRUE) ;
165   }
166
167   /* open a client connection to the JACK server */
168   client = jack_client_open (client_name, options, &status);
169
170   if (client == NULL){
171     /* if connection failed, say why */
172     printf ("jack_client_open() failed, status = 0x%2.0x\n", status);
173     if (status & JackServerFailed) {
174       printf ("Unable to connect to JACK server.\n");
175     }
176     exit (1);
177   }
178   /* if connection was successful, check if
179   * the name we proposed is not in use
180   */
181   if (status & JackNameNotUnique){
182     client_name = jack_get_client_name(client);
183     printf ("Warning: other agent with our name is running,");
184     printf ("%s' has been assigned to us.\n", client_name);
185   }
186
187   /* tell the JACK server to call 'jack_callback()'
188   * whenever there is work to be done.
189   */
190   jack_set_process_callback (client, jack_callback, 0);
191
192
```

```
193   /* tell the JACK server to call 'jack_shutdown()'
194   * if it ever shuts down,
195 either entirely, or if
196   * it just decides to stop calling us.
197   */
198   jack_on_shutdown (client, jack_shutdown, 0);
199
200
201   /* display the window size. */
202   //printf ("Sample rate: %d\n", jack_get_sample_rate (client));
203   printf ("Window size: %d\n", jack_get_buffer_size (client));
204   sample_rate = (float)jack_get_sample_rate(client);
205   int nframes = jack_get_buffer_size (client);
206   two_nframes = 2*nframes;
207
208   if (ir_len < nframes){
209     printf("Impulse responses with less than %d ", nframes);
210     printf("samples not supported (ir is %d samples long)\n", ir_len);
211   exit(1);
212   }
213
214   /* la respuesta de impulso no necesita tener un número
215   * de muestras que sea una potencia de dos; sin embargo,
216   * el número de muestras tiene queser un múltiplo de nframes
217   * (que sí es una potencia de 2), puesto que queremos usar
218   * la aceleración de fftw3
219   */
220
221   partitions = ceil(ir_len/nframes) + 1;
222   int   ir_aux_len  = partitions * nframes;
223   float ir_time_len = ir_len/sample_rate;
224   printf("padded ir len: %d (%.f ms)\n", ir_aux_len, 1000*ir_time_len);
225
226   // preparando la respuesta de impulso.
227
228   ir = (float*) calloc(ir_aux_len, sizeof(float));
229   int read_count = sf_read_float(audio_file, ir, ir_len);
230   if (ir_len != read_count){
231     printf("An error occurred while writing the IR buffer.\n");
232     exit(1);
```

```
233   }
234   sf_close(audio_file);
235   ir_len = ir_aux_len;
236
237   // calculamos la norma L2 de la respuesta de impulso acumulando
238   // cuadrados
239
240   float ir_L2_norm  = 0.0;
241   float ir_sup_norm = 0.0;
242   for (int i = 0; i < ir_len; i++){
243     // L2 norm:
244     ir_L2_norm += pow(ir[i], 2);
245
246     // supremum norm:
247     if (ir[i] > ir_sup_norm){
248       ir_sup_norm = ir[i];
249     }
250   }
251   printf("max ir val:    %.6f\n", ir_sup_norm);
252
253   // sacamos la raíz de la suma de cuadrados:
254
255   ir_L2_norm = sqrt(ir_L2_norm);
256   printf("max L2 ir val: %.6f\n", ir_L2_norm );
257
258   // normalizamos:
259
260   float ir_norm = ir_L2_norm + 0.01;
261
262   for (int i = 0; i < ir_len; i++){
263     ir[i] = 0.5*ir[i]/ir_norm;
264   }
265
266   // declaramos la fdl y las particiones del filtro:
267   fdl = malloc(partitions * sizeof(double complex*));
268   fir = malloc(partitions * sizeof(double complex*));
269   for (int i = 0; i < partitions; i++){
270     fdl[i] = (double complex *)
271            fftw_malloc(sizeof(double complex) * (nframes + 1) );
272     fir[i] = (double complex *)
```

```
273             fftw_malloc(sizeof(double complex) * (nframes + 1) );
274     }
275
276     //preparing FFTW3 buffers
277     //conv = (float*) calloc(2*nframes, sizeof(float));
278     buffer = calloc(2*nframes, sizeof(jack_default_audio_sample_t));
279
280     i_fft   = (double complex *)
281               fftw_malloc(sizeof(double complex) * (nframes + 1));
282     i_time  = (double         *)
283                    malloc(sizeof(double        ) * 2*nframes    );
284
285     ir_fft  = (double complex *)
286               fftw_malloc(sizeof(double complex) * (nframes + 1));
287     ir_time = (double         *)
288                    malloc(sizeof(double        ) * 2*nframes    );
289
290     o_fft   = (double complex *)
291               fftw_malloc(sizeof(double complex) * (nframes + 1));
292     o_time  = (double         *)
293                    malloc(sizeof(double        ) * 2*nframes    );
294
295     i_forward = fftw_plan_dft_r2c_1d(2*nframes,
296                                      i_time,
297                                      i_fft,
298                                      FFTW_MEASURE);
299     o_inverse = fftw_plan_dft_c2r_1d(2*nframes,
300                                      o_fft,
301                                      o_time,
302                                      FFTW_MEASURE);
303     ir_forward= fftw_plan_dft_r2c_1d(2*nframes,
304                                      ir_time,
305                                      ir_fft,
306                                      FFTW_MEASURE);
307
308     // aquí escribimos las particiones y la fdl
309     printf("number of partitions: %d\n", partitions);
310     int errors = 0;
311     for (int k = 0; k < partitions; k++){
312       for (int i = 0; i < nframes; i++){
```

```
313         // creamos particiones acolchonadas con zero
314         ir_time[i]           = ir[k*nframes + i];
315         ir_time[nframes + i] = 0.0;
316     }
317
318     fftw_execute(ir_forward);
319     for (int i = 0; i <= nframes; i++){
320     // se crean las particiones del filtro
321       fir[k][i] = ir_fft[i];
322
323       // se inicializa la fdl a cero
324       fdl[k][i] = 0.0 + I*0.0;
325       }
326     }
327
328     if (client == NULL){
329       /* if connection failed, say why */
330       printf ("jack_client_open() failed, status = 0x%2.0x\n", status);
331       if (status & JackServerFailed) {
332         printf ("Unable to connect to JACK server.\n");
333       }
334       exit (1);
335     }
336     /* create the agent input port */
337     input_port = jack_port_register (client,
338                                      "input",
339                                      JACK_DEFAULT_AUDIO_TYPE,
340                                      JackPortIsInput,
341                                      0);
342
343     /* create the agent output port */
344     output_port = jack_port_register (client,
345                                       "output1",
346                                       JACK_DEFAULT_AUDIO_TYPE,
347                                       JackPortIsOutput,
348                                       0);
349
350     /* check that both ports were created succesfully */
351     if ((input_port == NULL) || (output_port == NULL)) {
352     printf("Could not create agent ports. ");
```

```
353   printf("Have we reached the maximum amount of JACK agent ports?\n");
354     exit (1);
355   }
356
357   /* Tell the JACK server that we are ready to roll.
358   Our jack_callback() callback will start running now. */
359   if (jack_activate (client)) {
360     printf ("Cannot activate client.");
361     exit (1);
362   }
363
364   printf ("Agent activated.\n");
365
366   /* Connect the ports.  You can't do this before the client is
367    * activated, because we can't make connections to clients
368    * that aren't running.  Note the confusing (but necessary)
369    * orientation of the driver backend ports: playback ports are
370    * "input" to the backend, and capture ports are "output" from
371    * it.
372    */
373
374   printf ("Connecting ports... ");
375
376   /* Assign our input port to a server output port*/
377   // Find possible output server port names
378   const char **serverports_names;
379   serverports_names = jack_get_ports (client,
380                                       NULL,
381                                       NULL,
382                                       JackPortIsPhysical|JackPortIsOutput);
383   if (serverports_names == NULL) {
384     printf("No available physical capture (server output) ports.\n");
385     exit (1);
386   }
387   // Connect the first available to our input port
388   if (jack_connect (client,
389                     serverports_names[0],
390                     jack_port_name (input_port))) {
391     printf("Cannot connect input port.\n");
392     exit (1);
```

```
393   }
394
395   // free serverports_names variable for reuse in next part of the code
396   free (serverports_names);
397
398
399   /* Assign our output port to a server input port*/
400   // Find possible input server port names
401   serverports_names = jack_get_ports (client,
402                                       NULL,
403                                       NULL,
404                                       JackPortIsPhysical|JackPortIsInput);
405   if (serverports_names == NULL) {
406     printf("No available physical playback (server input) ports.\n");
407     exit (1);
408   }
409   // Connect the first available to our output port
410   // free serverports_names variable, we're not going to use it again
411   if (jack_connect (client,
412                     jack_port_name (output_port),
413                     serverports_names[0])) {
414     printf ("Cannot connect output ports.\n");
415     exit (1);
416   }
417   free (serverports_names);
418
419   printf ("done.\n");
420   signal(SIGINT, INThandler);
421
422   /* keep running until stopped by the user */
423   do{usleep (10);} while (working == 1);
424
425   jack_client_close (client);
426   printf("\nGoodbye!\n");
427
428   free(fdl);    free(fir);
429   free(tmp);    free(buffer);
430   free(ir);
431
432   exit (0);
```

```
433 }
```

C.4. Sonda de Respuestas de Impulso

Esta sonda de respuestas de impulso está inspirada en el trabajo de Angelo Farina. En Farina, 2000, propuso un método para medir respuestas de impulso que minimiza la distorsión armónica. Dicho método consiste de excitar el sistema usando una señal sinusoidal cuya frecuencia es una función logarítmica del tiempo. El siguiente código usa JACK para conectarse a las bocinas del sistema, reproducir ésta señal y grabar simultáneamente la respuesta.

```
/**
* A simple example of how to do FFT with FFTW3 and JACK.
*/
#include <unistd.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#include <jack/jack.h>

// to read/write audio files:
#include <sndfile.h>

// Include FFTW header and complex.h for compatibility
#include <complex.h>
#include <fftw3.h>

//sndfile stuff
SNDFILE * audio_file;
SF_INFO audio_info;
unsigned int audio_position = 0;

double complex  *i_fft, *i_time,
*t_fft, *t_time,
*ir_fft, *ir_time;

fftw_plan t_forward, i_forward,
ir_inverse;

double *t_wave, *i_wave, *ir, *i_b, *t_b;
```

```
33
34  int buffer_index = 0;
35  int working = 1;
36  int wave_index = 0;
37  int ir_index = 0;
38  double freq, sample_rate;
39  double sample_rate;
40  int freq_index, wave_index, wave_samples,
41      wave_secs, start_freq, end_freq,
42      write_count;
43
44  jack_port_t *input_port;
45  jack_port_t *output_port;
46  jack_client_t *client;
47
48
49  /**
50  * The process callback for this JACK application is called in a
51  * special realtime thread once for each audio cycle.
52  */
53
54  int jack_callback (jack_nframes_t nframes, void *arg){
55    jack_default_audio_sample_t *in, *out;
56    int i, j;
57
58    in  = (jack_default_audio_sample_t *)jack_port_get_buffer (input_port,
59                                                               nframes);
60    out = (jack_default_audio_sample_t *)jack_port_get_buffer (output_port,
61                                                               nframes);
62
63    for (i = 0; i < nframes; i++){
64      out[i] = t_wave[wave_index];
65      i_b[wave_index] = in[i];
66      t_b[wave_index] = t_wave[wave_index];
67      wave_index++;
68    }
69
70    if (wave_index >= wave_samples){
71      working = 0;
72    }
73
```

```
74    return 0;
75  }
76
77
78  /**
79  * JACK calls this shutdown_callback if the server ever shuts down or
80  * decides to disconnect the client.
81  */
82  void jack_shutdown (void *arg){
83    exit (1);
84  }
85
86
87  int main (int argc, char *argv[]) {
88    char audio_file_path[100];
89    const char *client_name = "ir_probe";
90    jack_options_t options = JackNoStartServer;
91    jack_status_t status;
92
93    if (argc != 5) {
94      printf("Syntax:\n");
95      printf("\tir_probe [DURATION (SECONDS)] [START FREQ] ");
96      printf("[END FREQ] [TARGET FILENAME]\n\n");
97      exit(1);
98    } else if (argc > 5) {
99      printf("too many arguments.\n");
100     exit(1);
101   } else {
102     wave_secs = atof(argv[1]);
103     start_freq = atof(argv[2]);
104     end_freq = atof(argv[3]);
105     strcpy(audio_file_path, argv[4]);
106   }
107
108   if (end_freq < 2*start_freq){
109     printf("end frequency must be at least twice the start frequency.\n");
110     exit(1);
111   }
112
113   /* open a client connection to the JACK server */
114   client = jack_client_open (client_name, options, &status);
```

```
115
116   if (client == NULL){
117     /* if connection failed, say why */
118     printf ("jack_client_open() failed, status = 0x%2.0x\n", status);
119     if (status & JackServerFailed) {
120       printf ("Unable to connect to JACK server.\n");
121     }
122     exit (1);
123   }
124   /* if connection was successful, check if the
125    * name we proposed is not in use */
126   if (status & JackNameNotUnique){
127 client_name = jack_get_client_name(client);
128     printf ("Warning: other agent with our name is running, ");
129     printf("`%s' has been assigned to us.\n", client_name);
130   }
131
132   /* tell the JACK server to call 'jack_callback()'
133    * whenever there is work to be done. */
134   jack_set_process_callback (client, jack_callback, 0);
135
136
137   /* tell the JACK server to call 'jack_shutdown()' if it ever shuts down,
138      either entirely, or if it just decides to stop calling us. */
139   jack_on_shutdown (client, jack_shutdown, 0);
140
141
142   /* display the current sample rate. */
143   printf ("Sample rate: %d\n", jack_get_sample_rate (client));
144   printf ("Window size: %d\n", jack_get_buffer_size (client));
145   sample_rate = (double)jack_get_sample_rate(client);
146   int nframes = jack_get_buffer_size (client);
147
148   // write info:
149
150   printf("writing audio file with path: %s\n", audio_file_path);
151   audio_info.samplerate = sample_rate;
152   audio_info.channels = 1;
153   audio_info.format = SF_FORMAT_WAV | SF_FORMAT_PCM_32 | SF_FORMAT_FLOAT;
154
155   audio_file = sf_open(audio_file_path, SFM_WRITE, &audio_info);
```

```
156   if (audio_file == NULL){
157     printf("%s\n", sf_strerror(NULL));
158     exit(1);
159   }
160
161
162   if (client == NULL){
163     /* if connection failed, say why */
164     printf ("jack_client_open() failed, status = 0x%2.0x\n", status);
165     if (status & JackServerFailed) {
166       printf ("Unable to connect to JACK server.\n");
167     }
168   exit (1);
169   }
170   /* create the agent input port */
171   input_port = jack_port_register (client,
172                                    "input",
173                                    JACK_DEFAULT_AUDIO_TYPE,
174                                    JackPortIsInput,
175                                    0);
176
177   /* create the agent output port */
178   output_port = jack_port_register (client,
179                                     "output",
180                                     JACK_DEFAULT_AUDIO_TYPE,
181                                     JackPortIsOutput,
182                                     0);
183
184   /* check that both ports were created succesfully */
185   if ((input_port == NULL) || (output_port == NULL)) {
186     printf("Could not create agent ports. Have we reached");
187     printf(" the maximum amount of JACK agent ports?\n");
188       exit (1);
189   }
190
191   // initialize test wave data:
192   wave_samples = wave_secs*sample_rate;
193   for (int n = 0; n < 22; n++){
194     if (pow(2, n) > wave_samples){
195       wave_samples = pow(2, n);
196       printf("rounding samples to %dth power of two. (%d samples, %f secs)\n",
```

```
197                 n, wave_samples, wave_samples/sample_rate);
198         break;
199       }
200     }
201     wave_secs = (double) wave_samples/sample_rate;
202     // allocate memory for wave measurement, initialize wave indexes.
203
204     wave_index = 0;
205     ir     = (double*) malloc(wave_samples* sizeof(double));
206     t_wave = (double*) malloc(wave_samples* sizeof(double));
207     i_b    = (double*) calloc(wave_samples, sizeof(double));
208     t_b    = (double*) calloc(wave_samples, sizeof(double));
209
210     // calculate samples for fader length:
211     int fade_samples = (int) 100*sample_rate/1000;
212
213     // create test wave:
214     double x, t;
215     double omega = 2*M_PI*start_freq;
216     double alpha = log(end_freq/start_freq);
217     double a = (omega * wave_secs)/alpha;
218     double b = alpha/wave_secs;
219
220     for (int i = 0; i < wave_samples; i++){
221       x = (double) i / sample_rate;
222       t_wave[i] = 0.5*sin(a*(exp(b*x) - 1));
223     }
224
225     // apply window function to wave:
226
227     for (int i = 0; i < fade_samples; i++){
228       t = (double) sin(M_PI*i/(2*fade_samples)) ;
229       t_wave[i] = pow(t,2)*t_wave[i];
230       t_wave[wave_samples-i-1] = pow(t, 2)*t_wave[wave_samples-i-1];
231     }
232
233     /* Tell the JACK server that we are ready to roll.
234        Our jack_callback() callback will start running now. */
235
236     if (jack_activate (client)) {
237       printf ("Cannot activate client.");
```

```
238     exit (1);
239   }
240
241
242
243   printf ("Agent activated.\n");
244
245   /* Connect the ports.  You can't do this before the client is
246    * activated, because we can't make connections to clients
247    * that aren't running.  Note the confusing (but necessary)
248    * orientation of the driver backend ports: playback ports are
249    * "input" to the backend, and capture ports are "output" from
250    * it.
251    */
252   printf ("Connecting ports... ");
253
254   /* Assign our input port to a server output port*/
255   // Find possible output server port names
256
257   const char **serverports_names;
258   serverports_names = jack_get_ports (client,
259                                       NULL,
260                                       NULL,
261                                       JackPortIsPhysical|JackPortIsOutput);
262   if (serverports_names == NULL) {
263     printf("No available physical capture (server output) ports.\n");
264     exit (1);
265   }
266   // Connect the first available to our input port
267   if (jack_connect (client,
268                     serverports_names[0],
269                     jack_port_name (input_port))) {
270     printf("Cannot connect input port.\n");
271     exit (1);
272   }
273   // free serverports_names variable for reuse in next part of the code
274   free (serverports_names);
275
276
277   /* Assign our output port to a server input port*/
278   // Find possible input server port names
```

```
279   serverports_names = jack_get_ports (client,
280                                       NULL,
281                                       NULL,
282                                       JackPortIsPhysical|JackPortIsInput);
283   if (serverports_names == NULL) {
284     printf("No available physical playback (server input) ports.\n");
285     exit (1);
286   }
287   // Connect the first available to our output port
288   // free serverports_names variable, we're not going to use it again
289   if (jack_connect (client,
290                     jack_port_name (output_port),
291                     serverports_names[0])) {
292     printf ("Cannot connect output ports.\n");
293     exit (1);
294   }
295   free (serverports_names);
296
297
298   printf ("done.\n");
299   int sleep_time = (1000*nframes)/sample_rate;
300   /* keep running until stopped by the user */
301   while(working == 1 ){
302     usleep (sleep_time);
303   }
304   printf("end of test signal.\n");
305
306   jack_client_close(client);
307   printf("calculating IR...\n");
308   /* the fft part goes here because we need to perform heavy duty calculations
309    * that we wouldn't be able to complete while running a jack client */
310
311   // these buffers are for the generated test wave
312   t_fft = (double complex *) fftw_malloc(sizeof(double complex) * wave_samples);
313   t_time = (double complex *) fftw_malloc(sizeof(double complex) * wave_samples);
314
315   // these are for the measured wave
316   i_fft = (double complex *) fftw_malloc(sizeof(double complex) * wave_samples);
317   i_time = (double complex *) fftw_malloc(sizeof(double complex) * wave_samples);
318
319   // these are for the impulse response output
```

```
320   ir_fft = (double complex *) fftw_malloc(sizeof(double complex) * wave_samples);
321   ir_time = (double complex *) fftw_malloc(sizeof(double complex) * wave_samples);
322
323   // these plans are for transforming the measured
324   // and test waves to the freq domain:
325   t_forward = fftw_plan_dft_1d(wave_samples,
326                                t_time, t_fft,
327                                FFTW_FORWARD,
328                                FFTW_MEASURE);
329   i_forward = fftw_plan_dft_1d(wave_samples,
330                                i_time,
331                                i_fft,
332                                FFTW_FORWARD,
333                                FFTW_MEASURE);
334
335   // this single plan is for writing the IR
336   ir_inverse = fftw_plan_dft_1d(wave_samples,
337                                 ir_fft,
338                                 ir_time,
339                                 FFTW_BACKWARD,
340                                 FFTW_MEASURE);
341
342   /* now that the client is closed we can calculate the IR at leisure.
343    * The i_time and t_time arrays
344 are full by this point and we can
345    * just call fft_execute */
346
347   for(int i = 0; i < wave_samples; i++){
348     i_time[i] = i_b[i];
349     t_time[i] = t_b[i];
350   }
351
352   // apply tukey window function to measurement
353   for (int i = 0; i < fade_samples; i++){
354     t = (double) sin(M_PI*i/(2*fade_samples)) ;
355     i_time[i] = pow(t,2)*i_time[i];
356     i_time[wave_samples-i-1] = pow(t, 2)*i_time[wave_samples-i-1];
357   }
358   fftw_execute(i_forward);
359   fftw_execute(t_forward);
360
```

```
361   /* now, we divide in the frequency domain
362    * to obtain the frequency response: */
363   for (int i = 0; i < wave_samples; i++){
364     ir_fft[i] = i_fft[i]/t_fft[i];
365   }
366
367   /* and we execute the inverse fft to get the actual IR */
368
369   fftw_execute(ir_inverse);
370
371   /* save only first 500ms of IR, set up L2 norm */
372   int ir_length = (int) 500*sample_rate/1000;
373   double ir_norm = 0;
374   fade_samples = 1000;
375   for (int i = 0; i < ir_length; i++){
376     ir[i] = creal(ir_time[i])/wave_samples;
377     ir_norm += pow(ir[i], 2);
378   }
379   ir_norm = sqrt(ir_norm);
380
381   // L2 normalize
382   for (int i = 0; i < ir_length; i++){
383     ir[i] = ir[i]/ir_norm;
384   }
385
386   // apply tukey window (optional)
387   /*
388   for (int i = 0; i < fade_samples; i++){
389     t = (double) sin(M_PI*i/(2*fade_samples)) ;
390     ir[i] = pow(t,2)*ir[i];
391     ir[ir_length-i-1] = pow(t, 2)*ir[ir_length-i-1];
392   }*/
393
394   /* write this to a file */
395   write_count = sf_writef_double(audio_file, ir, ir_length);
396   printf("writing %d samples to file.\n", write_count);
397
398   if (write_count != ir_length){
399     printf("\nEncountered I/O error. Exiting.\n");
400     sf_close(audio_file);
401     exit(1);
```

```
402   }
403   /* deallocate fftw3 buffers, close file*/
404   fftw_cleanup();
405   sf_close(audio_file);
406
407   printf("Success!\n");
408   exit (0);
409 }
```

Bibliografía

Farina, Angelo (2000). «Simultaneous measurement of impulse response and distortion with a swept-sine technique». En: *Audio engineering society convention 108*. Audio Engineering Society.

Garcia, Guillermo (2002). «Optimal filter partition for efficient convolution with short input/output delay». En: *Audio Engineering Society Convention 113*. Audio Engineering Society.

Gardner, William G (1994). «Efficient convolution without input/output delay». En: *Audio Engineering Society Convention 97*. Audio Engineering Society.

Helgason, Sigurdur (1968). *Lie groups and symmetric spaces. Battelle Rencontres: 1967 Lectures in Mathematics and Physics, 1–71*.

Hsu, Hwei P, Raj Mehra, Federico Velasco Coba y col. (1987). *Análisis de Fourier*.

Körner, Thomas William (1989). *Fourier analysis*. Cambridge university press.

Meijering, E. (2002). «A chronology of interpolation: from ancient astronomy to modern signal and image processing». En: *Proceedings of the IEEE* 90.3, págs. 319-342. DOI: 10.1109/5.993400.

Orfanidis, Sophocles J (1995). *Introduction to signal processing*. Prentice-Hall, Inc.

Proakis, John y Dimitri Manolakis (2006). *Digital signal processing: Principles, Algorithms, and Applications*. Pearson Prentice Hall, Pearson Education, Inc.

Rascón Estebané, Caleb (s.f.). *Notas del Curso de Procesamiento Digital de Audio*. URL: http://calebrascon.info/PDA/.

Scherz, Paul y Simon Monk (2016). *Practical electronics for inventors*. McGraw-Hill Education.

Schroeder, Manfred R y Benjamin F Logan (1961). «"Colorless" Artificial Reverberation». En: *IRE Transactions on Audio* 6, págs. 209-214.

Shilov, Georgi (1974). *Elementary Functional Analysis*. Dover.

— (1977). *Linear Algebra*. Dover.

Sobolev, Sergei Lvovich (1964). *Partial Differential Equations of Mathematical Physics*. Dover.

Wefers, Frank (2015). *Partitioned convolution algorithms for real-time auralization*. Vol. 20. Logos Verlag Berlin GmbH.