Benemérita Universidad Autónoma de Puebla

Facultad de Ciencias de la Electrónica

Licenciatura en Ciencias de la Electrónica

IMPLEMENTACIÓN Y EVALUACIÓN DEL ALGORITMO SIGMA DE LANCZOS EN DISPOSITIVOS PROGRAMABLES PARA LA ATENUACIÓN DEL FENÓMENO DE GIBBS

TESIS

Que para obtener el título de:
Licenciado en Electrónica
PRESENTA:

C. Juan José Meza Gutiérrez

Directores de Tesis: **Dra. María Monserrat Morín Castillo (FCE-BUAP)**
Dr. José Rubén Conde Sánchez (FCFM-BUAP)
Dr. José Jacobo Oliveros Oliveros (FCFM-BUAP)

Agosto 2022, Puebla, Pue. Méx.

Resumen

Una de las derivaciones a las que ha dado lugar el análisis de Fourier en sus diferentes vertientes de estudio, es el llamado fenómeno de Gibbs, que se caracteriza por la presencia de oscilaciones de hasta el 9 % de la amplitud del salto, al aproximar funciones con discontinuidades de salto por medio de series de Fourier. La presente tesis desarrolla la implementación del algoritmo Factor σ-Lanczos para la atenuación del fenómeno de Gibbs a través de dos metodologías de diseño propuestas en un SoC (System on Chip) Zynq-7000 contenido en la tarjeta de desarrollo PYNQ Z2. Un SoC tiene la doble ventaja de integrar, por un lado, un procesador tipo ARM programable por software; y por otro lado, un FPGA programable por Hardware empleando diferentes herramientas; características que los hacen adecuados para diseños que involucran la aceleración de algoritmos. Con esta idea, se presenta una primera arquitectura que es implementada completamente en la parte lógica del SoC utilizando VHDL y uso de IPs (Intelectual Properties); la segunda arquitectura implementada es desarrollada bajo la metodología de flujo de diseño de High Level Synthesis (HLS) utilizando tanto la programación por software como la programación de la lógica programable. Ambas arquitecturas permiten la síntesis de de hasta 16 componentes del desarrollo en series de Fourier de funciones periódicas.

Las arquitecturas se utilizaron para calcular las series de Fourier y su corrección por medio del σ-Factor de las funciones de onda cuadrada, diente de sierra y la llamada delta de Dirac. Se hizo un análisis de los resultados obtenidos en cada arquitectura, y se corrobora el incremento en la velocidad de convergencia de las series de Fourier en los casos que se utiliza el σ-Factor, frente a los casos en que no se utiliza.

Índice general

Agradecimientos

Al Dr. José Rubén Conde Sánchez, por los conocimientos, la confianza y el tiempo compartidos comnigo para el desarrollo de este proyecto.

A los doctores María Moserrat Morin Castillo y Jacobo Oliveros Oliveros por orientarme y proporcionarme los recursos necesarios para el desarrollo del mismo.

A los profesores de la Facultad de Ciencias de la Electrónica por sus contribuciones a mi formación personal y profesional.

A mis padres, hermano y amigos que me han acompañado durante todo este tiempo.

Dedicatoria

A Carlos G. Morales, porque me enseñaste que en las adversidades, nunca hay que dejar de intentar.

Índice de figuras

Índice de tablas

Introducción

Las series trigonométricas de Fourier y el análisis de Fourier son metodologías desarrolladas inicialmente para resolver problemas como el de la ecuación de onda y el de la ecuación de calor, y ha sido la base para el desarrollo del análisis armónico. Este procedimiento consiste en representar funciones muy generales, incluso con muchas discontinuidades por medio de series trigonométricas de la forma:

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left[a_n cos(nx) + b_n sen(nx) \right], \tag{1}$$

donde a_n y b_n con n = 0,1,2,..., son constantes, y se calculan usando las expresiones:

$$a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) cos(nx) dx, \tag{2}$$

$$b_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) sen(nx) dx. \tag{3}$$

Esto representa una ventaja significativa frente a otros métodos, entre los que se puede mencionar el método de series de potencias, que únicamente pueden representar funciones con derivadas de cualquier orden [9].

Las aplicaciones de este método son diversas, en el área de electrónica, por ejemplo: es una herramienta eficaz para determinar la respuesta de un sistema lineal a una señal de entrada periódica arbitraria; también, son una forma de representar el comportamiento de voltajes y corrientes alternos en sistemas eléctricos; inclusive en un sistema con características no lineales de corriente-voltaje, los armónicos inducidos al sistema por la fuente de alimentación pueden ser representados mediante series de Fourier [1]; esto da muestra de su importancia en diferentes áreas de la física e ingeniería.

Una de las áreas de estudio más interesantes de las series de Fourier es conocida como el Fenómeno de Gibbs [9], que se caracteriza por una mala aproximación a funciones definidas a trozos, específicamente en sus puntos de discontinuidad. Esta mala aproximación se ve reflejada en un sobreimpulso que tiende a un valor alrededor del 9 % de la amplitud del salto de la función, conforme el número de coeficientes utilizados en la serie tiende a infinito [12]; esto, en algunas aplicaciones es causa de serios problemas, como ejemplos: en el escaneo de imágenes por resonancia magnética (MRI[1]) el Fenómeno de Gibbs provoca la presencia de objetos inexistentes, en el caso de circuitos eléctricos, en un conmutador este fenómeno puede ocasionar saltos de

[1] Magnetic Resonance Imaging

voltaje que sobrepasen sus parámetros de funcionamiento, causando un daño permanente en el dispositivo [8].
Bajo el enfoque anterior, se pueden encontrar herramientas matemáticas para dar solución al Fenómeno de Gibbs, algunas de ellas se basan en el uso de filtros, las cuales permiten mejorar la exactitud de la aproximación lejos del punto de discontinuidad, otras opciones son el procedimiento de Gegenbauer, que consiste en re-expandir las series de Fourier con polinomios ultraesféricos para obtener exactitud exponencial en cualquier intervalo de analiticidad [11] y el método de sumación de Féjer, que saca el limite del promedio de las sumas parciales de Fourier, y ha demostrado dar buenos resultados en la eliminación del mencionado fenómeno [12].

El objetivo de esta tesis consiste en: *Implementar el algoritmo del factor σ-de Lanczos en dispositivos programables basados en el SoC*[2] *Zynq 7020.*
Para alcanzar este objetivo se utilizarán dos metodologías de diseño que se describen más adelante.

[2]System on a Chip

Objetivos

Los objetivos general y específicos de este trabajo de tesis se enuncian a continuación.

Objetivo General

Implementar en el SoC Zynq 7020 mediante dos metodologías de diseño distintas el algoritmo del factor σ-Lanczos para atenuar el fenómeno de Gibbs.

Objetivos Particulares

1. Implementar el algoritmo del factor σ-Lanczos a partir de una arquitectura desarrollada en VHDL.
2. Implementar el algoritmo del factor σ-Lanczos a través de una IP desarrollada mediante el flujo de diseño de HLS.
3. Comparar la cantidad de recursos de cómputo utilizados por cada una de las implementaciones.
4. Validar de manera experimental mediante la síntesis de diferentes funciones el aumento de la rapidez de convergencia de series de Fourier que proporciona el factor σ-Lanczos.

Organización de la Tesis

El presente documento está conformado por 5 capítulos, con el objetivo de resaltar la importancia del fenómeno de Gibbs y sus implicaciones en el análisis de Fourier, el *capítulo 1, "Antecedentes"* contiene una recapitulación histórica de los eventos que dieron pauta al desarrollo de las series de Fourier, el descubrimiento del fenómeno de Gibbs y las propuestas que se han planteado para solucionarlo.
El *capítulo 2*, titulado *Series de Fourier y Fenómeno de Gibbs*, otorga al lector los fundamentos matemáticos y conceptos teóricos que permiten el entendimiento de las series de Fourier, incluye una descripción general de la función Sinc, y se profundiza en la definición y descripción del fenómeno de Gibbs en dos funciones ampliamente utilizadas en sistemas electrónicos (la función de onda cuadrada y diente de sierra), así como un caso especial, la serie de deltas de Dirac, y se concluye con la definición y descripción del Factor σ-Lanczos para la atenuación de dicho fenómeno.
El *Capítulo 3*, contiene los conceptos clave acerca de los dispositivos programables reconfigurables en los que se realizó la implementación de las arquitecturas desarrolladas para este proyecto. Se describen los elementos que se encuentran dentro de una FPGA, además de que se da una breve descripción de la Familia de SoC Zynq del Fabricante Xilinx cuyo objetivo es proveer al usuario de estos dispositivos de las capacidades de un microprocesador de grado de aplicación con los beneficios de desarrollar arquitecturas para la aceleración de algoritmos en hardware utilizando segmento de lógica programable.
En el *Capítulo 4*, se describen las dos metodologías de diseño utilizadas para desarrollar las arquitecturas que realizan el cálculo de las series de Fourier y el factor σ-Lanczos. Además, se describen las características de las arquitecturas desarrolladas y su implementación en la tarjeta de desarrollo PYNQ-Z2.
En el *capítulo 5*, se presentan y analizan los resultados obtenidos al implementar las series de Fourier y su corrección por el factor σ-Lanczos en las arquitecturas desarrolladas. Finalmente, se muestran las conclusiones más importantes de esta tesis.

Capítulo 1

Antecedentes

1.1. Estado del Arte (antecedentes)

Las series trigonométricas y el análisis de Fourier son metodologías cuyo origen está asociado a dos problemas fundamentales: la *ecuación de onda* y la *ecuación de calor*. El primero de ellos en su forma más elemental, describe las vibraciones de una cuerda fija en sus extremos, mientras que el segundo caso describe como se conduce el calor a lo largo de un sólido a través del tiempo.
De forma en que se relacionan estas dos áreas de la matemática surge de siguiente manera [9]: El hecho de representar una función f definida en el intervalo $[-\pi, \pi]$ como una serie trigonométrica de la forma

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty}[a_n cos(nx) + b_n sen(nx)], \tag{I}$$

donde a_n y b_n con $n = 0, 1, 2, ...$, son constantes.

Lleva a la pregunta natural: ¿Cuáles son las condiciones que debe cumplir la función f (con $x \in [-\pi, \pi]$), para que existan las sucesiones $\{a_n\}_0^\infty$ y $\{b_n\}_1^\infty$ como coeficientes y que puedan expresar la expansión del tipo (I)?, además de la incógnita ¿Cómo hallar $\{a_n\}_0^\infty$ y $\{b_n\}_1^\infty$?.

Estas cuestiones aparecieron a mediados del siglo XVIII asociadas a los estudios de Leonard Euler (1701-1783) y de Daniel Bernoulli (1700-1782) sobre el problema de la *cuerda vibrante*.

Bernoulli, llega a plantearse la solución del problema de la cuerda vibrante en forma de serie trigonométrica a partir de consideraciones de tipo físico, que le llevan a pensar que la cuerda oscila involucrando varias frecuencias al mismo tiempo, cuyas amplitudes respectivas dependen de la forma inicial de la vibración, es decir, del modo en que se haya empezado a mover la cuerda. Esta posibilidad descubierta por Bernoulli, es lo que hoy llamamos *principio de superposición* y ha resultado ser un principio de gran importancia en muchas ramas de la Física Matemática.

Sin embargo, Euler entiende que esta idea de Bernoulli lleva a un resultado aparentemente paradójico, al preguntarse si una función "arbitraria"podría ser expresada como en (I). Teniendo en cuenta, que para los matemáticos de la época, las curvas se dividían en dos clases: curvas *"continuas"* y curvas "*geométricas*". En contraste con la terminología adoptada hoy en día, una curva se decía *"contínua"* si sus ordenadas y sus abscisas podían conectarse mediante alguna fórmula $y = f(x)$. Por otra parte, una curva se denominaba "geométrica" si podía dibujarse de

Figura 1.1: Retrato de Jean-Baptiste Joseph Fourier realizado por el pintor y dibujante francés Louis Léopold Boilly. Fuente: WIKIMEDIA COMMONS

.

alguna forma con trazos continuos o discontinuos. Pensaban por tanto que la segunda categoría de curvas era más amplia que la primera, ya que lo que nosotros denominamos como una función continua a trozos, puede dibujarse, pero no puede expresarse si no es con varias fórmulas. Así, si una función "arbitraria" podría expresarse, por ejemplo, como una serie de *senos* (es decir, como (I), pero con $a_n = 0$ para $n = 0, 1, 2, ...$), esto significaría que cualquier curva "geométrica" sería también una curva "continua", lo cual, para Euler y sus contemporáneos, era simplemente increíble.

Por otro lado, para contribuir más aún con este debate, la solución al problema de la cuerda vibrante de Bernoulli, compite con otra aportada por Jean le Rond D'Alembert (o Juan Le Rond d'Alembert, 1717-1783) en forma de una onda que avanza y otra que retrocede, que se determinan a partir de la posición y la velocidad iniciales de la cuerda. En particular, D'Alembert consideraba que la manera más natural de hacer que una cuerda empiece a vibrar era desplazarla de su posición de equilibrio tirando de algún punto de ella. Esto hace que su posición inicial se pueda representar mediante dos rectas que forman un determinado ángulo. Para D'Alembert la naturaleza de esta curva hacía imposible pensar en que pudiese expresarse como una serie trigonométrica, ya que se trata, como se ha comentado más arriba, de una curva "geométrica", mientras que la serie trigonométrica sería una curva "continua".

Un método de solución al problema planteado por Euler reapareció en una memorable sesión de la Academia Francesa de las Ciencias, el día 21 de diciembre de 1807, donde el matemático francés J. Fourier (1768-1830) presentó un trabajo que posteriormente abriría un nuevo capítulo en la historia de las matemáticas: la creación del *Análisis de Armónico* o también conocido como el *Análisis de Fourier*.

Fourier dedujo una ecuación que describía la conducción de calor a través de los cuerpos sólidos llamada la *ecuación del calor*. Pero no sólo la había deducido, sino que también desarrolló un método para resolverla, el *método de separación de variables*, que en cierto modo había sido utilizado ya por Bernoulli para su solución, aunque es Fourier quién lo empezó a utilizar de manera sistemática en la resolución de Ecuaciones en Derivadas Parciales.

La aplicación de la técnica de separación de variables a la ecuación de calor, lo llevó a escribir

la solución en la forma de serie trigonométrica e incluso llegó a afirmar que cualquier función periódica, de periodo 2π, se puede poner como una serie de la forma (I). Y para ello incluso encontró las *fórmulas (de Fourier)* que permiten calcular los coeficientes de la serie.

Aunque la representación de una función en serie trigonométrica se había considerado antes de Fourier, nadie antes que él puso de manifiesto la correspondencia entre función y coeficientes.

Sin embargo, el trabajo de Fourier no fue aceptado en su primera exposición, máxime teniendo como parte del auditorio a matemáticos como J. L. Lagrange (1736-1813), P. S. Laplace (1749-1827) y A. M. Legendre (1752-1833), que criticaron abiertamente la falta de rigor del tratamiento de Fourier. De hecho, Fourier tuvo que rehacer su trabajo ya que su memoria no fue aceptada en un primer momento. No obstante, finalmente sus ideas fueron aceptadas y fueron expuestas, años después, en su obra de 1822, *"Théorie Analytique de la Chaleor"*.

Hay que añadir que el estudio de las series de Fourier contribuyó de manera decisiva a clarificar la idea de función hasta el concepto moderno de nuestros días. Todo este tratamiento posterior está asociado a nombres tales como P. G. L. Dirichlet (1805-1859), B. Riemann (1826-1886), G. Cantor (1845-1918) y H. Lebesgue (1875-1941).

Una de las derivaciones interesantes a las que ha dado lugar el *análisis de Fourier*, es el llamado ***Fenómeno de Gibbs*** [12], que surge a mediados del siglo XIX, debido a que en 1848, el matemático inglés Henry Wilbraham en la corrección de un trabajo de Fourier que trataba sobre la convergencia de las series, observó que, en puntos cercanos a una discontinuidad de una función f, las sumas parciales de Fourier excedían en aproximadamente el 9 % del valor de salto de la discontinuidad.

Este trabajo de Wilbraham pasó desapercibido, hasta que hacía 1898 reapareció en un contexto diferente.

En 1898, el físico Albert Michaelson y su colega S. Stratton construyeron un sintetizador armónico: un dispositivo que reconstruye una señal periódica $\tilde{f}_N(x)$ de período T_0 y hasta 80 de sus componentes armónicos.

Michelson probó su dispositivo calculando los coeficientes de Fourier de distintas señales periódicas $\tilde{f}(x)$ y comparando la señal reconstruida $\tilde{f}_N(x)$ con la original. En líneas generales, $\tilde{f}_N(x)$ resultaba muy similar a $\tilde{f}(x)$. Sin embargo, cuando se utilizó como señal de prueba una onda cuadrada, la aproximación no fue tan buena.

Según Lanczos (1966), Michaelson no podía comprender las causas del problema y pensaba que su aparato podría estar funcionando inadecuadamente. Confió sus dudas al matemático Josiah Gibbs, quien investigó el fenómeno y publicó sus resultados en 1899.

Otra versión de este suceso es la de Gottlieb y Shu (1997), que comentan que Michelson y Stratton (1898) publicaron un trabajo en Nature, donde describían la construcción y el funcionamiento del analizador armónico, junto con algunos gráficos cuyo propósito era demostrar que la máquina calculaba correctamente los coeficientes de la serie de Fourier. Una de las curvas era una onda cuadrada que exhibía las oscilaciones de Gibbs, pero los autores no comentaban nada al respecto en ese trabajo. Sin embargo, parece que Michelson había notado algo particular, pues al poco tiempo (6 de Octubre de 1898) envía una carta a la revista donde comenta la dificultad de construir la función $f(x) = x$ a partir sus coeficientes de Fourier. En particular, argüia que la expresión (I) evaluada en $x = T_0(1/2 + k/N)$, donde k es pequeño, converge a distintos límites para diferentes valores de k. También observó que el mismo fenómeno ocurría para la derivada de la función (una onda cuadrada).

En cartas posteriores, a la misma revista, Michelson enfatiza que desde su punto de vista,

la convergencia debía ser uniforme en cualquier entorno de la discontinuidad.

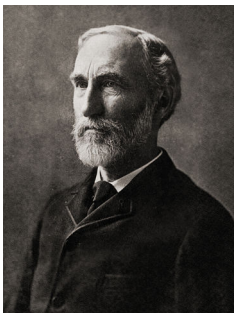


Figura 1.2: Josiah W. Gibbs (1839-1903). Fuente: The Scientific Papers of J. Willard Gibbs.

El siguiente personaje de interés en esta historia, Josiah Gibbs (1839-1903), responde las dudas de Michelson y describe las oscilaciones. Sin embargo, en su nota parecía implicar que la amplitud de estas oscilaciones decrecía con N.

El 27 de Abril de 1899, Gibbs publica el resultado correcto de su estudio, pidiendo disculpas, "I should like to correct a careless error" y mostrando que las oscilaciones no decaen, sino que el sobreimpulso tiende a un número constante.

El nombre "Fenómeno de Gibbs" fue utilizado por primera vez por Böcher en 1906, en un artículo donde extendía el resultado de Gibbs.

El primer intento por resolver el fenómeno de Gibbs ocurrió en 1904 , cuando el matemático húngaro L. Fejér (1880-1959), propuso una nueva forma de sumar los términos de la serie de Fourier. Su método consideraba sumar los promedios de las sumas parciales de la siguiente manera:

$$\theta_M(x) = \frac{1}{M+1}\sum_{N=0}^{M} s_N(x) = \frac{s_0(x) + s_1(x) + ... + s_N(x)}{M+1} \tag{1.1}$$

y pasar al límite.

$$s(x) = lim_{M\to\infty}\theta_M(x). \tag{1.2}$$

De esta manera, se obtiene una nueva sucesión de funciones que produce convergencia uniforme incluso en algunos casos en los que la serie original no la tiene. Este método es equivalente a filtrar la señal con un filtro de primer orden.

Otra forma de solución al fenómeno de Gibbs fue desarrollada por Cornelius Lanczos (1893-974), un reconocido físico matemático que tuvo un profundo impacto sobre los cimientos de la ciencia del siglo XX. Entre sus aportaciones destacadas se encuentra la *aproximación de Lanczos*, método para calcular numéricamente la función Gamma; y adicionalmente en el campo que nos compete propuso el llamado $\sigma - factor$ o también llamado *Factor Lanczos* para la atenuación del fenómeno de Gibbs. El desarrollo del Factor Lanczos y su implementación en dispositivos programables es la parte central de este documento, por lo tanto el $\sigma - Factor$ se explica a detalle en el apartado 2.4.2.

Capítulo 2

Series de Fourier y Fenómeno de Gibbs

2.1. Series de Fourier

El estudio de muchos problemas físicos que conducen a ecuaciones en derivadas parciales necesitan de series trigonométricas en la forma:

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left[a_n cos(nx) + b_n sen(nx)\right], \tag{2.1}$$

donde a_n y b_n con n = 0,1,2,..., son constantes, y se calculan usando las expresiones:

$$a_n = \frac{1}{\pi} \int_{-\pi}^{pi} f(x) cos(nx) dx, \tag{2.2}$$

$$b_n = \frac{1}{\pi} \int_{-\pi}^{pi} f(x) sen(nx) dx. \tag{2.3}$$

Esta representación en serie trigonométrica tiene como ventaja la capacidad de representar funciones muy generales, con muchas discontinuidades, frente a otros métodos como las series de potencias, que únicamente pueden representar funciones continuas con derivadas de cualquier orden.

Como se observa, si la serie 2.1 es uniformemente convergente, los coeficientes a_n y b_n se pueden obtener a partir de f, mediante las fórmulas 2.2 y 2.3. Sin embargo, necesitamos saber, si dada una función periódica esta admite un desarrollo mediante una serie trigonométrica que converge de manera uniforme, esto no se sabe, por eso es necesario definir ciertos números a_n y b_n que se usarán para construir la serie trigonométrica 2.1.

Cuando se usa este procedimiento, a los coeficientes a_n y b_n se les llama **coeficientes de Fourier** de la función f, y a la serie 2.1 se le llama **serie de Fourier** de f.

En otras palabras, una serie de Fourier es un tipo especial de serie trigonométrica, cuyos coeficientes se calculan aplicando 2.2 y 2.3 a cierta función f, y para formarla no es preciso suponer que f sea continua; basta con que las integrales 2.2 y 2.3 existan, y para ello es suficiente que f sea integrable sobre el intervalo $-\pi \leq x \leq \pi$.

El estudio de las series de Fourier en matemáticas ha contribuido de manera decisiva a clarificar

la idea de función, hasta formar el moderno concepto utilizado en nuestros días, y su tratamiento posterior a J. Fourier está asociado a nombres como P. G. L. Dirichlet (1805-1859), B. Riemann (1826-1866), G. Cantor (1845-1918) y H. Lebesgue (1875-1941) [12].

2.1.1. Funciones Pares, Impares y Sistemas Ortogonales

Debido a la utilidad de sus propiedades para calcular las series de Fourier de algunas funciones, en este apartado se enunciarán los conceptos de función par e impar.
Una función f: $\mathbb{R} \to \mathbb{R}$ se dice que es par, si para todo punto $x \in \mathbb{R}$ se cumple que

$$f(-x) = f(x). \tag{2.4}$$

Se dice que f es impar, si para todo punto x$\in \mathbb{R}$ se cumple

$$f(-x) = -f(x) \tag{2.5}$$

Así, $f(x) = x^2$ y $h(x) = cos(x)$ son funciones pares, mientras que $g(x) = x^3$ y $q(x) = sen(x)$ son impares, la gráfica de la función par es simétrica respecto al eje y la de una impar, es lo que se denomina antisimétrica.

Además, de las gráficas de funciones par e impar, se deduce que:

$$\int_{-a}^{a} f(x)dx = 2\int_{0}^{a} f(x)dx, \text{ si } f(x) \text{ es par,} \tag{2.6}$$

del mismo modo que

$$\int_{-a}^{a} f(x)dx = 0, \text{ si } f(x) \text{ es impar.} \tag{2.7}$$

Esto se debe a que las integrales representan las áreas (con signo) bajo las curvas, e incluso puede demostrarse mediante razonamientos analíticos basados en las expresiones 2.4 y 2.5.
Por otro lado, cuando se multiplican funciones pares e impares, su comportamiento se puede agrupar de la siguiente manera:

$$\begin{aligned} (par)(par) &= par, \\ (par)(impar) &= impar, \\ (impar)(impar) &= par. \end{aligned} \tag{2.8}$$

La demostración del segundo de estos enunciados se retoma de [9] en donde, de manera sencilla consideran lo siguiente:
Consideremos la función $F(x) = f(x)g(x)$, con f par y g impar. Entonces

$$F(-x) = f(-x)g(-x) = f(x)[-g(x)] = -f(x)g(x) = -F(x).$$

Lo que demuestra que el producto $f(x)g(x)$ es impar. Las otras expresiones se pueden demostrar de manera similar.
Utilizando este conjunto de propiedades, se puede deducir de manera rápida por ejemplo, que la función $x^3cos(x)$ es impar, puesto que $f(x) = x^3$ es impar, y $g(x) = cos(nx)$ es par, de tal modo que por (2.7) se puede inferir que

$$\int_{-\pi}^{\pi} x^3 cos(nx)dx = 0. \tag{2.9}$$

Sin necesidad de realizar la integración por partes.

2.2. Función Sinc

En esta sección definiremos a la *Función Sinc*, también llamada *seno cardinal*. La función sinc, se denota como *sinc(x)* y posee dos formas de definición. Su primera definición es la *sinc normalizada*, definida generalmente de la siguiente forma:

$$sinc_N(x) = \frac{sen(\pi x)}{\pi x}. \tag{2.10}$$

Esta definición está relacionada con el análisis de sistemas y señales en ingeniería, lo cual resulta importante en el procesamiento de señales digitales (señales unidimensionales, imágenes y vídeo) [9].
La segunda forma de definir la función *sinc*, se conoce como *sinc desnormalizada*, es utilizada en matemática (funciones de Bessel) y proyecciones cartográficas (Proyección de Winkel-Tripel) y se expresa de la siguiente manera:

$$sinc(x) = \frac{sen(x)}{x}. \tag{2.11}$$

En ambas definiciones, los valores de las funciones poseen una singularidad previsible en cero, lo cual, habitualmente se determina como una igualdad a 1. Una característica importante es que la *función sinc* es diferenciable en todo $\mathbb{R}$.

La definición formal de la función *sinc*, se da a continuación:
Definición

$$sinc(x) = \begin{cases} \frac{sen(x)}{x} & \text{si} \quad -\pi \leq x \leq \pi \ \wedge \ x \neq 0 \\ 1 & \text{si} \quad x = 0 \\ 0 & \text{en otro caso.} \end{cases} \tag{2.12}$$

Dos propiedades que complementan el concepto de la función *sinc*, son las siguientes [9]:

$$\int_{-\infty}^{+\infty} sinc_N(x)dx = 1, \tag{2.13}$$

$$\int_{-\infty}^{+\infty} sinc(x) = \pi. \tag{2.14}$$

2.3. Fenómeno de Gibbs

Dentro del campo del *análisis de Fourier*, una de las derivaciones más interesantes es el denominado *fenómeno de Gibbs* acerca del cuál se hizo un recorrido histórico en el apartado de antecedentes y se extiende en su análisis en esta sección a través de la función salto.

2.4. El fenómeno de Gibbs en la Función de Salto

Considérese la función salto como:

$$f(x) = \begin{cases} -1 & \text{si} \quad -\pi \leq x < 0 \\ 1 & \text{si} \quad 0 \leq x \leq \pi \end{cases} \tag{2.15}$$

La $N-ésima$ suma parcial correspondiente a su serie de Fourier esta dada por la expresión

$$s_N(x) = \frac{a_0}{2} + \sum_{n=1}^{N} \left[a_n cos(nx) + b_n sen(nx)\right], \tag{2.16}$$

donde los coeficientes de Fourier se calculan por medio de las expresiones 2.2 y 2.3
Dado que f es una función impar, $a_n = 0$ para todo n=0,1,2, ...
Por otro lado, b_n se puede calcular de forma explícita, obteniéndose como resultado:

$$b_n = \frac{2}{\pi}\left(\frac{1-(-1)^n}{n}\right),\ para\ n = 1, 2, ... \tag{2.17}$$

Por lo tanto, la suma parcial de Fourier para la función salto queda

$$s_N(x) = \sum_{n=1}^{N} \frac{2}{\pi}\left(\frac{1-(-1)^n}{n}\right) sen(nx). \tag{2.18}$$

Por otra parte, como b_n=0 si n es par, la suma parcial de Fourier puede reescribirse de la forma

$$\begin{aligned} s_{2n-1}(x) &= \frac{4}{\pi}\sum_{r=1}^{n} \frac{1}{2r-1} sen((2r-1)x) \\ &= \frac{2}{\pi}\left[sen(x) + \frac{sen(3x)}{x} + ... + \frac{sen((2n-1)x)}{2n-1}\right]. \end{aligned} \tag{2.19}$$

Se sabe que, si f y f' son continuas, salvo en un número finito de puntos de discontinuidad de tipo salto, las sumas parciales de Fourier convergen puntualmente a *f(x)* en los puntos de continuidad de f y a la media de los límites laterales en los puntos de discontinuidad. véase [10], cap. 5.
Este resultado se aplica al caso particular de la función salto y que presenta una singularidad en $x = 0$: una discontinuidad de tipo salto. En la Figura 2.1, se aprecia la forma en que cuando $x \neq 0$, la serie de Fourier aproxima el valor de la función en x, mientras que en $x = 0$ convergen a la media de los límites laterales, nula en este caso, puesto que $[f(0-)+f(0+)]/2 = (1-1)/2 = 0$. En este punto de discontinuidad $x = 0$ se aprecia con claridad el fenómeno de Gibbs. En efecto, se observa que la gráfica de la suma parcial de Fourier excede a la de la función salto en el punto de discontinuidad.

Por ejemplo, en la figura 2.2 a la izquierda del punto $x = 0$ se ve como la gráfica de las sumas parciales de Fourier sobresalen por debajo de la gráfica de $f(x)$, en las proximidades de (0,-1). De la misma forma, en la figura 2.3 se observa que la suma parcial de Fourier supera con nitidez a la de la función salto.

Retomando la figura 2.1 muestra que el fenómeno de Gibbs también se produce en los extremos de $x = \pm\pi$ del intervalo $(-\pi, \pi)$. Esto se debe a que la suma parcial de Fourier aproxima la extensión periódica de periodo 2π de la función salto, que presenta discontinuidades en los puntos de la forma $x = k\pi,\ k \in \mathbb{Z}$.

Adicionalmente, se sabe que para esta función en particular, el valor máximo de s_{2n-1} más cercano a $x = 0$ (por la derecha) es el punto $x = \frac{\pi}{2n}$ y que en este punto

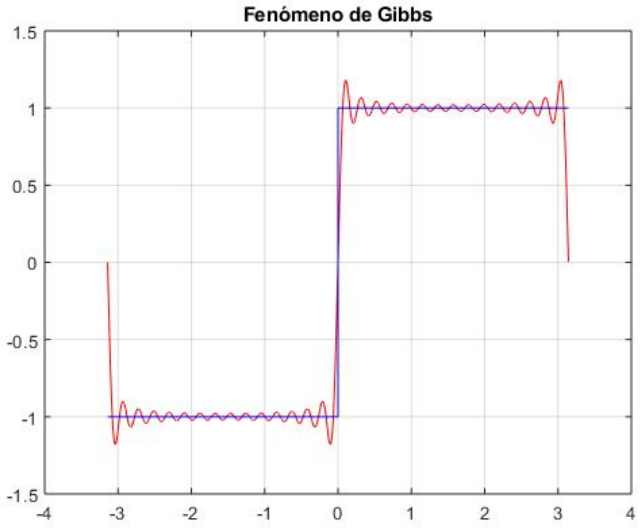


Figura 2.1: Suma parcial de Fourier $N = 30$.

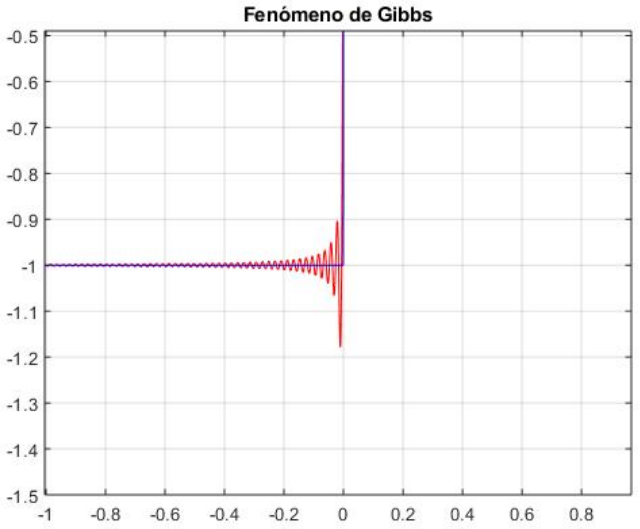


Figura 2.2: Suma parcial de Fourier $N = 300$, proximidades de $(0, -1)$.

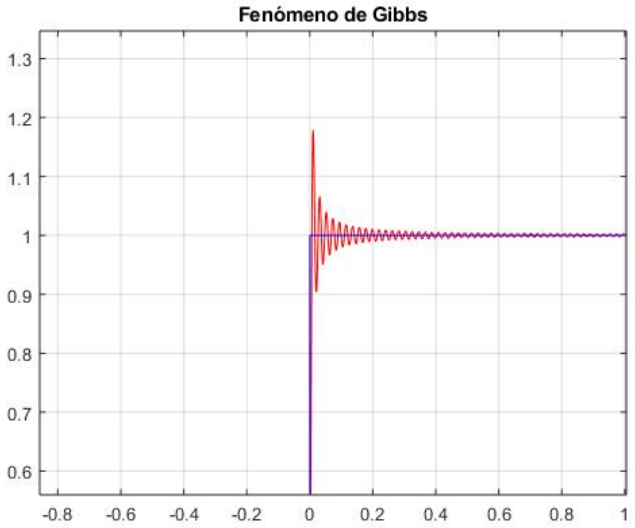


Figura 2.3: Suma parcial de Fourier $N = 300$, proximidades de $(0, 1)$.

$$\lim_{n\to\infty} s_{2n-1}\left(\frac{\pi}{2n}\right) = \frac{2}{\pi}Si(\pi) \approx 1.1790... \tag{2.20}$$

en donde

$$Si(x) = \int_0^x \frac{sen(t)}{t}dt. \tag{2.21}$$

Una nota adicional es que $Si(x)$ se aproxima numéricamente a 1.8519, pues de acuerdo con [12] la primitiva $sin(t)/t$, no se puede expresar con funciones elementales. El resultado de 2.20 indica que las aproximaciones de las sumas parciales de Fourier exceden al valor verdadero de la función, a la derecha es decir, a $f(0+) = 1$ en 0.18, lo que supone aproximadamente un 9 % de la longitud del salto, que en este caso es 2.
En este punto es conveniente retomar el Teorema de Rodríguez del Río [12]
Teorema *Sea f una función real de variable real, con un período 2π. Supongamos que f y f' son ambas continuas excepto para un número finito de discontinuidades de tipo salto en el intervalo $[-\pi, \pi]$. Sea $s_N(x)$ la suma parcial de orden N de Fourier. Entonces en un punto a de discontinuidad, las gráficas de las funciones $s_N(x)$ convergen al segmento vertical (ver figura 2.4) de longitud*

$$L = \frac{2}{\pi}Si(\pi)|f(a+) - f(a-)| \text{ centrado en el punto } (a, \frac{1}{2}(f(a+) + f(a-))). \tag{2.22}$$

La razón entre la longitud del segmento L (al que tienden las gráficas de las $s_N(x)$) y la longitud del salto de la discontinuidad, es decir,

$$\mathcal{L} = |f(a+) - f(a-)|, \tag{2.23}$$

se denomina **constante de Gibbs** y su valor

$$\frac{L}{\mathcal{L}} = \frac{2}{\pi}Si(\pi). \tag{2.24}$$

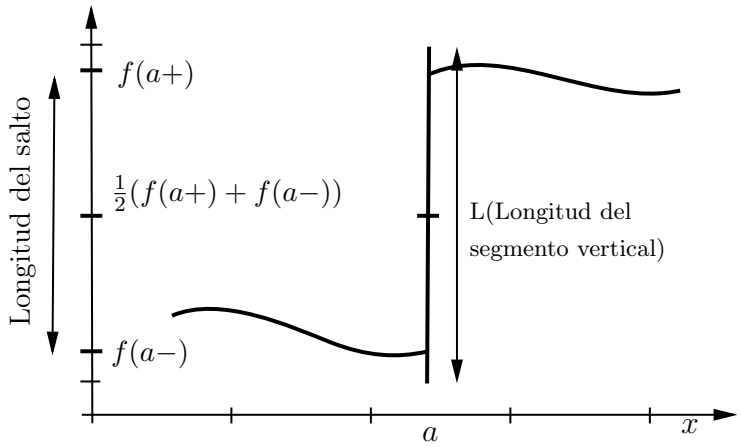


Figura 2.4: Fenómeno de Gibbs en punto de discontinuidad.

que coincide, evidentemente con el obtenido en el caso de la función salto en 2.20.

2.4.1. Fenómeno de Gibbs en Funciones Alternativas

"Función"Delta de Dirac

La "función"*delta de Dirac* fue introducida en 1926 por el físico inglés P.A.M. Dirac (1902-1984) en conexión con sus estudios sobre *Mecánica Cuántica*. Realmente no se trata de una función en el sentido ordinario del término, si no de una *distribución*, en este trabajo se denota como $\delta(x)$ [12].
Las propiedades principales de la función delta de Dirac son:

1. $\delta(x) = 0$ si $x \neq 0$.
2. $\delta(0)$ no está definida.
3. $\int \delta(x)dx = 1$, siempre que el intervalo de integración incluya a $x = 0$.
4. Si $g(x)$ es una función continua en $\mathbb{R}$, entonces $\int g(x)\delta(x)dx = g(0)$.

La gráfica de la función Delta de puede observar en la figura 2.5, ésta distribución es interpretada como una función que se anula en todos los puntos salvo en $x = 0$, en donde toma el valor infinito.
Para $a \neq 0$, la función $\delta(x - a)$, sería la misma función o distribución pero trasladada a $x = a$.
La función delta de Dirac, también es conocida como *Función impulso unitario*, y resulta ser un modelo útil en situaciones en las que, por ejemplo, se tiene un sistema mecánico sobre el que actúa una fuerza externa de gran magnitud durante un breve instante de tiempo. En el caso extremo en el que esta fuerza estuviera concentrada en un punto, esto se representa por la delta de Dirac.
La delta puede entenderse como el límite de una sucesión de funciones que, con una masa unitaria, se concentran infinitamente en torno a un punto.

Ahora, para calcular las sumas parciales de Fourier para la delta de Dirac.
Se revisa primero, ¿De qué manera puede expresarse con una serie trigonométrica una función

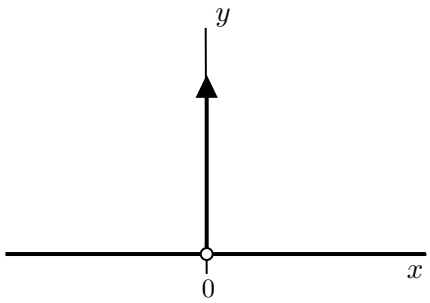


Figura 2.5: Delta de Dirac, $\delta(x)$.

tan singular como la Delta de Dirac? En realidad, primero es calcular las sumas parciales de Fourier correspondientes a una determinada combinación lineal de deltas de Dirac.

$$\Delta(x) = \sum_{z=par} \delta(x + z\pi) - \sum_{z=impar} \delta(x + z\pi) \tag{2.25}$$

que se obtiene al extender de manera periódica, con periodo 2π en la "función"que coincide con la delta de Dirac $\delta(x)$ en $x = 0$ y con $-\delta(x - \pi)$ en el punto $x = \pi$, es decir, con la δ trasladada y cambiada de signo, la reflejada impar de la $\delta(x)$.
Dicho de otra forma, la función denominada $\Delta(x)$, está compuesta por: impulsos unitarios *positivos* en los múltiplos pares de π es decir, puntos de la forma $..., -2\pi, 0, 2\pi, 4\pi, ...$; e impulsos unitarios *negativos* en los múltiplos impares de π, es decir, puntos de la forma $..., -3\pi, \pi, 3\pi, 5\pi, ...$ y su gráfica se podría representar como en la Figura 2.6.

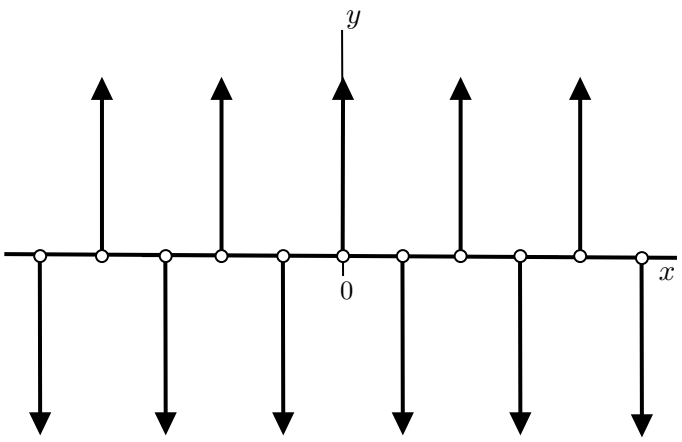


Figura 2.6: "Función" $\Delta(x)$.

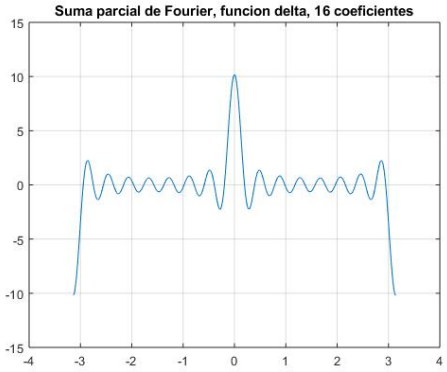


Figura 2.7: Suma parcial de Fourier de $\Delta(x)$, $N = 16$.

Aplicando las fórmulas 2.2 y 2.3 y tomando en cuenta que $\Delta(x)$ es una función par,

$$\begin{aligned} a_n = \frac{1}{\pi}\int_{-\pi}^{\pi}\Delta(x)cos(nx)dx &= \frac{2}{\pi}\int_{0}^{\pi}\Delta(x)cos(nx)dx \\ &= \int_{0}^{\pi}(\delta(x)-\delta(x-\pi))cos(nx) \\ &= \frac{2}{\pi}(cos(0)-cos(n\pi)) \\ &= \frac{2}{\pi}(1-(-1)^n) \end{aligned} \tag{2.26}$$

$$b_n = \frac{1}{\pi}\int_{-\pi}^{\pi}\Delta(x)cos(nx)dx = 0. \tag{2.27}$$

Por lo tanto, la suma parcial de Fourier de la función $\Delta(x)$ queda

$$s_{\Delta,N}(x) = \sum_{n=0}^{N} = \frac{2}{\pi}(1-(-1)^n)cos(nx) \tag{2.28}$$

que se puede expresar de la siguiente forma, dado que para los valores pares de k, los sumandos se anulan:

$$\begin{aligned} s_{\Delta,2n-1}(x) &= \frac{4}{\pi}\sum_{r=1}^{n}cos((2r-1)x) \\ &= \frac{4}{\pi}[cos(x)+cos(3x)+...+cos((2n-1)x)]. \end{aligned} \tag{2.29}$$

En las figuras 2.7 y 2.8 se observan las gráficas de la suma parcial de Fourier de la función $\Delta(x)$ en el intervalo $(-\pi,\pi)$ y en el intervalo $(-5\pi,5\pi)$ de modo que se puede apreciar la asimetría par de la función.

Algunas observaciones que se pueden hacer hasta ahora, a partir del análisis de la función par, son:

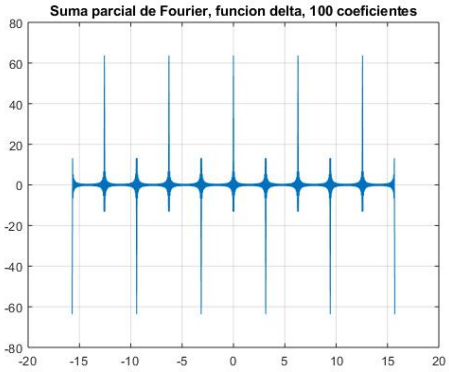


Figura 2.8: Suma parcial de Fourier de $\Delta(x)$, $N = 16$.

- La expresión 2.29 es exactamente la derivada de la expresión 2.19, y este es el motivo de la forma en que al inicio de este análisis se definió la función $\Delta(x)$. es decir:

$$\frac{d}{dx} = [s_{2n-1}(x)] = s_{\Delta,2n-1}(x). \tag{2.30}$$

- A partir de la distribución anterior; si la derivada de las sumas parciales de la función salto son las sumas parciales de la función Delta de Dirac, la integral de las sumas parciales de la delta serán las de la función salto. Es decir:

$$s_{2n-1(x)} = \int_0^x s_{\Delta,2n-1}(t)dt. \tag{2.31}$$

Mientras, $s_{\Delta,2n-1(x)}$ admite una expresión más cómoda:

$$\begin{aligned} s_{\Delta,2n-1(x)} &= \frac{4}{\pi}\sum_{r=1}^{n} cos((2r-1)x) \\ &= \frac{2}{\pi sen(x)}\sum_{r=1}^{n} 2sen(x)cos((2r-1)x) \\ &= \frac{2}{\pi sen(x)}\sum_{r=1}^{n}[sen(2rx) - sen(2(r-1)x)] \\ &= \frac{2}{\pi sen(x)}\left[\sum_{r=1}^{n} sen(2rx) - \sum_{r=0}^{n-1} sen(2rx)\right] \\ &= \frac{2}{\pi}\frac{sen(2nx)}{sen(x)}. \end{aligned} \tag{2.32}$$

Este hecho junto con 2.31, proporciona la siguiente expresión para las sumas parciales de Fourier de la función salto:

$$s_{2n-1}(x) = \frac{2}{\pi}\int_0^x \frac{sen(2nt)}{sen(t)}dt \tag{2.33}$$

al mismo tiempo que

$$\frac{d}{dx}[s_{2n-1(x)}] = \frac{2}{\pi}\frac{sen(2nx)}{sen(x)} \tag{2.34}$$

y se puede comprobar, de este modo que los puntos críticos de la función salto son los ceros de las sumas parciales de la Delta de Dirac. Es decir, que la altura que se alcanza en el primer máximo de la suma parcial de Fourier de la función salto, a la derecha de x=0, es la mitad del *área debajo del arco central de la gráfica 2.7*. Es decir,

$$s_{2n-1}\left(\frac{\pi}{2n}\right) = \frac{2}{\pi}\int_0^{\frac{\pi}{2n}} \frac{sen(2nt)}{sen(t)dt}. \tag{2.35}$$

Pasando al límite cuando $n \to \infty$ y en virtud de 2.20 necesariamente habría de obtener el mismo valor. Por tanto,

$$\int_0^{\frac{\pi}{2n}} \frac{sen(2nt)}{sen(t)}dt \to \int_0^{\pi} \frac{sen(t)}{t}dt = Si(\pi). \tag{2.36}$$

- En las gráficas de las figuras 2.7 y 2.8 parece observarse algo similar al fenómeno de Gibbs. Pero en este caso es distinto a lo que ocurría en la función salto. En la función salto, las gráficas de las sumas parciales excedían o quedaban debajo de la función en los alrededores de los puntos de discontinuidad. En la función Delta esta situación ocurre a lo largo de los puntos de continuidad de la función original, por ejemplo en los puntos del intervalo $(0, \pi)$. Además, este comportamiento no cambia aunque se aumente la cantidad de sumandos, debido a que en este caso tampoco se tiene convergencia puntual.

Función Diente de Sierra

La última función que se tomará en cuenta para nuestro análisis, esta definida por la expresión

$$f(x) = \begin{cases} -\frac{x}{2} - \frac{\pi}{2}; & si \quad -\pi \leq x < 0 \\ \\ -\frac{x}{2} + \frac{\pi}{2}; & si \quad 0 \leq x \leq \pi \end{cases} \tag{2.37}$$

Si se realiza el desarrollo de su serie de Fourier como se ha venido trabajando en las dos funciones anteriores, obtendremos su *N-ésima* suma parcial de Fourier que se expresa de siguiente manera:

$$S_N(x) = \sum_{n=1}^{N} \frac{1}{n} sen(nx), \tag{2.38}$$

cuya gráfica se encuentra en la figura 2.9. En ella se pueden apreciar claramente los efectos de Fenómeno de Gibbs, tanto en el punto de discontinuidad x=0, como en las oscilaciones decrecientes a lo largo del intervalo continuo de la función. Tanto la función Salto como la de diente de sierra son utilizadas ampliamente en aplicaciones de ingeniería, por lo que su tratamiento durante la implementación de este trabajo será de alta relevancia.

2.4.2. Tratamiento para atenuar el Fenómeno de Gibbs: Factor σ-Lanczos

Diferentes personajes han hecho aportaciones para la eliminación o atenuación del fenómeno de Gibbs, uno de estos personajes es **Cornelius Lanczos** (1893-1974), quien fue un físico matemático, que ha tenido un profundo impacto sobre los cimientos de la ciencia del siglo XX, gracias a resultados como la llamada *aproximación de Lanczos*; que es un método para calcular numéricamente la función Gamma. Sin embargo, en esta ocasión el tema de interés es su

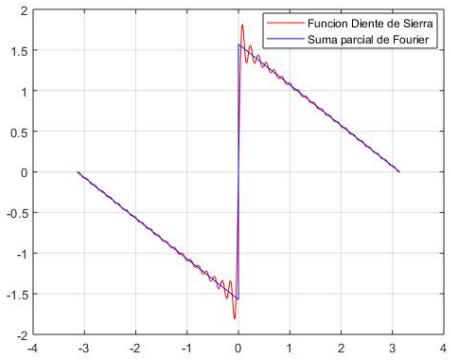


Figura 2.9: Función Diente de Sierra y su correspondiente suma parcial de Fourier con $N = 40$.

Figura 2.10: Cornelius Lanczos (1893-1974). Fuente: School of Mathematics and Statistics University of St Andrews, Scotland

σ-factor o también llamado *Factor Lanczos*, utilizada para atenuar el fenómeno de Gibbs.
El σ-Factor Lanczos, no sólo tiene la función de transformar una serie de Fourier divergente en una serie convergente, sino, también aumenta la velocidad de convergencia de una serie de Fourier.
Como hemos visto anteriormente, la lentitud de la convergencia de la serie de Fourier es particularmente indeseable si un punto de discontinuidad está involucrado. El efecto de esta discontinuidad es que la serie oscila alrededor de la función original con amplitudes que decrecen muy lentamente al incrementar el número de términos. Las oscilaciones de las series truncadas alrededor de la función original siempre están presentes, y por lo general, las amplitudes de estas oscilaciones son suficientemente pequeñas, sin embargo, son muy visibles e interfieren con la síntesis eficaz de armónicos.

En los antecedentes se mencionó que la llamada *media aritmética de Fejer* es el método que de manera exitosa elimina el fenómeno de Gibbs; y con este método la aproximación de la función cuadrada analizada en la sección anterior ahora ocurre por funciones monótonas completamente suaves que se mantienen debajo de la curva. De manera diferente, el método del σ-Factor no elimina las oscilaciones de la serie de Fourier, pero si atenúa las amplitudes de estas.
Para explicar en qué consiste el método del sigma factor e identificarlo en la serie de Fourier de la función cuadrada, se retoma la demostración de [9] en la que se parte del siguiente Teorema.

Teorema *Sea f :[a,b]*$\to \mathbb{R}$ *Riemann integrable, continua en x y sea* $\bar{S}(x,h)=\frac{1}{2h}\int_{t-h}^{t+h} f(x)dx$, *donde t*$\in$*(a,b).* Entonces

$$\lim_{h\to 0} \bar{S} = f(x).$$

Demostración A la suma $\bar{S}(x,h)$ se le conoce como el promedio de f en $[x-h, x+h]$. Considerando el mismo caso de la función cuadrada, se puede intentar sustituir la función oscilante S_n por su promedio $\bar{S}_n$ alrededor del punto x de la siguiente manera:

$$S_n(x) \to \bar{S}_n(x) = \frac{n}{\pi}\int_{t-\frac{\pi}{2n+1}}^{t+\frac{\pi}{2n+1}} S_n dx$$

$$= \frac{n}{\pi}\int_{t-\frac{\pi}{2n}}^{t+\frac{\pi}{2n}} \left[\frac{1}{2} + \frac{2}{\pi}\sum_{k=1}^{n}\frac{1}{2k+1} sen((2k+1)x)\right] dx,$$

de donde obtenemos

$$\bar{S}_n(x) = \frac{n}{\pi}\int_{t-\frac{\pi}{2n}}^{t+\frac{\pi}{2n}} \left[\frac{1}{2} + \frac{2}{\pi}\sum_{k=1}^{n}\frac{1}{2k+1} sen((2k+1)x)\right] dx,$$

$$= \frac{n}{\pi}\left[\frac{\pi}{2n} + \frac{2}{\pi}\sum_{k=1}^{n}\frac{1}{(2k+1)^2} cos((2k+1)x)\Big|_{t-\frac{\pi}{2n}}^{t+\frac{\pi}{2n}}\right]$$

así nos queda

$$\frac{n}{\pi}\left[\frac{\pi}{2n} + \sum_{k=1}^{n}\frac{1}{(2k+1)^2}\left(cos(2k+1)\left[\frac{2nx+\pi}{2n}\right] - cos(2k+1)\left[\frac{2nx-\pi}{2n}\right]\right)\right]. \qquad (2.39)$$

Utilizando identidades trigonométricas se obtiene $cos(x)cos(y) = cos(x+y)+sen(x)sen(y)$, despejando queda:

$$sen(x)sen(y) = \frac{cos(x-y) - cos(x+y)}{2},$$

sustituyendo lo anterior en la expresión 2.39 se tiene:

$$\bar{(}S)_n(x) = \frac{1}{2} + \frac{2}{\pi}\sum_{k=1}^{n} \frac{1}{2k+1} \underbrace{\left[\frac{sen(\frac{\pi}{2n}(2k+1))}{\frac{\pi}{2n}(2k+1)}\right]}_{\sigma} sen((2k+1)x). \tag{2.40}$$

En donde σ queda identificado como el σ-Factor Lanczos.
Del mismo modo en que se realizó este desarrollo para la función *onda cuadrada*, se puede seguir para cualquier función, y concluir que la serie de Fourier es corregida con un σ-Factor Lanczos de la forma

$$\begin{aligned} S_n(x) &= \frac{a_0}{2} + \sum_{k=1}^{n} \left[\frac{sen(\frac{\pi k}{n})}{\frac{\pi k}{n}}\right] (a_k cos(kx) + b_k sen(kx)) \\ &\equiv \frac{a_0}{2} + \sum_{k=1}^{n} \sigma_k (a_k cos(kx) + b_k sen(kx)). \end{aligned} \tag{2.41}$$

Capítulo 3

Dispositivos Programables Reconfigurables

Desde su introducción en la década de los 80, las FPGA (Field Programable Gate Array) han impactado de manera positiva en el ciclo de desarrollo de productos electrónicos. Esto se debe a que su tecnología permite a los desarrolladores de sistemas electrónicos, diseñar, depurar e implementar soluciones basadas en hardware sin tener que desarrollar circuitos integrados de propósito específico también conocidos como ASICs.
Inicialmente, las FPGA tenían lugar como componente central en sistemas digitales, remplazando a los *Programmable Logic Devices* (PLDs) y la tecnología TTL para implementar interfaces lógicas. Sin embargo, con el crecimiento del internet y la tecnología, han emergido nuevos mercados para el desarrollo de sistemas basados en FPGAs, por ejemplo: el prototipado de ASICs, los sistemas de DSP (digital signal processing) de alto desempeño, la implementación de *soft control processors* como MicroBlaze y PicoBlaze proporcionados por el fabricante Xilinx [4]. Además, la arquitectura original de las FPGA era una simple implementación de bloques de lógica programable. Pero con cada nueva generación, nuevas funciones programables han sido añadidas, reforzando algunas funciones específicas con el objetivo de reducir el costo o mejorar el desempeño de las FPGA en sistemas digitales, llegando incluso al desarrollo de un *System on Chip (SoC)* con un *hard block* en la FPGA que se encuentra disponible en la familia Zynq de Xilinx.
El uso de arquitecturas de la familia Zynq ha permitido acelerar el software que se ejecuta en el Procesador del SoC a través de la implementación de algoritmos que se ejecutan en la lógica programable, lo cual usualmente se ve reflejado en una reducción en el tiempo de ejecución y una significativa reducción de la cantidad de energía consumida en cada operación. En esta sección, se abordan los detalles de la arquitectura que conforma una FPGA, así como los detalles de los SoC de la familia Zynq-7000, haciendo énfasis en el SoC Zynq-7020 embebido en la tarjeta de desarrollo Pynq-Z2, ya que es la empleada en este proyecto.

3.1. Arquitectura de las FPGAs

La función principal de las FPGAs es implementar diseños para lógica programable que serán utilizados por los usuarios finales, de esta forma crear nuevos dispositivos hardware; están

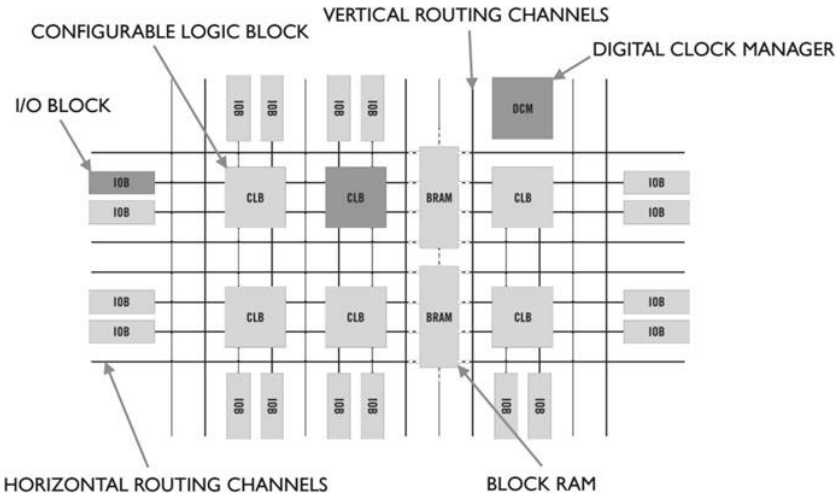


Figura 3.1: Arquitectura básica de una FPGA. Imagen tomada de [4].

constituidas como un arreglo de bloques de puertas lógicas embebidas en un entorno de interconexiones programables. En otras palabras, un FPGA se compone de elementos con recursos no comprometidos, que pueden ser seleccionados, configurados e interconectados por un usuario.
Al arreglo de compuertas lógicas también se le denomina *lógica programable*. En las orillas de su arquitectura se encuentran los bloques de entradas y salidas programables, diseñadas para ser utilizadas como interfaz entre la *lógica programable* y el mundo exterior. En la Figura 3.1 se aprecian las generalidades de estas arquitecturas.

Adicionalmente, a la arquitectura del FPGA se le han añadido nuevos recursos como cadenas de acarreo, bloques RAM y bloques DSP. Si bien, estos bloques son implementados como registros de corrimiento al ser construidos por medio de bloques dedicados específicamente a RAM o DSP permiten una reducción de costos de producción y mayor eficiencia en la gestión de recursos disponibles para el usuario.
Cabe mencionar que dentro de cada familia de dispositivos FPGA, todos los dispositivos comparten la misma arquitectura base, pero cada dispositivo contiene una diferente cantidad de lógica programable. Esto permite al usuario seleccionar el dispositivo correcto de acuerdo a la cantidad de recursos que requiere su aplicación.

Las FPGAs modernas operan en un rango de frecuencias de reloj entre los 100 a 500 MHz, para las cuales la mayoría de los diseños operan en un rango medio de frecuencias. Los que requieren más altas frecuencias son aquellos cuyo objetivo es el *Procesamiento Digital de Señales (DSP)* y que aprovechan la ventaja de contener *slices DSP* en el FPGA y de sus bloques RAM.
A continuación, se presentan algunos elementos básicos de las arquitecturas de FPGA. Para más detalles de cada uno de ellos, referirse a las guías de usuario y hojas de datos proporcionadas por Xilinx.

3.1.1. Interconexiones Programables

Distribuidas a través de la lógica programable es un conjunto de conexiones que pueden ser establecidas para conectar dos bloques lógicos en una FPGA. Esto permite al usuario construir redes de bloques lógicos arbitrarios. La arquitectura de las vías de interconexión varía de generación en generación y está oculta para el usuario a través de las herramientas de diseño proporcionadas por el fabricante.

3.1.2. Bloque de Lógica Programable

Es un arreglo de bloques de lógica programable embebidos en las interconexiones programables. En los dispositivos del fabricante Xilinx son denominados *CLBs (configurable logic blocks)*. Hoy en día, cada bloque de lógica programable está compuesto por una o más funciones lógicas implementadas por una *lookup table* configurable de 4-6 bits, una cadena de acarreo y registros configurables. En este caso, se usa la palabra *configurable* para indicar un bloque que puede ser configurado a través de la memoria de configuración de las FPGAs.
A la combinación de una LUT, una cadena de acarreo y un registro se le denomina *celda lógica*. Estas celdas son la unidad de medida de la capacidad de una FPGA, por ejemplo los dispositivos de mayor capacidad de Xilinx son de la familia Virtex UltraScale FPGA, y soportan hasta 4 millones de celdas lógicas, al tiempo que los más pequeños de la familia Spartan contienen apenas alrededor de 2000.

3.1.3. Memoria

En los diseños modernos, el acceso a memoria es extremadamente importante. Por esta razón, los diseños que se implementan en una FPGA comúnmente utilizan una combinación de memorias embebidas en la lógica programable y memorias externas de tipo DDR. Dentro de la lógica programable pueden implementarse diferentes tipos de memorias como registros discretos, registros de desplazamiento, RAM distribuidas, o bloques de RAM. En la mayoría de los casos, las herramientas de desarrollo proporcionadas por el fabricante seleccionan el mejor tipo de memoria para mapear cada memoria en el diseño del usuario.
Elementos como los registros o Flip-Flops son utilizados para almacenar datos de estado y registros de control, pipelining, y memorias FIFO de poca profundidad, es decir, un número reducido de registros. Los *registros de desplazamiento* o *shift registers* son comúnmente utilizados para generar elementos de retardo de señal o para pipeline balancing en diseños de procesamiento de señales. Las memorias RAM distribuidas se usan para generar memorias poco profundas de hasta 64 bits de profundidad y pueden ser tan anchas como sea necesario, mientras que los bloques de RAM son utilizados como buffers y memorias grandes que tienen soporte para definir tamaño de datos y cantidad de direcciones arbitrarias.

3.1.4. Bloques DSP

Las FPGAs modernas contienen multiplicadores discretos para realizar un procesamiento DSP eficiente.
Los dispositivos FPGA de Xilinx contienen bloques DSP conocidos como *DSP48*, los cuales soportan multiplicaciones de 18x25 bits, contienen un acumulador de 48 bits y un pre-sumador

de 25 bits. Además, soportan directamente operaciones matemáticas de tipo entero, al tiempo que se pueden utilizar librerías para hacer uso de operaciones de punto flotante de 32 y 64 bits. Al igual que con el ruteo de las conexiones entre celdas lógicas y la elección de elementos de memoria, las gestión de unidades DSP es realizada por las herramientas de desarrollo de Xilinx.

3.1.5. Bloques de entradas y salidas

Una de las características más importantes de los dispositivos FPGA es su capacidad para establecer una interfaz directamente a entradas y salidas externas (**I/O**) de diferentes tipos y formatos. Para soportar estas capacidades los dispositivos actuales contienen bloques llamados **I/O Block** o **IOB**. Estos bloques contienen poderosos buffers para conducir señales externas a la FPGA y receptores de entrada, además de que tienen registros para dichas señales y habilitadores de salida. IOBs típicamente soportan de 1.2 a 3.3 V CMOS, LVDS y múltiples estándares de entrada y salida para memorias como SSTL3.
La cantidad de entradas y salidas comúnmente son una limitación en los dispositivos digitales programables, por esta razón las FPGAs se encuentran disponibles en empaquetados de diferentes tamaños, permitiendo al usuario utilizar pequeños empaquetados de bajo costo y una pequeña cantidad de entradas y salidas o empaquetados de gran tamaño para usar con aplicaciones que requieren altas cantidades de señales.

3.2. La familia de SoC Zynq-7000

Los avances en el desarrollo de las tecnologías de semiconductores, particularmente el aumento de la escala de integración, ha permitido a los fabricantes y diseñadores de circuitos integrados la inclusión de una mayor cantidad de dispositivos semiconductores en un menor espacio físico.
Este hecho no sólo ha permitido el desarrollo de FPGAs con una mayor cantidad de celdas lógicas en encapsulados de pequeño tamaño, también dió pauta para que Xilinx desarrollara la familia de SoC Zynq-7000. La característica fundamental es la combinación de un procesador ARM Cortex-A9 de doble núcleo con un FPGA de las familias Artix o Kintex dentro de un mismo circuito integrado. [5]
Aunque procesadores dedicados ya habían sido acoplados a FPGAs en desarrollos anteriores, la propuesta de los dispositivos de la familia Zynq es brindar a los usuarios un procesador de grado de aplicación ARM Cortex-A9 capaz de ejecutar sistemas operativos como distribuciones de Linux, mientras la lógica programable está basada en arquitecturas de FPGAs de la serie 7 de Xilinx. La arquitectura de estos dispositivos está complementada por el estándar de interfaces industriales AXI, que proveen un gran ancho de banda, y conexiones de baja latencia entre las dos partes principales del SoC, lo que significa que el procesador y la lógica programable pueden ser utilizados de manera complementaria eliminando la necesidad de crear interfaces para dos dispositivos físicamente separados. En la Figura 3.2, se pueden apreciar los diferentes bloques que constituyen a los dispositivos de la Familia Zynq 7000.
Entre los beneficios más importantes de utilizar esta gama de dispositivos, se encuentran la simplificación de sistemas completos dentro de un solo chip, la reducción de tamaño físico de las implementaciones y reducción en costos de producción, que en combinación con metodologías

de diseño como HLS permiten reducir también el tiempo para lanzar productos funcionales al mercado.

3.2.1. Sistema de procesamiento (PS) SoC Zynq-7000

Todos los dispositivos Zynq comparten la misma arquitectura básica y todos ellos contienen como base del sistema de procesamiento un procesador de doble núcleo ARM Cortex-A9. Éste puede ser denominado "hard processor", debido a que está constituido y optimizado como un elemento de silicio dentro del SoC. Si bien, existen otras alternativas conocidas como los "soft processors", MicroBlaze de Xilinx, los cuales se implementan en la lógica programable de una FPGA, cuyas ventajas residen en que el número y la precisión de las instancias del procesador son flexibles. El uso de un "hard processor" tiene como objetivo alcanzar un desempeño bastante mayor. De cualquier forma, los dispositivos Zynq permiten la implementación de uno o más "soft processors" de MicroBlaze en la parte lógica del sistema.

Además del procesador ARM, el sistema de procesamiento de los dispositivos Zynq está conformado por un conjunto de recursos de procesamiento que comprenden la *APU*, (del inglés Application Processing Unit), diferentes interfaces periféricas, memorias cache, memorias de interfaces, interconexiones y circuitería para la generación de señales de reloj. Las comunicaciones entre el PS y las interfaces externas se gestiona a partir del sistema *MIO*, (del inglés; Multiplexed Input/Output), que provee 54 pines de conectividad flexible, esto significa que el mapeo entre pines y periféricos puede ser definido conforme es requerido. Adicionalmente, determinadas conexiones pueden implementarse a través del *Extended MIO* (EMIO), el cual es un camino indirecto desde el PS hacia las conexiones externas, pues pasar a través del segmento PL del SoC y comparte recursos de entradas y salidas con el mismo. Ambos elementos se pueden visualizar en el lado izquierdo de la Figura 3.1. El sistema EMIO puede ser utilizado como extensión de los 54 pines de MIO, o como interfaz entre el bloque de PS y los bloques IP implementados en PL. La lista de interfaces administradas por el sistema MIO se encuentra en la Tabla 3.1. Los detalles técnicos de estas interfaces se pueden consultar en [15].

3.2.2. Lógica Programable (PL)SoC Zynq-7000

La parte Lógica Programable (PL) de los dispositivos Zynq se puede observar en la Figura 3.3, en ella se resaltan sus elementos más importantes, cuyas funciones y características coinciden con los elementos que constituyen a todos las arquitecturas FPGA de Xilinx, que fueron comentadas en la sección 3.1.

3.2.3. Interfaces entre PS y PL

Como se expuso anteriormente, el mayor beneficio de utilizar los dispositivos de la familia Zynq reside en su capacidad para utilizar a sus dos bloques principales PS y PL de manera complementaría, permitiendo así crear sistemas completamente integrados [6]. La clave para realizar este tipo de implementaciones es el conjunto de interconexiones e interfaces del estándar AXI, que se describirá a continuación.

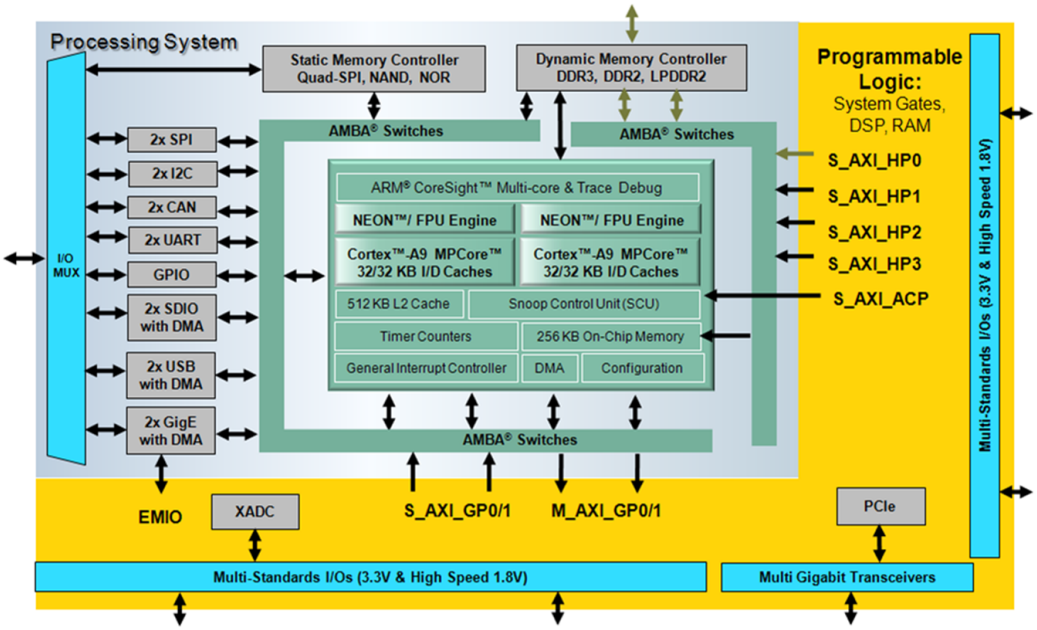


Figura 3.2: Diagrama a bloques de los SoC de la familia Zynq-7000, Imagen tomada de [5].

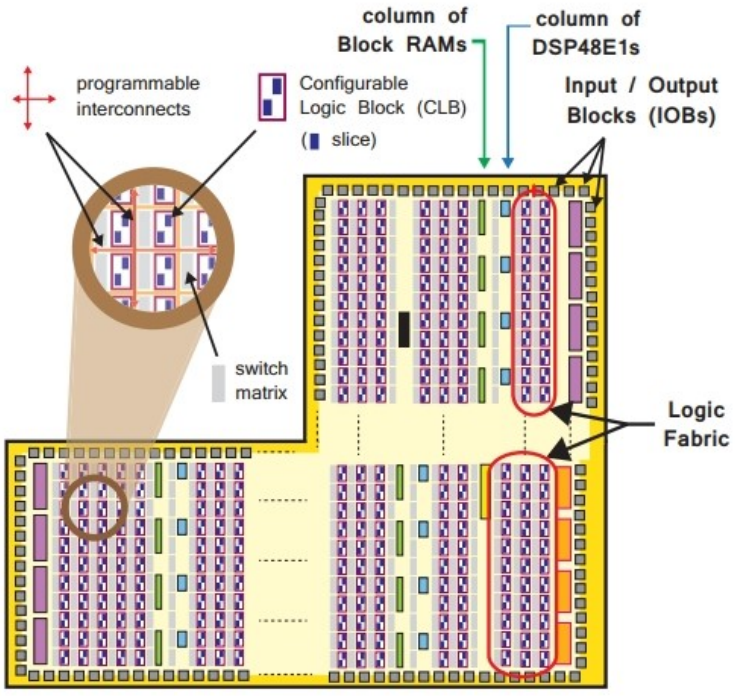


Figura 3.3: Segmento de Lógica Programable embebido en los dispositivos de la familia Zynq. Imagen tomada de [5].

Interfaz de Entrada/Salida	Descripción
SPI (x2)	Serial Peripheral Interface Protocolo estándar para comunicación serial basado en una interfaz de 4 pines. Puede ser usado en modo maestro o esclavo.
I2C (x2)	I2C bus Cumple con la especificación I2C, versión 2. Soporta modo maestro y esclavo.
CAN (x2)	Controller Area Network Interfaz de controlador de tipo bus, cumple con los estándares ISO 118980-1, CAN 2.0A y CAN 2.0B.
UART (x2)	Universal Asynchronous Receiver Transmitter Módulo interfaz de baja velocidad para comunicación serial. A veces usado para establecer conexiones de tipo terminal con una PC.
GPIO (x2)	General Purpose Input/Output El SoC incluye cuatro bancos GPIO de 32 bits cada uno.
SD (x2)	Para interfaz con una tarjeta de memoria SD.
USB (x2)	Universal Serial Bus Cumple con el estándar USB 2.0, y puede ser usado como host, device, o flexiblemente ("on-the-go"; modo OTG, en el cual puede actuar en modo host y device alternativamente).
GigE (x2)	Ethernet Periférico Ethernet MAC, soporta velocidades de 10 Mbps, 100 Mbps y 1 Gbps.

Tabla 3.1: Lista de entradas y salidas de interfaces periféricas de los dispositivos Zynq.

El estándar AXI

AXI del inglés *Advanced eXtensible Interface*, cuya versión actual es AXI4, es parte del estándar abierto ARM AMBA© 3.0, el cual es descrito por sus desarrolladores como "el estándar por defecto para comunicaciones *on-chip*". En el estándar se definen tres tipos diferentes de implementaciones AXI4, cada una de ellas representa diferentes protocolos de bus. La elección del protocolo para una conexión en particular, depende de las propiedades deseadas para dicha conexión.

- **AXI4 [3]** - Para enlaces de tipo *memory mapped*, provee el más alto desempeño de los tres protocolos; en este protocolo una dirección de memoria es transmitida, seguida de una ráfaga de datos de hasta 256 palabras.

- **AXI4-Lite [3]**- Es un tipo de enlace simplificado, soporta la transferencia de un solo dato por conexión (no ráfagas). También es un protocolo de tipo *memory mapped*; en este caso una dirección y una sola palabra es transmitida.

- **AXI4-Stream [2]**- Para transferencia de datos de alta velocidad. Soporta transferencia de ráfagas de datos de tamaño no restringido. En este protocolo no hay mecanismo de direccionamiento; este tipo de implementación es recomendado para un flujo directo de datos entre la fuente y el destino (*non memory mapped*).

En los casos anteriores, el término *memory mapped* describe a aquellos protocolos en los que se especifica una dirección dentro de la transacción solicitada por el dispositivo maestro (para lectura o escritura), la cual corresponde con la dirección en el espacio de memoria del sistema. En el caso de AXI4-Lite, que soporta la transferencia de un solo dato por transacción, el dato es escrito o leído desde la dirección especificada; en el caso de las ráfagas de AXI4, la dirección especificada corresponde al primer dato en ser escrito o leído durante la transferencia, y el dispositivo esclavo debe calcular las direcciones de los datos siguientes.

Capítulo 4

Metodología e implementación

4.1. Metodología

La parte central de la tesis consiste en diseñar e implementar arquitecturas para dispositivos programables del tipo FPGA y SoC Zynq 7020 que permitan el cálculo de series de Fourier de diferentes funciones con discontinuidades de tipo salto, en las que se aprecien las características del Fenómeno de Gibbs.

Una vez desarrolladas las dos arquitecturas (VHDL-HLS) que realizan el cálculo de la serie de Fourier, es necesario complementar la implementación con el módulo correspondiente al σ-Factor Lanczos como elemento atenuador de las oscilaciones producidas por el fenómeno de Gibss, de manera que los resultados obtenidos con ambas arquitecturas reflejen los beneficios y ventajas de emplearse.

Para cumplir con los objetivos esperados, se plantea la división del trabajo en diferentes tareas que corresponden con el proceso de diseño de un sistema embebido, ver Figura 4.1.
La descripción y desarrollo de cada tarea se presenta en esta sección.

- **Requerimientos del algoritmo:** En ésta etapa se analizan las características fundamentales del algoritmo a implementar, incluyendo la información acerca de las series de Fourier y el método del σ-Factor Lanczos.

- **Especificación detallada:** Se realiza una especificación precisa de las características técnicas requeridas en las arquitecturas a desarrollar, ésta etapa define la plataforma de implementación, las metodologías de diseño empleadas y las herramientas de diseño.

- **Arquitectura:** Éste punto analiza los módulos que forman parte del sistema sin entrar en detalles, de forma que cada uno realice su funcionalidad. El propósito de la arquitectura es describir la forma en que el sistema implementará sus funciones e iniciar el proceso de diseño.

- **Metodología de diseño I**: Consiste en desarrollar una arquitectura que satisfaga los requerimientos planteados en las etapas descritas anteriormente, aplicando la metodología

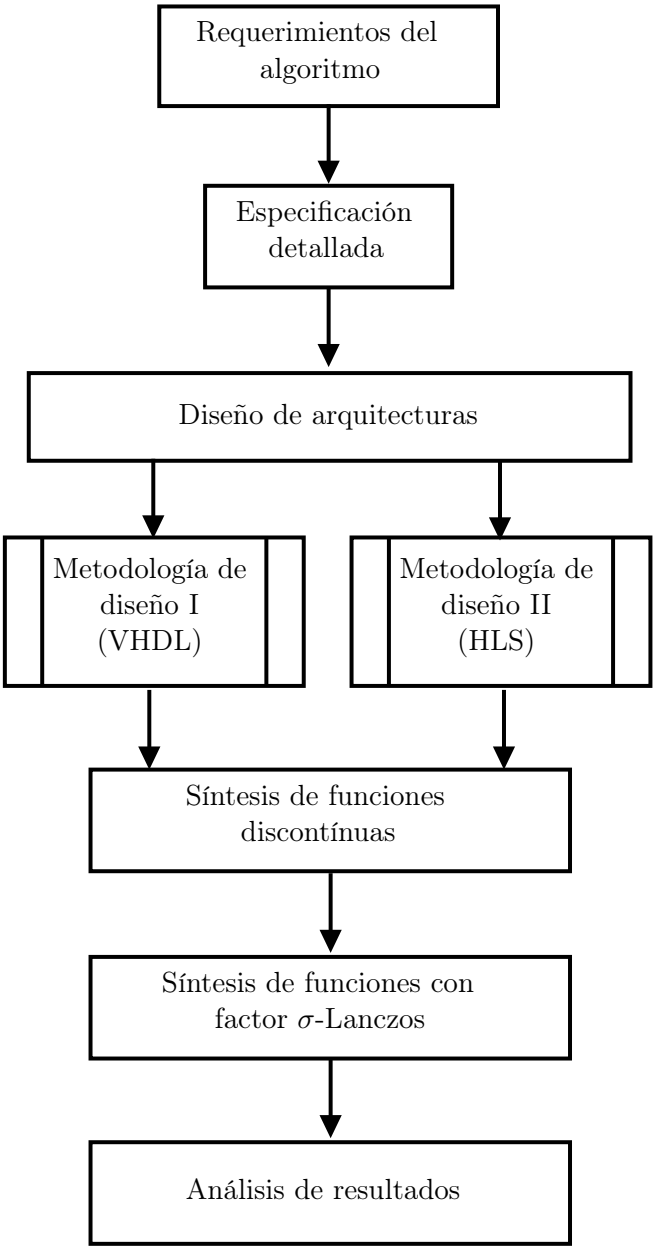


Figura 4.1: Diagrama de distribución de actividades para el desarrollo del proyecto.

de diseño por descripción de hardware en lenguaje VHDL[1] haciendo uso de módulos IP[2] proporcionados por el fabricante Xilinx [4].

- **Metodología de diseño II:** Desarrolla la arquitectura para la implementación del algoritmo que cumpla con los requerimientos planteados anteriormente, utilizando los recursos de la lógica programable (PL) y sistema de procesamiento (PS) del SoC Zynq 7020. El desarrollo se hace siguiendo la metodología de diseño de Síntesis de Alto Nivel (en inglés HLS), que permite la generación de módulos RTL a partir de funciones de código escritas en lenguaje C o C++ [14].

- **Síntesis de funciones discontinuas:** Éste punto sintetiza las arquitecturas desarrolladas de las series de Fourier de funciones con discontinuidades de salto, que mejor resaltan las características del Fenómeno de Gibbs.

- **Síntesis de funciones con factor σ-Lanczos**: Realiza el procedimiento anterior, utilizando el σ-Factor de Lanczos para atenuar las oscilaciones del Fenómeno de Gibbs.

- **Análisis de resultados:** Finalmente, analizar los datos obtenidos de la síntesis de las series de Fourier generadas con las dos arquitecturas y con ello, evaluar la atenuación del Fenómeno de interés. Además, se hace la comparativa de los recursos requeridos en la implementación entre arquitectura I y II.

4.2. Desarrollo

Ésta sección presenta los detalles de la aplicación de la metodología descrita anteriormente para el diseño y la implementación de las arquitecturas propuestas.

4.2.1. Requerimientos del Algoritmo

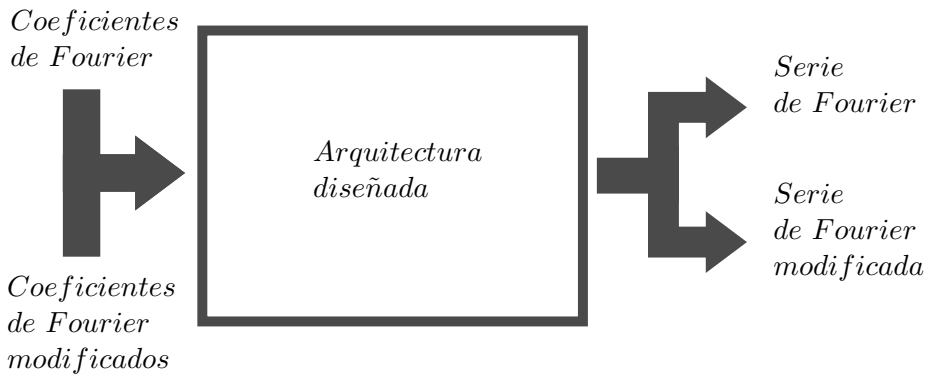


Figura 4.2: Diagrama general del sistema para el cálculo de la serie de Fourier y la serie de Fourier modificada por el σ-Factor.

[1] Very High Speed Integrated Circuit Hardware Description Language
[2] Intelectual Property

Anteriormente, se mencionó que el algoritmo a desarrollar tiene el propósito de realizarse en dispositivos programables: el cálculo de las series de Fourier de funciones periódicas definidas a trozos, que presenten discontinuidades de tipo salto, y posteriormente realizar el cálculo de sus series de Fourier, agregando el σ-Factor Lanczos como mecanismo de atenuación del Fenómeno de Gibbs. De esto, la primer interpretación del sistema se expone en la Figura 4.2. En ella, se observa que el bloque de la arquitectura a crear es *una caja negra* con dos bus de entrada de datos: uno de estos recibe los coeficientes de Fourier de la función a calcular (obtenidos de las expresiones 2.2 y 2.3). El otro bus, recibe los coeficientes de Fourier multiplicados por el σ-Factor (expresión 2.41). Una vez realizadas las operaciones matemáticas correspondientes son dispuestos los valores correspondientes a la componente de la serie de Fourier y a la componente de la serie de Fourier modificada.

4.2.2. Especificación detallada

Se mencionan algunas características de la tarjeta de desarrollo utilizada.

Tarjeta de desarrollo PYNQ-Z2

La tarjeta PYNQ-Z2, es como plataforma, del fabricante TUL©, creada bajo el *Xilinx University Program* para el desarrollo de sistemas embebidos y cuenta con la compatibilidad del *Framework PYNQ(Python Productivity for Zynq)*. Entre las especificaciones sobresalientes se enuncian las siguientes:

- Es basada en el núcleo SoC XC7Z020-1CLG400C de la familia Zynq.
  - Contiene un procesador de doble núcleo ARM Cortex-A9, a 650MHz.
  - Lógica Programable.
    - 13,300 logic slices, cada una con 6 LUTs de 6 entradas y 8 flip-flops.
    - 630 KB de memoria block RAM.
    - 220 bloques DSP.
    - Convertidor analógico-digital a 1 MSPS *on-chip* (XADC) con seis entradas analógicas .
  - Permite la programación a través del JTAG, Quad - SPI flash o tarjeta MicroSD.
- Memoria y almacenamiento
  - Memoria DDR3 de 512MB con ancho de palabra de 16 bits a 1050Mbps.
  - 16MB Quad -SPI Flash con identificador programado de fábrica EUI-48/64© de 48 bits globalmente único.
  - Una ranura de expansión tipo MicroSD.
- Alimentación
  - A través de puerto USB o por medio de regulador externo de 7V a 15V CD.
- Conectividad USB y Ethernet.

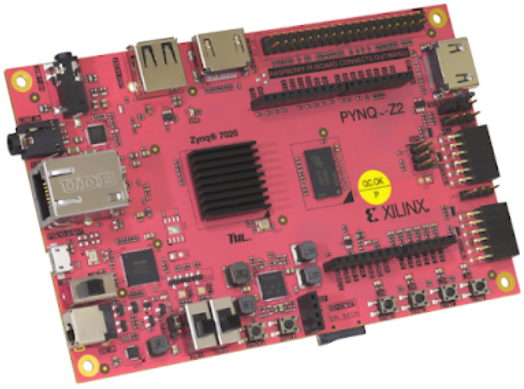


Figura 4.3: Tarjeta de desarrollo PYNQ-Z2.

  - Gigabit Ethernet PHY.
  - Puerto Micro USB para programación a través de JTAG.
  - Puerto Micro USB para comunicación a través de puente UART.
  - Puerto USB 2.0 OTG PHY (soporta modo host únicamente).
- Switches, Push-Buttons y LEDs
  - 4 push-buttons.
  - 2 switches.
  - 4 LEDs.
  - 2 LEDs RGB.

Adicionalmente, cuenta con puertos de expansión para la lógica programable (PMODs), interfaces de audio y vídeo, entre otros [13].

4.2.3. Herramientas de software y desarrollo

Para desarrollar las arquitecturas a implementar en el SoC de la tarjeta PYNQ-Z2, se utilizaron las herramientas de software proporcionadas por el fabricante Xilinx, descritas a continuación:

Vivado Design Suite

Vivado es un conjunto de herramientas de software para el diseño de soluciones de hardware sobre FPGAs de Xilinx bajo diferentes estándares de descripción de hardware como VHDL, Verilog y SystemVerilog [17], sus principales componentes son:

- **IP integrator**: permite crear complejos sistemas de harwdare instanciando e interconectando dentro de un panel de diseño *IP cores* y módulos del catálogo de Vivado.
- **IP packager**: es una herramienta para la creación de IPs *plug-and-play* para extender el catálogo de IPs de Vivado.

- **Vivado simulator**: es un simulador basado en eventos para diseños realizados en Lenguajes de Descripción de Hardware (VHDL, Verilog, SystemVerilog o diseños de lenguaje mixto). Permite realizar simulaciones funcionales, de comportamiento y temporales.
- **Vivado logic analizer**: permite la depuración de diseños basados en hardware a partir del *ILA IP Core* incluido en el catálogo de IPs de Vivado.
- Además: incluye IPs y documentación requeridas para diseñar sistemas que funcionen y utilicen todas las capacidades de los dispositivos Zynq-7000 AP SoC *soft cores* y/o los *hard cores* de la familia MicroBlaze.

Vitis IDE

El proyecto *Vitis unified software platform* es un conjunto de herramientas que combina todos los aspectos de desarrollo de software de Xilinx en un entorno unificado [18]. Esta plataforma de software soporta el flujo de desarrollo de software para sistemas embebidos de Xilinx, anteriormente conocido como *Software Development Kit*, y el flujo de desarrollo acelerado de aplicaciones para desarrolladores de software que desean utilizar los dispositivos FPGA de Xilinx, para aceleración de software.

En este trabajo, se hace uso de la herramienta *Vitis integrated design environment (IDE)*, parte de *Vitis unified software platform*, para el desarrollo de aplicaciones de software embebido en los procesadores del fabricante Xilinx, esto incluye a los mencionados *soft cores* MicroBlaze, y a los *hard cores* ARM Cortex A9 embebidos en los SoC de la familia Zynq. Vitis IDE está basado en el estándar de código abierto *Eclipse*, y sus características de desarrollo incluyen:

- Editor y compilador de código en lenguajes C y C++.
- Administración de proyectos.
- Configuración de construcción de aplicaciones y generación de *Makefiles* automática.
- Navegación de errores.
- Herramientas especiales para la configuración de FPGAs.
- Entre otras.

Las características de Vitis IDE lo hacen ideal para trabajar en conjunto con Vivado para el desarrollo de proyectos en dispositivos programables, que combinen diseños de hardware integrados con aplicaciones de software completamente funcionales.

Vitis HLS

Vitis HLS es una herramienta de síntesis de alto nivel, cuya finalidad es: a partir de funciones escritas en lenguaje C, C++ u OpenCL sintetizar módulos que se implementan en la lógica programable, bloques RAM y DSP. Al igual que otras herramientas, del *Vitis unified software platform*, Vitis HLS es compatible con la metodología de diseño de aceleración de aplicaciones de software, y usa el código descrito en C/C++ para desarrollar diseños de IPs RTL que se pueden añadir al catálogo de *Vivado Design Suite*.

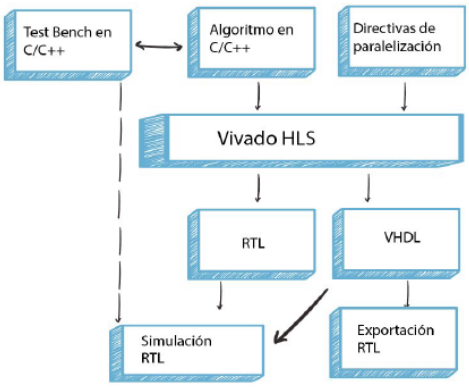


Figura 4.4: Flujo de diseño de IPs utilizando la metodología de Síntesis de Alto Nivel (HLS por sus siglas en inglés).

La herramienta Vitis HLS realiza de manera automática muchas de las modificaciones necesarias para optimizar e implementar el código en C/C++ en la lógica programable, de manera que genera una baja latencia y un alto desempeño en el algoritmo. Parte de las tareas fundamentales de esta herramienta, es inferir los *pragmas* requeridos para implementar el tipo correcto de interfaz de acuerdo a los argumentos de la función a sintetizar, además de crear *pipeline* entre los ciclos y las funciones. Aunque existe soporte para personalizar el código a implementar, de acuerdo a diferentes estándares de interfaces (como el AXI4) u otras especificaciones. El flujo de diseño de Vitis HLS consiste en:

1. Compilar, simular y depurar el algoritmo deseado en lenguaje C o C++, generando una función principal a ser sintetizada y un código de verificación del funcionamiento (*Test-Bench*).

2. Revisar los reportes generados para analizar y optimizar el diseño a través de *pragmas* y directivas de paralelización.

3. Sintetizar el algoritmo en lenguaje C dentro de un diseño RTL o en un archivo de HDL usando la herramienta Vitis HLS.

4. Verificar la implementación del diseño RTL usando la co-simulación RTL.

5. Empaquetar la implementación RTL dentro de un archivo de objeto compilado con extensión *'xo'*, o en su defecto exportarla en una IP RTL, para añadirla al catálogo de Vivado.

Este flujo de diseño se observa en la Figura 4.4.

4.2.4. Diseño de arquitecturas

En esta sección se presentan de manera abstracta las características de los sistemas que se desarrollarán e implementarán en las siguientes secciones. Como se mencionó en los objetivos, se realizará el diseño de dos arquitecturas. La primera es desarrollada en Lenguaje de Descripción de Hardware VHDL utilizando algunos elementos del catálogo de IPs de Vivado, ésta es

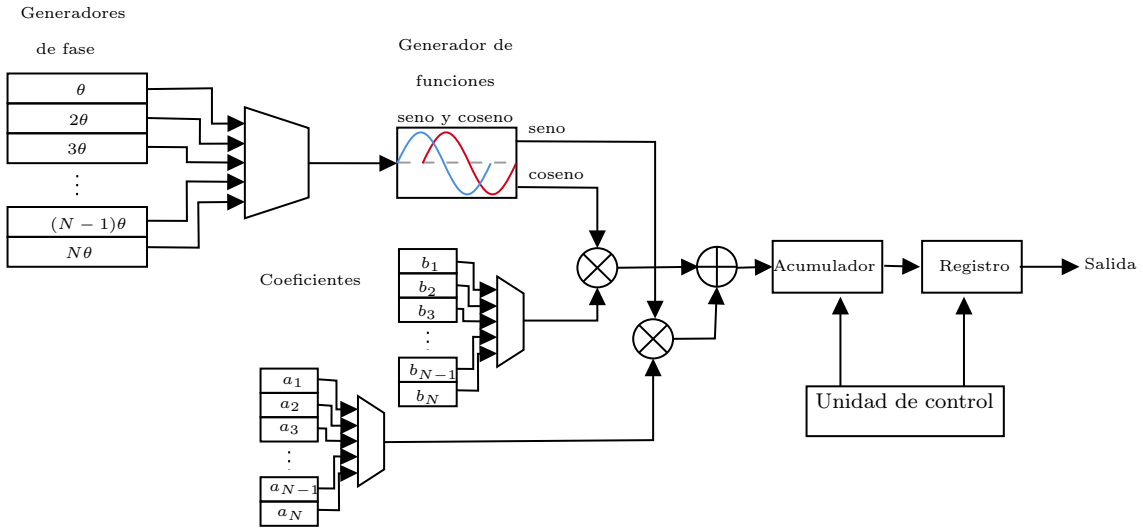


Figura 4.5: Diagrama a bloques de la arquitectura a implementar para el cálculo de series de Fourier usando la metodología de descripción de hardware.

implementada en la lógica programable del SoC. Debido a que en ésta metodología de diseño se describen e interconectan los elementos necesarios para el desarrollo del algoritmo de series de Fourier, se propone como base el diagrama de la Figura 4.5.

En él; se observan los elementos que permitirán realizar el cálculo de las series de Fourier a partir de la expresión I. Diferentes contadores se usarán como generadores de fase, datos que serán utilizados por un elemento generador de funciones para obtener los valores de seno y coseno, al tiempo que los coeficientes deberán ser extraídos de las memorias ROM uno a uno en el sistema para realizar los productos y posteriormente la suma. Éste resultado pasará a un acumulador en donde se emula el proceso de sumatoria, una vez completado este proceso para el número de coeficientes que utilizará el sistema, el resultado del acumulador pasará a un registro cuya salida se considera el resultado de la serie de Fourier calculada. Para efectos de sincronización de los diferentes bloques, se diseñará una unidad de control que permita gestionar los tiempos de cada una de las operaciones.

La segunda arquitectura a diseñar, pretende utilizar el sistema de procesamiento (PS) del SoC embebido en la tarjeta de desarrollo PYNQ, así como segmento de lógica programable (PL). Para ello, es necesario desarrollar una IP a partir del flujo de diseño de Síntesis de Alto Nivel (HLS) presentado en la Figura 4.4. Posteriormente, se deben seguir los pasos del proceso de desarrollo de sistemas embebidos basado en las herramientas Vivado y Vitis IDE, que se presenta en la Figura 4.6.

La IP desarrollada en HLS será la encargada de realizar las operaciones matemáticas que componen a la función I, mientras un algoritmo de control implementado en el sistema de procesamiento (PS) se encargará de gestionar los datos de entrada al módulo IP y realizará las operaciones necesarias para la presentación de los resultados, la comunicación entre ambas partes del sistema será establecida por las interfaces del estándar AXI4. Una abstracción de esta colaboración entre PS y PL del SoC, se puede apreciar en la Figura 4.7.

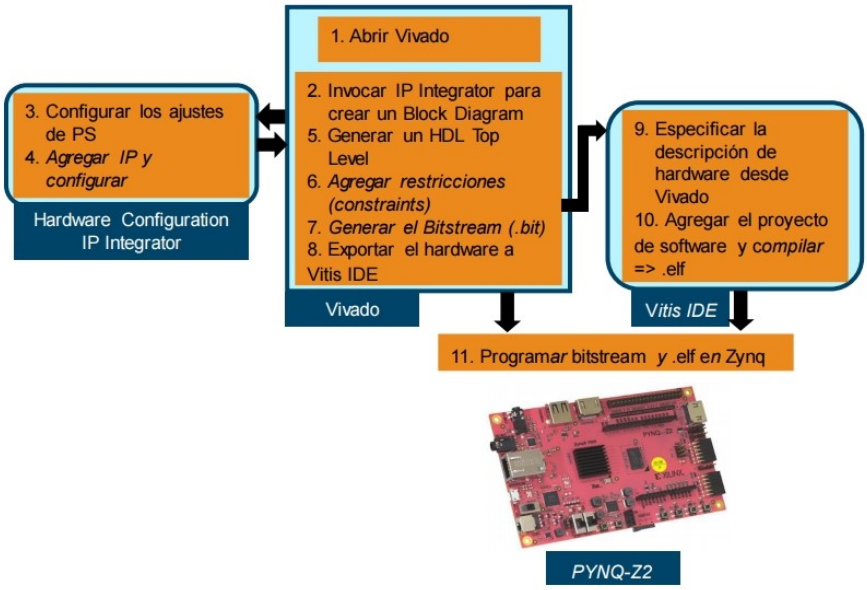


Figura 4.6: Proceso de desarrollo de sistemas embebidos por medio de las herramientas de Xilinx©; Vivado y Vitis IDE [7].

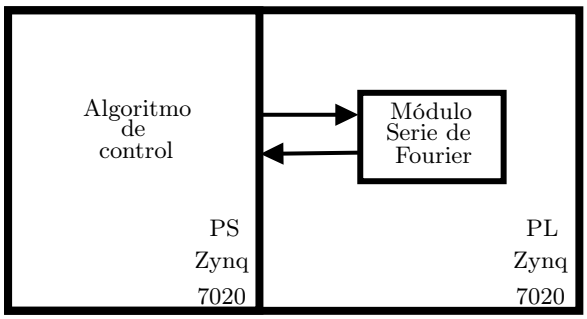


Figura 4.7: Abstracción del sistema para cálculo de series de Fourier utilizando metodología HLS

4.2.5. Metodología de diseño I: Descripción de Hardware

El trabajo de diseño de una arquitectura que calcule series de Fourier, y que también permita realizar el cálculo de las mismas utilizando el $\sigma-$Factor Lanczos se encuentra condensado en el diagrama de la Figura 4.8. Este diagrama es una extensión detallada del presentado en la Figura 4.5, en él se muestran todos los elementos necesarios para la implementación del sistema diseñado en Vivado. Los aspectos más importantes del diseño son:

- La arquitectura propuesta permite el cálculo de series de Fourier de máximo 16 términos.
- Su diseño se realiza a partir del lenguaje de descripción de hardware VHDL, y utiliza bloques IP predefinidos en el catálogo de Xilinx, para la implementación de operaciones específicas.
- El tipo de datos utilizados para realizar las operaciones es **entero con signo** de diferentes tamaños dependiendo de la etapa del sistema y la operación que se realiza.
- El diseño está enfocado en el uso eficiente de los recursos de lógica programable.

El diseño propuesto se divide en dos bloques de sistemas principales que se describen a continuación.

Elementos del algoritmo

Este bloque incluye a todos los componentes del sistema requeridos para realizar las operaciones matemáticas que implica la expresión (I).

Utiliza 16 contadores, asociados a 16 fases involucradas en la serie de Fourier, **(1, 2, .. 16)**θ de 8 bits controlados, a su vez, por un contador principal y un comparador, que determinan el incremento de forma secuencial en cada fase $(1\theta, 2\theta..n\theta)$ de las sinusoidales de la serie de Fourier.

Un multiplexor controlado por la señal de **counter_X** distribuye las fases hacia la entrada del componente parametrizable **DDS compiler**, el cual, calcula las funciones $sen(n\theta)$ y $cos(n\theta)$ [16], los parámetros en configuración del DDS son 8 bits en amplitud y 8 bits en fase, por lo que se tienen 256 valores distribuidos en la amplitud y 256 valores distribuidos en la partición de fase.

De manera paralela a la salida de los valores $sen(n\theta)$ y $cos(n\theta)$, $n = 1, 2, ..., 16$, se extraen los coeficientes de Fourier a_n y b_n almacenados en **memorias ROM** de 16 localidades, con formato de datos enteros con signo de 32 bits, para su posterior multiplicación. En el siguiente ciclo de reloj, los productos $a_n * cos(n\theta)$ y $b_n * sen(n\theta)$ son sumados, y el resultado es almacenado en un bloque **acumulador**, que como se ha mencionado en la sección anterior, realiza el efecto de la sumatoria requerido en la serie. Cada vez que se completa el cálculo para los 16 coeficientes, el resultado del acumulador es pasado a un **Shift Register** que almacena el resultado final, mientras espera el resultado de la siguiente secuencia.

4.2.6. Unidad de Control

Considerada la parte de diseño de mayor complejidad por ser la encargada de determinar la sincronización y concurrencia de los módulos, requiere una implementación a nivel de manipular

corrimientos en fase del reloj principal para establecer la sincronía general en la arquitectura. El diseño de este sistema está realizado para funcionar a partir de dos señales de reloj proporcionadas por el bloque *Digital Clock Manager* **DCM**, en donde, es requerimiento que *ssys_ clk* sea la señal principal utilizada por la mayoría de los elementos de la sección *componentes del algoritmo*, sin embargo, para mantener la sincronía de los elementos *acumulator* y *shift register* es necesario establecer a *ssys_ clk2* a una velocidad 2 veces más rápida que *ssys_ clk*, y mediante un arreglo de multiplexores utilizar a *ssys_ clk2* como elemento de control del reinicio del contenido del acumulador, la señal de reloj del mismo, y la señal de *bypass* que permite pasar el resultado de la sumatoria al *shift register*, así, se tiene el primer elemento en fase de la serie.

El esquema generado dentro del software Vivado de la arquitectura está disponible en el Anexo 1.

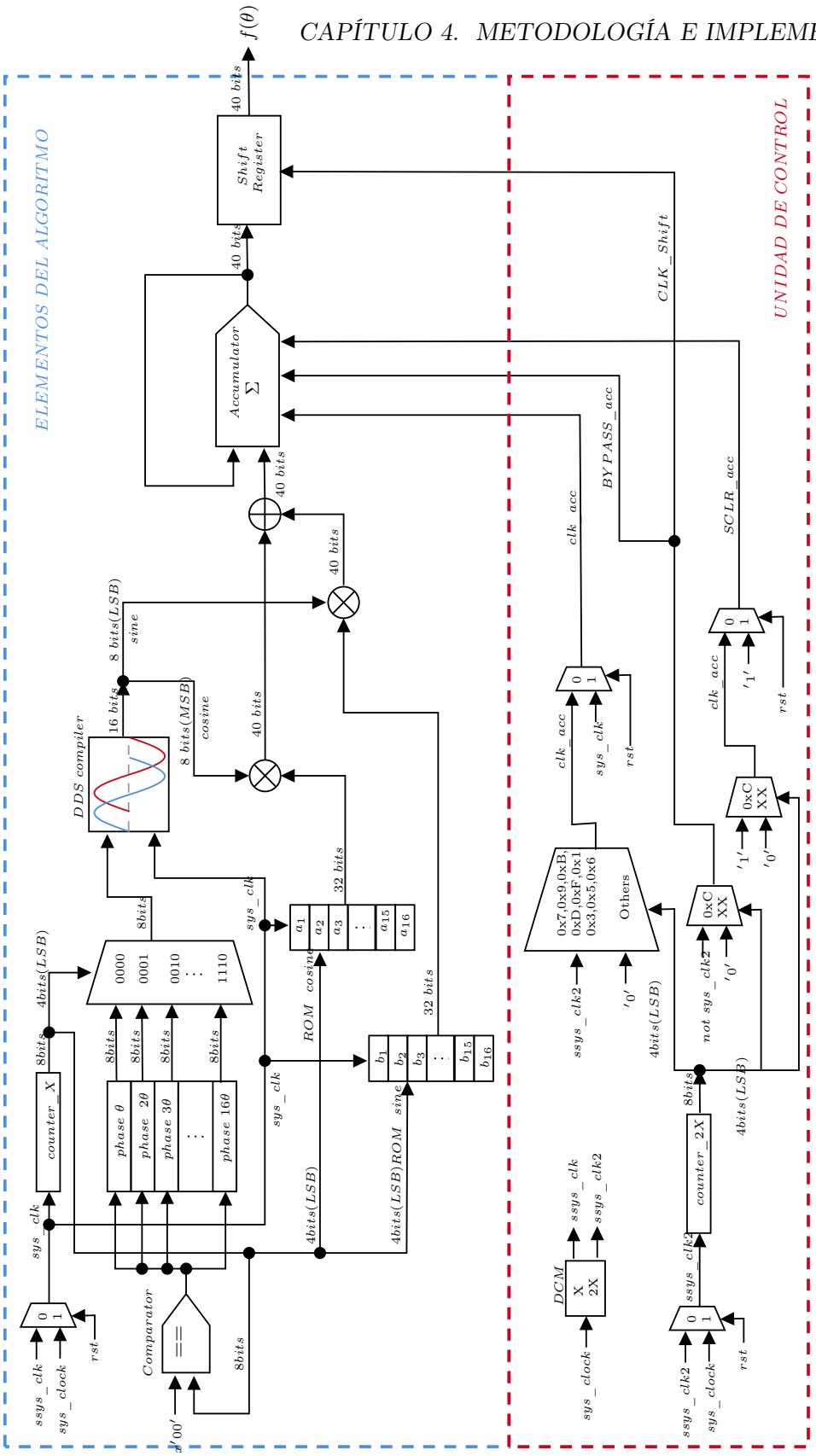


Figura 4.8: Arquitectura detallada del sistema diseñado con la metodología de Lenguaje de Descripción de Hardware.

4.2.7. Metodología de diseño II: Síntesis de Alto Nivel (HLS)

El diseño de la segunda arquitectura se realiza con ayuda de las herramientas proporcionadas por el software Vitis HLS, siguiendo el flujo de diseño para IPs personalizadas de la Figura 4.4. Para ello se realizan dos códigos en lenguaje C++, el Código 1 corresponde a la **función principal** que se sintetizará en una IP y se añadirá al catálogo de Vivado. Este está formado por dos funciones que durante el proceso de síntesis de alto nivel, serán traducidas a bloques descritos en VHDL. La primera función, denominada **serie**, realiza las operaciones correspondientes a la expresión $a_n cos(n*\theta) + b_n sen(n*\theta)$. La función **serie2**, reutiliza a la función serie y se encarga de calcular su resultado para los 16 coeficientes que son proporcionados para la síntesis de funciones. En este caso, es importante mencionar el uso de los pragmas de las líneas 16-19, pues estos definen las interfaces que se implementarán en la IP, para transmitir información entre la IP implementada en la lógica programable y el procesador ARM de la tarjeta de desarrollo en que se implementará el sistema.

Código 1: Función principal

```
1  // HLS math functions
2  #include <hls_math.h>
3  // Funcion de apoyo para calcular los productos de
4  // la serie de Fourier.
5  float serie(float A, float B, float x, int n){
6    float resultado = 0;
7    resultado = A*hls::cosf((n)*x) + B*hls::sinf((n)*x);
8    return resultado;
9  }
10 // Funcion principal,
11 // Itera sobre los 16 coeficientes de la serie
12 float serie2(float A[16], float B[16], float x){
13 #pragma HLS INTERFACE s_axilite bundle = entradas port = B
14 #pragma HLS INTERFACE s_axilite bundle = entradas port = A
15 #pragma HLS INTERFACE s_axilite bundle = entradas port = x
16 #pragma HLS INTERFACE s_axilite bundle = entradas port = return
17   float a = 0;
18   float b = 0;
19   float resultado = 0;
20   Ciclo:for (int n=0; n<16; n++){
21     a = A[n]; b = B[n];
22     serie:{
23     resultado+ = serie(a,b,x,(n+1));
24     }
25   }
26   return resultado;
27 }
```

El segundo código se denomina *Test Bench* o código de pruebas, y se utiliza como complemento a la función principal para validar su funcionamiento. Este mismo código se modificará más adelante para ser embebido en el procesador ARM del SoC Zynq. Su contenido se muestra a continuación.

Código 2: Test Bench

```
#include <stdio.h>
#include <math.h>
#include "test_core.h"
// Constantes
const float pi = 3.1416;
// Variables
float t = (2*pi) / 256;   // Tamano de paso de tiempo
float T[256];     // Vector de tiempo
float y[256];     // Vector de salida simulacion
float y_serie[256];   // Vector de salida hls
float resultado = 0;    // Almacena resultado serie simulacion
float resultado_hls = 0;// Almacena resultado serie HLS
float x = 0;        // Almacena tiempo de evaluacion
float A = 0;        // Almacena valor de coef actual A
float B = 0;        // Almacena valor de coef actual B
float calc_Error = 0;   // Almacena el error calculado
int main(){
  printf("\n\n\n");
  // Definicion de vectores de tiempo y salida
  for(int i=0;i<256;i++){
      y[i] = 0;
      y_serie[i] = 0;
      T[i] = -pi+t*i;
  }
  printf("\n\n\n");
  // Loop sobre periodo
  for(int k=0; k<255; k++){
    x = T[k];
    resultado = 0;
    resultado = serie2(coef_a,coef_b,x);
    y[k] = resultado;
  }
  // Despliegue de resultados en consola
    printf("\nValores calculados\n");
  for (int i=0; i<256;i++){
    printf("%f,",y[i]);
  }
  printf("\n\n\n");
}
```

Es importante observar que en el Código 2, el periodo de las funciones a sintetizar denominado T, está compuesto por 256 muestras, esto coincide con el número de muestras utilizadas en la arquitectura diseñada con la metodología de descripción de hardware y debido a las prestaciones del procesador y de la síntesis de alto nivel para generar bloques complejos de manera automática, el tipo de dato usado en las operaciones es *float*. Una vez verificado el funcionamiento del algoritmo, se realiza la síntesis de alto nivel y corroboran los resultados a partir de la simulación del bloque generado. Posteriormente, este se exporta para ser utilizado en Vivado, siguiendo el flujo de la Figura 4.6.

El sistema desarrollado en el IP Integrator de Vivado para la colaboración entre el PS y PL se muestra en la Figura 4.9. En este diagrama se destaca el uso del bloque IP **serie2_0**, desarrollado a partir de la Síntesis de Alto Nivel, y el bloque **ZYNQ7 Processing System**

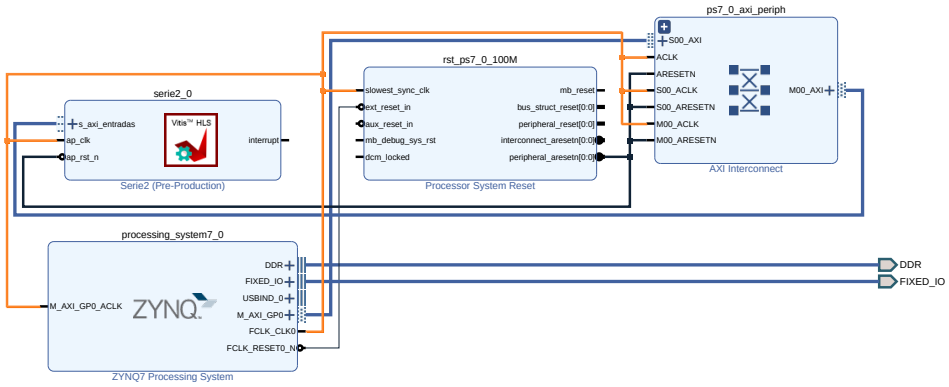


Figura 4.9: Sistema desarrollado mediante la metodología HLS en el que contribuyen el sistema de procesamiento (ZYNQ7 Processing System) y la IP serie2_ 0 sintetizada con las herramientas de Xilinx.

que invoca el uso del sistema de procesamiento embebido en el SoC de la familia ZYNQ.

Adicionalmente, se incluye un bloque de Reset para ambas arquitecturas, y la IP **AXI Interconect**, encargado de establecer comunicación entre el bloque PS y PL del sistema.

El programa de control y gestión de operaciones que se ejecutará en el sistema de procesamiento del SoC, se encuentra disponible en el Anexo 2.

4.3. Implementación

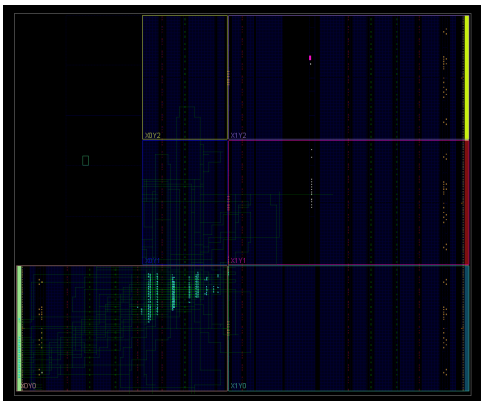
Ambas arquitecturas fueron implementadas en el SoC XC7Z020-1CLG400C embebido en la tarjeta de desarrollo PYNQ-Z2. En la Figura 4.10 se puede interpretar de manera visual la cantidad de recursos del SoC que utiliza cada una de las implementaciones, en la imagen 4.10a, las conexiones en color verde se distribuyen únicamente sobre la lógica programable debido a que ese es el objetivo del diseño, utilizar una cantidad de recursos limitada que permita al usuario disponer de la mayor parte de los recursos y el PS, para implementar otras aplicaciones.

Por otro lado, en la imagen 4.10b, la cantidad de recursos utilizados se incrementa considerablemente, no sólo en la lógica programable, sino que también se requiere del uso del bloque PS del SoC para implementar el algoritmo de control, y se pueden observar claramente las interfaces que se establecen para transmitir información entre la lógica programable y el procesador. La cantidad de recursos real de recursos utilizados se puede ver en la sección de Resultados.

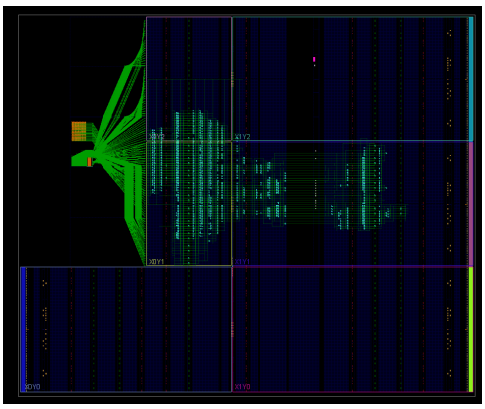
4.3.1. Síntesis de funciones discontinuas

Continuando con la metodología, una vez corroborado el funcionamiento de ambas arquitecturas en la tarjeta de desarrollo PYNQ-Z2, se procedió a implementar la síntesis de funciones discontinuas en ambas arquitecturas, que permitieran observar las características del Fenómeno de Gibbs. Las funciones elegidas para ser sintetizadas son las mismas que se analizaron en el capítulo 2 de este trabajo:

1. Función salto (expresión 2.15, sección 2.4).



(a) Implementación: Recursos utilizados por la arquitectura implementada en lenguaje VHDL.



(b) Implementación: Recursos utilizados por el sistema basado en una IP sintetizada en HLS.

Figura 4.10: Esquemas de recursos del SoC XC7Z020-1CLG400C utilizados por las arquitecturas implementadas.(En verde se muestran las conexiones establecidas entre los componentes básicos del SoC). En muchas aplicaciones, restringir la cantidad de recursos utilizados es primordial, la implementación de la Figura a) posee una ventaja considerable en ese sentido.

2. Serie de deltas de Dirac (expresión 2.25, sección 2.4.1)

3. Diente de Sierra (expresión 2.37, sección 2.4.1)

Se calcularon los primeros 16 coeficientes de Fourier y los mismos multiplicados por el $\sigma-$Factor de las 3 funciones, truncando los datos a los primeros 7 dígitos después del punto decimal. Para la arquitectura diseñada por la metodología de Síntesis de Alto Nivel, estos coeficientes se encuentran disponibles en el Código: "vectores.h" en el Anexo 3 de este documento. Debido a que la arquitectura diseñada con la metodología de descripción de hardware usa el tipo de dato *entero con signo*, es necesario multiplicar los valores del código vectores.h por el factor $1x10^7$, almacenarlos en las memorias ROM de los coeficientes, y re-sintetizar el proyecto cada vez que se cambia de función. Los resultados obtenidos para cada función se pueden observar en las gráficas 5.2 del siguiente capítulo.

Capítulo 5

Resultados

En este capítulo se presentan los resultados obtenidos en cuanto a consumo de recursos de la implementación las arquitecturas desarrolladas en la tarjeta PYNQ-Z2. También se muestran las gráficas de series de Fourier y series corregidas por el $\sigma-$Factor Lanczos para funciones periódicas con discontinuidades. Finalmente, se incluye la tabla de el error obtenido por las arquitecturas.

5.1. Resultados

En primer lugar, la Tabla 5.1 describe la cantidad de recursos de lógica programable del SoC XC7Z020-1CLG400C que son utilizados por las arquitecturas implementadas, datos que son de suma importancia si se pretende utilizar alguna de ellas en colaboración con otros proyectos.

		VHDL		HLS	
Recursos de lógica programable	Unidades disponibles	Unidades utilizadas	Unidades utilizadas(%)	Unidades utilizadas	Unidades utilizadas(%)
LUT	53200	521	0.98	3955	7.43
LUTRAM	17400	0	0	197	1.13
FF	106400	471	0.44	3228	3.03
BRAM	140	0.50	0.36	2	1.43
DSP	220	0.00	0.00	37	16.82
BUFG	32	7	21.88	1	3.13
MMCM	4	1	25.00	0	0

Tabla 5.1: Cantidad de recursos utilizados por cada arquitectura. Se puede observar un mayor consumo de recursos de hardware en la implementación del diseño realizado con la metodología HLS.

Una comparación gráfica de la cantidad de recursos utilizados por cada arquitectura, se encuentra disponible en la Figura 5.1, en donde se muestran los porcentajes de recursos de lógica programable del SoC requeridos por cada implementación.

Los resultados de la síntesis de funciones discontinuas por medio de Series de Fourier y su corrección por el método del σ-Factor, para las funciones salto, diente de sierra y serie de deltas de Dirac se pueden observar en el conjunto de Figuras 5.2. Las Figuras del lado izquierdo

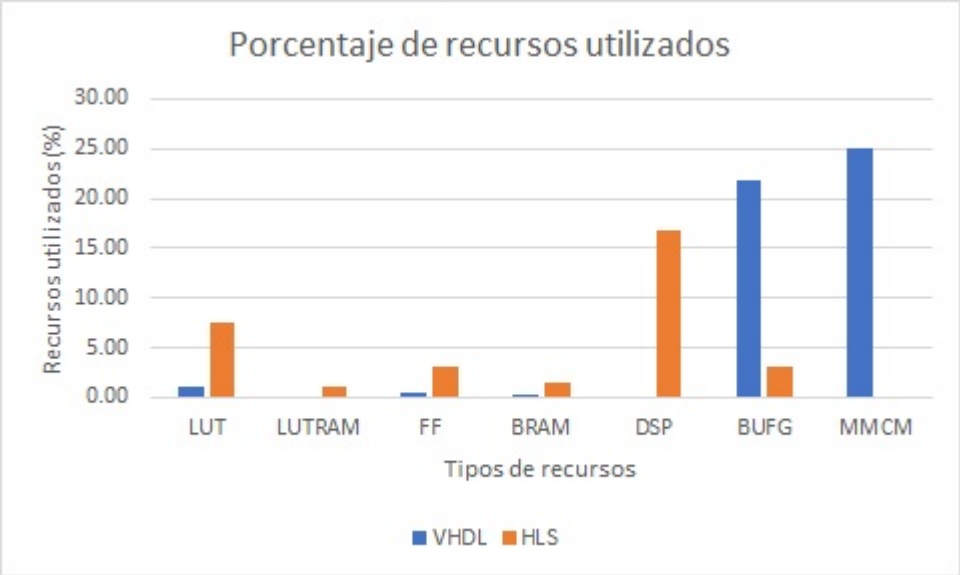


Figura 5.1: Porcentajes de recursos de lógica programable utilizados por cada implementación.

corresponden a los resultados de la implementación en la arquitectura diseñada a través de la metodología I (Descripción de hardware) y del lado derecho corresponden a la metodología II (HLS).

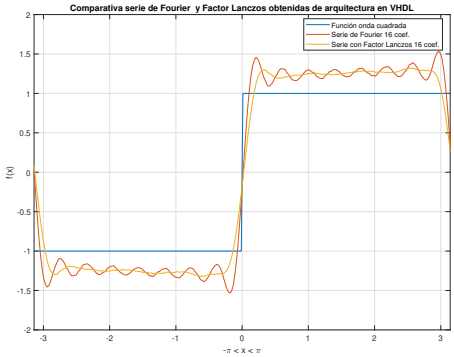
Para verificar los beneficios de la aplicación del σ-Factor al cálculo de las series de Fourier, se incluyen las Tablas 5.2 y 5.3. La primera, muestra el Error Absoluto Medio (MAE) definido por la expresión:

$$MAE = \frac{1}{n}\sum_{j=1}^{n}|y_j - \hat{y}_j|, \tag{5.1}$$

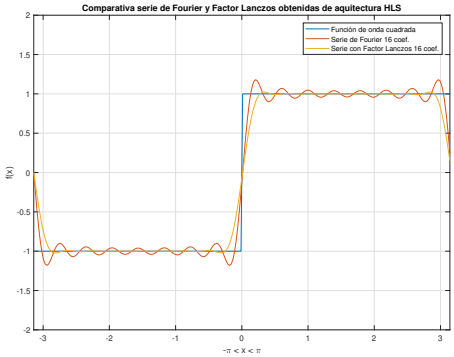
en donde y es la función discontinua que deseamos aproximar, $\hat{y}$ es la aproximación obtenida por series de Fourier o su corrección con el $\sigma-$Factor y n es el número de muestras que se han generado de dicha función. El error absoluto medio nos permite saber cuan lejos se encuentran las aproximaciones obtenidas con respecto a la función original. La segunda Tabla antes mencionada muestra el porcentaje de sobreimpulso generado por el Fenómeno de Gibbs en las aproximaciones realizadas, debido a que este es uno de los principales inconvenientes que se desea evitar.

5.2. Discusión de resultados

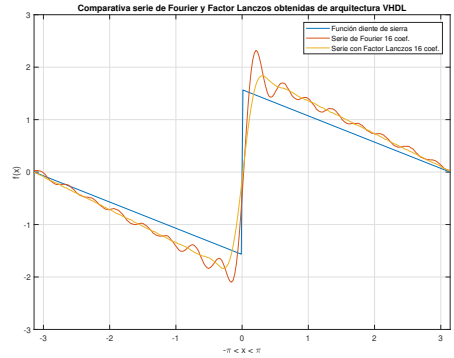
Los resultados obtenidos de la implementación de ambas arquitecturas resumidos en la Tabla 5.1 y la Figura 5.1 dan cuenta del tipo de componentes de lógica programable sobre el que se basa cada diseño. La arquitectura desarrollada en VHDL se caracteriza por un consumo predominante de elementos básicos de lógica programable, es decir; LUTs y Flip-Flops, esto se debe a que los componentes de esta arquitectura fueron cuidadosamente seleccionados para hacer un uso eficiente de los recursos de hardware, los bloques multiplicadores y sumadores utilizados no recurren a elementos de hardware complejos para su implementación. El tamaño de los buses de datos que conectan cada elemento del sistema es definido acorde al tamaño y



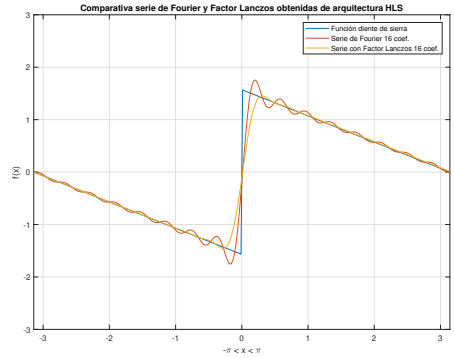
(a) Resultados: Función Salto calculada por la arquitectura VHDL.



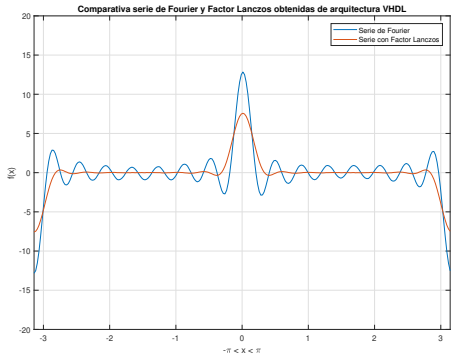
(b) Resultados: Función Salto calculada por la arquitectura HLS.



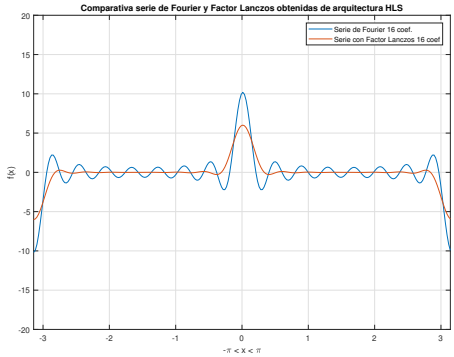
(c) Resultados: Función diente de sierra calculada por la arquitectura VHDL.



(d) Resultados: Función diente de sierra calculada por la arquitectura HLS.



(e) Resultados: Serie de deltas de Dirac calculada por la arquitectura VHDL.



(f) Resultados: Serie de deltas de Dirac calculada por la arquitectura HLS.

Figura 5.2: Conjunto de gráficas resultantes del cómputo de las series de Fourier y su corrección por medio del factor σ-Lanczos. Del lado izquierdo corresponden a los resultados obtenidos de la arquitectura desarrollada en lenguaje VHDL. Del lado derecho corresponden a los resultados obtenidos de la arquitectura desarrollada usando la metodología de flujo de diseño HLS. En ambos casos, las arquitecturas se implementaron en la tarjeta de desarrollo PYNQ-Z2

Error Absoluto Medio				
	Función Salto		Función Diente de Sierra	
Arquitectura	Fourier	Lanczos	Fourier	Lanczos
VHDL	0.2803	0.2755	0.2267	0.2197
HLS	0.0774	0.0654*	0.0682	0.0559*

*Tabla 5.2: Error Absoluto Medio de las funciones sintetizadas con las arquitecturas con respecto a las funciones discontinuas. *El error absoluto medio más pequeño se obtuvo con arquitectura diseñada mediante la metodología HLS y el uso del σ-Factor.*

Porcentaje de sobreimpulso				
	Función Salto		Función Diente de Sierra	
Arquitectura	Fourier	Lanczos	Fourier	Lanczos
VHDL	53.3	32.3	50.4	18.04
HLS	18.0	0.02	12.5	-7.2

Tabla 5.3: Porcentaje de sobreimpulso producido por el Fenómeno de Gibbs en la síntesis de funciones discontinuas utilizando las arquitecturas desarrolladas. Se incluyen los resultados por medio de series de Fourier y corregidos por el $\sigma-$Factor.

tipo de dato que transporta. El bloque DDS compiler que genera las funciones seno y coseno se implementa directamente en LUTs.

Las conexiones tipo BUFG fueron utilizadas para garantizar la integridad de la señales de reloj que gobiernan el sistema y el bloque MMCM (*Mixed-Mode Clock Manager*) para generar las dos señales de sincronización principales.

Por otro lado, la arquitectura diseñada bajo la metodología de síntesis de alto nivel, tiene un desarrollo de sus componentes internos ”transparente” para el desarrollador, esto se debe a que las herramientas de síntesis realizan este proceso. Por este motivo, se tiene muy poco control de la cantidad de recursos utilizados para implementar el sistema. Esto se refleja en una mayor cantidad de LUTs, LUTRAMs y Flip-Flops requeridos. También, con el fin de garantizar la exactitud de las operaciones implementadas en el algoritmo, esta arquitectura hace uso intensivo de los bloques DSP disponibles, especialmente para implementar las operaciones de multiplicación con datos de tipo float y el cálculo de las funciones seno y coseno.

En cuanto a la síntesis de funciones con discontinuidades, las gráficas de la Figura 5.2 dejan ver que ambas arquitecturas realizan la síntesis de series de Fourier. Sin embargo, la arquitectura desarrollada en VHDL presenta un problema de convergencia en los resultados obtenidos. Este problema puede ser causado por el truncamiento de los valores de seno y coseno en el DDS compiler. En el caso de los resultados obtenidos con la arquitectura diseñada con herramientas de HLS, este problema no existe, pues las operaciones implementadas con datos de tipo flotante garantizan la precisión de los resultados.

Se debe destacar que en ambas arquitecturas el uso del $\sigma-$Factor redujo considerablemente

las oscilaciones ocasionadas por el Fenómeno de Gibbs, esto es constatable en todas las gráficas de la Figura 5.2, en donde las líneas correspondientes a la síntesis de funciones por series de Fourier, corregidas por el σ-Factor, presentan oscilaciones más tenues, y el sobreimpulso en los puntos de discontinuidad es mucho menor que en caso de la síntesis únicamente por series de Fourier. Esta idea es reforzada por los datos de la Tabla 5.2, que muestra el error absoluto medio obtenido en la síntesis de las series de Fourier y su corrección por el método del $\sigma-$Factor con respecto a las funciones salto y diente de sierra. Se muestra que el error es menor en todos los casos que se aplica el σ-Factor, independientemente de la arquitectura.

Finalmente, los datos de la Tabla 5.3, dejan ver que el $\sigma-$Factor tiene efectos positivos al ser implementado como elemento atenuador del Fenómeno de Gibbs, reduciendo la magnitud del sobreimpulso en más de 20 % de su magnitud cuando se implementa la arquitectura desarrollada en VHDL y más de un 18 % cuando se usa la arquitectura basada en HLS. Por lo tanto, se considera una muy buena propuesta para aplicaciones en que evitar el sobreimpulso es una tarea crítica.

Conclusiones

1. El σ-Factor Lanczos ha demostrado reducir considerablemente la magnitud de las oscilaciones y el sobreimpulso causados por el Fenómeno de Gibbs.

2. El Fenómeno de Gibbs está presente en muchas aplicaciones de la electrónica. En este proyecto se han implementado dos arquitecturas para la síntesis de series de Fourier y el $\sigma-$Factor Lanczos como elemento para atenuar sus efectos.

3. El uso de una metodología enfocada en el diseño a partir de descripción de hardware, permite implementar sistemas administrando detalladamente los recursos utilizados.

4. Utilizar una metodología de diseño basada en síntesis de alto nivel (HLS) facilita el proceso de diseño, con resultados aceptables en cuanto a la gestión y uso de recursos de hardware.

Glosario

Bloque RAM Bloque solido de memoria RAM embebido dentro de la lógica programable de una FPGA.

CLB Del inglés *configurable logic blocks*, es la unidad básica de la arquitectura de una FPGA compuesta por una LUT, registros y cadenas de acarreo.

DSP Del inglés *digital signal processing*. Puede hacer referencia al procesamiento digital de señales o a bloques dentro de las FPGA dedicados para el alto desempeño en tareas dedicadas al procesamiento digital de señales.

FIFO (Memoria) Del inglés *First In, First Out,* es un tipo de memoria en la cual los primeros datos que entran son los primeros que saldrán de la misma.

FPGA Circuitos integrado reconfigurables compuesto de interconexiones programables que combinan bloques lógicos programables, de memoria embebida y de procesamiento de señales digitales.

IOB Del inglés *Input-Output Block* denota a los bloques de entradas y salidas incluidos en la arquitectura de una FPGA.

LUT Del inglés Lookup Table. Una celda lógica basada en memoria es considerado un bloque interno de la lógica programable de una FPGA cuya función es ser una tabla de verdad que asigna valores de salida a diferentes combinaciones en sus entradas.

LVDS Del inglés *Low-voltage differential signaling*, es un sistema de transmisión de señales de alta velocidad.

PL Del inglés *Programming Logic*, hace referencia al segmento de lógica programable disponible en los dispositivos de la familia Zynq.

PS Del inglés *Processing System, conjunto de elementos de procesamiento implementados en bloques de silicio de los SoC de la familia Zynq*.

RAM Random Access Memory, es un tipo de memoria en la que se pueden tanto leer como escribir datos, comunmnete utilizadas para almacenar datos de forma temporal.

RAM distribuida Memoria RAM en la que los elementos lógicos contenidos en la lógica configurable de la FPGA son conectados formando una memoria.

SoC Del inglés *System on a Chip*, hace referencia a la integración de todos los componentes que forman parte de un computador u otro sistema electrónico encapsulados en un solo chip o circuito integrado.

Referencias

[1] Diego Abril Fedrigo et al. Diseño de filtros para la eliminación de armónicos en sistemas de potencia industriales y comerciales. 2003.

[2] ARM AMBA. Axi4-stream protocol specification. *Volume IHI A*, 51, 2010.

[3] ARM ARM. Amba axi and ace protocol specification, 2011.

[4] Sanjay Churiwala and I Hyderabad. *Designing with Xilinx® FPGAs*. Springer, 2017.

[5] Louise Helen Crockett, Ross Elliot, Martin Enderwitz, and Robert Stewart. *The Zynq book: embedded processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 all programmable SoC*. 2014.

[6] FIUBA. Arquitectura Zynq, 2015. https://campus.fi.uba.ar/pluginfile.php/290152/mod_resource/content/5/Arquitectura_Zynq.pdf.

[7] FIUBA. Introducción al diseño de sistemas embebidos usando zynq, 2015. https://campus.fi.uba.ar/pluginfile.php/287102/mod_resource/content/7/Diseno_Sistemas_Embebidos

[8] David Gottlieb and Chi-Wang Shu. On the Gibbs Phenomenon and Its Resolution. *SIAM Review*, 39(4):644–668, 1997.

[9] Uribe Hernández José Miguel. El Fenómeno de Gibbs y el σ-factor Lanczos, 2011.

[10] Tyn Myint-u. *Partial differential equations of mathematical physics*. CUP Archive, 1978.

[11] Carolyn Oddy. The Gibbs Phenomenon and its Resolution, 2015.

[12] Roberto Rodríguez del Río and Enrique Zuazua Iriondo. Series de Fourier y Fenómeno de Gibbs. *Ene*, 2009.

[13] Tul. Pynq-z2 reference manual v1.0, 2018. URL https://dpoauwgwqsy2x.cloudfront.net/Download/pynqz2_user_manual_v1_0.pdf. Último acceso, 16 marzo 2022.

[14] Xilinx. Vivado Design Suite User Guide: High-Level Synthesis (ug902), 2019.

[15] Inc Xilinx. Zynq-7000 soc technical reference manual. Technical report, Technical report, July 2018. URL https://www. xilinx. com/support . . . , 2018.

[16] Inc. Xilinx. Dds compiler v6.0 logicore ip product guide. 2021.

[17] Inc. Xilinx. Vivado design suite user guide: Getting started (ug910). 2021.

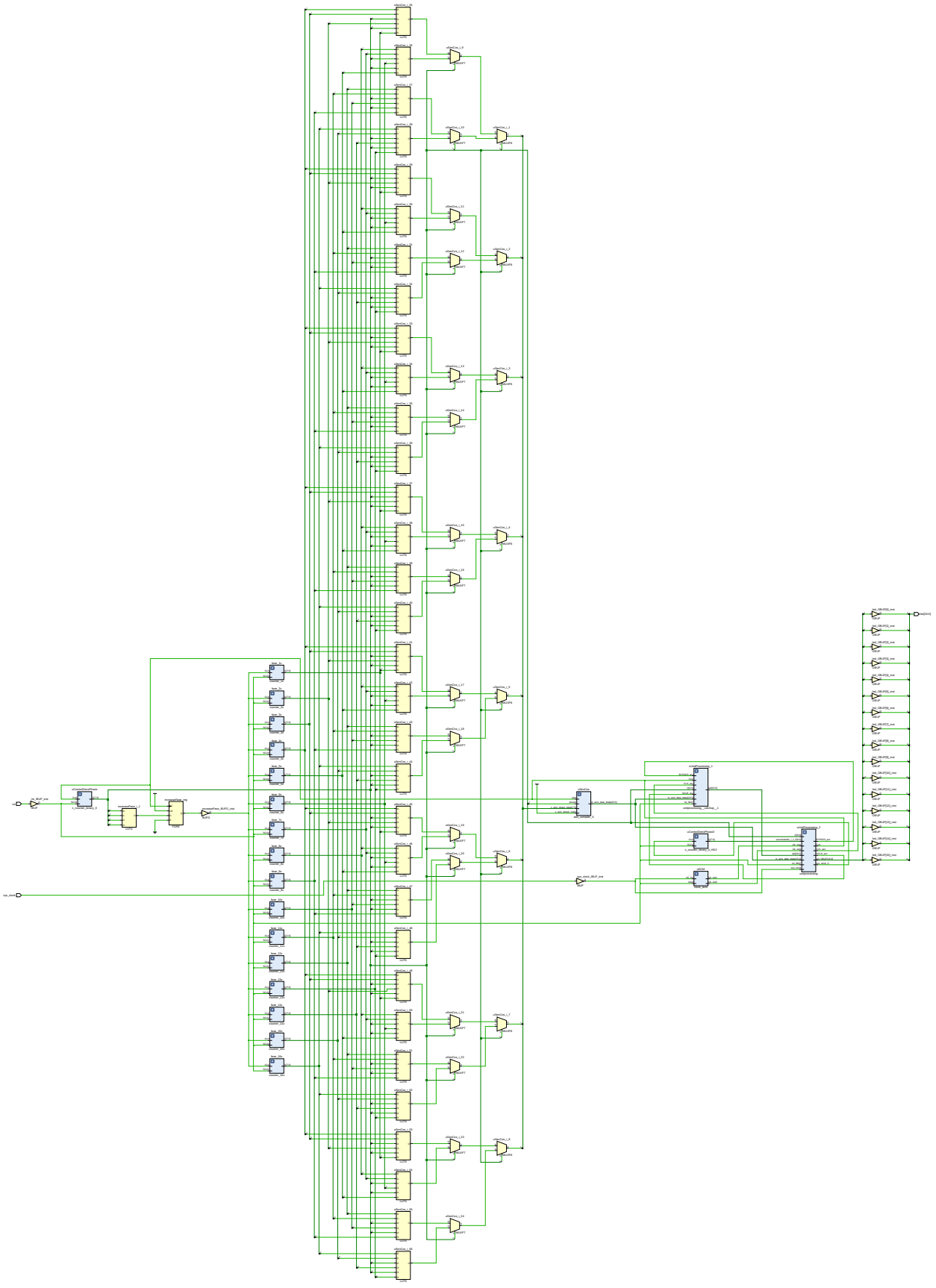
[18] Inc. Xilinx. Vitis unified software platform documentation: Embedded software development. 2022.

Anexos

5.3. Anexo 1. Diagrama Esquemático de la arquitectura implementada en VHDL, generado por Vivado

5.4. Anexo 1A. Código de la arquitectura implementada en VHDL, generado por Vivado

```
1  library IEEE;--,WORK;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_arith.all;
4  use IEEE.STD_LOGIC_unsigned.all;
5  entity top is
6      port ( rst : in STD_LOGIC;
7             sys_clock : in STD_LOGIC;
8             --sys_clk: in STD_LOGIC;
9             led : out STD_LOGIC_VECTOR(15 downto 0)
10            );
11 end top;
12 architecture top_arch of top is
13 signal vip_clk_in  : std_logic;
14 signal vip_clk_out : std_logic;
15
16 COMPONENT ila_0 PORT ( clk : IN STD_LOGIC;
17                         probe0 : IN STD_LOGIC_VECTOR(7 DOWNTO 0)
18                       ); END COMPONENT  ;
19 signal  probe0 : STD_LOGIC_VECTOR(7 DOWNTO 0);
20 COMPONENT c_accum_0 PORT ( B : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
21                           CLK : IN STD_LOGIC;
22                           BYPASS : IN STD_LOGIC;
23                           SCLR : IN STD_LOGIC;
24                           Q : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
25                         ); END COMPONENT;
26 signal     B_acc      :  STD_LOGIC_VECTOR(7 DOWNTO 0);
27 signal     CLK_acc    :  STD_LOGIC;
28 signal     BYPASS_acc :  STD_LOGIC;
29 signal     SCLR_acc   :  STD_LOGIC;
30 signal     sync_acc   :  STD_LOGIC_VECTOR(3 DOWNTO 0);
31 signal     Q_acc      :  STD_LOGIC_VECTOR(7 DOWNTO 0);
32 COMPONENT sumador PORT (
33                           A : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
34                           B : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
```



```
35                          CLK : IN STD_LOGIC;
36                          S : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
37                        ); END COMPONENT;
38 signal A_add   : STD_LOGIC_VECTOR(7 DOWNTO 0);
39 signal B_add   : STD_LOGIC_VECTOR(7 DOWNTO 0);
40 signal CLK_add : STD_LOGIC;
41 signal S_add   : STD_LOGIC_VECTOR(7 DOWNTO 0);
42 COMPONENT multiplicador PORT ( CLK : IN STD_LOGIC;
43                          A : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
44                          B : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
45                          P : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
46                        ); END COMPONENT;
47 signal CLK_m_coseno : STD_LOGIC;
48 signal A_m_coseno   : STD_LOGIC_VECTOR(3 DOWNTO 0);
49 signal B_m_coseno   : STD_LOGIC_VECTOR(3 DOWNTO 0);
50 signal coe_x_coseno : STD_LOGIC_VECTOR(7 DOWNTO 0);
51 signal CLK_m_seno   : STD_LOGIC;
52 signal A_m_seno     : STD_LOGIC_VECTOR(3 DOWNTO 0);
53 signal B_m_seno     : STD_LOGIC_VECTOR(3 DOWNTO 0);
54 signal coe_x_seno     : STD_LOGIC_VECTOR(7 DOWNTO 0);
55 COMPONENT ROMCoseno PORT ( a : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
56                          clk : IN STD_LOGIC;
57                          qspo_rst : IN STD_LOGIC;
58                          qspo : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
59                        ); END COMPONENT;
60 signal a_coseno         : STD_LOGIC_VECTOR(3 DOWNTO 0);
61 signal clk_coseno       : STD_LOGIC;
62 signal qspo_rst_coseno  : STD_LOGIC;
63 signal coe_coseno      : STD_LOGIC_VECTOR(3 DOWNTO 0);
64 COMPONENT ROMSeno PORT ( a : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
65                          clk : IN STD_LOGIC;
66                          qspo_rst : IN STD_LOGIC;
67                          qspo : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
68                        ); END COMPONENT;
69 signal a_seno         : STD_LOGIC_VECTOR(3 DOWNTO 0);
70 signal clk_seno       : STD_LOGIC;
71 signal qspo_rst_seno  : STD_LOGIC;
72 signal coe_seno       : STD_LOGIC_VECTOR(3 DOWNTO 0);
73 COMPONENT dds_compiler_0 PORT ( aclk : IN STD_LOGIC;
74                          aresetn : IN STD_LOGIC;
75                          s_axis_phase_tvalid : IN STD_LOGIC;
76                          s_axis_phase_tdata : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
77                          m_axis_data_tvalid : OUT STD_LOGIC;
78                          m_axis_data_tdata : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
79                        ); END COMPONENT;
80 signal aclk                : STD_LOGIC;
81 signal aresetn             : STD_LOGIC;
82 signal s_axis_phase_tvalid : STD_LOGIC;
83 signal s_axis_phase_tdata  : STD_LOGIC_VECTOR( 7 DOWNTO 0);
84 signal m_axis_data_tvalid  : STD_LOGIC;
85 signal m_axis_data_tdata   : STD_LOGIC_VECTOR(15 DOWNTO 0);
86 signal seno,coseno         : STD_LOGIC_VECTOR( 3 DOWNTO 0);
87 component clk_wiz_0 port  ( clk_out1          : out    std_logic;
88                           clk_out2          : out    std_logic;
89                           -- Status and control signals
```

```
90                          reset             : in     std_logic;
91                          clk_in1           : in     std_logic
92                      ); end component;
93 signal sys_clk,sys_clk2 : std_logic;
94 COMPONENT c_counter_binary_0 PORT ( CLK : IN STD_LOGIC;
95                          SCLR : IN STD_LOGIC;
96                          Q : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
97                        ); END COMPONENT;
98 signal Q_x1               : STD_LOGIC_VECTOR( 3 DOWNTO 0);
99 signal qControlGenePhase, Q_counter2x : STD_LOGIC_VECTOR( 3 DOWNTO 0);
100 signal increaseFase: std_logic;
101 COMPONENT counter_8x PORT (
102                          CLK : IN STD_LOGIC;
103                          SCLR : IN STD_LOGIC;
104                          Q : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
105                        ); END COMPONENT;
106 signal CLK_x8          : STD_LOGIC;
107 signal SCLR_x8         : STD_LOGIC;
108 signal Q_x8            : STD_LOGIC_VECTOR(3 DOWNTO 0);
109 COMPONENT counter_7x PORT ( CLK : IN STD_LOGIC;
110                          SCLR : IN STD_LOGIC;
111                          Q : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
112                        ); END COMPONENT;
113 signal CLK_x7          : STD_LOGIC;
114 signal SCLR_x7         : STD_LOGIC;
115 signal Q_x7            : STD_LOGIC_VECTOR(3 DOWNTO 0);
116 COMPONENT counter_6x PORT ( CLK : IN STD_LOGIC;
117                          SCLR : IN STD_LOGIC;
118                          Q : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
119                        ); END COMPONENT;
120 signal CLK_x6          : STD_LOGIC;
121 signal SCLR_x6         : STD_LOGIC;
122 signal Q_x6            : STD_LOGIC_VECTOR(3 DOWNTO 0);
123 COMPONENT counter_5x PORT (
124                          CLK : IN STD_LOGIC;
125                          SCLR : IN STD_LOGIC;
126                          Q : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
127                        ); END COMPONENT;
128 signal CLK_x5          : STD_LOGIC;
129 signal SCLR_x5         : STD_LOGIC;
130 signal Q_x5            : STD_LOGIC_VECTOR(3 DOWNTO 0);
131 COMPONENT counter_4x PORT (
132                          CLK : IN STD_LOGIC;
133                          SCLR : IN STD_LOGIC;
134                          Q : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
135                        ); END COMPONENT;
136 signal CLK_x4          : STD_LOGIC;
137 signal SCLR_x4         : STD_LOGIC;
138 signal Q_x4            : STD_LOGIC_VECTOR(3 DOWNTO 0);
139 COMPONENT counter_3x PORT ( CLK : IN STD_LOGIC;
140                          SCLR : IN STD_LOGIC;
141                          Q : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
142                        ); END COMPONENT;
143 signal CLK_x3          : STD_LOGIC;
144 signal SCLR_x3         : STD_LOGIC;
```

```
signal Q_x3           : STD_LOGIC_VECTOR(3 DOWNTO 0);
COMPONENT counter_2x PORT (
                        CLK : IN STD_LOGIC;
                        SCLR : IN STD_LOGIC;
                        Q : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
                       ); END COMPONENT;
signal CLK_x1,clkGen  : std_logic;
signal CLK_x2         : STD_LOGIC;
signal SCLR_x2        : STD_LOGIC;
signal Q_x2           : STD_LOGIC_VECTOR(3 DOWNTO 0);
COMPONENT counter_1x PORT ( CLK : IN STD_LOGIC;
                        SCLR : IN STD_LOGIC;
                        Q : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
                       ); END COMPONENT;
signal fase      : std_logic_vector(3 downto 0);
signal selecFase : std_logic_vector(2 downto 0);
signal ssys_clk,ssys_clk2  : std_logic;
COMPONENT shift_ff PORT ( D : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                        CLK : IN STD_LOGIC;
                        Q : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
                       ); END COMPONENT;
signal D_ff   : STD_LOGIC_VECTOR(7 DOWNTO 0);
signal Q_ff   : STD_LOGIC_VECTOR(7 DOWNTO 0);
signal clk_ff : std_logic;
signal SCLR_acc_t : std_logic;
signal CLK_acc_t  : std_logic;
begin
uClock : clk_wiz_0    port map (  -- Clock out ports
                                  clk_out1 => ssys_clk, -- 5MHz
                                  clk_out2 => ssys_clk2, -- 10MHz,
                                 -- Status and control signals
                                  reset => rst,
                                  -- Clock in ports
                                  clk_in1 => sys_clock  -- 100MHz
                         );
with rst select
 sys_clk2 <= ssys_clk2  when '0',
             sys_clock when others;
uControlGenePhase2 : c_counter_binary_0 PORT MAP ( CLK => sys_clk2,
                                                   SCLR => rst,
                                                      Q => Q_counter2x
                                                    );
with rst select
  sys_clk <= ssys_clk  when '0',
             sys_clock when others;
 uControlGenePhase : c_counter_binary_0 PORT MAP ( CLK => sys_clk,
                                                   SCLR => rst,
                                                   Q => qControlGenePhase
                                                   );
process(rst,sys_clk)
begin
    if rising_edge(sys_clk) then
        if qControlGenePhase(2 downto 0) = "000" then
            increaseFase <= '1';
        else
```

```
200             increaseFase <= '0';
201         end if;
202     end if;
203 end process;
204 
205 CLK_x1  <= increaseFase;--: STD_LOGIC;
206 fase_1x : counter_1x PORT MAP ( CLK => CLK_x1,
207                          SCLR => rst,
208                          Q => Q_x1
209                        );
210 
211 CLK_x2  <= increaseFase;--: STD_LOGIC;
212 fase_2x : counter_2x PORT MAP ( CLK => CLK_x2,
213                            SCLR => rst,
214                            Q => Q_x2
215                          );
216 CLK_x3  <= increaseFase;--: STD_LOGIC;
217 fase_3x : counter_3x PORT MAP ( CLK => CLK_x3,
218                            SCLR => rst,
219                            Q => Q_x3
220                          );
221 CLK_x4  <= increaseFase;--: STD_LOGIC;
222 fase_4x : counter_4x PORT MAP ( CLK => CLK_x4,
223                            SCLR => rst,
224                            Q => Q_x4
225                          );
226 CLK_x5  <= increaseFase;--: STD_LOGIC;
227 fase_5x : counter_5x
228                          PORT MAP (
229                            CLK => CLK_x5,
230                            SCLR => rst,
231                            Q => Q_x5
232                            );
233 
234 CLK_x6  <= increaseFase;--: STD_LOGIC;
235 fase_6x : counter_6x PORT MAP ( CLK => CLK_x6,
236                            SCLR => rst,
237                            Q => Q_x6
238                          );
239 
240 CLK_x7  <= increaseFase;--: STD_LOGIC;
241 fase_7x : counter_7x PORT MAP (
242                          CLK => CLK_x7,
243                          SCLR => rst,
244                          Q => Q_x7
245                      );
246 
247 CLK_x8  <= increaseFase;--: STD_LOGIC;
248 fase_8x : counter_8x PORT MAP ( CLK => CLK_x8,
249                          SCLR => rst,
250                          Q => Q_x8
251                      );
252    selecFase <= qControlGenePhase(2 downto 0);
253    with  selecFase select
254       fase <=    Q_x1 when "000",
```

```
255                 Q_x2 when "001",
256                 Q_x3 when "010",
257                 Q_x4 when "011",
258                 Q_x5 when "100",
259                 Q_x6 when "101",
260                 Q_x7 when "110",
261                 Q_x8 when others;
262
263  aclk                              <= sys_clk;
264  aresetn                           <= rst;
265  s_axis_phase_tvalid               <= '1';
266 s_axis_phase_tdata(3 downto 0)    <= fase;
267 uSenCos : dds_compiler_0 PORT MAP ( aclk                 => aclk,
268                                     aresetn              => aresetn,
269                                     s_axis_phase_tvalid  => s_axis_phase_tvalid,
270                                     s_axis_phase_tdata   => s_axis_phase_tdata,
271                                     m_axis_data_tvalid   => open,
272                                     m_axis_data_tdata    => m_axis_data_tdata
273                                   );
274 coseno    <=  m_axis_data_tdata( 3 downto 0);
275 seno <=       m_axis_data_tdata(11 downto 8);
276
277 a_coseno        <= "0"&qControlGenePhase(2 downto 0);--SOLO NECESITA UN ↩
      COEFICIENTE
278 clk_coseno      <= sys_clk;--: STD_LOGIC;
279 qspo_rst_coseno<= rst;--
280 --coe_coseno     <= ;--: STD_LOGIC_VECTOR(3 DOWNTO 0);
281 ROM_Coseno : ROMCoseno PORT MAP ( a         => a_coseno,
282                           clk       => clk_coseno,
283                           qspo_rst  => qspo_rst_coseno,
284                           qspo      => coe_coseno
285                         );
286 a_seno          <= "0"&qControlGenePhase(2 downto 0);--SOLO NECESITA UN ↩
      COEFICIENTE
287 clk_seno        <= sys_clk;--: STD_LOGIC;
288 qspo_rst_seno   <= rst;--
289
290 ROM_Seno : ROMSeno PORT MAP ( a         => a_seno,
291                           clk       => clk_seno,
292                           qspo_rst  => qspo_rst_seno,
293                           qspo      => coe_seno
294                         );
295 CLK_m_coseno <= sys_clk;--: STD_LOGIC;
296 A_m_coseno   <= coseno;--: STD_LOGIC_VECTOR(3 DOWNTO 0);
297 B_m_coseno   <= coe_coseno;--: STD_LOGIC_VECTOR(3 DOWNTO 0);
298
299 Multiplicador_coseno : multiplicador PORT MAP ( CLK => CLK_m_coseno,
300                             A => A_m_coseno,
301                             B => B_m_coseno,
302                             P => coe_x_coseno    );
303 CLK_m_seno    <=sys_clk;--: STD_LOGIC;
304 A_m_seno      <=seno;--: STD_LOGIC_VECTOR(3 DOWNTO 0);
305 B_m_seno      <=coe_seno;--: STD_LOGIC_VECTOR(3 DOWNTO 0);
306 --coe_x_seno      <=;--: STD_LOGIC_VECTOR(7 DOWNTO 0);
307 Multiplicador_seno : multiplicador PORT MAP ( CLK => CLK_m_seno,
```

```
308                          A => A_m_seno,
309                          B => B_m_seno,
310                          P => coe_x_seno);
311  A_add   <= coe_x_coseno;--: STD_LOGIC_VECTOR(7 DOWNTO 0);
312  B_add   <= coe_x_seno;--: STD_LOGIC_VECTOR(7 DOWNTO 0);
313  CLK_add <= sys_clk;--: STD_LOGIC;
314  uSumador : sumador  PORT MAP ( A   => A_add,
315                          B   => B_add,
316                          CLK => CLK_add,
317                          S   => S_add
318                        );
319  -- ======================================================
320  B_acc      <= S_add;    -- 8bits
321  sync_acc <= Q_counter2x(3 downto 0);
322
323  with sync_acc select
324      CLK_acc_t   <=   not sys_clk2    when "0111", --7
325                       not sys_clk2    when "1001", --9
326                       not sys_clk2    when "1011", --b
327                       not sys_clk2    when "1101", --d
328                       not sys_clk2    when "1111", --f
329                       not sys_clk2    when "0001", --1
330                       not sys_clk2    when "0011", --3
331                       not sys_clk2    when "0101", --5
332                       not sys_clk2    when "0110", --6
333                       --sys_clk2    when "001",
334                             '0'   when others; --'0'
335  with rst select
336        CLK_acc <= CLK_acc_t   when '0',
337                   sys_clk     when others;
338
339  with sync_acc select
340    BYPASS_acc <= not sys_clk2      when "0110",
341                  '0'      when others;
342
343  with sync_acc select
344    SCLR_acc_t   <= '1'      when "0110",
345                  '0'      when others;
346  with rst select
347    SCLR_acc <= SCLR_acc_t  when '0',
348                '1'         when others;
349
350  accumulador : c_accum_0 PORT MAP ( B      => B_acc,
351                          CLK    => CLK_acc,
352                          BYPASS => BYPASS_acc,
353                          SCLR   => SCLR_acc,
354                          Q      => Q_acc
355                        );
356 D_ff <= Q_acc;
357 uShift_ff : shift_ff PORT MAP ( D => D_ff, CLK => BYPASS_acc,--CLK_ff,
358                                 Q => Q_ff
359                                 );
360  led <= Q_ff & Q_ff;
361 end architecture;
```

5.5. Anexo 2. Programa de control PS, metodología HLS

```
#include <xparameters.h>
#include <xserie2.h>
#include "platform.h"
#include <stdio.h>
#include "vectores.h"
#include <math.h>
// Constantes
const float pi = 3.1416;
// Variables
float t = (2*pi) / 256; // Tamano de paso de tiempo
float T[256];   // Vector de tiempo
float y[256];   // Vector de salida simulacion
float y_serie[256]; // Vector de salida hls
float resultado = 0;  // Almacena resultado serie simulacion
float resultado_hls = 0;// Almacena resultado serie HLS
float x = 0;    // Almacena tiempo de evaluacion
float A = 0;    // Almacena valor de coef actual A
float B = 0;    // Almacena valor de coef actual B
int main(){
  printf("\n\n\n");
  // Definicion de vectores de tiempo y salida
  for(int i=0;i<256;i++){
    y[i] = 0;
    y_serie[i] = 0;
    T[i] = -pi+t*i;
    printf("\n\n\n");
  }
  // Loop de calculo de serie
  // Loop sobre periodo
  for(int k=0; k<255; k++){
    x = T[k];
    resultado = 0;
    resultado_hls = 0;
    //fourier2(coef_a,coef_b,x);
    // Loop sobre coeficiente
    for(int n=0; n<16;n++){
      A = coef_a[n]; // se instancia al valor -1
      B = coef_b[n]; // Debido a que los arrays se llaman desde A[0] y B[0]
      resultado = resultado + A*cosf((n+1)*x) + B*sinf((n+1)*x);
    }
    y[k] = resultado;
  }
  // Loop de comprobacion de resultados
  for (int i=0; i<256;i++){
    printf("%f,",y[i]);

  }
  printf("\n\n\n");
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 * Inicializacion de la plataforma
 * %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 */
init_platform();
```

```
54 //Initialize our core serie2
55 int status;
56 XSerie2 doSerie2;
57 XSerie2_Config *doSerie2_cfg;
58
59 doSerie2_cfg = XSerie2_LookupConfig(XPAR_SERIE2_0_DEVICE_ID);
60 if(!doSerie2_cfg){
61   printf("Error al cargar configuracion de doSerie2_cfg\n");
62 }
63 status = XSerie2_CfgInitialize(&doSerie2,doSerie2_cfg);
64 if (status!=XST_SUCCESS){
65   printf("Error al inicializar doSerie2\n");
66 }
67 //Llamada al hardware de vivado
68 printf("\n\nConfiguracion realizada exitosamente\n\n");
69 //%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
70 // Implementacion en hardware
71 //%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
72 // Cargar los coeficientes en memoria
73 u32 ub;
74 u32 ua;
75 int E_a;//Bandera de escritura
76 int E_b;//Bandera de escritura
77 //Lectura de datos
78 u32 ua2;
79 u32 ub2;
80 float fa2;
81 float fb2;
82 for (int i=0;i<16;i++){
83   ub = *((u32*)&coef_b[i]);// casteo flotante a u32
84   ua = *((u32*)&coef_a[i]);// casteo flotante a u32
85   E_a = XSerie2_Write_A_Words(&doSerie2,(i),&ua,1);
86   E_b = XSerie2_Write_B_Words(&doSerie2,(i),&ub,1);
87   if (E_a==0){
88     printf("\nEscritura del coeficiente A[%d] incorrecta",i);
89   }else
90     if (E_b==0){
91       printf("\nEscritura del coeficiente B[%d] incorrecta",i);
92     }else {
93       printf("\nEscritura de coeficientes correcta");
94       XSerie2_Read_A_Words(&doSerie2,i,&ua2,1);
95       XSerie2_Read_B_Words(&doSerie2,i,&ua2,1);
96       fa2 = *((float*)&ua2);
97       fb2 = *((float*)&ub2);
98       printf("\nCoeficiente A[%d]=%f",i,fa2);
99       printf("\nCoeficiente B[%d]=%f\n",i,fb2);
100    }
101  }
102  //Escritura del valor de x
103  u32 ux;
104  float resultado = 0;
105  u32  uresultado = 0;
106  for (int i=0;i<256; i++){
107    ux = *((u32*)&T[i]);
108    XSerie2_Set_x(&doSerie2, ux);
```

```
109         printf("\n\nValores de Serie de Fourier Calculados\n");
110     //Ejecucion de la IP
111     XSerie2_Start(&doSerie2);
112     //Espera hasta que la Ip termine sus operaciones
113     while(!XSerie2_IsDone(&doSerie2));
114     uresultado = XSerie2_Get_return(&doSerie2);
115     resultado  = *((float*)&uresultado);
116     printf("%f,",resultado);
117   }
118   return 0;
119 }
```

5.6. Anexo 3. Programa "vectores.h", metodología HLS

El siguiente código contiene los coeficientes utilizados para implementar síntesis de series de Fourier de 3 tipos de funciones discontinuas: Salto, serie de deltas de Dirac y diente de Sierra.

```
1  // Definicion de los vectores de coeficientes
2
3
4  // Funcion Salto
5  //float coef_a[]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
6  //float coef_b[]={1.2732395,  0,  0.4244131,  0,  0.254647, 0,  0.1818913,  ↩
       0,  0.1414710,  0,  0.1157490,  0,  0.0979415,  0,  0.0848826,  0};
7
8  // Funcion Salto Lanczos
9  //float coef_a[]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
10 //float coef_b[]={1.265074, 0, 0.400292, 0, 0.215668, 0, 0.129795, 0, ↩
       0.078518, 0, 0.044560, 0, 0.021317, 0, 0.005623, 0};
11
12 // Funcion Delta
13 //float coef_a[]={1.273240, 0, 1.273240, 0, 1.273240, 0, 1.273240, 0, ↩
       1.273240, 0, 1.273240, 0, 1.273240, 0, 1.273240, 0};
14 //float coef_b[]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
15
16 // Funcion Delta Lanczos
17 //float coef_a[]={1.265074,0,1.200875,0,1.078342,0, 0.908565,0, 0.706662,0, ↩
       0.490156,0, 0.277125, 0, 0.084338,0};
18 //float coef_b[]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
19
20 // Funcion Diente
21 //float coef_a[]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
22 //float coef_b[]={1.000000, 0.500000, 0.333333, 0.250000, 0.200000, 0.166667,↩
        0.142857, 0.125000, 0.111111, 0.100000,  0.090909,  0.083333,  0.076923,↩
         0.071429,  0.066667,  0.062500};
23
24 // Funcion Diente Lanczos
25 float coef_a[]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
26 float coef_b[]={0.993587, 0.487248, 0.314388, 0.225079, 0.169386, 0.130702, ↩
       0.101941,  0.079577,  0.061668,  0.047053,  0.034997,  0.025009,  ↩
       0.016743,0.009944,0.004416,0};
```