Escuela Técnica Superior de Ingenieros Informáticos

Universidad Politécnica de Madrid

Construcción de bloques multiplicadores para filtros FIR mediante programación genética guiada por gramáticas

Trabajo Fin de Máster

Máster Universitario en Inteligencia Artificial

AUTOR: Javier Rabanillo García
TUTOR/ES: Daniel Manrique Gamo y
Emilio Serrano Fernández

Julio 2019

AGRADECIMIENTOS

A Daniel y Emilio, mis tutores, por todas las apreciaciones recibidas durante todo el periodo de realización del TFM.

A mi hermano Juan, por ayudarme con algunas dudas que tenía sobre filtros, bloques multiplicadores y cuestiones de hardware. A Tamar, por todo el apoyo y la comprensión que recibí de ti al estar realizando tu propio trabajo final. A mi padre y a mi madre, por toda la experiencia y el apoyo que me habéis dado para que este trabajo pudiera seguir adelante. A mi amigo Luis Valero, por los recursos que me diste y todas esas conversaciones interesantes sobre genética.

A Francisco Laport, por la elaboración de la plantilla desde la que he redactado este TFM. A Antonio Jiménez, actual coordinador del máster, por toda la atención prestada para aquellos trámites que necesitaba realizar.

RESUMEN

Desde hace décadas, la demanda de procesamiento ha aumentado, haciendo que los procesadores digitales de propósito general se hayan convertido en uno de los principales obstáculos en la obtención de mayores rendimientos de transmisión debido al limitado número de operaciones que pueden realizar en un intervalo de tiempo. Es frecuente resolver este problema de rendimiento mediante el uso de circuitos integrados de aplicación específica que permiten una optimización de recursos acorde a cada problema de procesamiento.

Como alternativa a un proceso de diseño manual, a lo largo de los años han ido apareciendo diversos algoritmos que buscan obtener estos circuitos de forma automática. Un tipo de circuito integrado de aplicación específica es el bloque multiplicador, un dispositivo diseñado para realizar multiplicaciones paralelas de bajo coste que hace posible el funcionamiento de algunos filtros digitales como los de *respuesta finita al impulso* (FIR).

El objetivo del presente Trabajo Fin de Máster es el desarrollo de un algoritmo de Programación Genética Guiada por Gramáticas (PGGG) que permita obtener la estructura de un bloque multiplicador utilizado para el funcionamiento de un filtro FIR.

SUMMARY

Over the past decades, the processing requirements for digital processors have increased, making general-purpose digital processors become one of the main setbacks for obtaining better performances in data transmisions due to the limited number of operations that they can perform in a time interval. This problem is frequently resolved using application-specific integrated circuits that allow resource optimization appropiate to each processing problem.
As an alternative to a manual design process, several algorithms have been devised to obtain such circuits automatically over the years. One kind of these application-specific integrated circuits is called multiplier block, a device designed to perform low-cost parallel multiplications that allow the operation of some digital filters such as finite impulse response (FIR) filters.
This final master degree work aims to develop a Grammar-Guided Genetic Programming algorithm that obtains the multiplier block structure used for the operation of such an FIR filter.

Índice

Índice de figuras

1. INTRODUCCIÓN

Durante décadas, el desarrollo de sistemas de procesamiento digital ha supuesto un área extensa de investigación[4]. Con la evolución constante de las tecnologías digitales, han ido aumentando las exigencias en cuanto a la demanda sobre la capacidad de procesamiento, requiriendo de cada vez mayores anchos de banda para las transmisiones.
Esta creciente exigencia, ha convertido a los procesadores digitales de propósito general en uno de los principales obstáculos para la obtención de mayores rendimientos, debido a la limitación en el número de operaciones que pueden realizar en un intervalo de tiempo[4].
Este problema se ha resuelto frecuentemente mediante la utilización de circuitos integrados de aplicación específica que permiten una personalización adecuada para cada problema de procesamiento, obteniéndose así mayores rendimientos[4].

Como alternativa a un proceso de diseño manual, a lo largo de los años han ido apareciendo diversos algoritmos que buscan obtener estos circuitos de forma automática[47]. Un tipo de circuito integrado de aplicación específica es el bloque multiplicador, un dispositivo diseñado para realizar multiplicaciones paralelas de bajo coste que hacen posible el funcionamiento de algunos filtros digitales como los de *respuesta finita al impulso* (FIR)[47].
Los filtros FIR son un tipo de filtros digitales utilizados en problemas de filtrado y tratamiento de señales digitales. Todo filtro digital experimenta en su operación un retraso de fase en su banda de paso. En caso de que el filtro tenga fase lineal, la salida producida por el filtro equivale a una modificación de la señal original sin que se considere producto de distorsión[38]. A este respecto, una de las razones para utilizar filtros FIR frente a otros tipos de filtros es el requerimiento de esta fase lineal para el problema elegido[38].

Desde la década de 1950, la resolución de problemas complejos mediante heurísticas evolutivas ha adquirido una gran popularidad dentro de la Inteligencia Artificial, dando lugar a un nuevo campo de investigación conocido como la Computación Evolutiva[1].
Un problema es susceptible de ser resuelto por estos métodos si de él no se tiene conocimiento suficiente para formular reglas o fórmulas que permitan obtener resultados óptimos[32]. Mediante una generación aleatoria inicial y posterior recombinación de atributos entre soluciones candidatas, estos métodos obtienen soluciones satisfactorias para el problema planteado con una eficiencia mayor a la de una búsqueda aleatoria. Esta eficiencia es posible mediante el uso de un método de evaluación que permite guiar el algoritmo hacia las soluciones con mejores características del espacio de búsqueda[32].

Entre las diferentes ramas existentes en la Computación Evolutiva se encuentra la técnica utilizada en el presente trabajo: la Programación Genética Guiada por Gramáticas (PGGG). La PGGG es una rama de algoritmos evolutivos derivada del desarrollo durante décadas de los Algoritmos Genéticos y, más directamente, de la Programación Genética.
Al igual que sus predecesoras, esta rama persigue la simulación del proceso de adaptación natural mediante el uso de operadores genéticos que intercambian atributos entre soluciones. Además, las soluciones de este tipo de algoritmos evolutivos se constituyen en estructuras con forma de árbol, aspecto heredado de la Programación Genética, que se generan de acuerdo a una gramática formal que garantiza su válidez sintáctica[34].

El objetivo del presente Trabajo Fin de Máster es el diseño e implementación de un algoritmo de PGGG que permita obtener la estructura de un bloque multiplicador utilizado para el funcionamiento de un filtro FIR. En la realización del trabajo se tienen en cuenta los métodos ya existentes de forma que los resultados obtenidos sean competitivos.

Este trabajo se organiza en varios capítulos, comenzando por esta introducción que corresponde al **capítulo 1**. A continuación, siguen dos capítulos del estado del arte referidos a la técnica utilizada y al dominio del problema elegido para el trabajo.

El **capítulo 2** describe las técnicas utilizadas en este TFM relacionadas con la Computación Evolutiva, introduciendo de forma general las variantes existentes más destacadas.
El **capítulo 3** describe los conceptos, estructura y funcionamiento de los filtros FIR y Bloques Multiplicadores que el algoritmo evolutivo debe construir. Se presenta una introducción teórica de filtros digitales, además de una línea temporal de algoritmos de construcción de bloques multiplicadores considerados relevantes para la realización de este trabajo.

El **capítulo 4** describe de forma breve el planteamiento del problema propuesto para la realización de este trabajo.
El **capítulo 5** detalla el desarollo del algoritmo de PGGG con todas sus etapas: el método de codificación utilizado, el procedimiento de evaluación planteado, los operadores elegidos para la ejecución y algunos detalles de implementación.

El **capítulo 6** presenta el análisis de resultados obtenidos por el algoritmo durante el proceso de construcción del bloque multiplicador en función de una serie de parámetros cambiantes en función del tipo concreto de filtro FIR utilizado.

Para finalizar, el **capítulo 7** cierra el trabajo con las conclusiones y líneas futuras que se proponen para continuar el desarrollo dentro de este área.

2. COMPUTACIÓN EVOLUTIVA

2.1. Introducción

A lo largo de las décadas de 1950 y 1960, varios investigadores estudiaron por separado sistemas evolutivos con la idea de utilizar la evolución como una herramienta de optimización para problemas de ingeniería. La idea principal en torno a estos sistemas es la evolución de una población de soluciones candidatas a un problema dado, usando operadores inspirados en la variación genética natural y en la selección natural[37].

Durante las siguientes décadas, se fueron desarrollando los campos más destacados dentro de la Computación Evolutiva: las Estrategias Evolutivas, la Programación Evolutiva, los Algoritmos Genéticos y, más adelante, la Programación Genética[1].

2.1.1. Estrategias Evolutivas

Las Estrategias Evolutivas (EE.EE) nacen en la década de los 60 como un conjunto de reglas para el diseño y análisis automático de experimentos bajo condiciones de ruido ambiental que reciben ajustes graduales de variables hasta obtener el estado óptimo[2].

En 1964, Ingo Rechenberg y otros estudiantes realizaron el primer experimento satisfactorio de EE.EE para resolver problemas hidrodinámicos[36]. Este experimento superaba los resultados de la heurística univariante y del descenso por gradiente, demostrando la efectividad del método e impulsando experimentos que utilizaron la replicación y eliminación de genes para la evolución de variables, tanto su valor como el número de ellas presentes en el experimento. Como máximo, se utilizaron dos reglas. Una primera regla dicta un cambio de todas las variables al mismo tiempo de una forma suave y aleatoria. Otra segunda regla dice que el nuevo conjunto generado se mantiene solo si no disminuye la bondad de la solución. La primera regla recordaba a las mutaciones en la naturaleza y la segunda parecía modelar la supervivencia del más apto[2].

La primera implementación de EE.EE recibió el nombre de $(1+1)-ES$[2]. Este nombre se debe a que cada generación comienza con una solución progenitora de la que, mediante mutación, se genera otra solución descendiente. Estas soluciones posteriormente se comparan para conservar la mejor de ellas.

Para la mutación, se introduce un factor que regule la longitud media de ajuste: la *fuerza de mutación*[2]. Se elige una función de distribución y una fuerza de mutación distinta en función del problema a resolver, siendo ambos factores dependientes del espacio de búsqueda. Para variables numéricas, se introduce un ajuste δ al valor inicial y para obtener un nuevo valor ajustado $\tilde{y}$ de la siguiente manera:

$$\tilde{y} = y + \delta$$

Para variables continuas, la distribución más utilizada en las EE.EE es la distribución normal[2]. Para obtener el ajuste δ, se realiza un muestreo de varios valores de la distribución. Cada uno de estos valores tiene una probabilidad de ser seleccionado igual a la función de densidad de la distribución normal.
Para una media μ y una varianza σ^2, esta función de densidad se describe típicamente como:

$$\phi_{\mu,\sigma^2}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

En la distribución utilizada, μ equivale al valor inicial y. Para una fuerza de mutación σ, el ajuste δ tiene la siguiente probabilidad:

$$p(\delta) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(\tilde{y}-y)^2}{2\sigma^2}} = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{\delta^2}{2\sigma^2}}$$

Dado que el máximo de esta función de densidad está en el centro con tendencia decreciente hacia los extremos, los cambios más probables serán los más centrales, es decir, los que supongan menor ajuste.

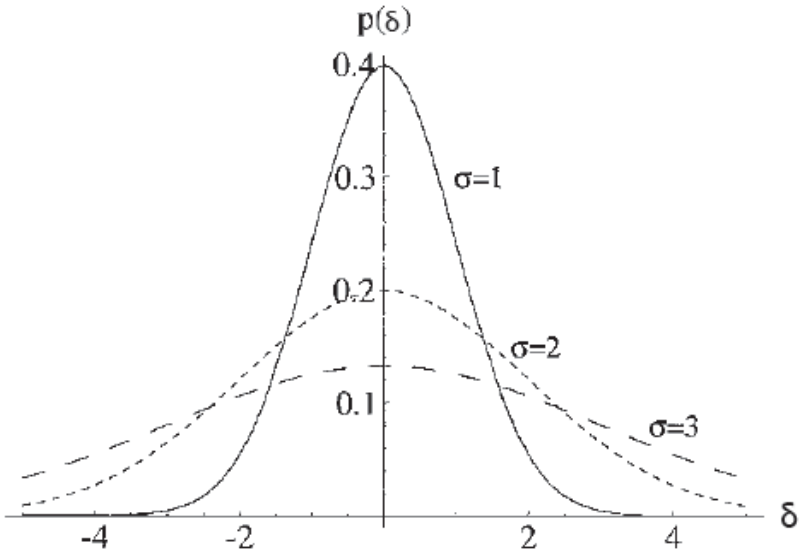


Fig. 1: Curva de $p(\delta)$ en función de la fuerza de mutación σ. Figura modificada de [2].

Esta limitación en los ajustes permite mantener una cercanía con el resultado previo, manteniendo un equilibrio entre la exploración y la explotación que hace posible un curso evolutivo constante y contribuye a la eficiencia del proceso de optimización[2][13].

Rechenberg presenta una nueva implementación, $(\mu + 1) - ES$[2], la primera *estrategia evolutiva multimiembro*, una EE dotada de una población de múltiples individuos. Esta implementación contaba con una población aumentada de μ progenitores y la presencia de un nuevo operador de cruce que genera un descendiente por cada generación. Para la generación de este descendiente, se eligen dos progenitores al azar para ser recombinados en un nuevo individuo que se muta posteriormente. De estos tres individuos, el peor de ellos es eliminado y los otros dos continuan en la siguiente generación. Rechenger afirma que el cruce puede acelerar substancialmente la evolución en cada generación. Además, idea un proceso de mutación sin control externo que podría dotar a la población de la capacidad de aprender a ajustar la fuerza de mutación por sí misma.

La década de 1960 eran tiempos de avance para los procedimientos de optimización numérica. Schwefel cree necesario comparar la eficiencia y robustez de las EE.EE con sus competidores de métodos tradicionales, llevando a cabo varios estudios. De la misma manera que en los métodos clásicos, la eficacia de las EE.EE depende enormemente del ajuste de parámetros internos, sobre todo el de la fuerza de mutación. De los estudios llevados a cabo, Schwefel descubre que $(\mu + 1) - ES$ tiene una tendencia hacia la reducción de la fuerza de mutación, un síntoma de convergencia. Esta observación lleva a Schwefel a introducir $(\mu+\lambda)-ES$ y $(\mu, \lambda)-ES$, dos nuevas versiones de *estrategia evolutiva multimiembro* con una población de μ progenitores y λ descendientes en cada generación, especialmente diseñadas para el procesamiento paralelo con λ procesadores, que permite la evaluación simultánea del conjunto de descendientes. La mayor diferencia entre estas versiones es la elección de soluciones que pasarán a la siguiente generación, lo que conocemos como reemplazo. El reemplazo de $(\mu + \lambda)$ descarta las μ peores soluciones de todos los $\mu+\lambda$ individuos. En cambio, (μ, λ) mantiene los λ sucesores, sin importar en ningún sentido la aptitud de los progenitores[2].

La comunidad alabó las primeras aplicaciones de las EE.EE. en el campo de la optimización experimental, aunque no existiera ninguna prueba de convergencia hacia un óptimo global en la resolución de estos problemas. No fue así con sus aplicaciones en el campo de la optimización numérica, quedando eclipsadas por el sustancial progreso obtenido en aquella década desde la optimización numérica clásica, aun con influencias de heurísticas determinísticas. No se percibía la necesidad de otros métodos, mucho menos uno aleatorio e inspirado en la biología operando con más de un punto de búsqueda a la vez. La versión $(\mu + 1)$ fue ridiculizada por introducir la idea de utilizar hasta el peor de los μ progenitores, en vez de enfocar todos los esfuerzos en la mejor solución. También se criticó que $(\mu + \lambda)$ no utilizara la información recogida por los nuevos descendientes hasta que no estuvieran todos generados. Por último, (μ, λ) se percibió como un sinsentido por abrir la posibilidad a perder soluciones buenas o intermedias en favor de otras peores[2].

2.1.2. Programación Evolutiva

La Programación Evolutiva (PE) fue ideada por Lawrence J. Fogel en 1960, mientras trabajaba como asistente de investigación en torno a heurísticas y a simulación de redes neuronales primitivas. Fogel pensaba que ambas técnicas tenían una limitación fundamental: no modelaban la evolución, el proceso que produce criaturas de creciente intelecto[14].

Según Fogel[14], el comportamiento inteligente tiene como componente principal la predicción, habilidad necesaria para predecir entre una serie posible de entornos de forma que se puedan cumplir objetivos. Este planteamiento sugirió a Fogel la realización de experimentos utilizando la evolución simulada de máquinas de estados finitos para predecir una serie temporal estocástica. De esta manera, el entorno a predecir para los individuos en la PE, se describe como una secuencia de símbolos procesables en una máquina de estados finitos, por lo tanto, pertenecientes a un alfabeto finito.

Los organismos vivos son fruto de la dualidad que existe entre su genotipo y su fenotipo. El genotipo corresponde a la codificación genética oculta. El fenotipo es la manifestación del genotipo, el comportamiento, la fisiología y la morfología del organismo. Esta dualidad sugiere dos enfoques a la hora de simular la evolución: simulaciones genotípicas y simulaciones fenotípicas[14]. Las simulaciones genotípicas equiparan las soluciones candidatas a cromosomas y genes que se manipulan utilizando operadores genéticos. Las simulaciones fenotípicas se enfocan en los comportamientos de las soluciones, buscando mantener relaciones conductuales entre progenitores y descendientes. Dándose importancia a esta relación de comportamientos, se consigue evitar cualquier deriva en la dirección de la búsqueda, puesto que cualquier descendiente divergente es descartado o directamente no generado. Esta relación de comportamiento se traduce necesariamente en una cercanía en la aptitud[13].
De los dos tipos de simulación antes mencionados, la PE pertenece al segundo conjunto, el de la simulación fenotípica. Ésta se desarrolla sin la intención de imitar estrictamente la naturaleza, por lo que no cuenta con un operador de cruce. Se ha explicado esta carencia desde un punto de vista formal argumentando que la PE es una abstracción de la evolución de distintas especies, imposibilitando la acción de este operador [36].

El operador de variación principal de la PE es el operador de mutación. Este operador está gobernado por una distribución de probabilidad, típicamente uniforme, que decide si esta operación se realiza en la presente iteración. El número de descendientes puede ser fijado previamente, pero también puede ser elegido mediante una distribución de probabilidad[14].

Dada la estructura de las máquinas de estados finitos, existen cinco tipos de mutación:

1. Cambio de un símbolo de salida.
2. Cambio de una transición de estado.
3. Adición de un estado.
4. Eliminación de un estado.
5. Cambio de estado inicial.

Una población de estas máquinas debe producir un símbolo de salida por cada símbolo de entrada que reciba del entorno. Cuando se genera este símbolo de salida, se evalua en función del siguiente símbolo del entorno, todavía desconocido, mediante una función de beneficio o *payoff function*.
Después de conocerse el entorno al completo, es decir, que se haya recorrido toda la serie y realizado todas las predicciones, se aglutinan los resultados individuales en una función compuesta, p.ej. una media, para obtener la aptitud de cada individuo. Las máquinas que obtengan una aptitud por encima de la media pasan a convertirse en progenitores de la siguiente generación.

Como ejemplo introductorio, Fogel postuló el problema de clasificación de primalidad de los números naturales. Para el entorno, se utilizó una secuencia de dígitos binarios que representaban la primalidad de cada número natural en orden creciente. A modo de ejemplo, para una secuencia de 10 dígitos, el entorno se representa con la cadena binaria 1110101000. En condiciones normales, la comprobación de primalidad de un número es directa, pero la predicción de este resultado en base a una secuencia de números observados no pareció a Fogel un problema trivial[14].
El algoritmo aprendió a reconocer múltiplos de dos y tres como números no primos, habiendo también evidencias de algún tipo de reconocimiento de múltiplos de cinco. Fogel, más tarde, remarcó que el programa había descubierto satisfactoriamente las inherentes propiedades cíclicas de la primalidad, sintetizando una definición casi completa de primalidad sin conocimiento previo. Fogel esperaba que el programa no fuera un predictor perfecto de primalidad, ya que ninguna máquina de estados finitos puede representar la primalidad de forma total.

En 1992, se realizó la primera conferencia de Programación Evolutiva. Justo antes de esta conferencia, se dieron los primeros contactos entre las comunidades de PE y EE.EE. Era aparente para las dos comunidades, que a pesar de desarrollar sus técnicas de forma independiente, estaban recorriendo caminos muy similares en la simulación del proceso de evolución. Es por esto que, desde entonces, los miembros de las EE.EE participaron en las sucesivas conferencias de Programación Evolutiva.

2.1.3. Algoritmos Genéticos

Los Algoritmos Genéticos (AAGG) fueron ideados en los años 60 por John H. Holland. [37]
En 1975, Holland presentó el marco teórico que sentaría la base de este nuevo campo de investigación en *Adaptation in Natural and Artificial Systems*[21]. Más adelante, David E. Goldberg aplicó este marco teórico para la resolución de problemas de gestión de gaseoductos en *Computer-aided gas pipeline operation using genetic algorithms and rule learning*[17].
La interacción creciente durante las décadas sucesivas entre investigadores de diferentes ramas de la Computación Evolutiva, ha empezado a desdibujar las fronteras entre AAGG, EE.EE., PE y otras propuestas evolutivas y actualmente el término *algoritmo genético* se utiliza para designar un concepto más amplio al presentado originalmente por Holland[37].

El propósito principal de Holland era el estudio en profundidad del fenómeno de adaptación de los organismos vivos tal y como ocurre de forma natural. Holland justifica este estudio como una herramienta de resolución de problemas de optimización de gran complejidad y con incertidumbre como son la biología genética, la planificación económica, diseño de sistemas de control, psicología fisiológica, teoría de juegos o inteligencia artificial. Estos problemas suelen tener una búsqueda incierta de la solución óptima que suele forzar a la aceptación de soluciones provisionales subóptimas. Todos estos problemas tienen en esencia las mismas limitaciones de *no linealidad*, *máximo local* e *interacción entre variables* para los que Holland propone un nuevo marco que permita unificarlos en un problema más general: el proceso adaptativo[21].
En un principio, Holland se enfoca en el componente genético, el factor que determina las características de un organismo. Todo genotipo se compone de unas unidades mínimas llamadas genes, que aparecen en forma de variantes llamadas alelos. Estos alelos permiten asociar a cada gen unas características flexibles. Pudiendo contener un ser vivo miles de genes con varios alelos, la verdadera complejidad de estos sistemas genéticos no viene del espacio combinacional resultante sino de las interacciones que se producen entre las expresiones de estos genes. Debido al *efecto epistático*, que hace que los efectos de estas expresiones no sean aditivos, conocer el alelo responsable de una mayor eficiencia del fenotipo de un organismo supone una dificultad. La adaptación mediante cambios en la estructura genética se convierte en una búsqueda de *conjuntos de alelos coadaptados* dependientes en gran parte de las características del entorno[21].
Holland elige una representación lineal para el genotipo de las soluciones. Cada solución forma una secuencia de longitud fija de genes. Para la representación de los alelos, se elige un alfabeto finito que contenga todas las variantes contempladas. A modo de ejemplo, en una representación binaria se utiliza el alfabeto $\{0, 1\}$. Para una longitud l, una solución A se representa como:

$$A = (a_1, a_2, a_3, \ldots, a_l) = a_1 a_2 a_3 \ldots a_l$$

Posteriormente, se ha justificado la elección de esta longitud fija dentro de los AAGG de acuerdo a ventajas deseables en el proceso de diseño de soluciones. Con una longitud fija, toda solución tiene una interpretación asegurada, ya que se facilita un modelo sencillo para obtenerla. Además, esta restricción en la longitud garantiza que cualquier operación realizada obtenga soluciones válidas de la misma longitud. Estos dos requisitos reciben las denominaciones de *propiedad de buena definición* y *propiedad de cierre*[24].

El proceso adaptativo se compone de varios elementos que son las *estructuras genéticas posibles*, el *entorno* del sistema sometido a adaptación, el *plan adaptativo* encargado de obtener un *conjunto de operadores* que actúe sobre las estructuras y una *medida de aptitud* de las estructuras para el entorno dado que permita orientar la búsqueda hacia mejores soluciones. [21]
La aptitud de cada individuo de una población está claramente relacionada con la influencia que tiene sobre el desarrollo futuro de esa población. Un plan adaptativo que tenga como base una relación de proporcionalidad entre la influencia y la aptitud recibe el nombre de *plan reproductivo*. Si los descendientes de una población son simples copias de sus progenitores, se preserva la aptitud de los progenitores, pero no hay ninguna posibilidad de mejora. Si los descendientes se obtienen únicamente mediante pura variación aleatoria se puede obtener un gran número de nuevas variantes, pero no existe ninguna garantía de que estas nuevas variantes avancen en la misma dirección que las ya obtenidas. Debido a este problema, los planes reproductivos necesitan una selección adecuada de operadores genéticos que permitan una búsqueda fructífera y sostenible. [21]

Siguiendo el principio de imitación de la naturaleza, la única manera de obtener futuras poblaciones es por medio de la reproducción de los individuos de la población actual. Estos individuos sirven para mantener una representación de los valores ya observados y actúan como la principal fuente de variación. Conocer el estado de la población permite determinar su futuro sin más información adicional puesto que existe la misma influencia sobre el futuro para dos sistemas que acaben convergiendo en el mismo estado. Para obtener la nueva generación, se crea una población auxiliar mediante la copia de individuos de la población actual en un número proporcional a la aptitud determinado de forma estocástica. De esta manera, los conjuntos de alelos coadaptados que aparecen en individuos de aptitud por encima de la media ocuparán una mayor proporción dentro de la población sucesiva. A esta nueva población auxiliar, se le aplican los operadores genéticos y se obtiene la nueva generación. Los operadores genéticos son los encargados de alterar y redistribuir alelos dentro de una población. Se deben diseñar estos operadores de tal manera que se evite separar alelos cercanos entre sí, intentando que estos *conjuntos de alelos coadaptados* funcionen como unidades de transferencia.

Inicialmente, Holland presenta tres operadores genéticos: el cruce, la mutación y la inversión.
El cruce biológico es un proceso que obtiene recombinación de alelos mediante el intercambio entre pares de cromosomas. En el cruce propuesto por Holland, el *cruce basado en un punto*, se eligen dos soluciones para las que se escoge un número aleatorio x entre $[1, l-1]$ que servirá para marcar el punto tras el que se intercambian los genes de una solución con la otra. Esta selección del punto de cruce permite que la separación de genes cercanos entre sí sea menos probable, induciendo a enlaces de genes que tendrán más posibilidad de retenerse cuanto más cortos sean. [21]

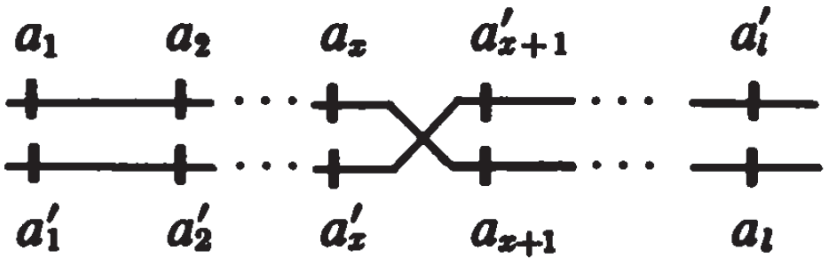


Fig. 2: Cruce propuesto por Holland[21]

La mutación biológica es un proceso espontáneo en el que un alelo de un gen se ve reemplazado por otro, dando lugar a una nueva estructura. Para el operador de mutación, se eligen de forma aleatoria varios puntos $x_1, x_2, \ldots x_h$ dentro de la solución que sufrirán un cambio bajo una probabilidad.

La generación de nuevas estructuras es una labor llevada a cabo principalmente por el operador de cruce. Sin embargo, el operador de cruce solo tiene la capacidad de reorganizar alelos presentes en la población. La actuación del cruce es dependiente, por tanto, de la generación inicial de estructuras o del grado de conservación de los diferentes alelos. En caso de que la población ya no contenga un determinado alelo es el momento oportuno para la acción del operador de mutación. En una situación de convergencia prematura en óptimo local, la mutación puede generar combinaciones inalcanzables para el cruce que permitan la continuación de la búsqueda. Si la probabilidad de mutación es lo suficientemente baja, la presente tendencia de las soluciones más aptas a ser predominantes en la población se mantiene inalterada a la vez que se permite la entrada de nuevas y mejores estructuras generadas por este operador. [21]
Finalmente, el operador de inversión altera los enlaces entre genes intercambiando el orden en el que aparecen dentro de la solución. Por medio de esta operación, las soluciones resultantes siempre son equivalentes a las originales, cambiando solamente la posibilidad de separación en el cruce de varios tipos de genes. La inversión requiere representación binominal para los genes de forma que, de modificarse su posición, se pueda diferenciar el propósito del contenido genético $\delta_i(A)$ mediante un identificador del gen al que pertenece. [21]

De esta manera, una solución A previamente representada como:

$$A = (a_1, a_2, a_3, \ldots, a_l)$$

se representa como:

$$A = ((1, a_1), (2, a_2), (3, a_3), \ldots, (l, a_l))$$

Para este operador, se elige una solución de la población y se escogen dos números aleatorios x_1 y x_2 para designar los extremos del intervalo de genes que se va a invertir. Dada una solución $A = 10011$ y unos números $x_1 = 1$ y $x_2 = 5$, la inversión realiza la siguiente operación:

$$A = (1, 1), (2, 0), (3, 0), (4, 1), (5, 1)$$

$$\Downarrow$$

$$A = (1, 1), (4, 1), (3, 0), (2, 0), (5, 1)$$

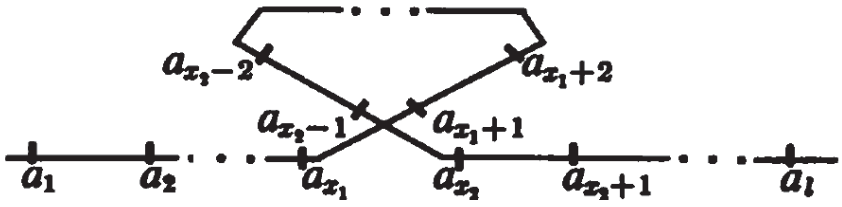


Fig. 3: Inversión propuesta por Holland[21]

Los planes reproductivos contribuyen a que conjuntos de alelos coadaptados presentes en individuos con una aptitud por encima de la media ocupen una proporción creciente de las poblaciones sucesivas. El incremento de esta proporción se hace visible en el curso de varias generaciones siempre que exista una correlación entre este conjunto de alelos y una mayor aptitud[21]. Para describir este proceso en el que el plan va encontrando los conjuntos de alelos responsables de incrementos generales de la aptitud surge la *hipótesis de los bloques de construcción*[49].
Para formalizar esta hipótesis, Holland presenta el concepto de *esquema*. Un esquema es un subconjunto del espacio de soluciones definido por valores en común de determinados genes. Para la representación del esquema, los valores no comunes de estas soluciones se representan con el símbolo □, un valor indefinido entre los disponibles.

Para una solución con representación binaria y longitud 3, existen las siguientes combinaciones de soluciones:

$$A_0 = 000, A_1 = 001, A_2 = 010, A_3 = 011$$
$$A_4 = 100, A_5 = 101, A_6 = 110, A_7 = 111$$

Para este espacio de búsqueda, se pueden obtener los siguientes esquemas:

$$A_4, A_5, A_6, A_7 \in \xi_1 = 1\square\square$$
$$A_0, A_1, A_2, A_3 \in \xi_2 = 0\square\square$$
$$A_0, A_1 \in \xi_3 = 00\square$$
$$A_3, A_7 \in \xi_4 = \square 11$$

Con el concepto de esquema se observa que, con la evaluación de una solución, la búsqueda obtiene información de todos los esquemas a los que esta solución pertenece, permitiendo al plan avanzar en la dirección adecuada. Holland denomina a este fenómeno *paralelismo intrínseco*.

Para el aprovechamiento óptimo de este paralelismo intrínseco, la representación de soluciones elegida supone un factor importante. El número de esquemas a los que puede pertenecer una solución es directamente proporcional al número de genes contenidos en la solución.

Esto se puede comprobar mediante la siguiente comparación de representaciones:

- 6 genes con 10 alelos dan lugar a un espacio de soluciones de 10^6 con un espacio de esquemas de $11^6 = 1,77 \cdot 10^6$. Cada solución puede formar parte de $2^6 = 64$ esquemas.

- 20 genes con 2 alelos dan lugar un espacio de soluciones similar al anterior con $2^{20} = 1,05 \cdot 10^6$ soluciones, a la vez que pueden existir $3^{20} = 3,48 \cdot 10^9$ esquemas. En cambio, cada solución puede formar parte de tantos esquemas como soluciones hay en el espacio, $2^{20} = 1,05 \cdot 10^6$.

Para explicar la diferencia entre las dos representaciones anteriores, Holland describe el *principio de los alfabetos mínimos*[49]. Este principio afirma que la utilización de una representación con menos alelos obliga a la utilización de un mayor número de genes para ocupar espacios de búsqueda similares. Esta mayor presencia de genes en una solución produce un aumento en el número de esquemas a los que puede pertenecer su representación. Debido a este principio, los AAGG han adoptado la representación binaria como su estándar.

2.1.4. Programación Genética

La idea sobre la que se fundamenta la Programación Genética es la de la generación mediante un proceso de evolución de programas que tengan la capacidad de resolver problemas sin importar su complejidad. Esta idea inicial precede históricamente a la propia Computación Evolutiva.

La primera instancia escrita de esta idea se puede encontrar en *Computing Machinery and Intelligence* de Alan Turing[46]. Esta es la primera publicación en la que aparece el conocido como Test de Turing, un procedimiento con el propósito de generar interacciones humano-máquina que sean indistinguibles de interacciones humanas para un observador externo.
Para la resolución de un problema tan complejo como el comportamiento adulto humano, Turing decide analizar el proceso que desarrolla el estado mental de una persona adulta. La simulación de este proceso debe partir del estado mental inicial, *el nacimiento*, e integrar la educación formal recibida, además de experiencias aprendidas en otros ámbitos.
Para no hacer demasiado evidente el razonamiento que debe tomar el observador, se desea permitir la posibilidad de que la máquina se construya de tal manera que los propios creadores no sean capaces de describir su modo de funcionamiento a un nivel de detalle suficiente. Por último, se menciona la necesidad de un factor de aleatoriedad, ya que una comprobación iterativa de todas las posibles combinaciones de soluciones puede suponer un problema intratable.

Tres décadas después, Richard Forsyth presenta *BEAGLE - A Darwinian Approach to Pattern Recognition* (1980) [15], un algoritmo evolutivo que utiliza para sus soluciones candidatas la representación de árboles. Forsyth afirma obtener buenos resultados con esta implementación.
El problema utiliza una base de datos de casos médicos. Cada caso contiene resultados de pruebas realizadas al paciente. Además, cada caso contiene también el destino vital del paciente, si al final el paciente se había curado o había fallecido. Para los resultados de pruebas se utilizan valores numéricos, pero para el destino vital del paciente se utilizan valores booleanos.
Conociendo todos estos datos, BEAGLE debe encontrar la regla que mejor relacione los resultados de las pruebas con el destino vital del paciente. El algoritmo contiene, bajo estándares actuales, tres operadores de mutación (GROW, SLIM y MUTATE), un operador de cruce (MATE) y un operador de control semántico (TIDY), que elimina redundancias y expresiones que obtienen resultado no booleano para el destino vital del paciente.
Los operadores de mutación modifican la estructura de los árboles de forma aleatoria, mediante la adición de nodos, la eliminación de términos o subexpresiones, el intercambio de subárboles y la alteración de operadores y operandos.
Cada generación consta de tres fases: *crecimiento*, *reproducción* y *exploración*.

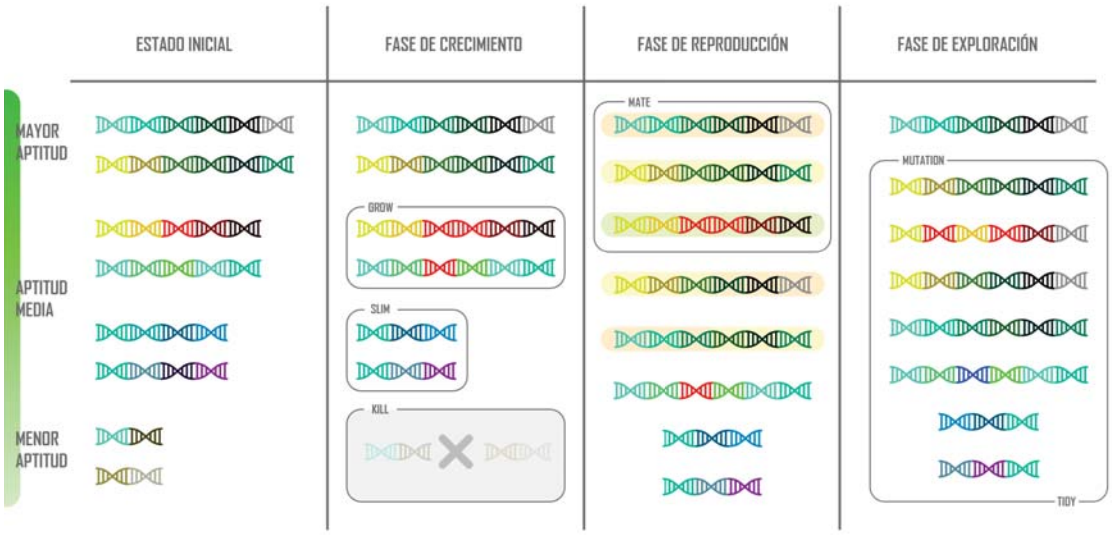


Fig. 4: Fases de una iteración del algoritmo BEAGLE

Como se puede ver en la Fig. 4, la utilización de operadores se realiza en función del ranking de las soluciones dentro de la población:

1. La fase de crecimiento comienza eliminando el cuarto menos apto de la población mediante el procedimiento KILL. El segundo cuarto menos apto de la población sobrevive, pero se somete al operador SLIM, que disminuye su estructura, simulando la falta de alimento de estas soluciones. El segundo cuarto más apto recibe el trato contrario mediante el operador GROW, que aumenta la estructura de estas soluciones. El cuarto más apto de la población no recibe ninguna modificación por parte de estos operadores, ya que se considera que son lo suficientemente buenas.

2. En la fase de reproducción, se selecciona la mitad más apta de la población superviviente para restaurar el tamaño de población inicial.

3. En la fase de exploración, se realiza una mutación más agresiva a los individuos por debajo del octavo más apto de la población utilizando el operador MUTATE, que puede generar estructuras no válidas que serán corregidas por el operador TIDY.

La idea de evolucionar programas se encontraba presente entre los estudiantes de John Holland. En la primera conferencia de AAGG en 1985[7], Nichael Cramer publicó dos lenguajes de representación de programas. Estos lenguajes utilizaban las instrucciones de la Fig. 5.

```
(:INC VAR) ;; add 1 to the variable VAR
(:ZERO VAR) ;; set the variable VAR to 0
(:LOOP VAR STAT) ;; perform the statement STAT VAR times
(:GOTO LAB) ;; jump to the statement with label LAB
```

Fig. 5: Instrucciones disponibles en el lenguaje JB[7]

El primer lenguaje, JB, representaba soluciones como secuencias de instrucciones. Cada instrucción tenía como primer elemento un identificador de operación y dos argumentos, de tipo variable o referencia a otra instrucción.

```
(0 0 1 3 5 8 1 3 2 1 4 3 4 5 9 9 2)
```

(a) Representación de la solución

```
(0 0 1) ;;main statement-> (:BLOCK AS_0 AS_1)
(3 5 8) ;;auxiliary statement 0 -> (:ZERO V_5)
(1 3 2) ;;auxiliary statement 1 -> (:LOOP V_3 AS_2)
(1 4 3) ;;auxiliary statement 2 -> (:LOOP V_4 AS_3)
(4 5 9) ;;auxiliary statement 3 -> (:INC V_5)
```

(b) Interpretación de la solución

Fig. 6: Ejemplo de solución JB[7]

Debido a esta estructura no jerárquica, este lenguaje tiene un problema de generación de bucles entre instrucciones. Cada programa cuenta con una instrucción inicial cuyo identificador de operación puede modificarse accidentalmente mediante una mutación que impida ejecutar más instrucciones.

Para resolver estos problemas, se diseña el segundo lenguaje, TB, que pasa a representar las soluciones con una estructura de árbol en la que cada instrucción contiene las instrucciones a las que hace referencia, garantizando que no se generen los bucles anteriormente mencionados.

```
(0 (3 5) (1 3 (1 4 (4 5) ) ) )
```

Fig. 7: Solución adaptada al nuevo lenguaje TB[7]

Para evitar el problema anterior de destrucción de instrucciones principales, se limita la actuación de la mutación a los elementos hoja y a las instrucciones que los contienen.

A finales de los 80, John R. Koza presentó dos publicaciones[24][25] en torno a la utilización de una representación de árbol dentro de los AAGG para la evolución de programas.
En la primera de ellas, *Non-Linear Genetic Algorithms for Solving Problems*, Koza analiza aspectos de la representación más convencional de los AAGG: la cadena binaria de longitud fija.
Esta representación, según Koza, contiene varias limitaciones debido a algunos aspectos de su diseño.
La primera limitación tiene que ver con la longitud fija de las cadenas. Como ya hemos comentado anteriormente, una cadena de longitud fija supone unas ciertas garantías de estabilidad en la representación, el cumplimiento de las propiedades de *buena definición* y *cierre*. Sin embargo, esta limitación también conlleva una simplificación arbitraria de las soluciones utilizadas. En problemas en los que se necesita conocer información de muchas iteraciones, el diseñador del algoritmo está obligado a descartar posible información relevante debido a la inviabilidad de que la representación de soluciones pueda crecer indefinidamente. Esta simplificación limita la complejidad de las soluciones hasta el punto que la solución óptima que se desea obtener podría no ser representable. Esta incapacidad de representación de soluciones contrasta fuertemente con la intención de que toda solución tenga una interpretación, la propiedad de buena definición.
Existen otras representaciones que pueden cumplir la propiedad del cierre dado un diseño adecuado de operadores que mantengan las soluciones sintácticamente correctas. Dados los efectos adversos anteriores, resulta relevante considerarlas.
La segunda limitación viene de la representación binaria y del concepto de *paralelismo implícito* de Holland, inicialmente paralelismo intrínseco. La utilización de un espacio de búsqueda unidimensional limita la resolución de problemas a todos aquellos para los que se pueda realizar una representación binaria de forma viable y ninguna mejora en la eficiencia de la búsqueda es más prioritaria que su implementación.
La resolución de la mayoría de problemas complejos recae frecuentemente en diseñar relaciones jerárquicas entre el problema que se desea resolver y otros subproblemas más sencillos, tarea que la representación de cadena de caracteres dificulta en gran medida. Para la resolución de tales problemas de una forma realista es necesario que no se imponga ninguna limitación a priori de tamaño, forma o complejidad a las estructuras de datos utilizadas.
En la segunda publicación, *Hierarchical Genetic Algorithms operating on populations of computer programs*, Koza describe en mayor profundidad los aspectos relacionados con la nueva representación de árbol. Se presentan las soluciones del nuevo algoritmo como árboles representados por una expresión LISP que se genera desde la raíz hasta las hojas conteniendo funciones y otros elementos. El conjunto de soluciones disponibles para cada problema dependerá siempre del conjunto de expresiones relevantes para el dominio concreto del problema elegido.
Para comparar la eficiencia entre el uso de esta nueva representación y la representación convencional se analizan los factores relacionados con la *hipótesis de los*

bloques de construcción y la formación de esquemas. En esta nueva representación, los esquemas se definen como árboles que contienen elementos con valor específico o valor indeterminado. El conjunto de subárboles derivados de los valores indeterminados es infinito, pero este conjunto se puede hacer finito fijando el número de elementos contenidos en estos subárboles. De esta forma, la nueva representación hace uso del paralelismo implícito similar al de la representación original de Holland utilizando como bloques de construcción los subprogramas contenidos en soluciones con alta aptitud. La búsqueda tiende a concentrarse en subestructuras cada vez más pequeñas que pueden dar lugar a expresiones con cada vez más aptitud.
Koza considera que estos nuevos AAGG jerárquicos son una extensión natural de los lineales basados en cadenas. En los AAGG convencionales, cualquier caracter de la representación se considera análogo a los elementos básicos del ADN, los nucleótidos. Para el procesamiento de la aptitud media de los correspondientes esquemas se utiliza la aptitud observada para esta información pasiva. En cambio, en el procesamiento de la aptitud media de los esquemas para los AAGG jerárquicos contribuye la acción proactiva de su expresión, análoga a la labor de una proteína dentro de una célula, de manera que se emplea en esencia la misma técnica para reconocer los genes responsables de mayores aptitudes.

En 1992, Koza publicó el libro *Genetic programming: On the programming of computers by means of natural selection*[27], considerado como el punto de inicio de la Programación Genética. Las soluciones en Programación Genética (PG) se representan mediante una jerarquía de funciones, como podemos ver en la Fig. 8. Cada una de estas funciones se representa como un subárbol cuya raíz es el operador y los elementos derivados de ella son sus argumentos, que pueden contener otras funciones.

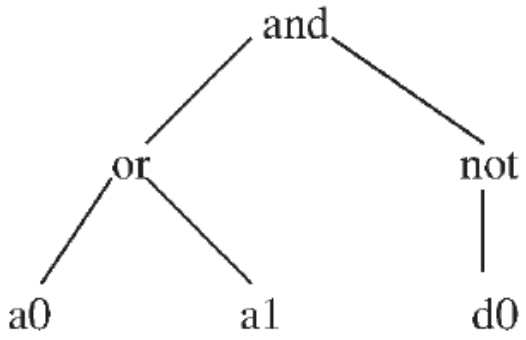


Fig. 8: Representación de la expresión (a_0 OR a_1) AND $\bar{d}_0$[48]

Para el planteamiento de esta nueva representación jerárquica, se trata nuevamente la propiedad del cierre. Koza plantea el cumplimiento de esta propiedad como un problema a resolver en la PG, el *problema del cierre*. Este problema se resuelve cuando se cumple dicha propiedad, es decir, cuando el algoritmo solo utilice soluciones sintácticamente válidas.

Para la mayoría de representaciones, Koza afirma que el problema del cierre se soluciona desde la generación mediante el diseño exhaustivo de las funciones utilizadas debiendo cada función gestionar todo valor que se pueda obtener de cualquiera de ellas. Por último, afirma que la resolución del problema del cierre es algo deseable pero que no debe convertirse en un requisito para la implementación, ya que no existe hasta la fecha ningún método generalizado en torno a este problema.

Además, Koza presenta tres métodos de inicialización de las soluciones: Full, Grow y Ramped Half-and-Half. El método Full genera árboles de forma aleatoria con todos sus caminos hasta las hojas de una longitud igual a una profundidad máxima. Todos los caminos tienen la misma longitud, limitando el método la selección aleatoria a elementos no terminales hasta que se llegue a la profundidad máxima, donde solo se eligen terminales.
El método Grow está más enfocado a la variabilidad de formas permitiendo la selección de cualquier elemento, terminal o no terminal, hasta llegar a la profundidad máxima donde solo se pueden elegir terminales.
El método Ramped Half-and-Half genera una distribución homogénea de subárboles de una profundidad mínima de 2 hasta la profundidad máxima. Una vez obtenidos estos subárboles, se aplica a cada mitad de los elementos hoja generados un método diferente de entre los anteriores Full y Grow hasta formar completamente el árbol.

Por último, Koza presenta una gran variedad de operadores que modifican los árboles de varias maneras, siendo los más destacables a lo largo del tiempo el cruce y la mutación. El cruce que presenta Koza realiza un intercambio de subárboles elegidos de entre los dos progenitores de forma aleatoria mediante una distribución uniforme, ocupando cada árbol la posición del otro en los nuevos descendientes de una forma similar a la utilizada en los AAGG de Holland.
La mutación se realiza mediante la selección aleatoria de un nodo cualquiera del árbol. A continuación, se elimina el subárbol correspondiente y se procede a generar de forma aleatoria un nuevo subárbol. Este proceso tiene paralelismos con los métodos de inicialización estando limitado por el mismo parámetro de profundidad máxima. Koza resta importancia al operador de mutación, ya considerado algo secundario en los AAGG de Holland. La desaparición total del alelo de un gen en la búsqueda representa una situación mucho menos importante puesto que ya no se limita el intercambio de representación entre varios genes, produciéndose el intercambio de alelos a lo largo de toda la representación.

2.2. Operadores en Programación Genética

Dado que la PG no es más que una especialización de los AAGG, los algoritmos desarrollados dentro de estos dos campos de investigación suelen fundamentarse en un marco común de cuatro operaciones básicas:

2.2.1. Inicialización

Cada algoritmo comienza con una *generación inicial* que pone el punto de partida de la búsqueda a efectuar. Una vez inicializada la población, se da paso a un *ciclo de generaciones* que termina cuando se cumple una condición de parada, como se puede ver en la Fig. 9. La condición de parada puede darse en un número de generaciones, cuando se llegue a una solución objetivo o si determinadas variables de progreso se mantienen estables durante ciertas generaciones.

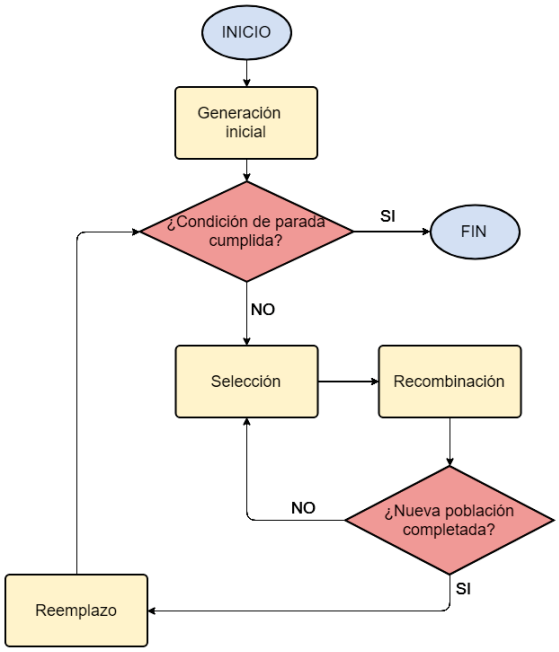


Fig. 9: Diagrama de flujo resumen de la ejecución

Tiempo después de la presentación de los métodos originales de Koza, varios investigadores proponen nuevos métodos que solucionan algunas de sus dificultades.

Iba[22] realiza un experimento con el método GROW y descubre que este realiza una generación no uniforme de árboles distribuidos en su mayoría en los extremos del intervalo de tamaño permitido. Para resolver este problema, presenta como mecanismo de generación uniforme el método RAND-Tree que utiliza un lenguaje de Dyck para representar relaciones de filiación entre nodos.

Bohm y Geyer-Schulz presentan un nuevo método conocido como Uniform[31] que utiliza tablas precomputadas para una generación exhaustiva de árboles para un intervalo factible de nodos. Se selecciona un nodo mediante una distribución uniforme derivada de estas tablas. Si se obtiene un nodo no terminal, se asignan tamaños de subárboles para cada hijo y se realizan las respectivas llamadas recursivas hasta completar el árbol.

Chellapilla[6] presenta un nuevo método de inicialización para árboles en Programación Evolutiva conocido como RandomBranch[31]. Este método está orientado al control del número de nodos y tiene un muestreo uniforme de subárboles. El tamaño que tiene el programa se elige aleatoriamente en un intervalo acotado por una *longitud* máxima. El algoritmo utilizado para este método es el de la Fig. 10.

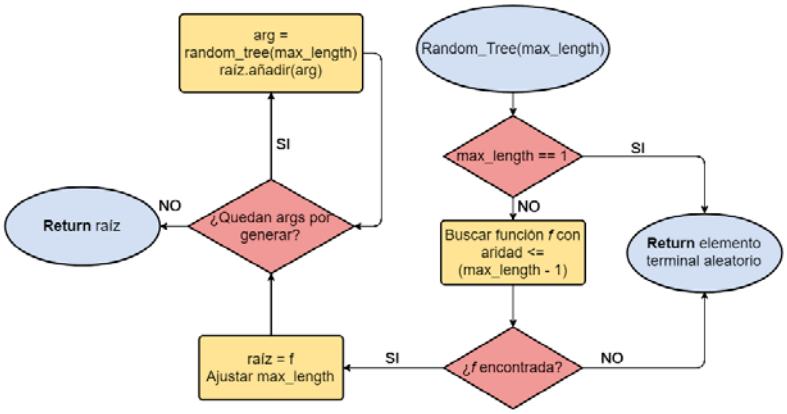


Fig. 10: Algoritmo RandomBranch

Para este algoritmo, se busca una función f con una aridad que no supere la longitud máxima. De haberla, se asigna esta función a la raíz y se actualiza la longitud máxima como $max\_length = \frac{max\_length-1}{f.aridad}$. Posteriormente, se realizan llamadas recursivas con la nueva longitud hasta generar todos los argumentos de la función principal.

Sean Luke[30] analiza el método GROW y encuentra algunos problemas que le sirven para desarrollar los métodos PTC1 y PTC2. Estos dos métodos se describen brevemente mediante las figuras 11 y 12. Ambos métodos tienen un tamaño esperado de árbol y cada elemento tiene una probabilidad de selección q y una aridad b[31].

PTC1 se inicia obteniendo la probabilidad de selección de elementos no terminales frente a terminales para mantener el tamaño esperado. Esta probabilidad puede ser precalculada para un tamaño de árbol E_{tree} y un conjunto de no terminales N, tal que:

$$p = \frac{1 - \frac{1}{E_{tree}}}{\sum_{n \in N} q_n b_n}$$

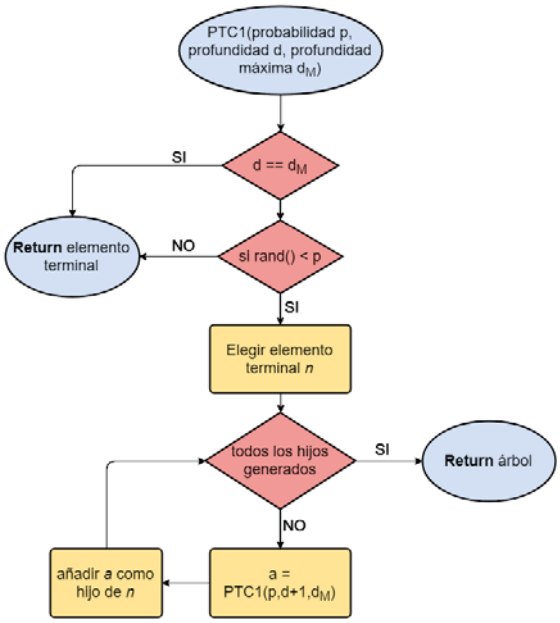


Fig. 11: Algoritmo de inicialización PTC1

PTC2 realiza un procedimiento recursivo como el método anterior. Esta versión utiliza una cola como método de control de individuos por añadir para no exceder el tamaño esperado. Con este método, se garantiza un tamaño final de árbol comprendido entre la aridad máxima de los elementos no terminales y el tamaño esperado. Para cada elemento que se añade a la cola se determina una posición en el árbol enlazada al nodo padre del que se deriva.

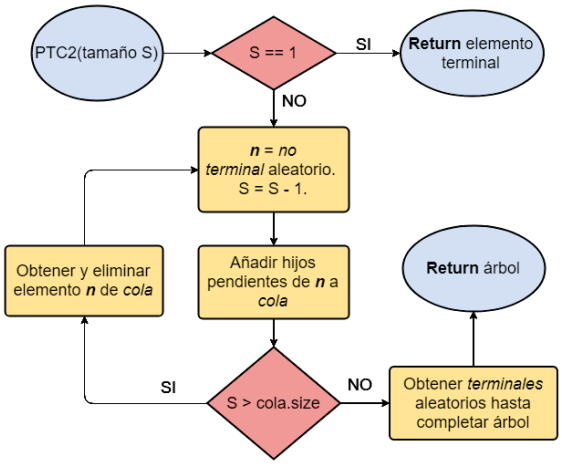


Fig. 12: Algoritmo de inicialización PTC2

2.2.2. Selección

Cada generación del algoritmo comienza con el operador de *selección*, el operador encargado de elegir los individuos que forman parte del proceso de recombinación. La selección original para estos algoritmos es la selección proporcional a la aptitud también denominada *selección por ruleta*. La selección de un individuo se lleva a cabo mediante la obtención de un número aleatorio en el intervalo $[0, \sum_i f_i]$, siendo $\sum_i f_i$, la suma de aptitudes de toda la población. De esta manera, en cada selección se obtiene un individuo i de aptitud f_i con la probabilidad de selección $p = \frac{f_i}{\sum_i f_i}$.
James Baker[32] presentó una variante de la selección por ruleta denominada *muestreo universal estocástico* (SUS). Esta selección obtiene los N individuos necesarios para la nueva población de descendientes mediante una división uniforme de la ruleta. Para determinar la primera sección de la selección, se calcula un número aleatorio entre 0 y $\frac{\sum_i f_i}{N}$. Posteriormente, para calcular las siguientes posiciones hasta llegar a la suma total de aptitud, se suma de forma iterativa $\frac{\sum_i f_i}{N}$ a la posición inicial. Esta variante permite una mayor eficiencia realizando solo un cálculo por cada generación. Además, garantiza que se dé selección del individuo más apto al menos una vez, algo que no ocurre en la ruleta convencional.

La selección por ruleta se mantiene eficiente siempre que la relación numérica entre las aptitudes sea de la proporción correcta. Sin embargo, en situaciones de convergencia en la que los valores de aptitud son muy similares, la selección tiende a aproximarse a una búsqueda aleatoria. Por el contrario, si las soluciones más aptas tienen un valor de aptitud desproporcionado con el resto, el proceso se convierte en una selección determinística del más apto. Para estos casos es conveniente utilizar una selección que trate la aptitud como una medición comparativa más difusa, para lo que aparecen los métodos como la selección truncada o la selección por torneo.
La *selección por torneo* elige el individuo más apto de entre t individuos seleccionados aleatoriamente. Esta selección está dotada de gran popularidad por su gran versatilidad ya que no es dependiente de la aptitud, no requiere ningún cálculo previo y funciona bien con algoritmos paralelos. Por último, el ajuste de la presión de selección no supone ninguna dificultad ya que todo el proceso depende en última instancia del parámetro t. Para un valor $t = 1$, la selección es una búsqueda aleatoria, siendo común elegir un valor superior a este.

Además de estos diferentes métodos de selección, existen opciones dentro de los AAGG en cuanto al procedimiento de generación de sucesores con aplicación en la PG, como los *AAGG de estado estacionario*, una variante que genera una cantidad pequeña de sucesores independiente del número de progenitores[32].

2.2.3. Recombinación

La *recombinación* es el proceso que genera nuevos individuos dentro del ciclo de generaciones. Es la parte del algoritmo más dependiente de la representación. Este proceso utiliza fundamentalmente el operador de cruce, en el que se utilizan dos

progenitores para dar lugar a dos descendientes. Para cada descendiente obtenido, se da una baja probabilidad de mutación en la que un elemento de la representación cambia aleatoriamente.
Al contrario que otros algoritmos de Computación Evolutiva[32], los AAGG y la PG convencionalmente seleccionan individuos por parejas para inmediatamente ser recombinados, realizando estos dos procesos de forma concurrente como se refleja en la Fig. 9.

A pesar de que el *cruce Koza* se convirtió en el cruce convencional para el campo de la Programación Genética, posteriores investigaciones encontraron varios efectos derivados de la utilización de este cruce que hacen la búsqueda un proceso más difícil en la práctica. Dada la libertad bajo la que se realizan los intercambios en este cruce, existe una tendencia de este operador a desagregar los subárboles que conforman los bloques de construcción. Esta desagregación evita la resolución completa del problema del cierre para problemas complejos, que disminuye también la velocidad de convergencia. Además, la falta de restricciones de intercambio de árboles de distinta longitud y profundidad provoca un crecimiento desmedido en las soluciones que produce una caída en el rendimiento. Este efecto es conocido en inglés como *code bloat* o *explosión de código*[34].

Para el problema de la desagregación de los bloques de construcción, D'haeseleer[12] presenta dos nuevos cruces *de preservación de contexto*. Ambos cruces utilizan un método de localización en base a coordenadas. Se distingue la denominación de cada cruce en función de la intensidad de restricción para el intercambio, de tal forma que tenemos un cruce *fuerte* (SCPC) y otro *débil* (WCPC).
Las coordenadas de este sistema se van formando desde la raíz, que representa el vacío, (). A medida que se van formando hijos de este elemento raíz, se va añadiendo a sus coordenadas el ordinal que corresponde a cada elemento, teniendo los hijos de la raíz las coordenadas $(1), \ldots (n)$. A su vez, los hijos de un elemento (k), se representan como $(k,1), \ldots (k,n)$.
Para un elemento elegido en el primer progenitor, el cruce fuerte de preservación de contexto solo permite el intercambio de otro elemento del segundo progenitor que posea las mismas coordenadas, mientras que el cruce débil permite intercambiar este mismo nodo del segundo progenitor además de cualquiera de sus nodos hijos.
Estos cruces imponen unas restricciones bastante severas ya que impiden los movimientos horizontales de subárboles. Esta limitación genera grandes problemas de diversidad que entorpecen la búsqueda ya que se produce un encapsulamiento de bloques de construcción comparable al que podrían tener los AAGG más convencionales con genes que no se pueden intercambiar entre sí y que deben seleccionarse y evolucionar de forma independiente. Como con los Algoritmos Genéticos, este encapsulamiento se puede contrarrestar mediante el uso del operador de mutación que facilita la reintroducción de subárboles perdidos.
D'haeseleer concluye en que la utilización conjunta del cruce Koza y el cruce fuerte de preservación de contexto mejora enormemente la eficiencia de la búsqueda.

Para la resolución del problema de explosión de código, Langdon[28][29] presenta operadores de cruce y de mutación *de tamaño justo*. Ambos operadores utilizan como referencia una longitud l que se obtiene del nodo elegido en el primer progenitor, además de una longitud máxima prefijada. La *mutación de tamaño justo* genera subárboles de forma aleatoria mediante dos distribuciones uniformes diferentes.
La primera de ellas, denominada *mutación 50 %-150 %*, elige una longitud para el nuevo subárbol dentro del intervalo $l \pm \frac{l}{2}$. Si $l + \frac{l}{2}$ superara el máximo prefijado, se elige otro nodo. Este método muestrea árboles en cuanto a su longitud en torno a l. Las longitudes con mayores combinaciones, las más largas, son también las más improbables, introduciéndose en esta distribución un sesgo implícito de parsimonia, la simplificación de estructuras superfluas.
En la segunda de ellas, la nueva longitud se elige aleatoriamente de entre los subárboles de la propia solución de la misma forma que el muestreo de la distribución anterior. Aunque la longitud promedio obtenida con estas dos distribuciones tiende a l, existe la posibilidad de obtener soluciones demasiado pequeñas desde las que no se pueda volver a crecer. Por esto, en caso de que la mutación elija un nodo raíz, el reemplazo se lleva a cabo mediante un método de generación inicial, el Ramped Half-and-half.
El *cruce de tamaño justo* solo permite cruzar el subárbol seleccionado con uno del otro progenitor que no tenga una longitud mayor de $l+2l$. Para elegir algún subárbol del segundo progenitor, se calculan todas las longitudes de los subárboles existentes y se utiliza una ruleta en la que tienen más ponderación los subárboles con una longitud más aproximada al original. [9]

Años después, Crawford-Marks y Spector[9] presentan modificaciones de los operadores de Langford para dar lugar al *cruce justo* y la *mutación justa*. La mutación funciona exactamente de la misma manera, salvo que el intervalo de longitud permitido es el de $l \pm \frac{l}{4}$.
El nuevo cruce elige aleatoriamente un subárbol del segundo progenitor y, si su longitud entra dentro del intervalo $l \pm \frac{l}{4}$, se procede al cruce. En caso contrario, se elige otro subárbol. En caso de no realizarse el cruce tras determinados intentos, se procede a elegir el que más se acerque al intervalo de todos los seleccionados.

2.2.4. Reemplazo

Una vez terminada la fase de recombinación, comienza el turno del operador de reemplazo. El operador de reemplazo es el encargado de preparar la población para la siguiente generación. El reemplazo tradicional de los AAGG y la PG elige la nueva población de descendientes tal y como se obtiene de la recombinación. Más adelante, aparecen nuevas variantes como la elitista o la de estado estacionario que introducen individuos de la población de progenitores dentro de la generación de descendientes, permitiendo retener las mejores soluciones. Esta nueva variante favorece fuertemente la explotación, aumentando el riesgo de convergencia prematura[32].

2.3. Programación Genética basada en gramáticas

Para resolver el problema del cierre es imprescindible que se garantice la generación de soluciones válidas[34]. Para ello, se adopta dentro de la PG el uso de gramáticas formales para la sistematización de la representación. La gramática utilizada sustituye a los conjuntos de funciones y argumentos utilizados hasta entonces en la PG convencional. Esta nueva estructura regulariza la integración de todos los elementos de la representación, eliminando la necesidad de un diseño exhaustivo en torno a los tipos de datos utilizados[3].

Una gramática generativa es un conjunto de reglas de derivación o *producciones* que describen las transformaciones necesarias para generar una palabra que pertenezca a ese lenguaje formal, una estructura sintácticamente válida. [19]
Toda gramática se define formalmente como la tupla $(N, \sum, P, S)$:

- **N** es el conjunto de elementos no terminales. Los elementos no terminales se representan en mayúscula y son los elementos que pueden formar parte de los antecedentes de las producciones.
- $\sum$ es el conjunto de elementos terminales, que se presentan en minúscula y no pueden derivarse en otros elementos.
- **P** es el conjunto de producciones.
- **S** es el elemento inicial desde el que comienzan las producciones.

2.3.1. Programación Genética de Gramáticas Libres de Contexto

En 1995, Whigham presenta en *Grammatically-based genetic programming*[48] una nueva variante que utiliza gramáticas libres de contexto como solución al problema del cierre. Una gramática libre de contexto es aquella para la que sus producciones se realizan sin importar el contexto en el que se encuentre su antecedente.
Whigham presenta la siguiente gramática $G = (N, \sum, P, S)$ para representar expresiones booleanas:

$$G = \begin{cases} N = \{S, B, T\} \\ \sum = \{and, or, not, if, a_0, a_1, d_0, d_1, d_2, d_3\} \\ P = \begin{cases} S \Rightarrow B \\ B \Rightarrow and\ BB \mid or\ BB \mid not\ B \mid if\ BBB \mid T \\ T \Rightarrow a_0 \mid a_1 \mid d_0 \mid d_1 \mid d_2 \mid d_3 \end{cases} \end{cases}$$

Con esta nueva gramática, se obtienen árboles como el de la Fig. 13, equivalentes al de la Fig. 8. Para esta nueva variante, la representación de las soluciones se interpreta sin necesidad de identificar relaciones jerárquicas entre elementos, concatenando los elementos terminales de izquierda a derecha.

Para el método de inicialización, Whigham propone un planteamiento similar al del método GROW de Koza, permitiendo un tamaño variable acotado por una longitud máxima. Además, se introduce variabilidad en esta longitud máxima de forma que cada individuo de la población se genere de forma diferente.
Antes de empezar el proceso de generación, se calcula el mínimo número de pasos de derivación para obtener una palabra de la gramática. Una producción del tipo $A \Rightarrow \beta$ tal que $\beta \in \sum^*$ tiene una longitud de uno.
Una vez calculadas estas longitudes, se genera el nodo inicial y se elige una producción para la que sus nodos no sobrepasen la longitud máxima prefijada para esta solución. Se realiza el mismo proceso para cada uno de los no terminales obtenido de esta primera producción hasta completar el árbol.

El operador de cruce propuesto elige de forma aleatoria un nodo no terminal del primer progenitor. A continuación, se busca otro nodo no terminal con el mismo símbolo dentro del segundo progenitor para proceder a realizar el cruce mediante el intercambio de los subárboles respectivos.
Finalmente, tenemos el operador de mutación. Este operador comienza eligiendo un nodo no terminal del árbol de forma aleatoria y se procede a eliminar los subárboles que tiene por debajo. A continuación, para una profundidad máxima fija, se realiza un procedimiento similar a la generación con el nodo como raíz desde la que comienzan las producciones.

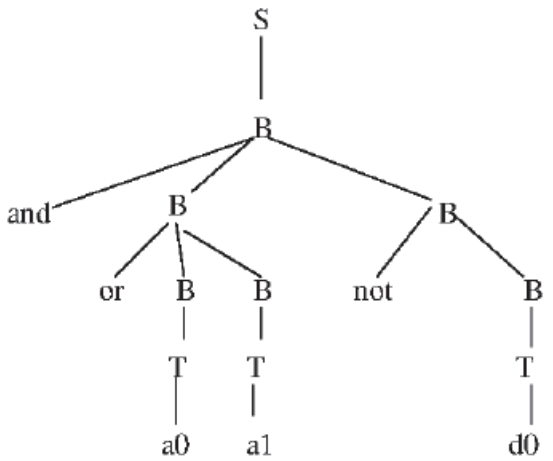


Fig. 13: Representación formada por una gramática generativa[48]

Con el paso de los años, la variante de Whigham se ha convertido en la principal de entre las que utilizan gramáticas, recibiendo el nombre de *Programación Genética Guiada por Gramáticas* (PGGG) en publicaciones sucesivas. [3][34]
El *cruce Whigham* es uno de los más utilizados en PGGG, llegando a convertirse en el estándar de facto debido a su simplicidad y eficiencia. [34]

En 2005, Manrique, Ríos y Rodríguez-Patón[33] proponen el nuevo *cruce basado en gramáticas* (GBX). Este cruce busca superar las limitaciones producidas por la restricción del cruce Whigham, que deja varias alternativas de selección sin explorar. El nuevo cruce consta de los siguientes pasos:

1. De la misma manera, el nuevo cruce comienza eligiendo de forma aleatoria un nodo no terminal del primer progenitor, el denominado *nodo de cruce*.
2. Para este nodo elegido se procede a explorar todas las producciones en las que el padre de este nodo es antecedente. Se almacenan en una lista R todos los consecuentes.
3. Se distingue la producción que genera el nodo de cruce como la *derivación principal* y se procede a calcular la 3-tupla $T(l, p, C)$:
   - l es la longitud de derivación, el número de caracteres que componen el consecuente.
   - p son las coordenadas del nodo.
   - C es el consecuente.
4. Se eliminan de R todos los consecuentes con longitud diferente a l.
5. Se eliminan todos aquellos que difieran de C en una posición que no sea la del nodo de cruce.
6. Se obtienen los símbolos en la posición del nodo de cruce de los consecuentes de R y se almacenan en el conjunto X, el conjunto de nodos no terminal que pueden ser elegidos para el cruce.
7. Se elige aleatoriamente un nodo del segundo progenitor que tenga un símbolo contenido en X.
8. Por último, se procede a intercambiar ambos subárboles.

En 2007, García-Arnau, Manrique, Ríos y Rodríguez-Patón[16] proponen un nuevo método de inicialización derivado del de Whigham, el *método de inicialización basado en gramáticas* (GBIM). En esta versión, solo hay una profundidad máxima única. Además, se introduce el concepto de longitud para cualquier elemento, permitiendo así conocer la *longitud mínima del axioma*, por ende la longitud mínima de las palabras pertenecientes a la gramática. El cálculo de las longitudes mínimas se realiza de la siguiente forma:

- $L(a) = 0 \mid a \in \sum$
- $L(A \Rightarrow a) = 1 \mid a \in \sum$
- $L(A \Rightarrow \alpha) = 1 + \text{máx}_{\beta \in \alpha} L(\beta) \mid \alpha, \beta \in \sum \cup N$
- $L(A) = \text{mín}\, L(A \Rightarrow \alpha)$

En 2007, Couchet, Manrique, Ríos y Rodríguez-Patón[8] presentan un operador de cruce y otro de mutación *basados en gramáticas*, GBC y GBM. Ambos operadores poseen mecanismos de control de profundidad de las soluciones que utilizan el cálculo de longitudes introducido en el método de inicialización GBIM.
El cruce GBC representa una nueva versión mejorado del cruce GBX. Este nuevo cruce realiza casi el mismo algoritmo que la versión original, con las siguientes modificaciones:

- El algoritmo comienza almacenando los elementos no terminales del primer progenitor en un conjunto NT. Si no se encuentra ningún elemento en NT para seleccionar como nodo de cruce CN1, se obtienen descendientes idénticos a los progenitores.
- El paso 7 del GBX se desdobla en los pasos 8 y 9 del GBC:

    8. Se elige un nodo no terminal de X.
        - En caso de que X esté vacío, se elimina el símbolo del nodo de cruce de NT y se vuelve al paso 2.
        - Se introduce en PN los nodos del segundo progenitor con el símbolo obtenido, en forma de coordenadas como las del cruce SCPC.
    9. Si PN está vacío, se elimina del conjunto X el símbolo elegido y se vuelve al paso 8. Sino, se elige aleatoriamente un elemento en este conjunto, que pasará a ser el nodo de cruce CN2.

Una vez completado el paso 9, el cruce continúa así:

10. Se calculan las nuevas profundidades de los dos subárboles resultantes. En caso de que alguna exceda el valor de profundidad máxima, se elimina CN2 de PN y se vuelve al paso 9.
11. Si los símbolos de CN1 y CN2 son idénticos, se procede a realizar el cruce.
12. Si son símbolos diferentes, se obtiene la derivación generada por el padre de CN2 y se intercambian los nodos CN1 y CN2 con sus subárboles.
13. Si la derivación obtenida coincide con algún consecuente de las producciones del padre de CN2, se procede finalmente a realizar el cruce intercambiando los subárboles. Sino, se elimina CN2 de PN y se vuelve al paso 9.

La mutación GBM comparte el mismo procedimiento que el cruce GBC hasta el paso 7, el del cálculo de X. En este operador, se utilizan las mismas variables y conjuntos que en el anterior, salvo para los nodos elegidos para la mutación que se denotan como MN. A partir del paso 7, el operador de mutación procede a realizar lo siguiente:

8. Si X está vacío, se elimina el símbolo del nodo de mutación MN de NT y se vuelve al paso 2. Sino, se elige un símbolo CS de forma aleatoria.

9. Se calcula la longitud de mutación permitida ML restando la profundidad de MN a la profundidad máxima.

10. Se asigna el valor 0 a la profundidad actual CD.

11. Se almacenan en PP las producciones P con el símbolo CS como antecedente, si $CD + L(P) \leq ML$. Si PP está vacío, se elimina el símbolo CS de X y se vuelve al paso 8.

12. Se elige de forma aleatoria una producción de PP.

13. Para cada no terminal obtenido, se obtienen producciones como en el paso 11 que cumplan que $CD+1+L(P) \leq ML$. Se realiza este paso de forma recursiva hasta obtener solo elementos terminales.

14. Finalmente, el subárbol generado se sustituye por el subárbol que tiene MN de raíz.

2.3.2. Otras variantes

En 1998, C. Ryan, J.J. Collins y Michael O'Neill[42] presentan el nuevo paradigma evolutivo denominado *Evolución Gramatical*. Este nuevo paradigma introduce una distinción de representación entre el genotipo y el fenotipo, la materialización de una idea parecida a la *dualidad ADN-proteína* planteada años antes. Se recupera una representación lineal con longitud variable para el genotipo que recuerda a los primeros antecedentes de la PG mientras que se mantiene la representación de árbol para el fenotipo. En esta variante, se utiliza el genotipo como una especie de semilla que determina las diferentes producciones que se llevan a cabo para generar el fenotipo[3].

En 2003, Hoai, McKay y Abbass[20] proponen una nueva variante derivada de la PGGG que utiliza *gramáticas de adición de árboles*. Las gramáticas de adición de árboles son un superconjunto de las gramáticas libres de contexto que utilizan mecanismos de reescritura de árboles. Para esta reescritura de árboles, estas gramáticas no utilizan producciones, sino *árboles elementales*, que pueden ser de tipo inicial o auxiliar[3].

Toda gramática de este tipo se define formalmente como la tupla (N, T, I, A, S).

- **N** es el conjunto de no terminales.
- **T** es el conjunto de terminales.
- Para los árboles elementales, **I** representa el conjunto de los iniciales y **A** el de los auxiliares.
- **S** es el elemento inicial.

Todos los elementos de un árbol elemental pueden tener símbolos no terminales. Además, los nodos hoja, los nodos en la *frontera*, también pueden tener terminales. Los árboles elementales iniciales se denotan con la letra α, mientras que los auxiliares se denotan con la letra β.

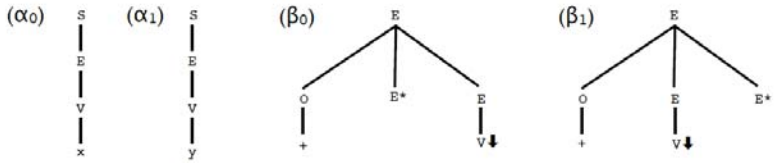


Fig. 14: Ejemplos de árboles elementales[20]

Un árbol auxiliar con un símbolo no terminal X se dice que es de tipo X. La frontera de este tipo de árboles debe contener un nodo especial denominado *nodo pie* con el mismo símbolo que el de su raíz. El nodo pie de cada árbol auxiliar se marca con un asterisco. El resto de nodos de la frontera con símbolo no terminal se marcan con una flecha $\downarrow$.

Las dos operaciones disponibles en la PGGG de adición de árboles son la *adición* y la *sustitución*, ambas combinan árboles elementales para dar lugar a *árboles derivados*. La sustitución requiere un árbol inicial α y un auxiliar β. α debe contener en la frontera un nodo con el símbolo raíz de β. Si esta condición se cumple, se sustituye este nodo frontera por β.
La adición da como resultado un nuevo árbol γ a partir de los árboles α, tanto inicial como derivado, y β, auxiliar. Si α contiene un nodo interno con el mismo símbolo que la raíz y el nodo pie de β, se procede a lo siguiente:

1. Se localiza el nodo interno y se desconecta de α el subárbol α_1, que queda por debajo de este nodo.

2. Se sustituye β en α.

3. Finalmente, α_1 toma el lugar del nodo pie de β.

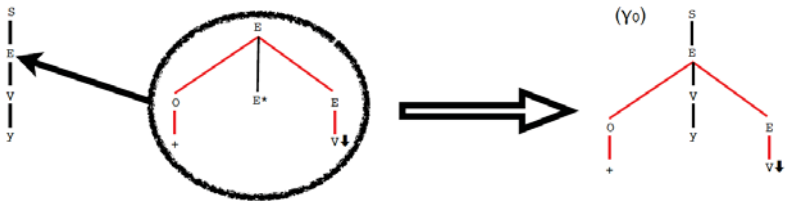


Fig. 15: adición de α_1 y β_0 para formar γ_0[20]

La PGGG de adición de árboles introduce la diferencia entre un *árbol derivado* y un *árbol de derivación*. El árbol derivado es la representación más convencional de árboles en la que aparecen todos los nodos que los componen. El árbol de derivación, en cambio, es una representación resumida de la solución en función de sus árboles elementales.

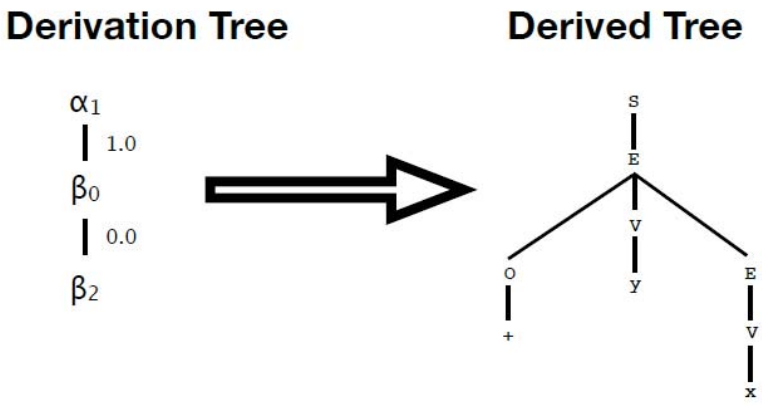


Fig. 16: Estructuras utilizadas en la PGGG de adición de árboles[20]

La separación entre los dos tipos de árboles provee una equivalencia natural entre el genotipo y el fenotipo, dado que un árbol de derivación permite obtener su árbol derivado, como se muestra en la Fig. 16.
Los árboles de derivación en gramáticas de adición de árboles tienen un planteamiento mucho más modular que los de gramáticas libres de contexto y se acercan más a una representación semántica. Según Hoai et al.[20], estos nuevos árboles de derivación se generan de una forma mucho más simple y ordenada, garantizando la validez de las soluciones tras cualquier operación, una propiedad que denominan *factibilidad* que resuelve el problema del cierre.

3. FILTROS FIR Y BLOQUES MULTIPLICADORES

3.1. Filtros Digitales

El término *filtro* hace referencia a cualquier sistema que discrimina la entrada que recibe en función de alguno de sus atributos. Los *filtros digitales* reciben una secuencia discreta, de la que obtienen otra secuencia discreta que experimenta variaciones de amplitud o fase como salida[35].

3.1.1. Sistemas discretos: características generales

La longitud finita en la representación digital causa complicaciones en el análisis de los sistemas de tratamiento digital. Con el fin de simplificarlo, es posible generalizar las señales y sistemas digitales concibiéndolos como señales y sistemas discretos en el tiempo. Los sistemas discretos se pueden clasificar de la siguiente manera[38]:

- **Sistemas estáticos y dinámicos**. Un sistema estático es aquel en el que su salida en cualquier instante depende de la muestra de entrada en dicho instante, pero no de muestras pasadas o futuras de la entrada. En cualquier otro caso, se tiene un sistema dinámico, dotado de memoria.

- **Sistemas invariantes y variantes**. Un sistema es invariante en el tiempo si las características internas del sistema se mantienen constantes en el tiempo.

- **Sistemas lineales y no lineales**. Un sistema es lineal si satisface el principio de superposición. Este principio exige que la respuesta a una suma ponderada de señales sea igual a la correspondiente suma ponderada de las respuestas a cada una de las señales individuales de entrada, cumpliéndose que $f[a_1x_1(n) + a_2x_2(n)] = a_1f[x_1(n)] + a_2f[x_2(n)]$

- **Sistemas causales y no causales**. Se dice que un sistema es causal si la salida del mismo en cualquier instante depende de las entradas actuales y pasadas, pero no las entradas futuras. Para el procesamiento de señales en tiempo real, los sistemas no causales son físicamente irrealizables, pudiendo ser utilizados para aplicaciones de procesamiento fuera de línea como la del tratamiento de imágenes.

- **Sistemas estables e inestables**. Un sistema es estable si, para un intervalo finito, toda entrada acotada en ese intervalo genera otra salida acotada.

3.1.2. Filtros IIR y FIR

En función de cómo se comporta un filtro tras ser estimulado desde el estado de reposo, se puede hablar de dos tipos de filtros: Filtros IIR (Infinite Impulse Response) y Filtros FIR (Finite Impulse Response). Los filtros IIR, al contrario que los filtros FIR, no vuelven a su estado de reposo una vez que son estimulados[38].

3.1.3. Tipos de filtros en función de su respuesta

Los filtros se pueden clasificar en torno a su respuesta deseada, es decir, los mantenimientos o cambios de amplitud dentro y fuera de un intervalo de frecuencias. Estos intervalos se representan mediante frecuencias de corte f_c o f_b y f_a.
En función del intervalo que se decida mantener, tenemos los siguientes tipos de filtros[40]:

- Filtro paso bajo: $[-f_c, f_c]$
- Filtro paso alto: $[-1, -f_c]$, $[f_c, 1]$
- Filtro paso banda: $[-f_a, -f_b]$, $[f_b, f_a]$
- Filtro para banda: $[-1, -f_a]$, $[-f_b, f_b]$, $[f_a, 1]$

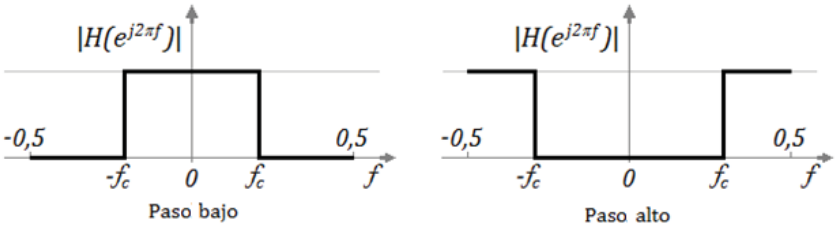


Fig. 17: Filtros paso bajo y paso alto[40]

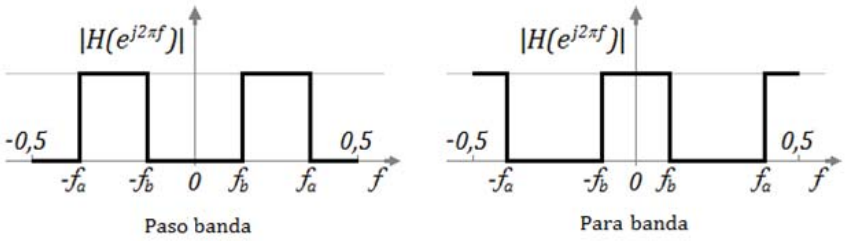
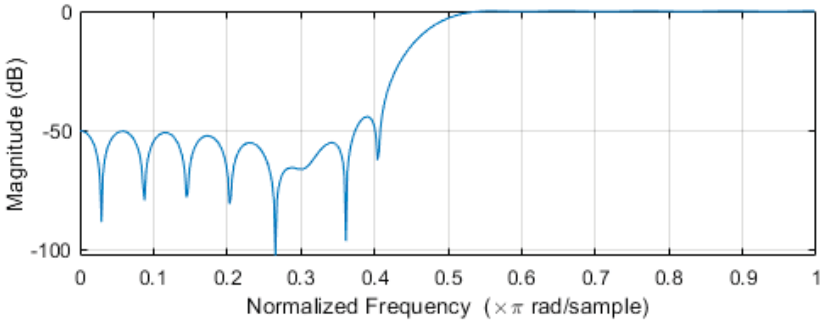
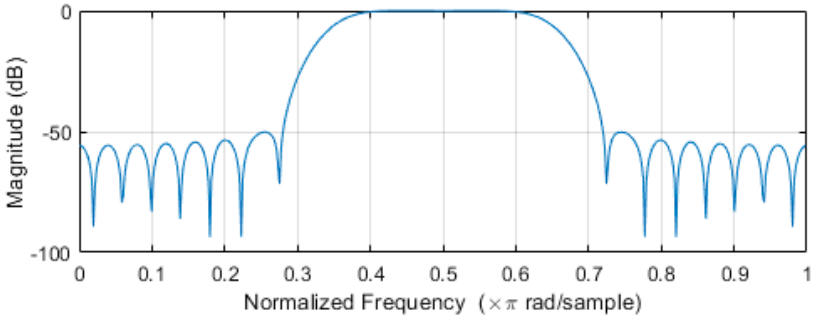


Fig. 18: Filtros paso banda y para banda[40]

En las Fig. 17 y 18 se representan resultados ideales, mientras que en la Fig. 19 se muestran resultados obtenidos de filtros prácticos, de tipo FIR.



(a) FIR paso alto. $f_c = 0{,}48$.



(b) FIR paso banda. $f_b = 0{,}35$ y $f_a = 0{,}65$.

Fig. 19: Respuesta real para cada tipo de filtrado[45]

3.2. Filtros FIR

Los sistemas discretos lineales e invariantes, entre los que están incluídos los filtros FIR, suelen poseer unos parámetros constantes independientes de entrada y salida denominados *coeficientes*, denotados como $\{b_k\}$[38]. Estos coeficientes son las variables que determinan el cálculo de la salida del filtro.
Se habla de *orden* N para designar una cualidad relacionada con el número de sus coeficientes. El número mínimo de coeficientes que un filtro FIR puede poseer es uno, a lo que corresponde un orden de N=0.

3.2.1. Fase lineal: filtros simétricos y antisimétricos

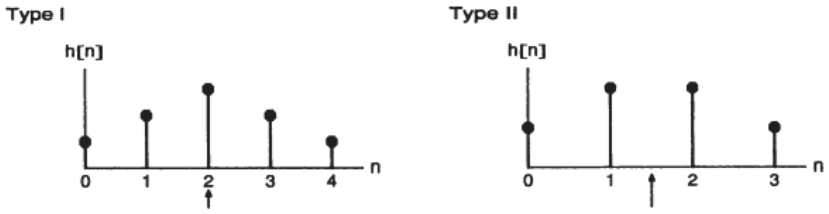
Todo filtro experimenta en su operación un retraso de fase en su banda de paso. En caso de que un filtro tenga una fase lineal, su salida equivale a una versión retardada y con amplitud escalada de la señal de entrada original, sin que se considere que existe ninguna distorsión. Una de las razones para elegir un filtro FIR sobre un filtro IIR es la necesidad de que el filtro utilizado tenga esta característica de fase lineal[38]. Un filtro FIR tiene fase lineal si su respuesta al impulso unidad, h(n), cumple que:

$$h(n) = \pm h(N-n), \text{ para } n = 0, 1, ..., N$$

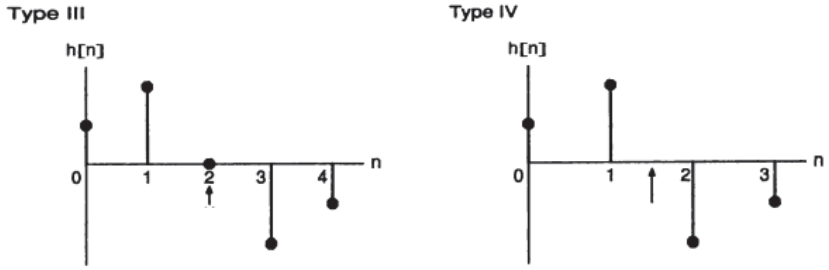
Esta condición limita a los FIR de fase lineal a dos tipos: filtros simétricos y filtros antisimétricos[38].

- Los coeficientes de los simétricos siguen la equivalencia $h(n) = h(N - n)$. Los coeficientes de este tipo tienen una suma total de 1.
- Los coeficientes de los antisimétricos siguen la equivalencia $h(n) = -h(N-n)$. Los coeficientes de este tipo tienen una suma total de 0.

En la Fig. 20, se pueden comprobar cuatro tipos, incluyendo separación entre número par e impar de coeficientes.



(a) FIR simétricos



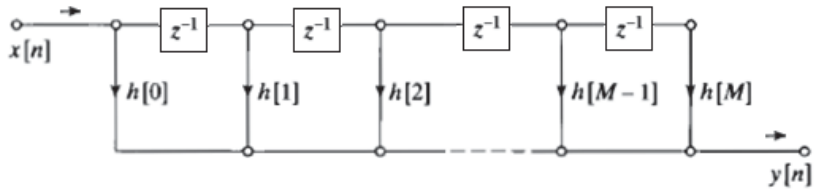
(b) FIR antisimétricos

Fig. 20: Tipos de filtrado en función de los coeficientes[40]

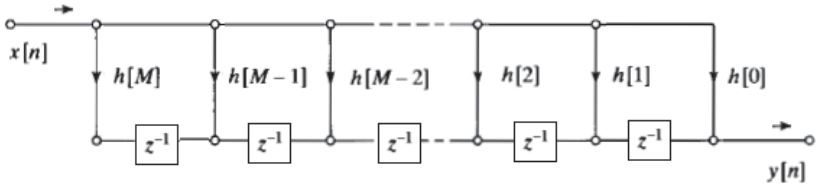
Paralelamente, esta condición de simetría proporciona una ventaja de procesamiento a los FIR de fase lineal, que solo necesitan realizar $\sim \frac{N+1}{2}$ operaciones, frente a $N + 1$ de un FIR no simétrico[38].

3.2.2. Funcionamiento

A nivel de hardware, el filtro FIR es un circuito compuesto por varios acumuladores interconectados entre sí, de tal forma que sus salidas se van sumando. Estos acumuladores capturan la última entrada recibida y transmiten simultáneamente la anteriormente almacenada. Existen varias formas de construir este circuito[38]. En este trabajo se destacan dos: la forma directa y la forma directa traspuesta. Estas dos formas, contenidas en la Fig. 21, obtienen conjuntos de salida idénticos para idénticos conjuntos de entrada. Esta situación muestra el cumplimiento del principio de superposición de los filtros FIR, sistemas discretos lineales.



(a) Forma directa



(b) Forma directa traspuesta

Fig. 21: Estructura de un filtro FIR[40]

Dado un filtro FIR de orden N y una entrada $x(n)$, la salida $y(n)$ es tal que[38]:

$$y(n) = \sum_{k=0}^{N} b_k x(n-k)$$

A modo de ejemplo, para un filtro de orden N=4, se obtiene el conjunto de salidas siguiente:

$$\begin{aligned}
y(0) &= b_0x(0) \\
y(1) &= b_0x(1) + b_1x(0) \\
y(2) &= b_0x(2) + b_1x(1) + b_2x(0) \\
y(3) &= b_0x(3) + b_1x(2) + b_2x(1) + b_3x(0) \\
y(4) &= b_0x(4) + b_1x(3) + b_2x(2) + b_3x(1) + b_4x(0) \\
y(5) &= b_0x(5) + b_1x(4) + b_2x(3) + b_3x(2) + b_4x(1) \\
&\vdots
\end{aligned}$$

3.2.3. Obtención de coeficientes

En el libro *Tratamiento Digital de Señales* de Proakis y Manolakis[38], se presentan métodos de cálculo de coeficientes de *filtros ideales* que varían en función de la salida deseada, además de la introducción de errores en el proceso de cuantificación de estos coeficientes en soporte digital.

Para la realización de este trabajo, se ha optado por utilizar una función implementada en MATLAB para este procedimiento. Se utilizará la función $fir1$[45], más concretamente $fir1(n, Wn, ftype)$, a la que se proporciona como argumentos el orden (n), la frecuencia de corte (Wn) y el tipo de filtrado $(ftype)$, dejando todo lo demás en su valor por omisión. Esta función devuelve como resultado un vector de coeficientes de valor real que posteriormente se utiliza para construir el filtro de referencia de este trabajo denominado *filtro Matlab*.

Los argumentos de esta función son los siguientes:

- n: El orden del filtro. Para filtros paso alto y para banda de orden impar, la función calculará los filtros correspondientes de orden $n + 1$.
- Wn: Restricción de frecuencia. Intervalo normalizado de frecuencias, en el intervalo $(0, 1)$, que delimitan el filtrado.
- ftype: Tipo de filtrado deseado. Se encuentran las opciones de filtro paso bajo ('low'), paso alto ('high'), pasa banda ('bandpass') y otras opciones más específicas.

A continuación, se presenta un ejemplo de llamada a esta función para un filtro paso bajo de orden N=5, y $f_c = 0,5$:

```
>> fir1(5,0.5,'low')

ans =

   -0.0078    0.0645    0.4433    0.4433    0.0645   -0.0078
```

Fig. 22: Resultado ejemplo de la función *fir1*

3.3. Bloques Multiplicadores

Dada la necesidad de los filtros FIR de realizar series de multiplicaciones, la implementación de estos filtros resulta costosa mediante métodos convencionales de operación. Para mejorar la eficiencia, se introduce el elemento denominado *bloque multiplicador*[47].
Un bloque multiplicador es una implementación en hardware que realiza multiplicaciones de bajo coste computacional mediante sumas, restas y desplazamientos binarios. Existen dos tipos de bloques multiplicadores en función de las entradas y salidas deseadas: paralelo y multiplexado. El bloque multiplicador paralelo devuelve varias salidas de forma simultánea para una entrada, mientras que el multiplexado devuelve una salida a partir de una entrada y un factor de control que es el que determina la operación realizada. En este trabajo, se considera utilizar el bloque multiplicador paralelo por ser el que mejor se ajusta a las necesidades del filtro FIR, resultando su incorporación dentro del filtro FIR en un circuito más simple. Como consecuencia, la estructura del filtro FIR que se utiliza en este trabajo es la forma directa traspuesta (Fig. 21b), ya que es la que permite utilizar una sola entrada para realizar de forma simultánea todas las operaciones de multiplicación.
Dadas las operaciones disponibles, los bloques multiplicadores suelen obtener resultados equivalentes a los de la multiplicación de la entrada por un número entero. Para obtener estos números enteros, es necesario utilizar un factor de desplazamiento denominado *fractional bits* sobre los coeficientes $\{b_k\}$ menores que la unidad. Con estos *bits fraccionarios* f, se determina el factor de desplazamiento hacia la izquierda para obtener los *coeficientes fraccionarios* $\{c_k\}$, además del factor de desplazamiento hacia la derecha que se debe realizar a la salida del bloque multiplicador para que se aproxime al resultado esperado:

$$c_k = [b_k 2^f] \Leftrightarrow b_k \approxeq \frac{c_k}{2^f}$$

La construcción de bloques multiplicadores está íntimamente ligada al problema de la Multiplicación Múltiple de Constantes (MCM)[47], un problema que ha sido objeto de estudio durante años y designado como NP-completo[5]. La Multiplicación Múltiple de Constantes es, a su vez, una extensión del problema de la Multiplicación Simple de Constantes (SCM). La SCM consiste en la obtención de la configuración que multiplica una variable x por una constante t únicamente mediante sumas, restas y desplazamientos binarios. Siendo la MCM una extensión de esta última, su problema correspondiente es la obtención de la configuración que multiplica esa variable x por varias constantes $t_1 \dots t_n$ en paralelo utilizando el denominado bloque multiplicador. Pero el resultado de una MCM no constituye una suma exacta de las SCM que la componen, puesto que puede haber resultados intermedios comunes, resultando en una estructura de menor tamaño[47].

3.3.1. Representaciones binarias

El método más directo para obtener una multiplicación como sumas y desplazamientos consiste en descomponer en varios operandos los bits de la constante en

su representación binaria. Existen dos estrategias para llevar a cabo este método: transformar bits con valor 1 en operandos positivos o transformar bits con valor 0 en operandos negativos. Para t = 71:

$$71x = 1000111_2x = x \ll 6 + x \ll 2 + x \ll 1 + x$$

$$71x = 1000111_2x = (x \ll 7 - x) - x \ll 5 - x \ll 4 - x \ll 3$$

Un método más sofisticado que el anterior es el de la forma canónica (CSD) de la representación binaria, que amplía la representación a 0, $\bar{0}$, 1 y $\bar{1}$. Este método mejora los resultados anteriores al utilizar solo dos operadores:

$$71x = 1000111_2x = 100100\bar{1}_{CSD}x = x \ll 6 + x \ll 3 - x$$

Estos métodos basados en representación binaria poseen una limitación en el rango de topologías debido a esta representación. Para evitar esto, se desarrollaron posteriores métodos basados en grafos[47].

3.3.2. BHA y BHM

En 1991, Bull y Horrocks[4] presentan el algoritmo BHA que permite obtener grafos para generar bloques multiplicadores conectados a filtros FIR. En la Fig. 23, se muestra un diagrama de los dos elementos combinados.

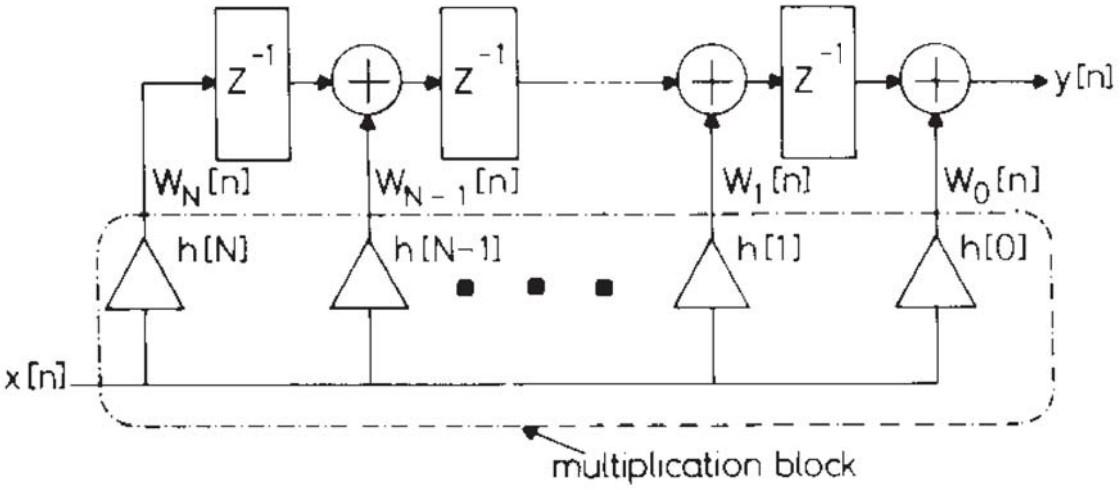


Fig. 23: Bloque multiplicador dentro del circuito de un Filtro FIR de estructura traspuesta[4]

Este algoritmo consta de cuatro versiones incrementales en cuanto a los operadores utilizados: *Solo adición* (Algoritmo 1), *adición y substracción*, *adición y desplazamiento* (Algoritmo 2) y, finalmente, *Adición, substracción y desplazamiento* (Algoritmo 3).

El algoritmo 1, en sus iteraciones, realiza operaciones mediante combinación de elementos ya existentes para obtener lo que se denominan sumas parciales. Estas sumas parciales, representación matemática de cada nodo del resultante bloque, se almacenan en el conjunto denominado psum (partial sums). Al inicio de la ejecución, se inicializa el conjunto psum con los fundamentales [0, 1], que representan el resultado de una salida sin entrada y la propia entrada del circuito, respectivamente. Cada iteración tiene dos fases: búsqueda de sumas parciales y la sintetización del coeficiente.

Algorithm 1 BHA: Adición. Versión modificada de [4].

```
1:  index ← −1
2:  psum ← {0, 1}
3:  repeat
4:      index ← index + 1
5:      error ← h′[index]
6:      repeat
7:          error ← (h′[index])
8:          while error <> 0 do
9:              if error ∈ psum[i] then error ← 0
10:             else
11:                 n ← length(psum) − 1
12:                 (x, y) ← minimise(error − (psum[n − x] + psum[n − y]))
13:                 psum[n] ← psum[n − x] + psum[n − y]
14:                 error ← error − psum[n]
15:             end if
16:         end while
17:     until error = 0
18:     repeat
19:         list ← list + combineElements(psum)
20:     until h′[index] ∈ list
21:     psum ← psum + list
22: until index > length(h′) − 1
```

En la primera fase, la de búsqueda, el algoritmo asigna el coeficiente a sintetizar a la variable error, variable designada a guiar la búsqueda hasta que, valiendo este error cero, se dé la iteración por terminada. El error pasa directamente a valer cero en el caso de que ya exista un fundamental en el conjunto psum con su mismo valor. En caso contrario, el algoritmo ajusta el error al valor del coeficiente y busca en el conjunto psum una suma de fundamentales que minimice ese error al substraer su resultado.
En la segunda fase, la de la obtención, se combinan las sumas parciales hasta obtener el fundamental equivalente al coeficiente. Al final de cada iteración, se recogen todas las sumas parciales obtenidas en el conjunto psum.

El algoritmo 2 incluye, además, el operador de desplazamiento. Con este operador, al conjunto inicial de sumas parciales, se le añaden todos los desplazamientos de la entrada que no superen el coeficiente máximo a sintetizar.
El ajuste de la variable error al principio de cada iteración sufre una modificación: el error se ajusta a la forma reducida del coeficiente, esto es, la división de este coeficiente entre 2 hasta que sea impar.
En la fase de obtención del coeficiente, se realiza el procedimiento de combinación hasta que se obtenga la forma reducida del coeficiente. Además, se añaden también todos los desplazamientos de las nuevas sumas parciales que no superen el fundamental a sintetizar.

Algorithm 2 Adición y desplazamiento. Versión modificada de [4].

1: $index \leftarrow -1$
2: $psum \leftarrow \{0, 1\} + allpow(2, 1, max(h'))$
3: **repeat**
4: $index \leftarrow index + 1$
5: $error \leftarrow h'[index]$
6: **repeat**
7: **if** $h'[index] \in psum[i]$ **then** $error \leftarrow 0$
8: **else**
9: $error \leftarrow reduce(h'[index])$
10: **end if**
11: **while** $error <> 0$ **do**
12: $n \leftarrow length(psum) - 1$
13: $(x, y) \leftarrow minimise(error - (psum[n - x] + psum[n - y]))$
14: $psum[n] \leftarrow psum[n - x] + psum[n - y]$
15: $error \leftarrow error - psum[n]$
16: **end while**
17: **until** $error = 0$
18: **repeat**
19: $list \leftarrow list + combineElements(psum)$
20: **until** $reduce(h'[index]) \in list$
21: $list \leftarrow list + allpow(2, list, h'[index])$
22: $psum \leftarrow psum + list$
23: **until** $index > length(h') - 1$

El algoritmo 3 incluye todas las operaciones posibles. Esto añade la posibilidad de que la combinación a minimizar en cada iteración sea tanto una suma como una resta, además de que el error resultante pueda ser negativo. La versión *adición y substracción* incorpora un desplazamiento similar al del algoritmo 2 al inicio de cada iteración. En la publicación original, no se menciona explícitamente si se gestiona esta redundancia. En el presente trabajo, se presenta la versión definitiva con todas las incorporaciones respectivas.

Algorithm 3 Adición, substracción y desplazamiento. Versión modificada de [4].

1: $index \leftarrow -1$
2: $psum \leftarrow \{0, 1\} + allpow(2, 1, max(h'))$
3: **repeat**
4: $index \leftarrow index + 1$
5: $error \leftarrow h'[index]$
6: **repeat**
7: **if** $h'[index] \in psum[i]$ **then** $error \leftarrow 0$
8: **else**
9: $psum \leftarrow psum + allpow(2, psum, h'[index])$
10: $n \leftarrow length(psum) - 1$
11: $error \leftarrow reduce(h'[index] - psum[n])$
12: **end if**
13: **while** $error <> 0$ **do**
14: $n \leftarrow length(psum) - 1$
15: $(x, y, op) \leftarrow minimise(error - (psum[n - x] \text{ op } psum[n - y]))$
16: $psum[n] \leftarrow psum[n - x] \text{ op } psum[n - y]$
17: $error \leftarrow error - psum[n]$
18: **end while**
19: **until** $error = 0$
20: **repeat**
21: $list \leftarrow list + combineElements(psum)$
22: **until** $reduce(h'[index]) \in list$
23: $list \leftarrow list + allpow(2, list, h'[index])$
24: $psum \leftarrow psum + list$
25: **until** $index > length(h') - 1$

En 1995, Dempster y Macleod[11] describen las limitaciones del método BHA y se proponen varias soluciones:

- *Las sumas parciales para obtener un coeficiente solo pueden llegar a tener un valor menor que el coeficiente y no valores que lo excedan.* Solución: generar sumas parciales por encima del valor del coeficiente. Esto permite soluciones similares a las del método CSD, por ejemplo, permitiendo que se obtenga la operación $7 = 8 - 1$ que supera a $7 = 4 + 2 + 1$.

- *Se pueden introducir sumas parciales de resultado par en el conjunto psum.* Solución: reducir todas las sumas parciales en factores de 2 hasta que sean impares e introducirlas al conjunto psum y solo entonces generar todos sus múltiplos desplazados a la izquierda. Esto maximiza el número de sumas parciales disponible en estadios finales del algoritmo, maximizando su flexibilidad.

- *Los coeficientes se procesan en orden numérico.* Solución: Ordenar los coeficientes en orden incremental de coste de coeficiente, obtenido con el algoritmo MAG[10].

Algorithm 4 funciones BHA. Elaboración propia.

```
1: function ALLPOW(b,x,m)
2:     n ← 1
3:     while b^n x ≤ m do
4:         list ← list + b^n x
5:         n ← n + 1
6:     end while
7:     return list
8: end function
9:
10: function REDUCE(x)
11:     while x mod 2 = 0 do
12:         x ← x \ 2
13:     end while
14:     return x
15: end function
```

3.3.3. RAG-n

En la publicación del BHM, Dempster y Macleod[11] presentan además un algoritmo híbrido que consta de una parte óptima y otra heurística.

La parte óptima consta de los siguientes pasos:

1. Se comienza reduciendo todos los coeficientes a fundamentales impares, dividiéndolos por 2 hasta ser impares y se almacenan en el conjunto incompleto.

2. Se evalúan todos los costes de coeficientes utilizando la tabla de búsqueda generada por el algoritmo MAG, que se detalla en *Constant integer multiplication using minimum adders*[10].

3. Se eliminan del conjunto incompleto todos los fundamentales de coste 0, es decir, todas las potencias de 2 que han quedado reducidas a 1, o fundamentales repetidos.

4. Se crea el conjunto grafo para el almacenamiento de los fundamentales seleccionados. En este conjunto, se introducen todos los fundamentales de coste 1, p. ej. $5 = 4 + 1$, y se eliminan del conjunto incompleto.

5. Se buscan sumas de parejas de fundamentales en el conjunto grafo con múltiplos de potencia de 2 de esos mismos fundamentales. Si se produce alguno de los coeficientes contenido en el conjunto incompleto, se elimina del conjunto incompleto y se coloca en el conjunto grafo.

6. Repetir el paso 5 hasta que no se añadan más fundamentales al conjunto grafo.

La parte heurística introduce el concepto de distancia o adder distance. La distancia de un vértice es el número de sumadores que se necesitan para llegar a un nuevo vértice.
Esta parte consta de los siguientes pasos:

7. Comprobar cada elemento del conjunto incompleto para dos instancias de vértices de distancia 2:

   - El caso en el que para llegar al coeficiente de interés hace falta sumar un fundamental de coste 1 a otro fundamental ya obtenido.
   - El caso en el que para llegar al coeficiente de interés hace falta sumar un fundamental de coste 0 a la suma de dos fundamentales ya obtenidos.

Si se encuentran tales casos, se añaden los dos fundamentales (el coeficiente de interés y el intermedio) y se elimina el coeficiente del conjunto incompleto.

8. Repetir los pasos 6 y 7 hasta que no se forme ningún fundamental de distancia 1 o 2.

9. Si se llega a este punto, hay coeficientes que están a mayor distancia que 2 del grafo existente, o a una distancia 2 con una topología no incluida por los dos ejemplos en el paso 7. Se sintetizará el coeficiente por medio de la tabla MAG ya precalculada[47].

10. Repetir desde el paso 6 al 9 hasta que todos los coeficientes estén sintetizados.

3.3.4. Hcub

En 2007, Voronenko y Püschel[47] presentan una síntesis de todos los algoritmos MCM previos que da lugar al nuevo algoritmo Hcub. El nuevo algoritmo recibe inspiración de su precedente RAG-n, siendo también este un algoritmo híbrido. El algoritmo Hcub hereda una parte óptima prácticamente idéntica a la del algoritmo RAG-n, además de introducir una nueva parte heurística[47].
En este nuevo algoritmo, se utiliza el término *fundamentales* del BHA como intercambiable al de *constantes*.

Esta implementación utiliza sumas, restas y desplazamientos, como cualquier otra de MCM, pero, como método de optimización del proceso de búsqueda, se introduce una simple operación parametrizada denominada A-operation o A_p.

Dadas dos entradas u, v y una salida, una A_p realiza una suma o resta y un número arbitrario de desplazamientos tal que:

$$A_p(u, v) = |(u \ll l_1) + (-1)^s(v \ll l_2)| \gg r = \left|2^{l_1}u + (-1)^s 2^{l_2}v\right|2^{-r}$$

Cada A_p tiene un conjunto de parámetros que definen la operación, denotada como configuración: $p = (l_1, l_2, r, s)$

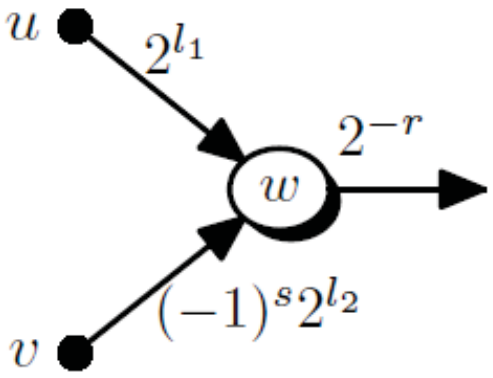


Fig. 24: Diagrama de una A-operation[47]

Todo grafo dirigido tiene un equivalente de igual coste compuesto únicamente por fundamentales impares[10]. Este tipo de grafos son relevantes porque ayudan a reducir los grados de libertad a la hora de construir un grafo, simplificando así la búsqueda. Para obtener grafos de fundamental impar, se debe restringir la configuración de las A_p para que solo se obtengan resultados impares de ellas, mediante la utilización de al menos una entrada sin desplazar hacia la izquierda o un desplazamiento hacia la derecha. Esta nueva A_p restringida recibe la denominación de A^{odd}. La A^{odd} es la operación inicial sobre la que se construyen los grafos, sus restricciones exigen que se realice un desplazamiento hacia la derecha a todo el conjunto objetivo hasta contener solo números impares.

Para expresar todo el potencial de configuraciones de una A_p, se define el conjunto de vértices fundamentales A_* como el conjunto de todas las posibles configuraciones de A_p que, dada una entrada fija, tengan salidas diferentes a las entradas:

$$A_*(u,v) = \{A_p(u,v) \mid \text{p es válida}\} - u - v$$

En el caso de que las entradas sean conjuntos:

$$A_*(U,V) = \bigcup_{\substack{u\in U \\ v\in V}} (A_*(u,v)) - U - V$$

$$A_*(U \cup V, W) = A_*(U,W) \cup A_*(V,W) - U - V$$

Algorithm 5 Algoritmo Hcub[47].

```
1: Right shift elements of T until odd
2: R ← {1}
3: W ← {1}
4: S ← {1}
5: while T ≠ ∅ do
6:     while W ≠ ∅ do
7:         R ← R ∪ W
8:         S ← (S ∪ A*(R, W)) − W
9:         W ← ∅
10:
11:        for t ∈ S ∩ T do
12:            Synthesize(t)
13:        end for
14:    end while
15:
16:    if T ≠ ∅ then
17:        s ← H(R, S, T)
18:        Synthesize(t)
19:    end if
20: end while
21:
22: function SYNTHESIZE(s)
23:    W ← W + s
24:    T ← T − s
25: end function
```

En el algoritmo, se utilizan varios conjuntos para almacenar las soluciones:

- **T** (target) para las constantes objetivo pendientes de sintetizar.
- R (result) para el resultado.
- S (successors) para generar sucesores de los nodos ya obtenidos.
- W (worklist) como conjunto auxiliar para almacenar las constantes sintetizadas en la iteración.

En el algoritmo 5, la parte óptima se encuentra en las líneas 6-14 y la heurística en las líneas 16-19. Al final de cada una de las partes, se sintetizan todas las constantes encontradas en la iteración, se borran del conjunto objetivo y se llevan al conjunto auxiliar. La parte heurística solo tiene lugar en caso de que una iteración de la parte óptima no consiga sintetizar algún objetivo.

En la parte óptima, se calculan solo los sucesores directos, los sucesores que están a una A_p de los elementos ya obtenidos. Se define el conjunto S como:

$$S = \{s \mid dist(R, s) = 1\} = A_*(R, R)$$

La operación que aparece en la línea 8 corresponde al cálculo de los sucesores en cada iteración. El resultado final se presenta como la derivación de lo siguiente:

$$\begin{aligned} S_{new} = A_*(R_{new}, R_{new}) = A_*(R_{new}, R \cup W) = \\ A_*(R_{new}, R) \cup A_*(R_{new}, W) = \\ A_*(R \cup W, R) \cup A_*(R_{new}, W) = \\ A_*(R, R) \cup A_*(W, R) \cup A_*(R_{new}, W) - W = \\ S \cup A_*(R, W) \cup A_*(R_{new}, W) - W \end{aligned}$$

Ya que $A_*(R, W) \subset A_*(R_{new}, W)$, se obtiene finalmente:

$$S_{new} = (S \cup A_*(R_{new}, W)) - W$$

Para la parte heurística, se introduce el concepto de A-distance, equivalente a la adder distance del algoritmo RAG-n, el número mínimo de operaciones necesarias para llegar a un fundamental. El objetivo de la parte heurística está en disminuir esta A-distance entre el conjunto resultado y el conjunto objetivo. Para ello, se busca un sucesor que minimice esta distancia a un elemento objetivo al formar parte del conjunto resultado.

Se introduce la *función de beneficio*, que maximiza la distancia perdida de la siguiente manera:

$$B(R, s, t) = dist(R, t) - dist(R + s, t)$$

Para elementos próximos, la función devuelve como máximo 1. Para objetivos más lejanos, solo se puede estimar la distancia, que se va haciendo más imprecisa, pero también menos importante. Por todo esto, se introduce la *función de beneficio ponderada*:

$$\bar{B}(R, s, t) = 10^{-dist(R+s,t)}(dist(R, t) - dist(R + s, t))$$

Para calcular la distancia entre el conjunto resultado añadido el sucesor y el conjunto objetivo, se introducen las siguientes funciones heurísticas:

Función de máximo beneficio $H_{maxb}(R, S, T)$:

$$H_{maxb}(R, S, T) = \underset{s \in S}{\operatorname{argmax}} \left(\max_{t \in T} \sum \bar{B}(R, s, t)\right)$$

Función de beneficio acumulado $H_{cub}(R, S, T)$, que da nombre al algoritmo:

$$H_{cub}(R, S, T) = \underset{s \in S}{\operatorname{argmax}} \left(\sum_{t \in T} \bar{B}(R, s, t)\right)$$

4. PLANTEAMIENTO DEL PROBLEMA

Para la resolución del problema NP-completo que es la generación automática de bloques multiplicadores, existen iniciativas dedicadas al desarrollo de herramientas de generación automática de circuitos integrados como la del Proyecto Spiral.

Spiral es un sistema de generación de programas para transformaciones lineales, entre ellas la transformada de Fourier. El propósito de Spiral consiste en la automatización del desarrollo de librerías de alto rendimiento[39].

El proyecto Spiral proporciona una gran variedad de generadores de filtros FIR/IIR y de bloques multiplicadores (`http://www.spiral.net`). En la realización de este trabajo, se utiliza el generador de bloques multiplicadores de MCM (Multiple Constant Multiplication) paralela[43] que se encuentra en la correspondiente sección de su página web. Este generador permite elegir un algoritmo MCM de entre los comentados en este trabajo además de otros parámetros como los bits fraccionarios y una opción de optimización secundaria como podemos ver en la Fig. 25. En la misma página, se puede descargar la librería *synth*, el código fuente en lenguaje C utilizado por el generador.

La solución proporcionada por este generador mediante el algoritmo Hcub se denomina *solución Spiral* en el presente trabajo.

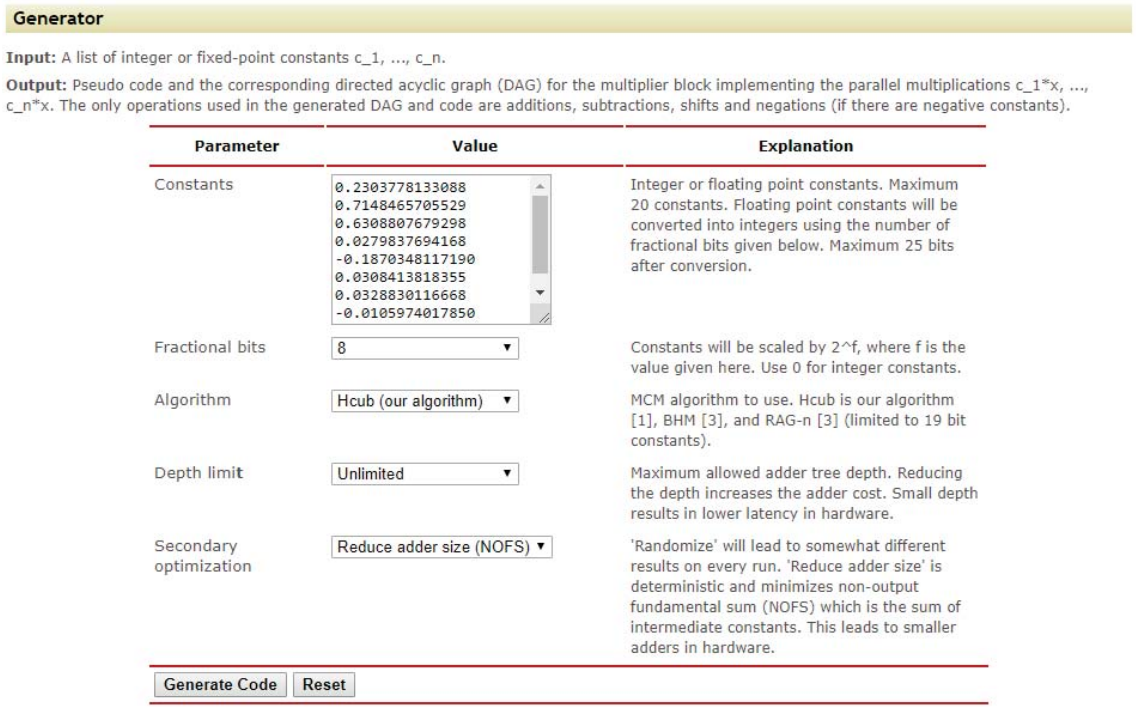


Fig. 25: Formulario del generador de bloques[43]

A pesar de la funcionalidad que brinda, el generador de bloques multiplicadores de Spiral posee las siguientes limitaciones, tanto en su versión web como descargable:

1. La versión web devuelve siempre una misma solución para los mismos parámetros, limitando los filtros a un orden máximo de $N = 21$. Sin embargo, la versión descargable no posee ninguna limitación en cuanto al orden y conserva el comportamiento estocástico del algoritmo Hcub, siempre que no se elija la optimización secundaria de área.

2. La versión web admite tanto los coeficientes originales $\{b_k\}$, los coeficientes del *filtro Matlab*, como los fraccionarios $\{c_k\}$, siempre que estos últimos se introduzcan seleccionando cero bits fraccionarios. En la versión descargable, en cambio, solo se permite la introducción de coeficientes fraccionarios $\{c_k\}$.

3. En caso de introducir los coeficientes originales en la versión web, es necesario transformarlos en coeficientes fraccionarios. Para cada uno de ellos, se realiza el desplazamiento correspondiente a los bits fraccionarios para posteriormente realizar un truncamiento.

Toda reducción de la parte decimal de los coeficientes, introduce un error a la salida del filtro $y(n)$ tal que:

$$\Delta y(n) = \sum_{k=0}^{N} \frac{\partial y(n)}{\partial b_k} \Delta b_k \approx \sum_{k=0}^{N} \Delta b_k = \sum_{k=0}^{N} \frac{c_k}{2^f} - b_k$$

Para un filtro paso bajo de orden $N = 10$, $f_c = 0{,}4$ y bit fraccionario $f = 15$, la Tab. 1 muestra los distintos errores introducidos en función del método de reducción:

k	0, 10	1, 9	2, 8	3, 7	4, 6	5
b_k	-1,24 $\cdot 10^{-18}$	-0,0126	-0,0246	0,0635	0,2747	0,3980
$2^f b_k$	-4,0677 $\cdot 10^{-16}$	-414,2522	-809,1159	2080,9360	9004,5727	13043,7187
$c_k = \lfloor 2^f b_k \rfloor$	0	-414	-809	2080	9004	13043
Δb_k	1,24 $\cdot 10^{-18}$	7,70 $\cdot 10^{-6}$	3,54 $\cdot 10^{-6}$	-2,86 $\cdot 10^{-5}$	-1,75 $\cdot 10^{-5}$	-2,19 $\cdot 10^{-5}$
$c_k = [2^f b_k]$	0	-414	-809	2081	9005	13044
Δb_k	1,24 $\cdot 10^{-18}$	7,70 $\cdot 10^{-6}$	3,54 $\cdot 10^{-6}$	1,95 $\cdot 10^{-6}$	1,30 $\cdot 10^{-5}$	8,58 $\cdot 10^{-6}$

Tab. 1: Coeficientes fraccionarios y su error asociado Δb_k para dos métodos de reducción decimal

En caso de que la parte decimal reducida esté más cerca de su límite superior, el error introducido por este truncamiento introduce un error añadido mayor que el de un redondeo. Para el truncamiento, obtenemos $\sum \Delta b_k = -9,16 \cdot 10^{-7}$ y $\sum |\Delta b_k| = 1,36 \cdot 10^{-4}$, mientras que para el método de aproximación a la unidad más cercana obtenemos $\sum \Delta b_k = 6,10 \cdot 10^{-7}$ y $\sum |\Delta b_k| = 6,10 \cdot 10^{-5}$.
De esta manera, se concluye que la versión web del generador Spiral exige la selección manual de los coeficientes fraccionarios para obtener resultados óptimos.

4. En ambas versiones, se recibe un mensaje de error cuando el valor de un coeficiente $\{c_k\}$ es cero. En la versión web, se recibe un error como el de la Fig. 26 para coeficientes $\{b_k\}$ cuyo tratamiento produce este valor.

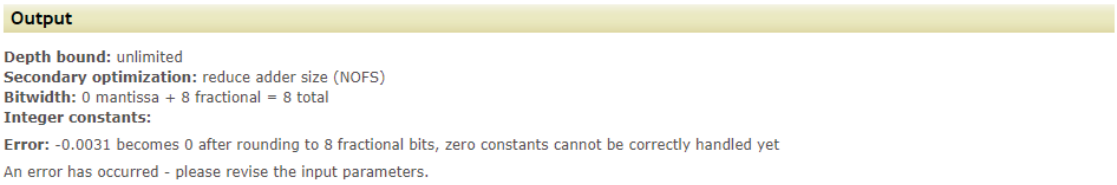


Fig. 26: Resultado con mensaje de error para coeficientes de un filtro paso bajo de orden $N = 3$, $f_c = 0{,}7$ y bits fraccionarios $f = 8$[43]

Para remediar las dificultades anteriores, se propone un método evolutivo para generar los bloques multiplicadores que cumpla con dos objetivos:

1. Minimizar el error introducido en la reducción decimal. Este problema resulta trivial para los métodos evolutivos. La función de evaluación se formula en relación al valor real correspondiente a los coeficientes del *filtro Matlab* de forma que se ven recompensadas las soluciones que cuentan con los coeficientes más próximos.

2. Sintetizar el circuito responsable de los coeficientes que minimizan el error introducido. Este problema resulta más complejo, existiendo múltiples soluciones que satisfacen el objetivo. Como criterio diferenciador, se considera óptima aquella configuración que también minimiza el área del circuito.

Para ello, se plantea la implementación de un algoritmo de PGGG con los siguientes parámetros:

- Un archivo que contiene la gramática elegida para resolver el problema.
- Características del filtro elegido (orden, frecuencia de corte, tipo) que permiten obtener el *filtro Matlab* tal y como se muestra en la Fig. 22.
- Bit fraccional que fija el tamaño de los coeficientes a sintetizar para la solución.
- La *solución Spiral*, la solución de referencia que la ejecución debe alcanzar o superar.

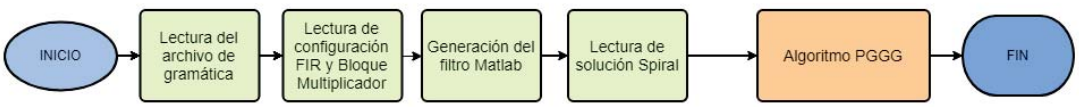


Fig. 27: Esquema general de la implementación

5. SOLUCIÓN PROPUESTA

En este capítulo, se explica el desarrollo del algoritmo PGGG a partir del planteamiento presentado en el anterior capítulo.
Primero, se comentan los parámetros evolutivos y los operadores genéticos elegidos para el algoritmo. Para finalizar, se describe el proceso de modelado de las soluciones y las métricas elegidas para su evaluación: error, área y ranking.

Los parámetros del algoritmo de PGGG son los de una población de 100 individuos, una probabilidad de cruce de 100 % y una de mutación del 1 %.
La condición de parada es la obtención de una solución que domine a la *solución Spiral*, no pudiendo la ejecución realizar más de 300 generaciones.
La profundidad máxima de la codificación para las soluciones debe ser una variable dependiente de los bits fraccionales utilizados, es decir, del tamaño de los coeficientes a sintetizar. Para 8 bits fraccionales, se decide utilizar una profundidad máxima de 30.

Se seleccionan diferentes operadores de entre los mencionados en el capítulo 2 para la inicialización, la selección, el cruce, la mutación y el reemplazo. Para cada ejecución, se utilizan los siguientes operadores:

- Para la inicialización, se utiliza el método GBIM.
- Para la selección, se permite utilizar tanto la selección por ruleta o el torneo probabilístico.
- Para el cruce y la mutación, se implementan los operadores de Whigham y los basados en gramáticas GBC y GBM.
- Para el reemplazo, se permite incluir a los progenitores en la formación de la población de la siguiente generación. Además, se permite prohibir la presencia de soluciones duplicadas en la nueva población, una característica frecuentemente utilizada junto con la anterior[41].

Para la finalización de la implementación del algoritmo, se debe llevar a cabo el proceso de diseño para elegir características relevantes para la resolución del problema.
A continuación, se procede a detallar este proceso que comienza con la codificación que permite la definición de las soluciones. Para finalizar, se procede a explicar el proceso de selección de métricas para la evaluación de esta codificación.

5.1. Estructura y codificación de las soluciones

La etapa de codificación comienza con un proceso de análisis y diseño de la estructura de los bloques multiplicadores.
A partir de esta realidad previa, se construye un modelo que represente este tipo de circuitos de forma que se ajuste a la realidad en sus aspectos más relevantes.
Posteriormente, se procede a adaptar este modelo para que pueda ser representado mediante una gramática libre de contexto.

5.1.1. Diseño de soluciones

La construcción de las soluciones se realiza mediante la combinación de nodos, los elementos básicos de construcción que proporcionan las operaciones al circuito. Mediante esta combinación, se obtiene una representación de bloques multiplicadores en forma de grafo.

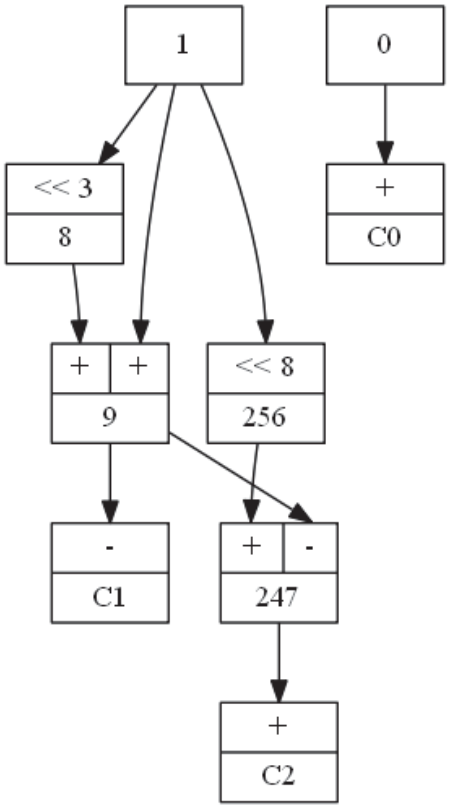


Fig. 28: Ejemplo de bloque multiplicador con salidas que sintetizan el conjunto de coeficientes $\{-9, 0, 247\}$

La Fig. 28 se obtiene mediante la herramienta de visualización de grafos Graphviz[18], construida en el lenguaje descriptivo DOT. Esta herramienta ha sido utilizada anteriormente[40] para representar bloques multiplicadores con una caracterización de la que se recibe influencia en este trabajo.

Dada la caracterización anterior, se decide utilizar los siguientes tipos de nodos para construir los bloques multiplicadores:

- **Entrada**: Los nodos entrada pueden aparecer de dos formas diferentes. Para la representación de la entrada introducida al circuito se toma como referencia el impulso unitario, 1. En el caso de que un bloque multiplicador precise que una de sus salidas tenga valor nulo, se representa esta entrada como un impulso nulo, 0. Los nodos entrada no reciben información de ningún otro nodo. Ningún nodo intermedio puede recibir la entrada nula, que debe ir directamente a la salida.

- **Desplazador**: Los nodos desplazadores realizan un desplazamiento binario de la entrada que reciben. En el presente trabajo, no se contemplan los desplazamientos hacia la derecha, ya que no suelen ser utilizados más allá de la teoría y pueden presentar complicaciones de representación. Para un desplazamiento de n posiciones de una entrada E, se representa el nodo desplazador como $n(E)$.

  $$n(E) = E \ll n = 2^n E$$

  Este nodo puede recibir una entrada de cualquier nodo que no sea otro desplazador, ya que la operación resultante equivale a la de un desplazador de grado mayor. Se pone esta limitación porque no existe ninguna ventaja en la combinación de desplazadores.

- **Operador**: Los nodos operadores son los nodos encargados de sumar o restar dos entradas. Para dos entradas E_1 y E_2, se representa un sumador como $++(E_1, E_2)$ y un restador como $+-(E_1, E_2)$. Este nodo no tiene restricciones de ningún tipo.

- **Salida**: Los nodos salida son nodos finales, no pueden tener conexiones de salida hacia otros nodos del bloque. Estos nodos pueden contener un negador que modifique el signo de la salida, representándose la salida como $-\{E\}$. En caso de no tener este negador, se representan como $+\{E\}$. Se decide utilizar los separadores distintivos { y } para dar más claridad visual a la representación.

Para las salidas de la Fig. 28, se obtienen las siguientes representaciones:

$$C0 = +\{0\}$$
$$C1 = -\{++(3(1), 1)\}$$
$$C2 = +\{+-(8(1), ++(3(1), 1))\}$$

La representación total del bloque se obtiene concatenando la representación de cada una de las salidas de la siguiente manera:

$$+\{0\}; -\{++(3(1), 1)\}; +\{+-(8(1), ++(3(1), 1))\}$$

5.1.2. Obtención de la gramática utilizada

Para generar la estructura de bloques multiplicadores, se decide utilizar una gramática libre de contexto. Se utilizan dos parámetros para la gramática utilizada: **d**, el límite de desplazamiento y **s**, el número de salidas del bloque a construir.

Se comienza asociando un elemento de la gramática a cada tipo de nodo:

- $\mathbf{1} \in \sum$ para la entrada introducida al circuito.
- $\mathbf{0} \in \sum$ para la entrada nula.
- $\mathbf{L} \in N$ para el desplazador.
- $\mathbf{A} \in N$ para el operador.
- $\mathbf{B} \in N$ para la salida.

A continuación, se procede a crear producciones a partir de la representación para cada tipo de nodo de forma que se interconecte con otros nodos.
Un nodo desplazador L puede recibir la salida de un operador sobre la que realizar un desplazamiento D:

$$\begin{aligned} L &\Rightarrow D(A) \\ D &\Rightarrow 1 \mid \ldots \mid d \end{aligned}$$

Un nodo operador A puede recibir las salidas tanto de operadores como desplazadores sobre las que realizar una operación O:

$$\begin{aligned} A &\Rightarrow O(N,N) \\ N &\Rightarrow L \mid A \\ O &\Rightarrow ++ \mid +- \end{aligned}$$

Un nodo salida B con un signo I puede recibir la salida de un nodo intermedio, operador o desplazador, o de una entrada nula:

$$\begin{aligned} B &\Rightarrow I\{N\} \mid I\{0\} \\ I &\Rightarrow + \mid - \end{aligned}$$

Es necesario incluir la entrada introducida al circuito, **1**, dentro de alguna de las producciones anteriores. Se decide colocar en las producciones recursivas de L y A, de forma que la recursión pueda finalizar:

$$\begin{aligned} L &\Rightarrow D(A) \mid 1 \\ A &\Rightarrow O(N,N) \mid 1 \end{aligned}$$

Por último, la producción que parte del elemento inicial de la gramática y que agrupa todas las salidas:

$$S \Rightarrow B[;B]^{s-1}$$

Integrando todas las producciones que se enuncian anteriormente y que definen las posibles conexiones entre los diferentes nodos del circuito, se obtiene la gramática $G = (N, \sum, P_{s,d}, S)$. Además, se introducen algunas producciones probabilísticas:

$$
G = \begin{cases}
N = \{S, B, R, N, L, A, I, D, O\} \\
\sum = \{,, +, -, ++, +-, ;, (,), \{, \}, 0, \ldots, d\} \\
P_{s,d} = \begin{cases}
S \Rightarrow B[;B]^{s-1} \\
B \Rightarrow I\{R\} \\
R \Rightarrow N \mid 0 \begin{cases} P(R \Rightarrow N) = 95\,\% \\ P(R \Rightarrow 0) = 5\,\% \end{cases} \\
N \Rightarrow L \mid A \\
L \Rightarrow D(A) \mid 1 \begin{cases} P(L \Rightarrow D(A)) = 60\,\% \\ P(L \Rightarrow 1) = 40\,\% \end{cases} \\
A \Rightarrow O(N, N) \mid 1 \begin{cases} P(A \Rightarrow O(N, N)) = 60\,\% \\ P(A \Rightarrow 1) = 40\,\% \end{cases} \\
I \Rightarrow + \mid - \\
D \Rightarrow 1 \mid \ldots \mid d \\
O \Rightarrow ++ \mid +-
\end{cases}
\end{cases}
$$

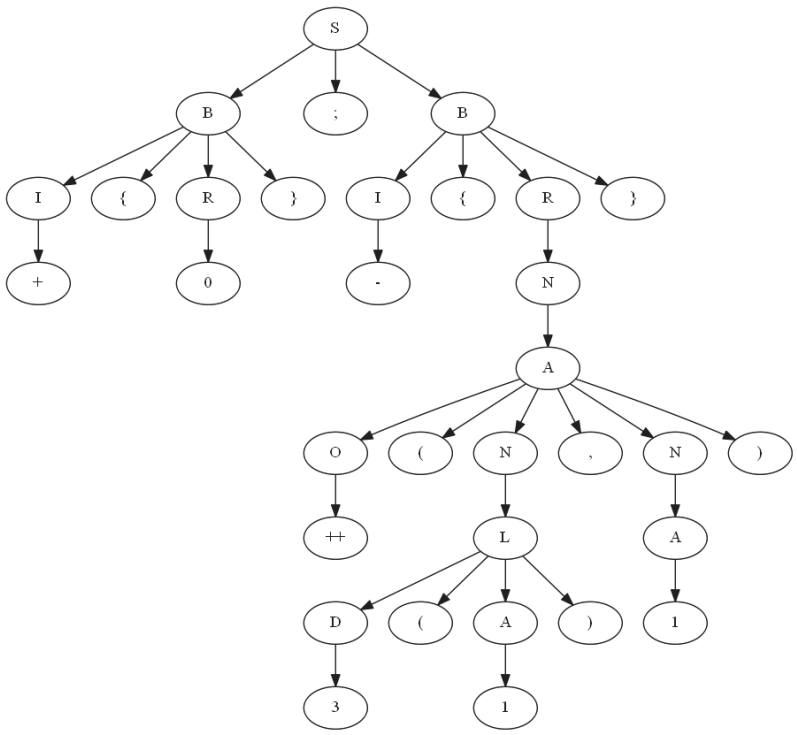


Fig. 29: Árbol de derivación que representa la palabra $+\{0\}; -\{++(3(1), 1)\}$

5.2. Evaluación

La función de evaluación es el mecanismo de los métodos evolutivos que hace posible guiar la búsqueda mediante la valoración de soluciones, proceso que permite distinguir las mejores soluciones de una población.

Para la evaluación de los bloques multiplicadores, se decide utilizar dos criterios de referencia:

1. El error del filtro FIR
2. El área del bloque multiplicador

5.2.1. Error de filtrado

El error de filtrado es el error de aproximación experimentado por la salida de un filtro FIR con respecto a la salida considerada ideal. El error de salida ε_y se calcula en función de la salida ideal esperada $y^t(i)$ de la siguiente manera:

$$\varepsilon_y = \sum_{i=0}^{N} \frac{|y^t(i) - y(i)|}{y^t(i)}$$

Este es el método directo de cálculo de la precisión de filtrado para una determinada onda de entrada y por ello el que lo representa con mayor realismo.
El principal problema de utilizar este método para la evaluación de bloques multiplicadores reside en la selección de las ondas utilizadas para la evaluación de forma que no se incurra en un sobreajuste hacia un tipo de onda determinada.

Por ello, se recurre al método indirecto de cálculo del error centrado en el valor de los coeficientes:

$$\Delta y = \sum_{k=0}^{N} |\Delta b_k|$$

5.2.2. Área del bloque multiplicador

Para el cálculo del área de un bloque multiplicador, es necesario conocer la precisión de la señal p que necesite el filtro FIR para recibir y transmitir. Esta precisión determina el valor máximo de entrada que puede procesar el circuito, debiendo ser capaces los componentes de procesar múltiplos de este valor. El área de cada nodo del circuito es directamente proporcional a esta longitud de entrada, equivaliendo la suma de todos los nodos a la del bloque multiplicador. El único tipo de nodo que no tiene área es el desplazador, ya que no es más que un desplazamiento de conexiones en el circuito.

La función de área de los operadores y negadores sigue una función lineal de tipo $A = kx - b$, siendo x una variable relacionada con el tamaño de las entradas. Las constantes k y b para cada tipo de componente se pueden obtener a partir de la librería *firgen* de Spiral[44], escrita en Perl. La variable x requerida para el cálculo del área depende necesariamente de la precisión de señal p así como de la longitud binaria de las entradas.

Dado que no se consideran los desplazadores hacia la derecha, es posible reducir el área de los operadores en función del número de ceros hasta el primer uno por la derecha que pueden tener los operandos E_1 y E_2, que se decide denotar como la función $r0(E_i)$. Sabiendo que todos los números procesados dentro del circuito son múltiplos de una entrada inicial, se puede deducir el número de ceros introducidos por la derecha en cada operando en función de su valor con respecto a la entrada.
Se puede concluir que el número de ceros por la derecha para los múltiplos de la entrada, $r0(k \cdot e)$, es como mínimo el de los presentes en la representación binaria del factor que multiplica la entrada, $r0(k)$. A continuación, se presenta la comprobación de esta propiedad para una entrada $e = \ldots 1$, cuyo valor final da lugar al mínimo número de ceros por la derecha para todos los múltiplos de cualquier entrada:

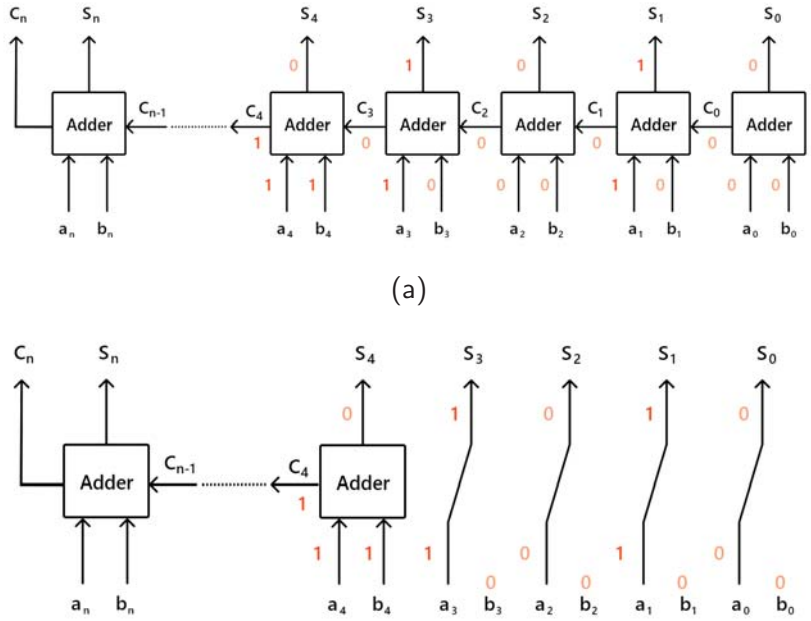
$$
\begin{aligned}
&r0(e) = r0(\ldots 1_2) = r0(1) = 0\\
&r0(2e) = r0(e \ll 1) = r0(\ldots 10_2) = r0(10_2) = r0(2_{10}) = 1\\
&r0(3e) = r0(e \ll 1 + e) = r0(\ldots 11_2) = r0(11_2) = r0(3_{10}) = 0\\
&r0(4e) = r0(e \ll 2) = r0(\ldots 100_2) = r0(100_2) = r0(4_{10}) = 2\\
&\ldots
\end{aligned}
$$

Esta posibilidad de reducción del área se debe a que los ceros introducidos por la derecha son bits que en algunos casos representan el elemento neutro de la operación y, por tanto, no tienen ningún efecto en ella. Para la suma, se ahorran tantas posiciones como el mayor número de ellos en ambos operadores, mientras que para la resta solo es relevante el número en el segundo operando.
De darse estas circunstancias, es posible construir un circuito como el de la Fig. 30 que devuelva las cifras que recibe del operando no neutro sin necesidad de ningún componente, por tanto, obteniendo un ahorro de área.

En la construcción del circuito, existe la posibilidad de que se formen circuitos innecesarios de operadores que den como resultado el valor nulo. La optimización de la Fig. 30 también aplica en este caso para sumadores con al menos una entrada nula y para restadores con $E_2 = 0$, pasando a considerarse estos de área cero. Además, un restador con $E_1 = 0, E_2 \neq 0$, se trata como un negador a efectos de área.

Finalmente, las funciones de cada tipo de nodo son las siguientes:

$$
\begin{aligned}
A_{sumador} &= 69,8544 \cdot (p + \lceil \log_2 \text{máx}(E_1, E_2) \rceil - \text{máx}(r0(E_1), r0(E_2))) - 39,9168\\
A_{restador} &= 76,5071 \cdot (p + \lceil \log_2 \text{máx}(E_1, E_2) \rceil - r0(E_2)) - 39,9167\\
A_{negador} &= 46,5696 \cdot (p + \lceil \log_2 E \rceil) - 53,2224
\end{aligned}
$$



(a)

(b)

Fig. 30: Comparación entre los circuitos de un sumador convencional (a) y otro equivalente con el área optimizada (b) para $E_1 = \ldots 11010$ y $E_2 = \ldots 10000$ y todos sus múltiplos

5.2.3. Eficiencia de Pareto

Dado que el problema a resolver tiene más de un criterio de optimización, éste recibe el nombre de *problema multiobjetivo*. Es posible que puedan existir conflictos entre los distintos objetivos en este tipo de problema, impidiendo que se pueda dar una optimización simultánea de cada objetivo[23].

La solución más general para evaluar este tipo de problemas está en el diseño de una función que aglutine todas las funciones objetivo de forma que su resultado se pueda tratar como un solo objetivo[23]. El problema de este enfoque está en la selección adecuada de pesos que permita llegar a una conclusión acertada. En la práctica, tal selección puede resultar en un problema muy complejo hasta para un experto en el dominio del problema, pudiendo acabar obteniendo soluciones completamente diferentes a las deseadas en caso de hacerlo incorrectamente[23].

Alternativamente, se puede enfocar este problema realizando una búsqueda en la que cada solución de un conjunto de soluciones cumpla con todos los objetivos a un nivel aceptable sin que sea *dominada* por otras soluciones[23]. Para un problema de optimización, se dice que una solución x domina a otra solución y si $f(x) > f(y)$ para al menos una función objetivo, siempre que para el resto de ellas $f_i(x) \geq f_i(y)$[23].

Cuando existe una solución que no es dominada por ninguna otra en el espacio de soluciones, se habla de una *solución óptima de Pareto*. Esta solución se considera óptima ya que no se puede mejorar el resultado de un objetivo sin que se empeore en otro[23]. El conjunto de soluciones óptimas que se puede obtener de manera factible para un problema recibe el nombre de *conjunto óptimo de Pareto*, para el que sus correspondientes valores de funciones objetivo dan lugar a la *frontera de Pareto*[23]. Cada solución recibe una posición en el *ranking de Pareto* en función del número de soluciones en la población que tengan dominancia sobre ella, estando en la primera posición las soluciones de frontera.

Para la evaluación de las soluciones, se decide utilizar el ranking de Pareto para los objetivos de error y área. Además, entre soluciones con el mismo ranking de Pareto, reciben evaluaciones mejores aquellas soluciones que posean menor error.

6. RESULTADOS

En este capítulo dedicado a los resultados es necesario elegir tanto las configuraciones concretas utilizadas para obtener los resultados como los casos particulares de bloques multiplicadores que se van a construir que tengan una reducida complejidad de forma que sea factible su resolución.

En cuanto a las configuraciones utilizadas para obtener los resultados, se decide fijar los siguientes operadores con las siguientes justificaciones:

- Se ha observado que los datos de evolución de los dos tipos operadores de cruce y mutación son bastante semejantes, haciendo la comparativa poco interesante. Se decide utilizar los operadores GBC y GBM por los mecanismos de control de profundidad que poseen, que permiten una ejecución más estable.
- Se decide fijar la inclusión de los progenitores en la formación de las nuevas generaciones, ya que de otra manera la eficiencia del algoritmo sufre en gran medida perdiéndose los avances obtenidos.
- Finalmente, se prohíben las soluciones duplicadas en la nueva población. Las soluciones duplicadas dan lugar a problemas de diversidad en la población que hacen que esta caiga en convergencia prematura.

Los operadores de selección se utilizan de la siguiente manera:

- La ruleta otorga a cada solución un peso de $\frac{1}{\sqrt{ranking}}$, de forma que la probabilidad de selección disminuya suavemente para las soluciones con peor ranking.
- El torneo probabilístico utiliza los parámetros $t = 2$, $p = \frac{1}{3}$, siendo t el número de individuos seleccionados y p la probabilidad de selección del individuo.

Se decide que los casos particulares de bloques multiplicadores a resolver tengan en común los parámetros de orden $N = 2$, 8 bits fraccionales, precisión de señal de 128 bits y tipo *paso bajo*, de forma que sean soluciones de complejidad similar y de fácil comparación. Las salidas de los bloques multiplicadores objetivo quedan acotadas por debajo del valor de 256. Se diseñan cuatro ejemplos que difieran en la frecuencia de corte con los siguientes valores:

1. $f_c = 0{,}5$
2. $f_c = 0{,}9$
3. $f_c = 0{,}1$
4. $f_c = 0{,}4$

Se procede a recoger los resultados de 100 ejecuciones para cada uno de estos ejemplos y para cada una de las configuraciones elegidas para las que solo varía el operador de selección, configuraciones que en adelante reciben el nombre de *configuración ruleta* y *configuración torneo*.

Las secciones dedicadas a cada ejemplo tienen la siguiente estructura:

1. Introducción al objetivo del problema. Se comienza ilustrando el objetivo a resolver con una tabla que contiene los siguientes campos:
   - los coeficientes del *filtro Matlab*, los coeficientes originales $\{b_k\}$.
   - los coeficientes del *filtro Spiral*, coeficientes fraccionarios $\{c_k\}$ obtenidos mediante el truncamiento de los coeficientes originales tras ser multiplicados por el factor de desplazamiento 2^f de los bits fraccionales.
   - el error Δb_k asociado al truncamiento de cada coeficiente.
2. A continuación, se muestra una figura de la estructura de la *solución Spiral* obtenida a partir de los coeficientes del filtro Matlab y la opción de optimización de área. Esta figura está acompañada de las características de error y área que se obtienen mediante el método de evaluación del presente trabajo, de forma que sea visible el objetivo que el algoritmo debe superar.
3. Más adelante, se comentan de forma breve los resultados obtenidos por ambas configuraciones.
4. Para finalizar, se muestran las gráficas de evolución de error y área obtenidas para las ejecuciones detenidas tras 300 generaciones, con dos series para cada configuración utilizada y otra estática para marcar el objetivo, el valor obtenido en la *solución Spiral*. Además, se presentan diagramas de caja con los resultados de área obtenidos.

6.1. Ejemplo 1

Los parámetros utilizados para el filtro FIR y el bloque multiplicador en este ejemplo son los de $N = 2$, $f_c = 0{,}5$, tipo *paso bajo*, 8 bits fraccionales y precisión de señal de 128 bits. Para estos parámetros, se obtiene la siguiente tabla:

k	0, 2	1
b_k	$4,6221 \cdot 10^{-2}$	0,9075
$2^f b_k$	11,8327	232,3345
$c_k = \lfloor 2^f b_k \rfloor$	11	232
Δb_k	$-3,2527 \cdot 10^{-3}$	$-1,3070 \cdot 10^{-3}$

Tab. 2: Coeficientes correspondientes a la configuración elegida con el error asociado Δb_k para los coeficientes enteros de Spiral

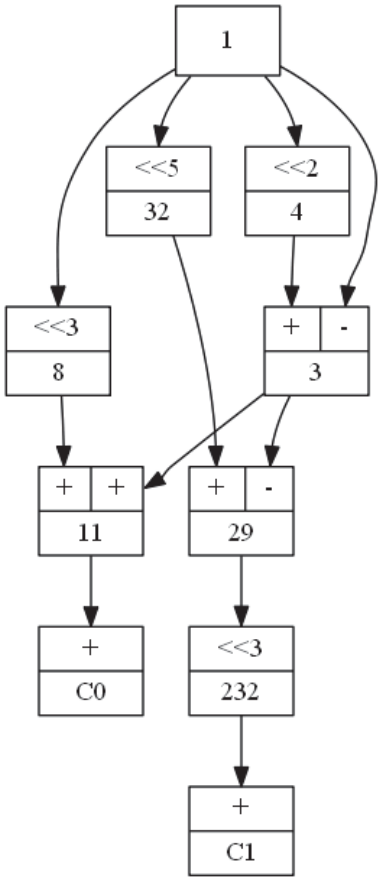


Fig. 31: Solución obtenida mediante el generador Spiral

La solución contenida en la Fig. 31 sintetiza los coeficientes $\{11, 232\}$. Por medio de 3 nodos operadores, realiza las siguientes operaciones:

$$\begin{cases} 11x = x \ll 3 + (x \ll 2 - x) \\ 232x = (x \ll 5 - (x \ll 2 - x)) \ll 3 \end{cases}$$

La solución Spiral tiene un error de $7,8125 \cdot 10^{-3}$ y un área de $28943,0058$. El error óptimo obtenido es de $2,6140 \cdot 10^{-3}$, presente en la generación 300 en el 100 % de los casos para la configuración ruleta y 98 % para la configuración torneo. El 2 % restante corresponde a errores de $3,9062 \cdot 10^{-3}$. El área óptima obtenida es de $19036,9870$, representando un valor atípico en ambas configuraciones.

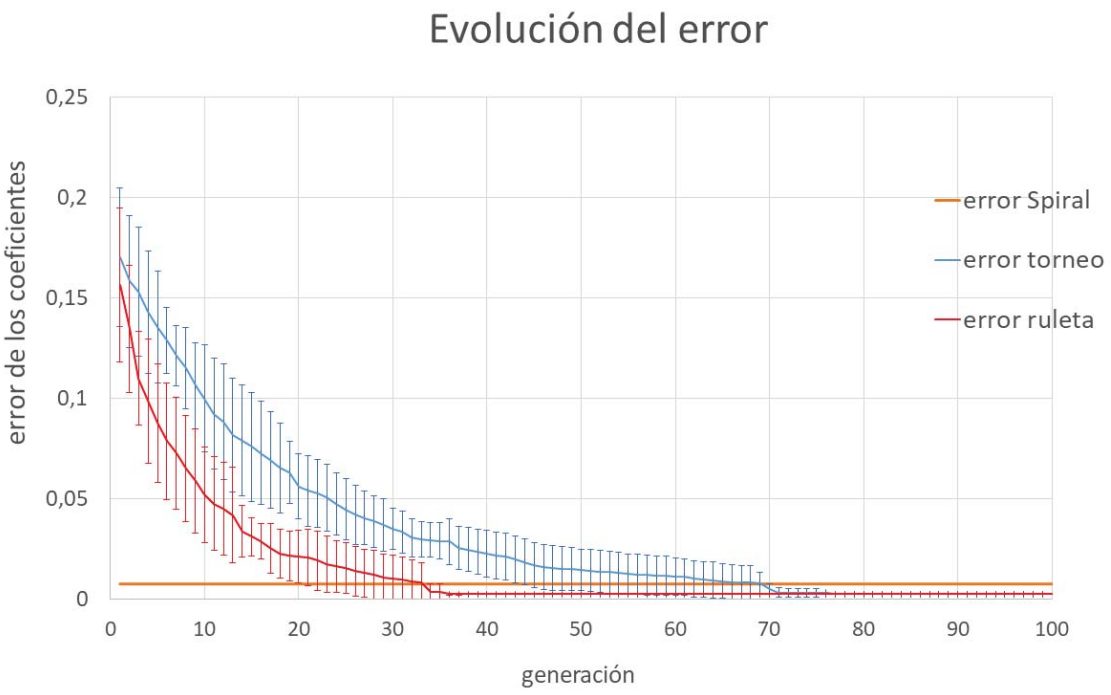


Fig. 32: Gráfica de evolución del error mínimo

En la Fig. 32, se presenta el valor medio del error mínimo con barras de error que representan la desviación típica. Para esta gráfica, se ve que la configuración ruleta alcanza el objetivo en una duración de 35 generaciones, mientras que la configuración torneo tiene una duración de algo menos de 70.

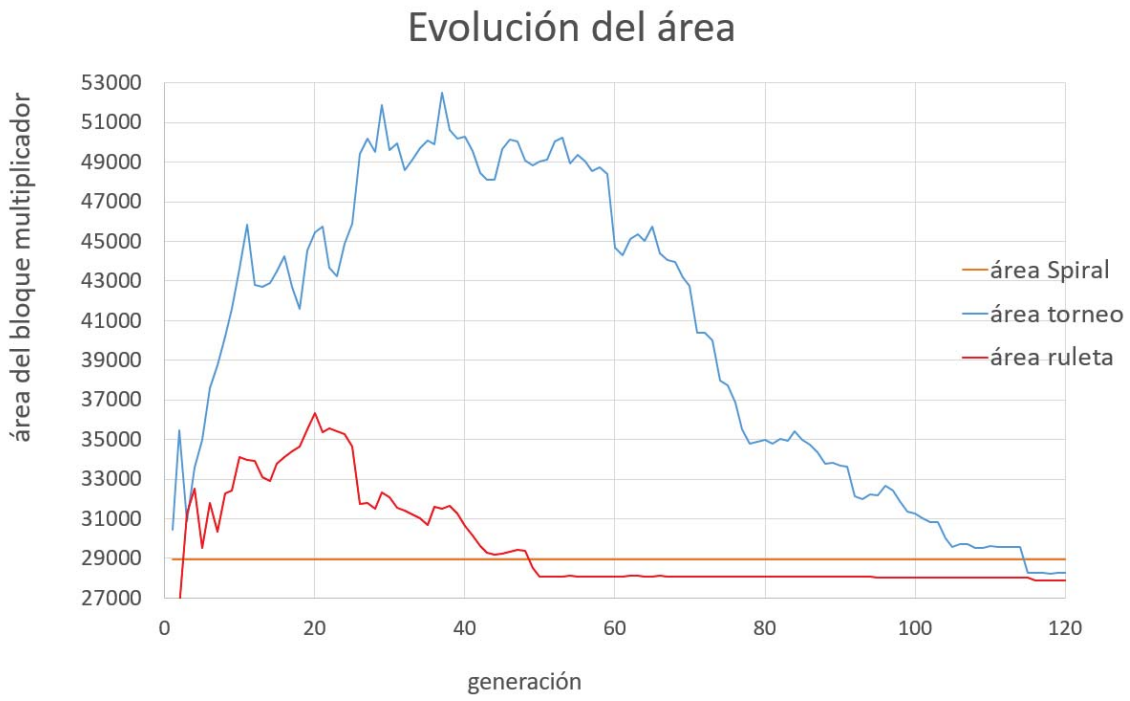


Fig. 33: Gráfica de evolución del área media correspondiente al error mínimo

La Fig. 33 representa el valor medio del área que tienen las soluciones con error mínimo. Para el área, la configuración ruleta alcanza el objetivo en unas 50 generaciones, mientras que la configuración torneo lo hace pasadas 110 generaciones.

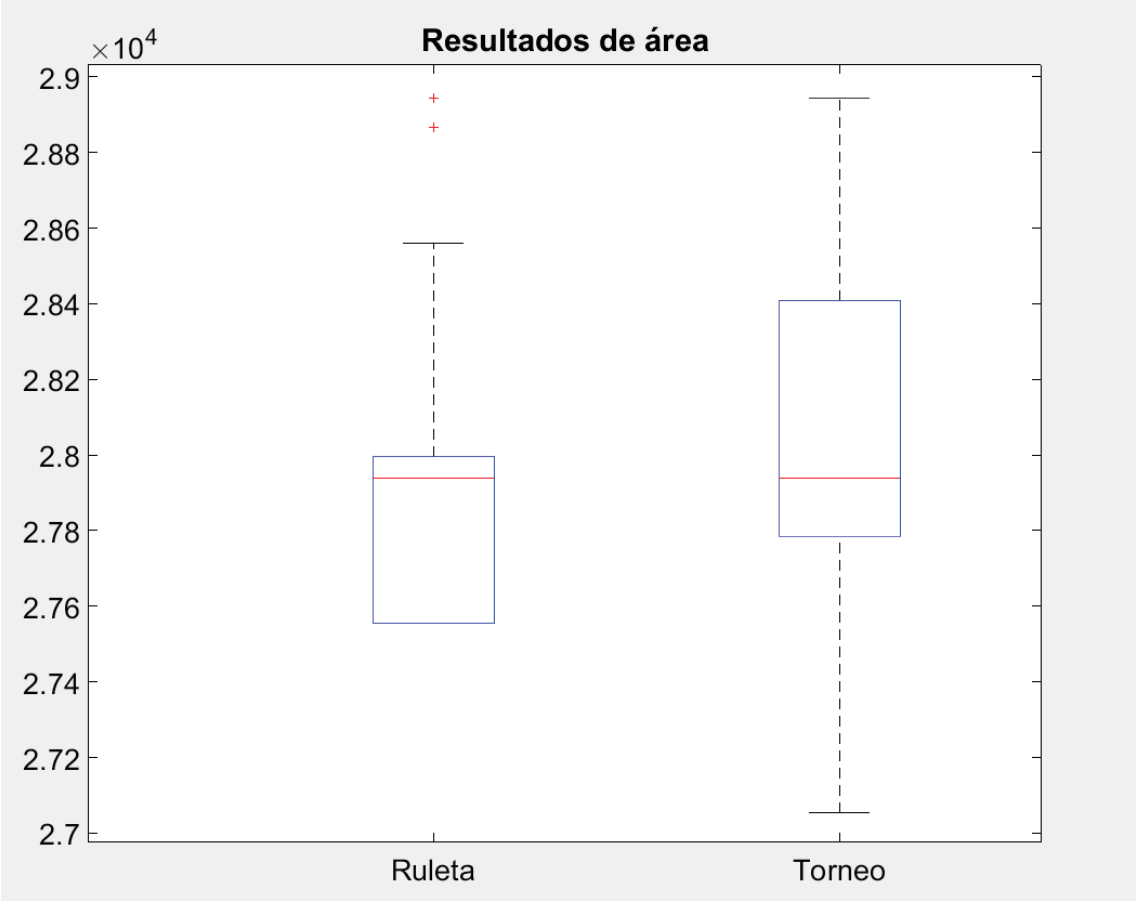


Fig. 34: Resultados de área representados por diagrama de caja

La Fig. 34 contiene los resultados finales de área, incluyéndose los valores atípicos en la Fig. 35. Para estas figuras, ambas configuraciones tienen la mediana idéntica de valor $27938,43$. La configuración ruleta se distribuye más hacia valores inferiores de esta mediana, al contrario que la configuración torneo que lo hace hacia valores superiores. Además, los bigotes de esta última son más amplios, lo que indica una desviación típica superior y por tanto es un sistema menos predecible.

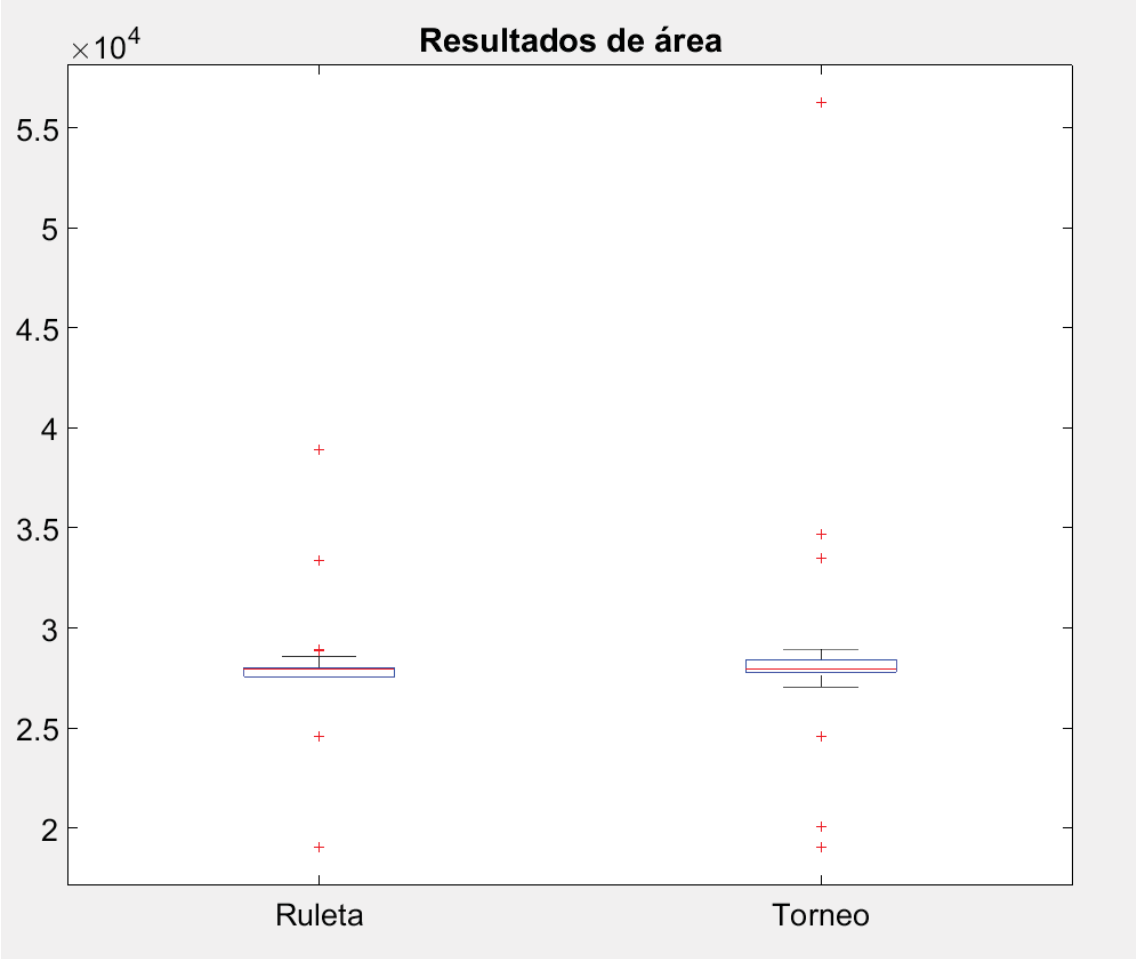


Fig. 35: Resultados de área representados por diagrama de caja, incluyendo los valores atípicos

En la Fig. 36, se presentan tres soluciones óptimas equivalentes obtenidas con el sistema evolutivo propuesto. Utilizando solo dos nodos operadores, las soluciones realizan las siguientes operaciones :

$$a) \begin{cases} 12x = x \ll 3 + x \ll 2 \\ 232x = x \ll 8 - (x \ll 3 + x \ll 2) \ll 1 \end{cases}$$

$$b) \begin{cases} 12x = x \ll 3 + x \ll 2 \\ 232x = (x \ll 7 - (x \ll 3 + x \ll 2)) \ll 1 \end{cases}$$

$$c) \begin{cases} 12x = (x \ll 1 + x) \ll 2 \\ 232x = x \ll 8 - (x \ll 1 + x) \ll 3 \end{cases}$$

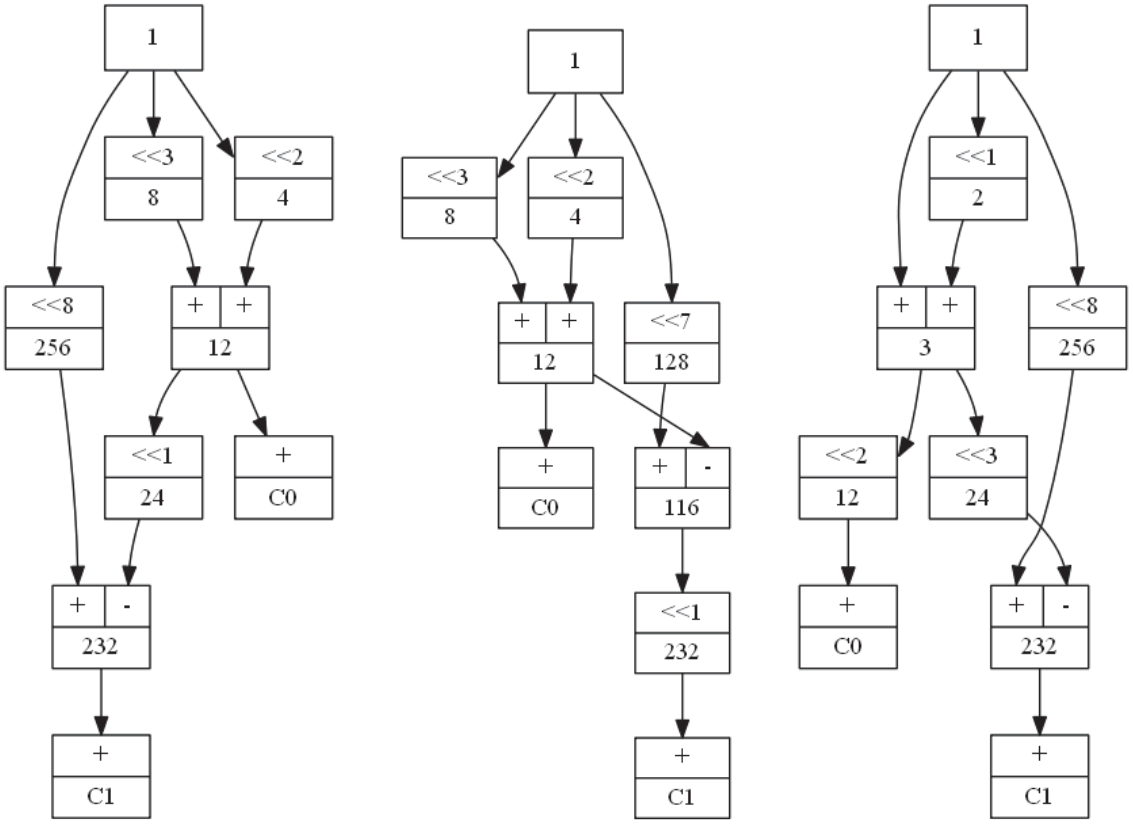


Fig. 36: Soluciones óptimas

6.2. Ejemplo 2

Los parámetros utilizados para el filtro FIR y el bloque multiplicador en este ejemplo son los de $N = 2$, $f_c = 0{,}9$, tipo *paso bajo*, 8 bits fraccionales y precisión de señal de 128 bits. Para estos parámetros, se obtiene la siguiente tabla:

k	0, 2	1
b_k	$8,5931 \cdot 10^{-3}$	0,9828
$2^f b_k$	2,1998	251,6003
$c_k = \lfloor 2^f b_k \rfloor$	2	251
Δb_k	$-7,8062 \cdot 10^{-4}$	$-2,3450 \cdot 10^{-3}$

Tab. 3: Coeficientes correspondientes a la configuración elegida con el error asociado Δb_k para los coeficientes enteros de Spiral

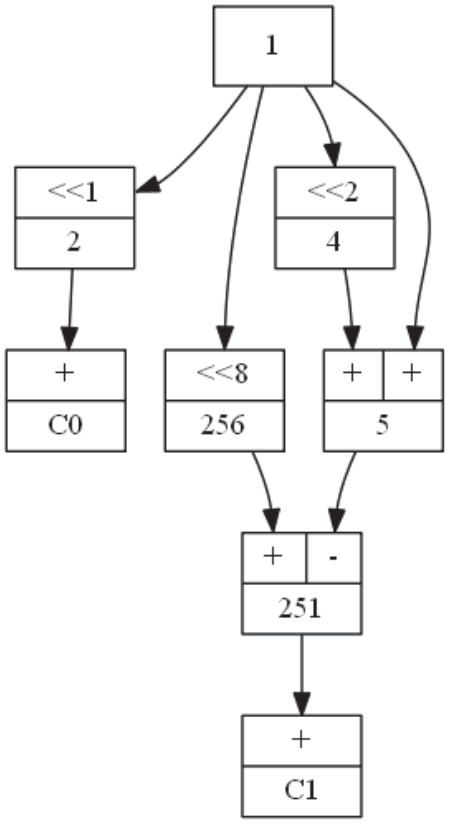


Fig. 37: Solución obtenida mediante el generador Spiral

La solución contenida en la Fig. 37 sintetiza los coeficientes $\{2, 251\}$. Por medio de 2 nodos operadores, realiza las siguientes operaciones:

$$\begin{cases} 2x = x \ll 1 \\ 251x = x \ll 8 - (x \ll 2 + x) \end{cases}$$

La solución Spiral tiene un error de $3,9063 \cdot 10^{-3}$ y un área de $19266,5086$. El error óptimo obtenido es de $3,1225 \cdot 10^{-3}$, presente en la generación 300 en el 100 % de los casos para ambas configuraciones. El área óptima obtenida es de $10212,0475$, la mediana en la configuración torneo y el límite inferior de la caja en ambas configuraciones.

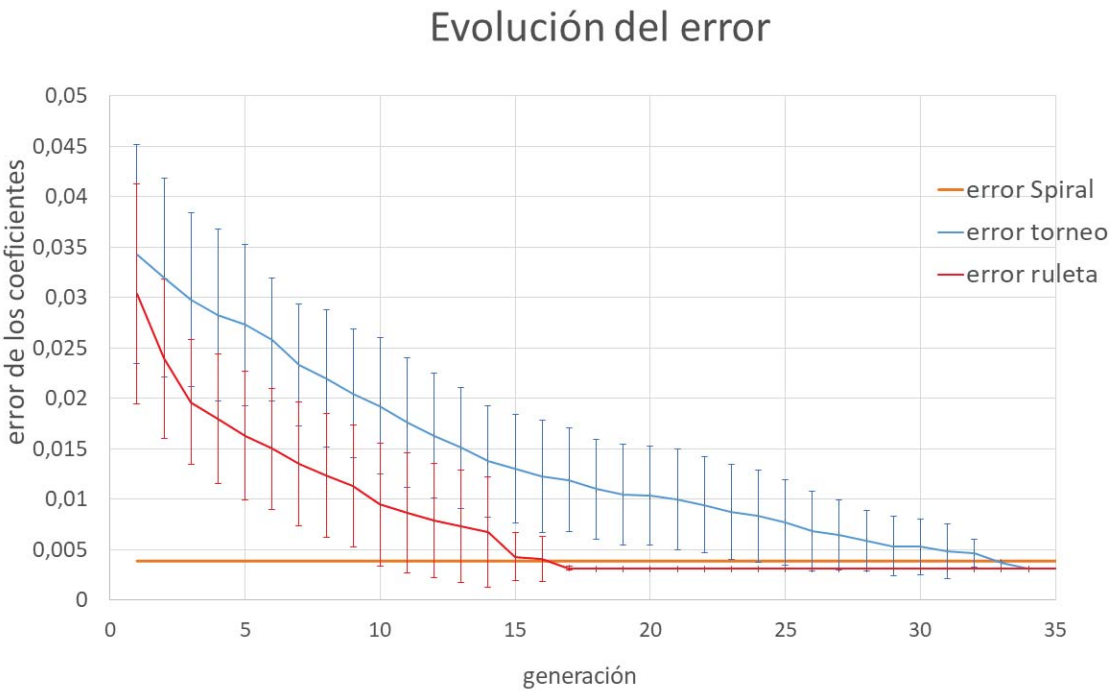


Fig. 38: Gráfica de evolución del error mínimo

En la Fig. 38, se presenta el valor medio del error mínimo con barras de error que representan la desviación típica. Para esta gráfica, se ve que la configuración ruleta alcanza el objetivo en una duración de 17 generaciones, mientras que la configuración torneo tiene una duración de 35.

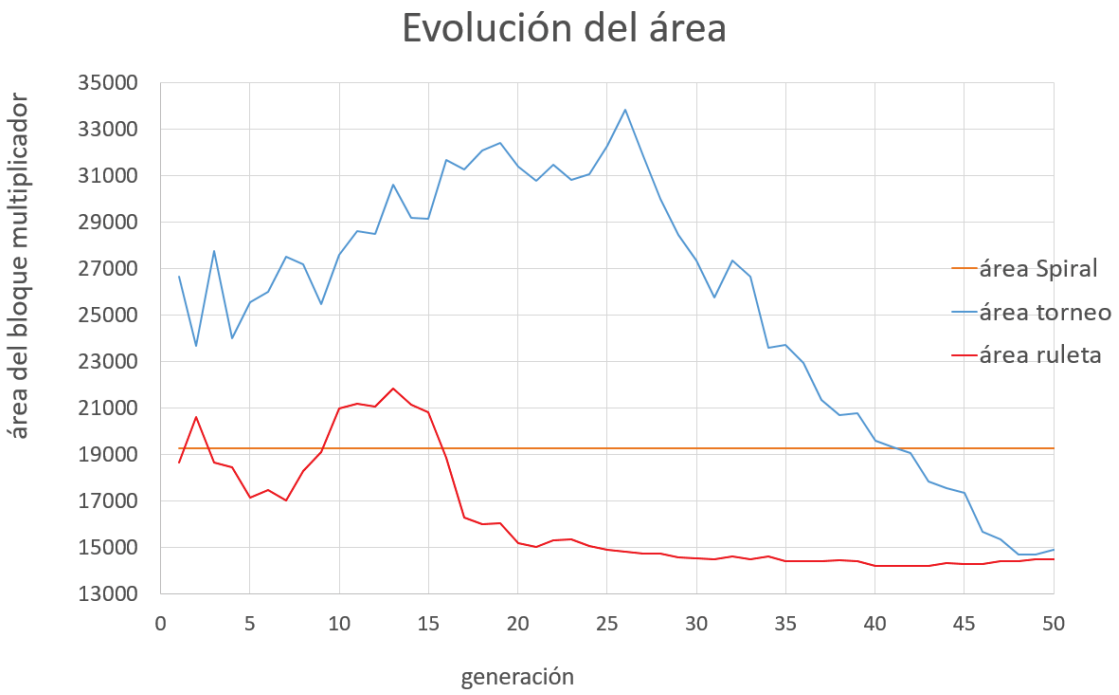


Fig. 39: Gráfica de evolución del área media correspondiente al error mínimo

La Fig. 39 representa el valor medio del área que tienen las soluciones con error mínimo. Para el área, la configuración ruleta alcanza el objetivo pasadas las 15 generaciones, mientras que la configuración torneo lo hace pasadas 40 generaciones.

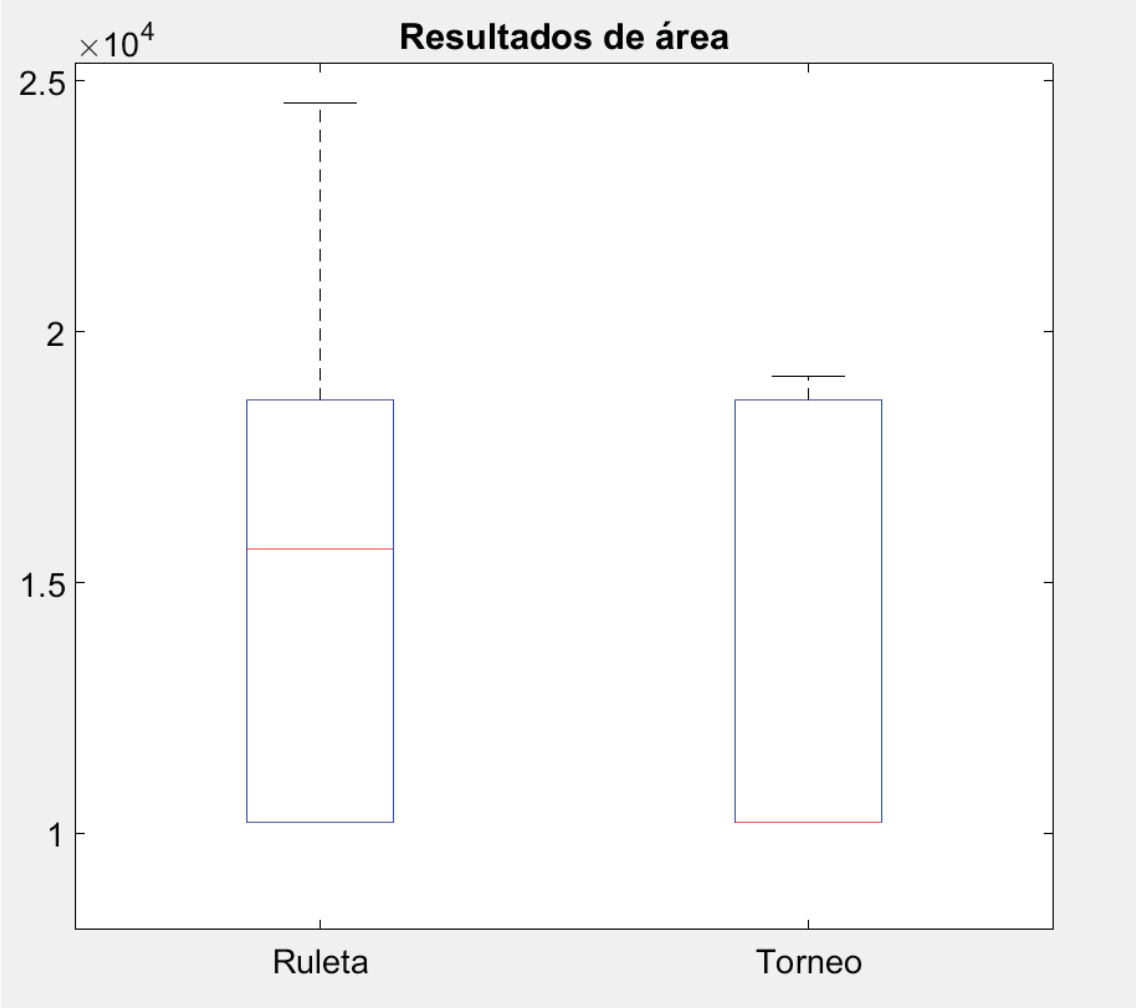


Fig. 40: Resultados de área representados por diagrama de caja

La Fig. 40 contiene los resultados finales de área. Para esta figura, los límites de las cajas de ambas configuraciones son idénticos. Sin embargo, la mediana de la configuración torneo coincide con su límite inferior, mientras que para la configuración ruleta está muy por encima. La configuración ruleta tiene el bigote superior mucho más amplio debido a una mayor desviación típica, lo que convierte a esta configuración en más impredecible.

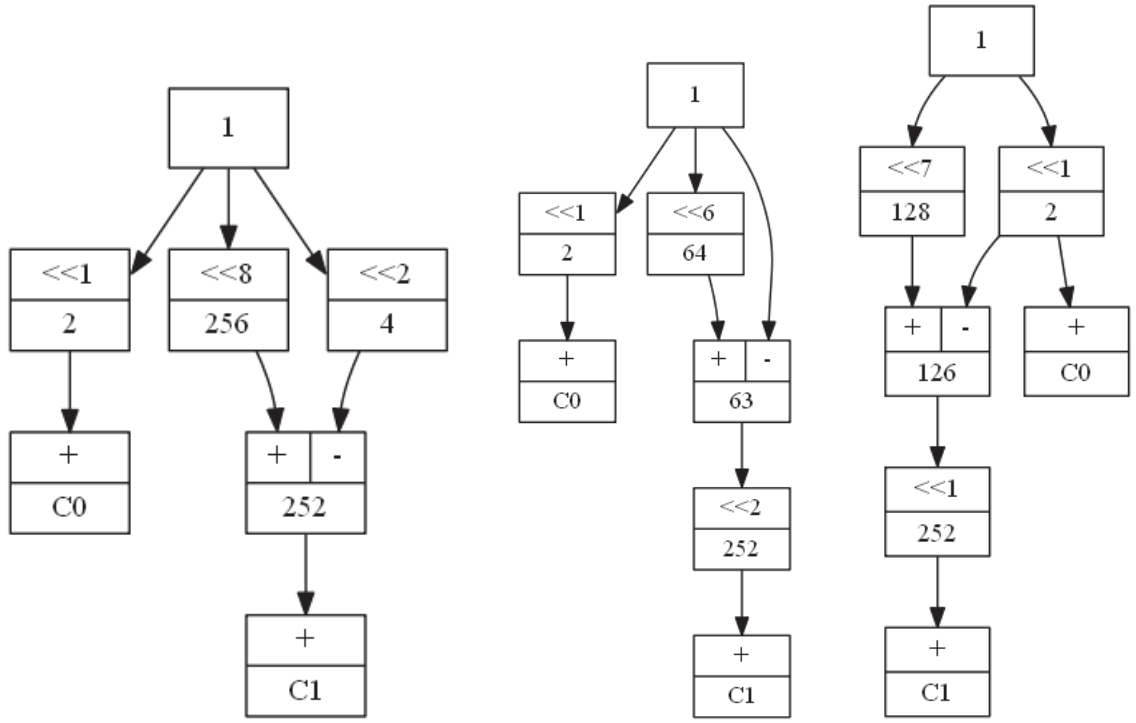


Fig. 41: Soluciones óptimas

En la Fig. 41, se presentan tres soluciones óptimas equivalentes obtenidas con el sistema evolutivo propuesto. Utilizando solo un nodo operador, las soluciones realizan las siguientes operaciones:

$$a) \begin{cases} 2x = x \ll 1 \\ 252x = x \ll 8 - x \ll 2 \end{cases}$$

$$b) \begin{cases} 2x = x \ll 1 \\ 252x = (x \ll 6 - x) \ll 2 \end{cases}$$

$$c) \begin{cases} 2x = x \ll 1 \\ 252x = (x \ll 7 - x \ll 1) \ll 1 \end{cases}$$

6.3. Ejemplo 3

Los parámetros utilizados para el filtro FIR y el bloque multiplicador en este ejemplo son los de $N = 2$, $f_c = 0{,}1$, tipo *paso bajo*, 8 bits fraccionales y precisión de señal de 128 bits. Para estos parámetros, se obtiene la siguiente tabla:

k	0, 2	1
b_k	$6,79901 \cdot 10^{-2}$	$0,86401$
$2^f b_k$	17,4054	221,1890
$c_k = \lfloor 2^f b_k \rfloor$	17	221
Δb_k	$-1,5839 \cdot 10^{-3}$	$-7,3841 \cdot 10^{-4}$

Tab. 4: Coeficientes correspondientes a la configuración elegida con el error asociado Δb_k para los coeficientes enteros de Spiral

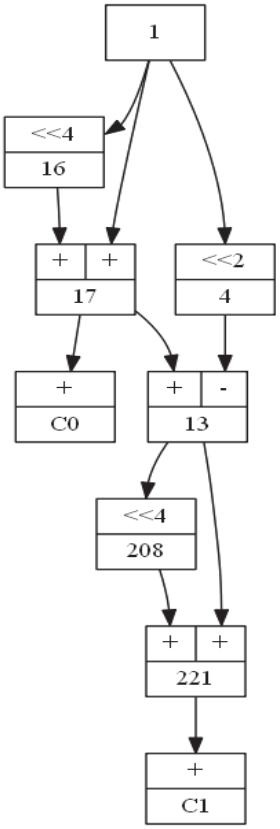


Fig. 42: Solución obtenida mediante el generador Spiral

La solución contenida en la Fig. 42 sintetiza los coeficientes $\{17, 221\}$. Por medio de 3 nodos operadores, realiza las siguientes operaciones:

$$\begin{cases} 17x = x \ll 4 + x \\ 221x = (x \ll 4 + x - x \ll 2) \ll 4 + (x \ll 4 + x - x \ll 2) \end{cases}$$

Al contrario que en ejemplos anteriores, los coeficientes enteros de Spiral en la Tab. 4 están correctamente redondeados y, por lo tanto, son óptimos. La solución Spiral tiene un error de $3,9062 \cdot 10^{-3}$ y un área de $28064,8369$.

El error óptimo obtenido es $3,9062 \cdot 10^{-3}$, el mismo que el del objetivo. Este error se obtiene en la generación 300 en el 74 % de los casos para la configuración ruleta y en el 87 % para la configuración torneo. En la Fig. 43 se muestra que ambas configuraciones tienen como mediana el error óptimo. Los valores típicos de la configuración torneo están contenidos en este valor, mientras que la configuración ruleta se distribuye hacia valores muy superiores.
El área óptima obtenida es de $18883,9726$ (Fig. 49a) para un error subóptimo de $1,1718 \cdot 10^{-2}$ en la configuración torneo. El área óptima obtenida (Fig. 49b) para una solución de error óptimo es de $28517,2273$, un valor próximo al objetivo obtenido en ambas configuraciones. Más adelante, en las Fig. 46, 47 y 48 se presentan diagramas de caja que muestran la dispersión de estos resultados.

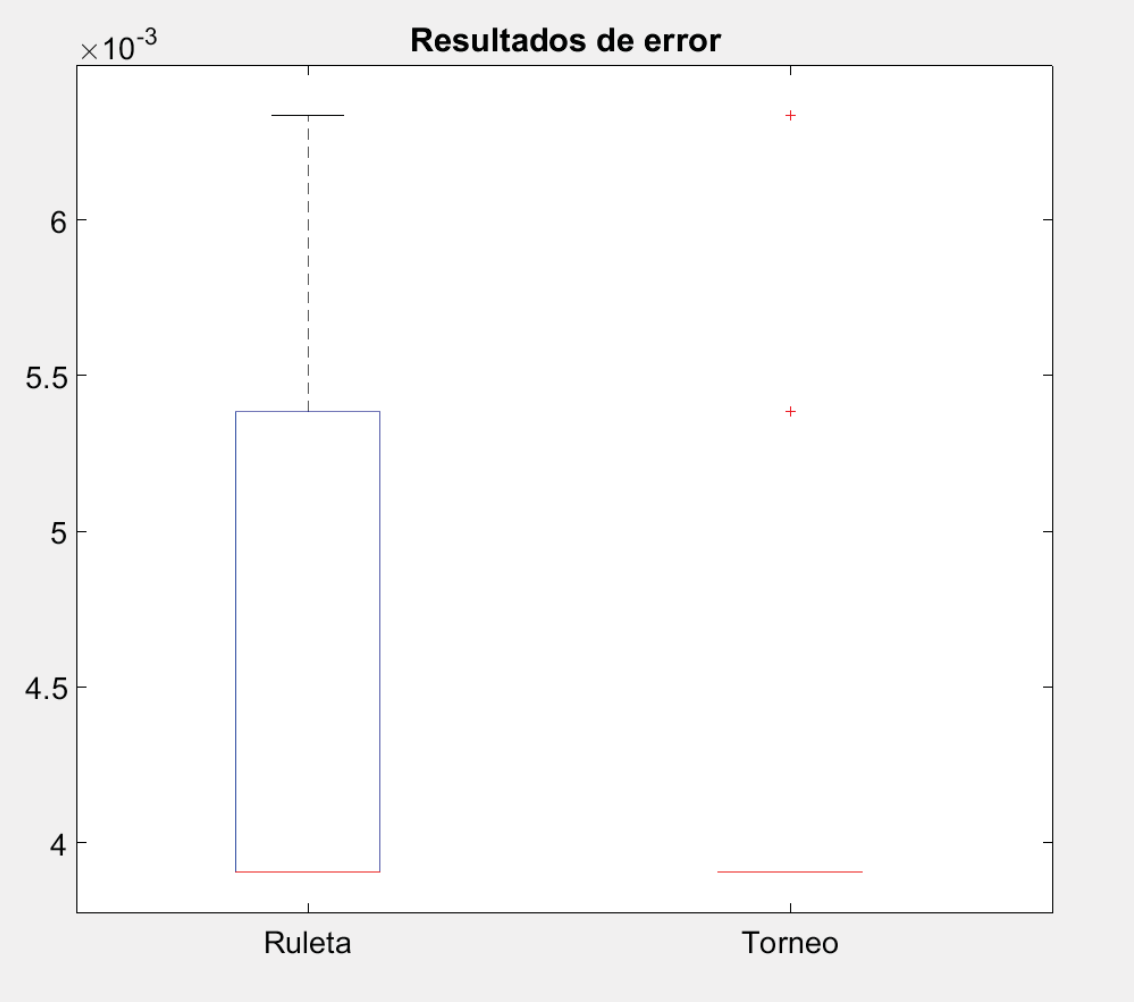
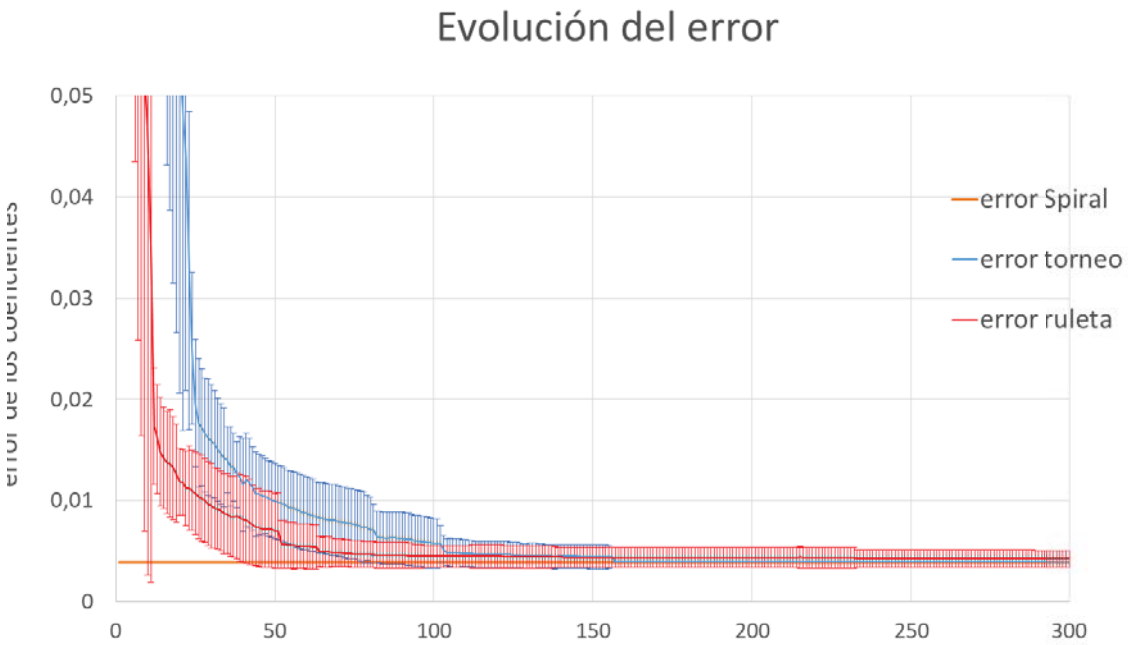


Fig. 43: Resultados de error representados por diagrama de caja



(a) Gráfica de evolución del error mínimo

(b) Gráfica de evolución del error mínimo con vista ampliada

Fig. 44: Gráfica de evolución del error mínimo

En la Fig. 44, se presenta el valor medio del error mínimo, incluyendo la subfigura 44a barras de error para la desviación típica. La configuración torneo es la única que alcanza el objetivo pasadas 150 generaciones.

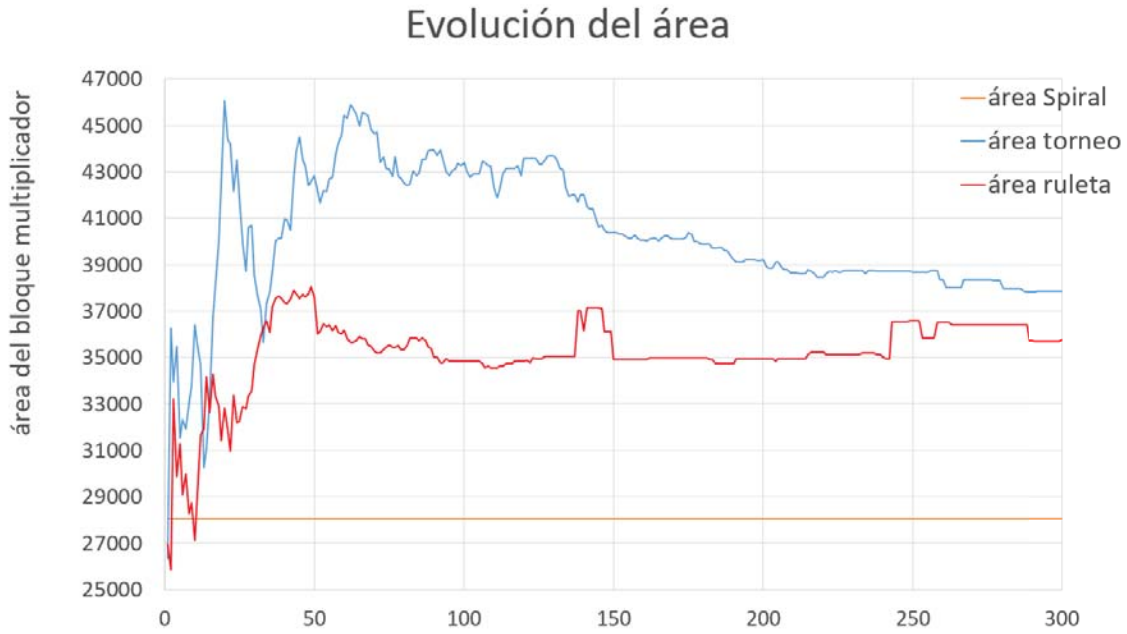


Fig. 45: Gráfica de evolución del área media correspondiente al error mínimo

La Fig. 45 representa el valor medio del área que tienen las soluciones con error mínimo. Para el área, ninguna configuración supera en su valor medio el objetivo. Este resultado va en consonancia con no haber obtenido soluciones que *dominen* a la *solución Spiral*, dado que el error mínimo está distribuido en torno al valor óptimo.

En la Fig. 46, se representa un diagrama de caja para los resultados de área obtenidos en cada configuración para soluciones que posean error óptimo. La configuración ruleta tiene una mediana de $37335,514$, un valor más bajo que la mediana de la configuración torneo, $37747,9869$. Además, la configuración ruleta tiene una dispersión mucho más concentrada, siendo una configuración más predecible en la obtención de área para este ejemplo.

Fig. 46: Diagrama de caja para los resultados de área correspondientes al error óptimo

En las Fig. 47 y 48, se presentan resultados de área para cada configuración en función del error obtenido, dada la dispersión de este último. En estos diagramas de caja, se aprecia una relación discontinua entre cada valor de error y sus correspondientes valores de área obtenidos. Esto se debe a la configuración de coeficientes que deben ser sintetizados y la dificultad que existe para encontrar expresiones comunes entre ellos. El error óptimo debe sintetizar $\{17, 221\}$, mientras que las dos siguientes configuraciones de error deben obtener $\{18, 221\}$ y $\{17, 222\}$.

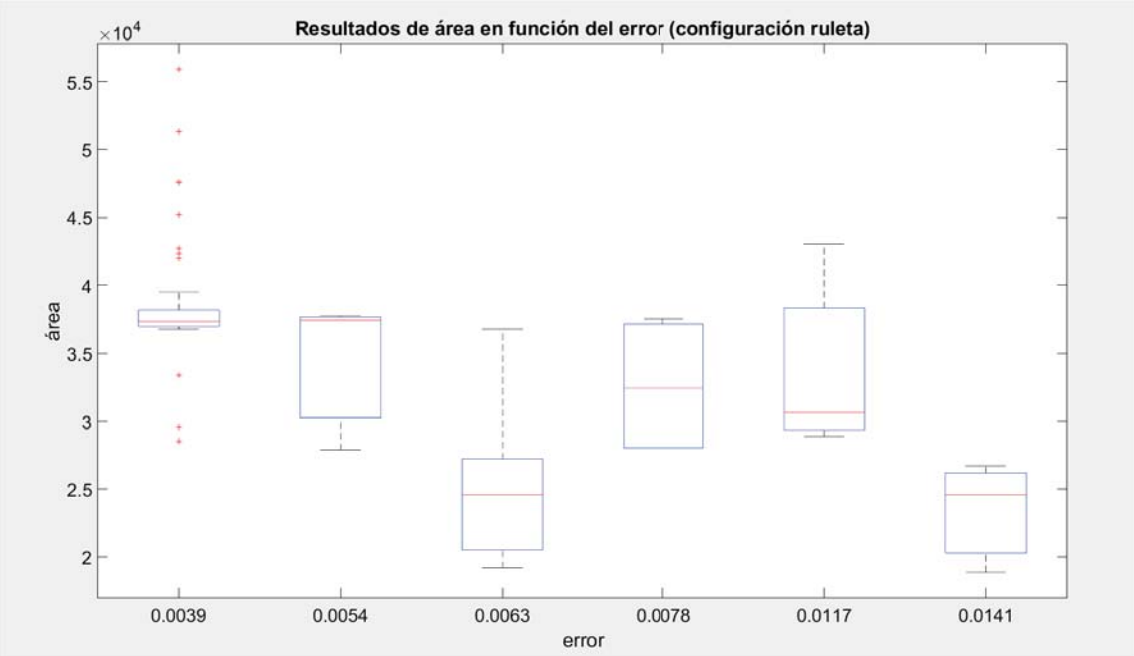


Fig. 47: Diagrama de caja para los resultados de ruleta de la configuración torneo en función del valor de error

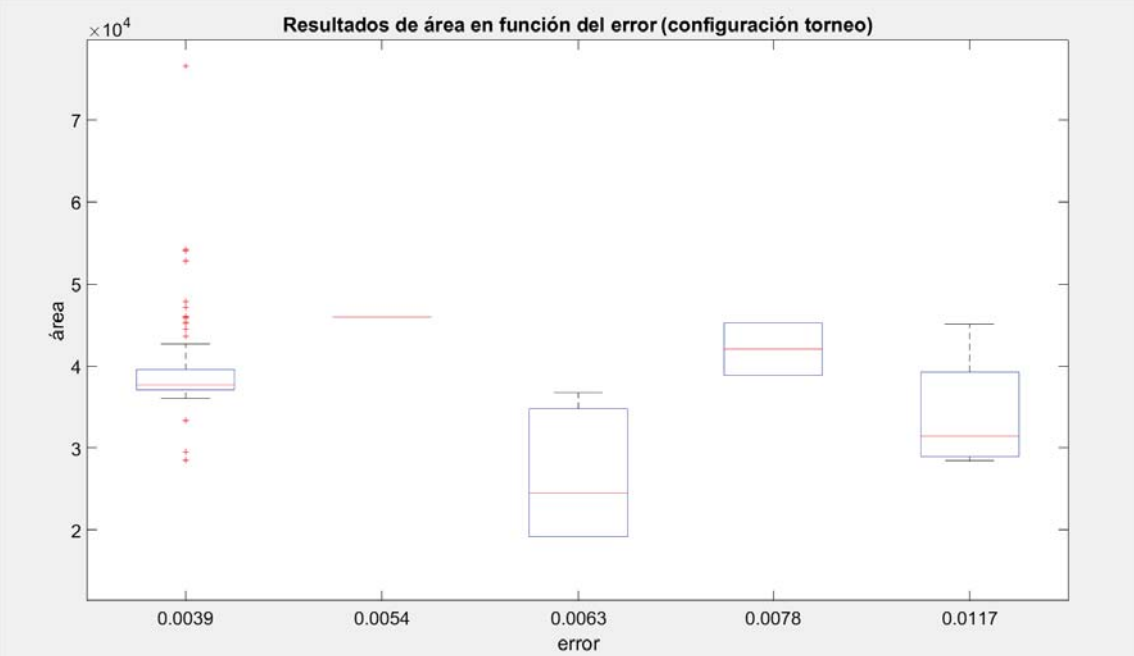
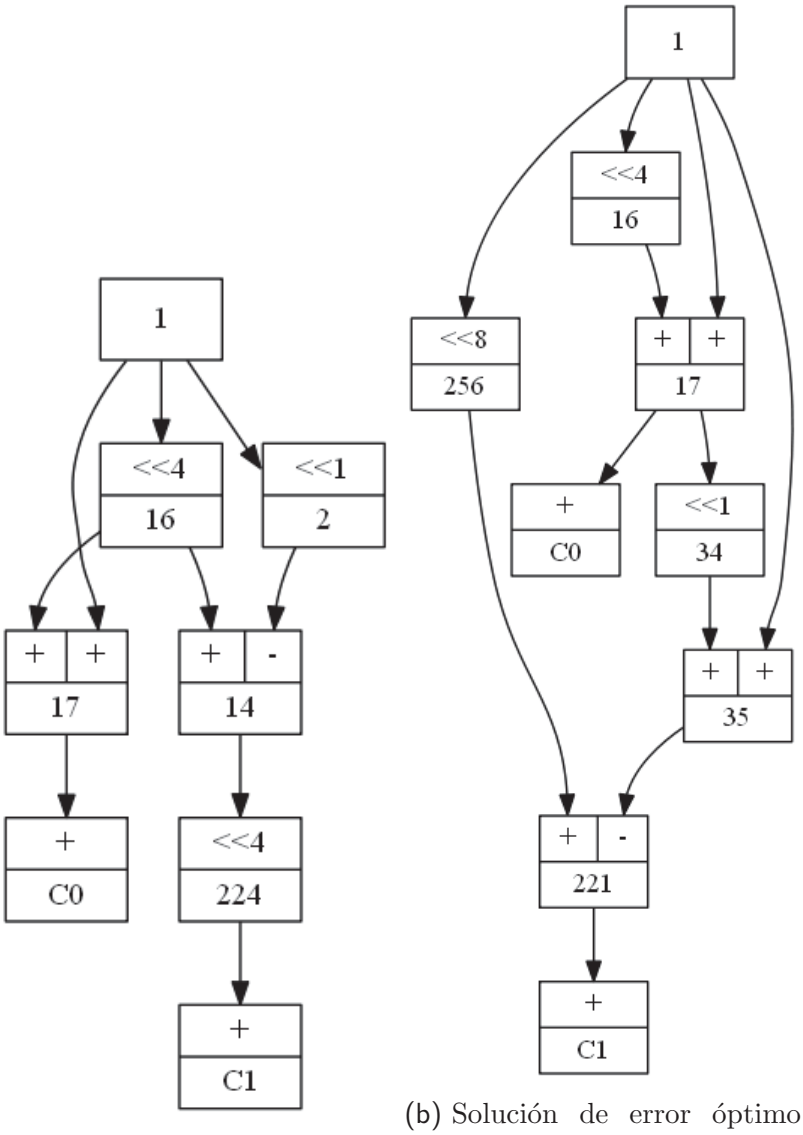


Fig. 48: Diagrama de caja para los resultados de área de la configuración torneo en función del valor de error



(a) Solución de área óptima

(b) Solución de error óptimo con área mínima

Fig. 49: Soluciones subóptimas encontradas

En la Fig. 49, se presentan las mejores soluciones obtenidas para este ejemplo con el sistema evolutivo propuesto. Utilizando de dos a tres nodos operadores, las soluciones realizan las siguientes operaciones:

$$a) \begin{cases} 17x = x \ll 4 + x \\ 224x = (x \ll 4 - x \ll 1) \ll 4 \end{cases}$$

$$b) \begin{cases} 17x = x \ll 4 + x \\ 221x = x \ll 8 - ((x \ll 4 + x) \ll 1 + x) \end{cases}$$

6.4. Ejemplo 4

Los parámetros utilizados para el filtro FIR y el bloque multiplicador en este ejemplo son los de $N = 2$, $f_c = 0{,}4$, tipo *paso bajo*, 8 bits fraccionales y precisión de señal de 128 bits. Para estos parámetros, se obtiene la siguiente tabla:

k	0, 2	1
b_k	$5,4006 \cdot 10^{-1}$	$0,8919$
$2^f b_k$	13,8256	228,3487
$c_k = \lfloor 2^f b_k \rfloor$	13	228
Δb_k	$-3,2251 \cdot 10^{-3}$	$-1,3622 \cdot 10^{-3}$

Tab. 5: Coeficientes correspondientes a la configuración elegida con el error asociado Δb_k para los coeficientes enteros de Spiral

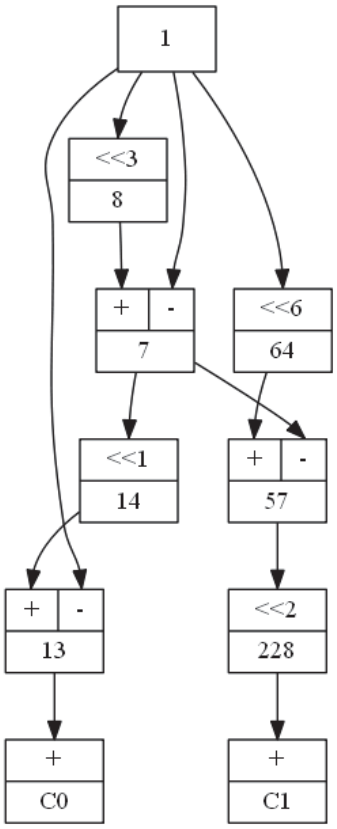


Fig. 50: Solución obtenida mediante el generador Spiral

La solución contenida en la Fig. 50 sintetiza los coeficientes $\{11, 232\}$. Por medio de 3 nodos operadores, realiza las siguientes operaciones:

$$\begin{cases} 13x = (x \ll 3 - x) \ll 1 - x \\ 228x = (x \ll 6 - (x \ll 3 - x)) \ll 2 \end{cases}$$

La solución Spiral tiene un error de $7,8125 \cdot 10^{-3}$ y un área de $30253,6067$. El error óptimo obtenido es de $2,7244 \cdot 10^{-3}$, presente en el 91 % de los casos para la configuración ruleta y en el 82 % para la configuración torneo. El área óptima obtenida es de $19093,5359$, representando un valor atípico en ambas configuraciones.

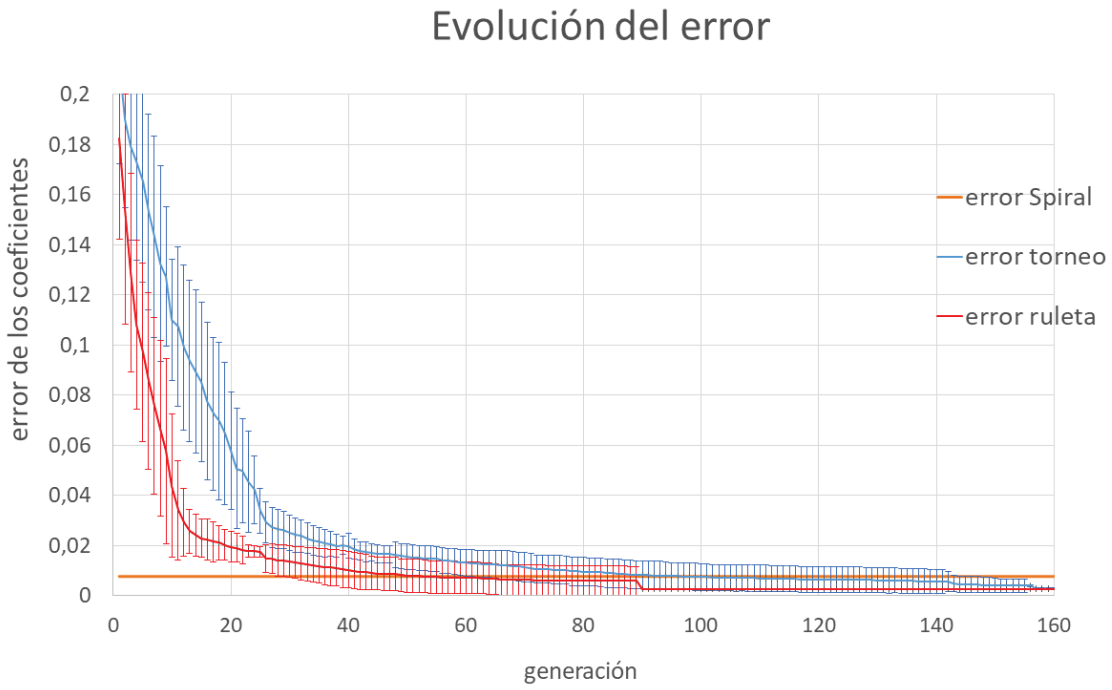


Fig. 51: Gráfica de evolución del error mínimo

En la Fig. 51, se presenta el valor medio del error mínimo con barras de error que representan la desviación típica. La configuración ruleta alcanza el objetivo en una duración de 90 generaciones, mientras que la configuración torneo tiene una duración de unas 100.

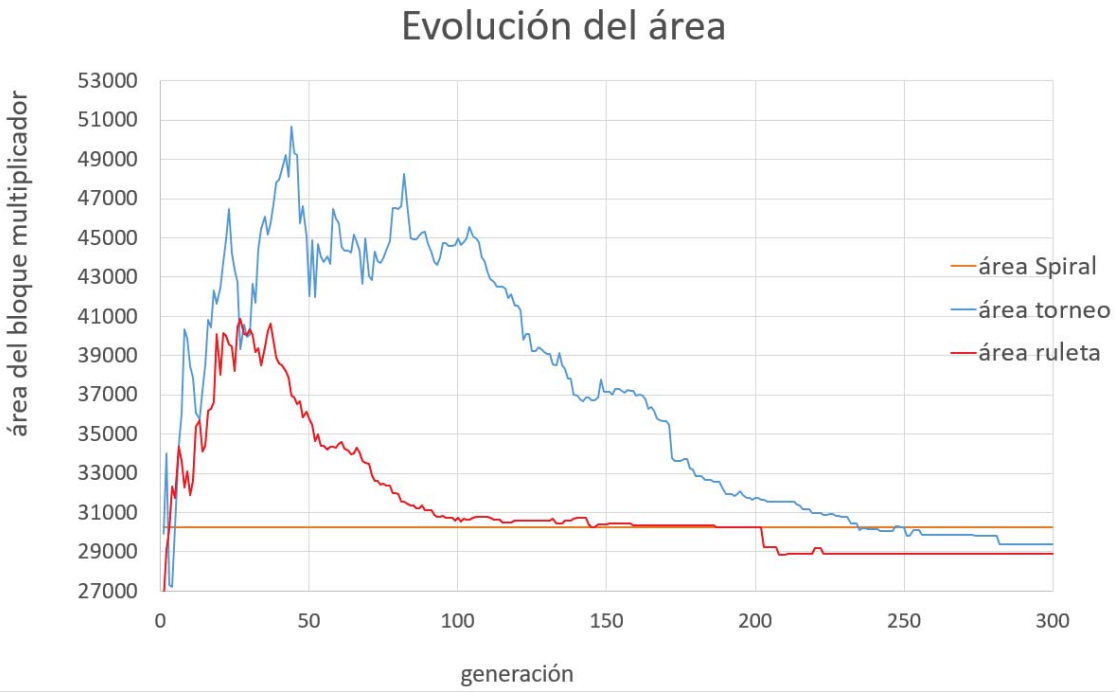


Fig. 52: Gráfica de evolución del área media correspondiente al error mínimo

La Fig. 52 representa el valor medio del área que tienen las soluciones con error mínimo. Para el área, la configuración ruleta alcanza el objetivo pasadas las 200 generaciones, mientras que la configuración torneo lo hace pasadas 225 generaciones.

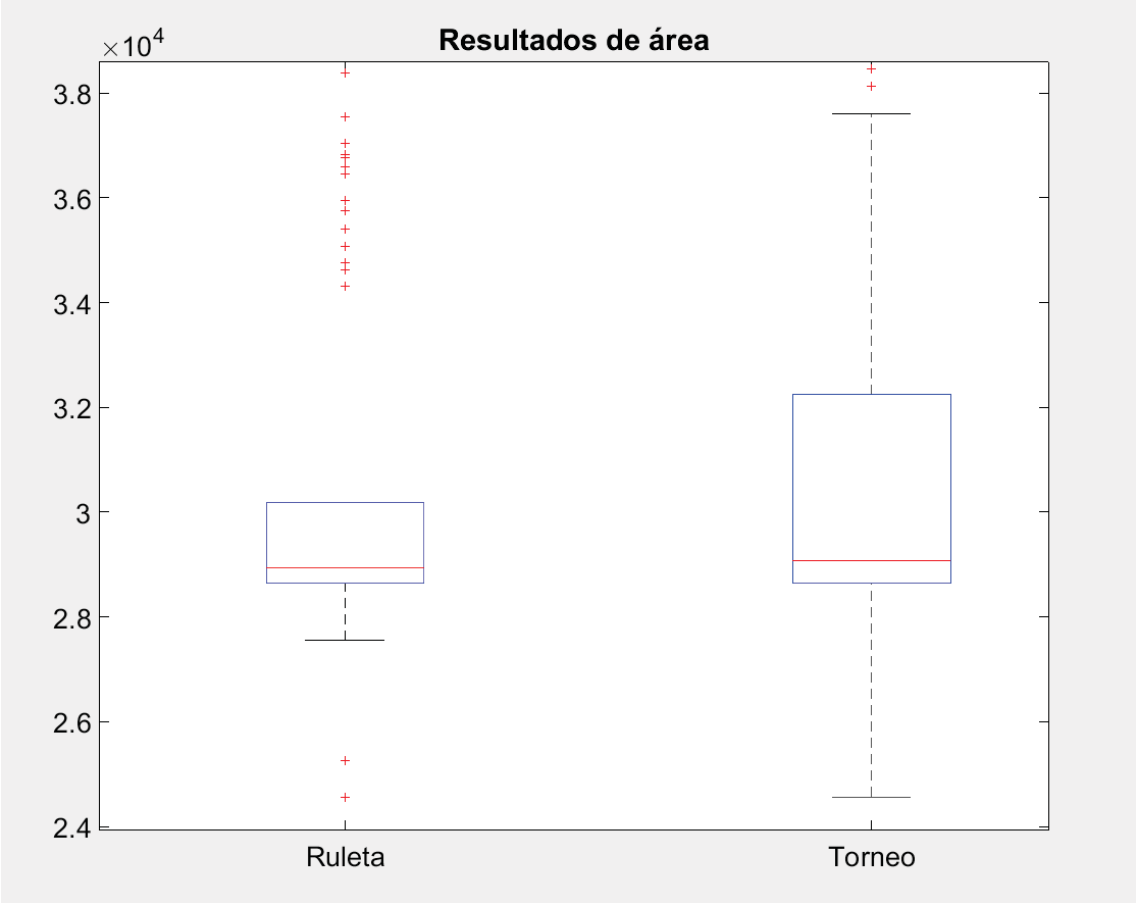


Fig. 53: Resultados de área representados por diagrama de caja

La Fig. 53 contiene los resultados finales de área. Para esta figura, ambas configuraciones tienen una mediana muy parecida, teniendo la configuración ruleta la mediana de 28943, 0058, mientras que la configuración torneo tiene una de 29076, 06. La configuración torneo tiene una distribución más amplia que la configuración ruleta por ambos extremos de la caja, siendo la primera una configuración más impredecible para la obtención de valores de área.

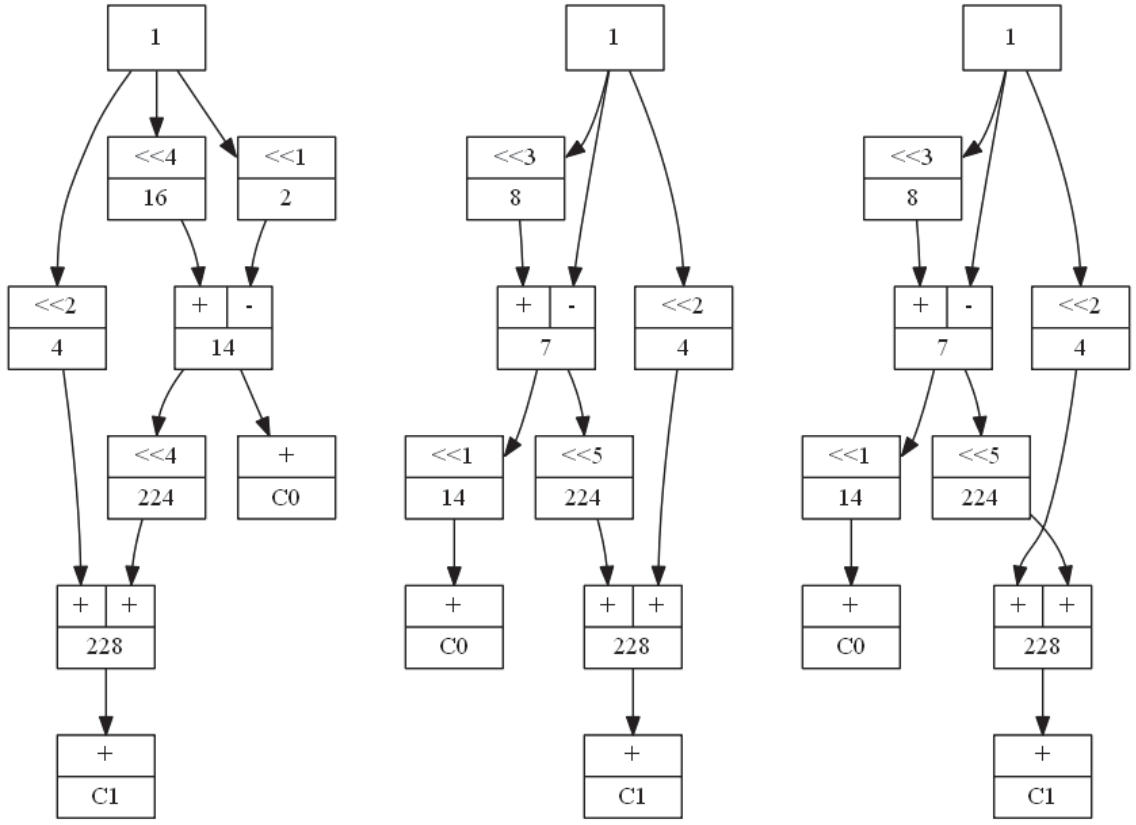


Fig. 54: Soluciones óptimas

En la Fig. 54, se presentan tres soluciones óptimas equivalentes obtenidas con el sistema evolutivo propuesto. La solución de la Fig. 54b solo difiere de la de la Fig. 54c en el orden de los operandos del sumador final que sintetiza el segundo coeficiente. Utilizando solo dos nodos operadores, las soluciones realizan las siguientes operaciones:

$$a) \begin{cases} 14x = x \ll 4 - x \ll 1 \\ 228x = x \ll 2 + (x \ll 4 - x \ll 1) \ll 4 \end{cases}$$

$$b),\ c) \begin{cases} 14x = (x \ll 3 - x) \ll 1 \\ 228x = (x \ll 3 - x) \ll 5 + x \ll 2 \end{cases}$$

7. CONCLUSIONES Y LÍNEAS FUTURAS

Se ha presentado el proceso realizado para el diseño y la implementación de un algoritmo de PGGG que obtiene de forma automática estructuras de bloque multiplicador para ser utilizadas dentro de un determinado filtro FIR.

Se puede afirmar que el sistema propuesto cumple los dos objetivos que se formularon en el planteamiento del problema: la minimización del error y la sintetización del circuito responsable de esta minimización. Si bien el sistema propuesto es capaz de obtener las estructuras responsables de dotar al filtro FIR de un error mínimo, la obtención de la estructura con área óptima no está garantizada si esta requiere que los coeficientes se formen mediante expresiones comunes de cierto grado de complejidad. Los resultados mostrados en el capítulo anterior reflejan esta limitación en la resolución del problema obteniéndose expresiones comunes entre coeficientes de dos elementos de longitud, no pudiendo igualar expresiones de mayor longitud o que se repiten dentro de la sintetización del mismo coeficiente como es el caso de $x \ll 4 + x - x \ll 2$, expresión que aparece en la *solución Spiral* del ejemplo 3 como parte de la sintetización de la salida de valor $221x$:

$$\begin{cases} exp = x \ll 4 + x - x \ll 2 \\ 221x = exp \ll 4 + exp \end{cases}$$

A partir de las gráficas de evolución mostradas en los ejemplos, se puede concluir que el proceso de optimización implementado tiene dos fases definidas.
La primera de ellas corresponde a un proceso de exploración de posibles estructuras para los bloques que termina cuando se encuentran una o más estructuras que minimizan el error, que tienden a tener un área excesiva. Una vez que este error ha sido optimizado, comienza la segunda etapa en la que la optimización del área pasa a ocupar un papel central, obteniéndose estructuras con cada vez menos área hasta llegar al objetivo.
Dado que la optimización de área tiende a ocurrir en las etapas finales, es importante que se seleccionen unos límites de ejecución acordes a cada problema a resolver de forma que se ajuste a su dificultad correspondiente y no lo haga terminar prematuramente.

Se proponen las siguientes líneas futuras:

1. *Inclusión de los desplazamientos hacia la derecha.*

La representación utilizada en el presente trabajo utiliza exclusivamente desplazamientos hacia la izquierda. Además de todos los beneficios que supone la exclusión de los desplazamientos hacia la derecha para el proceso de codificación y para el ahorro de área, también se introducen ciertas limitaciones al problema con esta restricción. Excluyendo la posibilidad de que se realicen desplazamientos hacia la derecha, se impone que el dominio de los coeficientes fraccionarios $\{c_k\}$ esté limitado al de los números enteros, dado que es imposible la obtención de un número fuera de este conjunto por medio de las operaciones disponibles. Sin embargo, los desplazamientos hacia la derecha introducen una nueva operación que permite aumentar este dominio al de todos aquellos números racionales que se pueden obtener a partir de la división de un número entero por una potencia de dos.

Este aumento del dominio de los coeficientes permite una mejor aproximación de los coeficientes fraccionarios $\{c_k\}$ a los coeficientes originales $\{b_k\}$, reduciéndose así de forma general el error mínimo que se pueda introducir para cada coeficiente del filtro construido.

2. *Mejora del método de evaluación para el error de filtrado.*

Para el cálculo del error de filtrado se ha elegido un método por aproximación. Para una mayor fiabilidad de esta medida, es necesario realizar su cálculo por medio de un método directo. Dado el problema de sobreajuste que se produce al evaluar las soluciones con una onda determinada, es preciso que se haga una investigación en torno a los efectos de este sobreajuste para dar lugar a un modelo que permita resolver este problema.

3. *Aumento de la complejidad de la solución objetivo.*

Tras resolver el problema de construcción de bloques multiplicadores para 8 bits fraccionales y filtros de orden $N = 2$, parece oportuno ampliar esta investigación hacia bloques más complejos. Esto puede ser interesante porque cuanto mayor sea el orden del filtro más se acerca el resultado de filtrado a resultados ideales para ese tipo de filtro. Además, en la búsqueda de mayor optimización del error, resulta inevitable explorar bits fraccionales mayores, ya que permiten que los errores asociados a cada coeficiente sean menores al desplazarse mayor porción decimal por encima de la unidad. Esta diferencia en la magnitud del error se puede comprobar comparando el error obtenido en el ejemplo del capítulo 4 con los obtenidos en el capítulo 6.

4. *Formación de expresiones comunes de mayor complejidad.*

Se han observado limitaciones a la hora de construir circuitos que contengan expresiones comunes de cierta complejidad entre los coeficientes a sintetizar. Se propone una línea de trabajo destinada al refinamiento del sistema evolutivo de forma que favorezca la creación de este tipo de estructuras tan necesarias para obtener resultados óptimos.

8. ANEXOS

Anexo 1: Cálculo de coeficientes y errores

En este anexo, se presenta el código Matlab utilizado para el cálculo de los coeficientes y errores presentados en la Tab. 1 del capítulo 4 y en todas las tablas del capítulo 6.

```
format longEng
coefsOriginales = fir1(orden,fc,'low')
coefsDesplazados = 2^f * coefsOriginales

format short
coefsSpiral = fix(coefsDesplazados)
coefsOptimos = round(coefsDesplazados)

format shortEng
eAbsSpiral = coefsSpiral/2^f - coefsOriginales
eAbsOptimos = coefsOptimos/2^f - coefsOriginales
```

Anexo 2: Código de los diagramas de caja

En este anexo, se presenta el código Matlab utilizado para componer todos los diagramas de caja del capítulo 6.

```
% datos de area fc=0.5
nombres = {'Ruleta', 'Torneo'};
datos = [28943.0058216176, 27555.8977337308,
    19036.9870424338, 27555.8977337308, 27938.4337162014,
    27938.4337162014, 27994.9825299588, 28943.0058216176,
    27938.4337162014, 27938.4337162014, 27938.4337162014,
    27555.8977337308, 27555.8977337308, 19036.9870424338,
    28866.4986251235, 27938.4337162014, 28943.0058216176,
    27938.4337162014, 27938.4337162014, 27994.9825299588,
    27938.4337162014, 27938.4337162014, 27994.9825299588,
    27994.9825299588, 27938.4337162014, 27861.9265197073,
    27938.4337162014, 27555.8977337308, 27994.9825299588,
    24562.1376412647, 27938.4337162014, 28943.0058216176,
    27938.4337162014, 24562.1376412647, 27555.8977337308,
    24562.1376412647, 28254.4410531706, 27938.4337162014,
    27938.4337162014, 27994.9825299588, 27938.4337162014,
    28943.0058216176, 27555.8977337308, 24562.1376412647,
    27938.4337162014, 27555.8977337308, 24562.1376412647,
    27938.4337162014, 27994.9825299588, 27938.4337162014,
    27938.4337162014, 27555.8977337308, 27994.9825299588,
    28943.0058216176, 27938.4337162014, 27938.4337162014,
    27861.9265197073, 24562.1376412647, 28866.4986251235,
    27938.4337162014, 28943.0058216176, 27938.4337162014,
    27938.4337162014, 28078.1425206985, 27938.4337162014,
    28560.469839147, 27861.9265197073, 27555.8977337308,
    33393.7299127838, 33393.7299127838, 27938.4337162014,
    27555.8977337308, 24562.1376412647, 27938.4337162014,
    27555.8977337308, 27994.9825299588, 24562.1376412647,
    27861.9265197073, 28943.0058216176, 28254.4410531706,
    28943.0058216176, 27994.9825299588, 27938.4337162014,
    27994.9825299588, 19036.9870424338, 27938.4337162014,
    38925.5317972956, 28254.4410531706, 28560.469839147,
    24562.1376412647, 27994.9825299588, 27938.4337162014,
    27861.9265197073, 19036.9870424338, 27938.4337162014,
    24562.1376412647, 27938.4337162014, 27938.4337162014,
    27555.8977337308, 27994.9825299588;
27938.4337162014, 28943.0058216176, 27555.8977337308,
    27861.9265197073, 27994.9825299588, 28943.0058216176,
    27994.9825299588, 28943.0058216176, 27994.9825299588,
    27555.8977337308, 19036.9870424338, 27555.8977337308,
```

```
    24562.1376412647, 19036.9870424338, 28254.4410531706,
    28943.0058216176, 27938.4337162014, 27555.8977337308,
    28943.0058216176, 27994.9825299588, 27938.4337162014,
    28943.0058216176, 19036.9870424338, 27994.9825299588,
    27938.4337162014, 27938.4337162014, 27938.4337162014,
    27555.8977337308, 27994.9825299588, 28254.4410531706,
    27938.4337162014, 27555.8977337308, 27994.9825299588,
    24562.1376412647, 27938.4337162014, 34697.6780099308,
    27994.9825299588, 56276.0354631573, 28943.0058216176,
    27994.9825299588, 28560.469839147, 28943.0058216176,
    27938.4337162014, 27938.4337162014, 27938.4337162014,
    27861.9265197073, 27994.9825299588, 27938.4337162014,
    27938.4337162014, 27053.6120325455, 24562.1376412647,
    27994.9825299588, 27555.8977337308, 24562.1376412647,
    28560.469839147, 28943.0058216176, 20041.55914785,
    27938.4337162014, 27861.9265197073, 27861.9265197073,
    27938.4337162014, 27938.4337162014, 27555.8977337308,
    28866.4986251235, 27994.9825299588, 28943.0058216176,
    28943.0058216176, 27938.4337162014, 27938.4337162014,
    27994.9825299588, 28866.4986251235, 27938.4337162014,
    28943.0058216176, 27708.9121267191, 27555.8977337308,
    27555.8977337308, 27861.9265197073, 27555.8977337308,
    28943.0058216176, 28560.469839147, 27555.8977337308,
    27938.4337162014, 27555.8977337308, 33463.5843150323,
    27861.9265197073, 27938.4337162014, 28254.4410531706,
    27994.9825299588, 28943.0058216176, 28943.0058216176,
    28254.4410531706, 27555.8977337308, 27938.4337162014,
    27994.9825299588, 20041.55914785, 27994.9825299588,
    28943.0058216176, 28943.0058216176, 27938.4337162014,
    27994.9825299588];
boxplot(transpose(datos),nombres)
ax = gca
ax.FontSize = 20
title('Resultados de area')
```

```
1 % datos de area fc=0.9
2 nombres = {'Ruleta', 'Torneo'};
3 datos = [18654.4510599632, 10212.0475651603,
    18654.4510599632, 15660.690967497, 15660.690967497,
    15660.690967497, 19113.4942389279, 10212.0475651603,
    19113.4942389279, 10212.0475651603, 19113.4942389279,
    15660.690967497, 10212.0475651603, 10212.0475651603,
    15660.690967497, 15660.690967497, 10212.0475651603,
    10212.0475651603, 10212.0475651603, 10212.0475651603,
    10212.0475651603, 19036.9870424338, 10212.0475651603,
    19113.4942389279, 10212.0475651603, 10212.0475651603,
    15660.690967497, 15660.690967497, 10212.0475651603,
    15660.690967497, 10212.0475651603, 19113.4942389279,
    15660.690967497, 10212.0475651603, 10212.0475651603,
    10212.0475651603, 15660.690967497, 10212.0475651603,
    10212.0475651603, 18654.4510599632, 19113.4942389279,
    10212.0475651603, 19113.4942389279, 18654.4510599632,
    19113.4942389279, 19113.4942389279, 15660.690967497,
    10212.0475651603, 19113.4942389279, 18654.4510599632,
    10212.0475651603, 10212.0475651603, 18654.4510599632,
    10212.0475651603, 10212.0475651603, 10212.0475651603,
    15660.690967497, 19113.4942389279, 15660.690967497,
    10212.0475651603, 10212.0475651603, 19113.4942389279,
    19965.0519513558, 15660.690967497, 15660.690967497,
    19113.4942389279, 19113.4942389279, 15660.690967497,
    19113.4942389279, 10212.0475651603, 10212.0475651603,
    19113.4942389279, 10212.0475651603, 19113.4942389279,
    10212.0475651603, 19113.4942389279, 10212.0475651603,
    15660.690967497, 15660.690967497, 15660.690967497,
    10212.0475651603, 10212.0475651603, 15660.690967497,
    10212.0475651603, 10212.0475651603, 18654.4510599632,
    19113.4942389279, 10212.0475651603, 10212.0475651603,
    15660.690967497, 19113.4942389279, 10212.0475651603,
    15660.690967497, 24562.1376412647, 10212.0475651603,
    19113.4942389279, 10212.0475651603, 15660.690967497,
    15660.690967497, 18654.4510599632;
4 10212.0475651603, 10212.0475651603, 10212.0475651603,
    10212.0475651603, 10212.0475651603, 10212.0475651603,
    10212.0475651603, 19036.9870424338, 10212.0475651603,
    19113.4942389279, 19113.4942389279, 10212.0475651603,
    19113.4942389279, 10212.0475651603, 15660.690967497,
    10212.0475651603, 19113.4942389279, 10212.0475651603,
    18654.4510599632, 15660.690967497, 15660.690967497,
    15660.690967497, 15660.690967497, 15660.690967497,
```

```
    19036.9870424338, 10212.0475651603, 19113.4942389279,
    10212.0475651603, 15660.690967497, 15660.690967497,
    10212.0475651603, 15660.690967497, 10212.0475651603,
    10212.0475651603, 10212.0475651603, 10212.0475651603,
    15660.690967497, 19113.4942389279, 10212.0475651603,
    15660.690967497, 10212.0475651603, 10212.0475651603,
    15660.690967497, 10212.0475651603, 10212.0475651603,
    10212.0475651603, 10212.0475651603, 10212.0475651603,
    19113.4942389279, 10212.0475651603, 19036.9870424338,
    19113.4942389279, 19113.4942389279, 10212.0475651603,
    10212.0475651603, 15660.690967497, 18654.4510599632,
    10212.0475651603, 15660.690967497, 15660.690967497,
    15660.690967497, 10212.0475651603, 19036.9870424338,
    10212.0475651603, 10212.0475651603, 15660.690967497,
    19113.4942389279, 15660.690967497, 10212.0475651603,
    19113.4942389279, 10212.0475651603, 10212.0475651603,
    10212.0475651603, 15660.690967497, 10212.0475651603,
    10212.0475651603, 19113.4942389279, 19113.4942389279,
    10212.0475651603, 10212.0475651603, 10212.0475651603,
    19113.4942389279, 19113.4942389279, 10212.0475651603,
    15660.690967497, 10212.0475651603, 10212.0475651603,
    10212.0475651603, 19113.4942389279, 10212.0475651603,
    15660.690967497, 15660.690967497, 19113.4942389279,
    19113.4942389279, 19113.4942389279, 10212.0475651603,
    10212.0475651603, 19113.4942389279, 10212.0475651603,
    10212.0475651603];
boxplot(transpose(datos),nombres)
ax = gca
ax.FontSize = 20
title('Resultados de area')
```

```
% datos de error fc=0.1
nombres = {'Ruleta', 'Torneo'};
datos = [0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00633566695599004,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00633566695599004,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.0117187499999998,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00633566695599004, 0.00633566695599004,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00633566695599004,
    0.00390624999999984, 0.00390624999999984,
    0.00633566695599004, 0.00781249999999984,
    0.00390624999999984, 0.0117187499999998,
    0.00390624999999984, 0.00781249999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00538308304400996, 0.01414816695599,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.0117187499999998,
    0.00633566695599004, 0.00390624999999984,
    0.00390624999999984, 0.01414816695599,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.0117187499999998, 0.00781249999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.01414816695599,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00633566695599004, 0.00633566695599004,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
```

```
    0.00390624999999984, 0.00781249999999984,
    0.00390624999999984, 0.0117187499999998,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00538308304400996,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00633566695599004,
    0.00538308304400996, 0.00633566695599004;
0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.0117187499999998, 0.00633566695599004,
    0.00633566695599004, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00633566695599004, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.0117187499999998, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.0117187499999998, 0.00390624999999984,
    0.0117187499999998, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00633566695599004,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00633566695599004, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00538308304400996, 0.00390624999999984,
    0.00633566695599004, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
```

```
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00781249999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00781249999999984,
    0.00390624999999984, 0.00390624999999984];
boxplot(transpose(datos),nombres)
ax = gca
ax.FontSize = 20
title('Resultados de error')
```

```
1 %datos de area para error optimo fc=0.1
2 nombres = {'Ruleta', 'Torneo'};
3 datos = [29555.0633935706, 37069.4019794514,
    42714.3030000426, NaN, 36763.3731934749,
    36763.3731934749, 37418.6739906941, NaN,
    45205.7773913235, 37069.4019794514, 37308.9021199264,
    33393.7299127838, 47567.5199478514, NaN,
    37308.9021199264, 36763.3731934749, 37418.6739906941,
    55913.4578634235, NaN, NaN, 51313.0472640117,
    37418.6739906941, 37767.9452988911, 29555.0633935706,
    42015.7589775572, 39537.5893692485, 42365.0309888, NaN,
    36763.3731934749, 42365.0309888, NaN, NaN,
    29555.0633935706, NaN, 37335.5139999544, NaN,
    39537.5893692485, 37418.6739906941, 42714.3030000426,
    29555.0633935706, 36763.3731934749, 38150.4812813617,
    37348.8195884455, 38150.4812813617, NaN, NaN,
    33393.7299127838, 37308.9021199264, 37418.6739906941, NaN
    , NaN, 38187.0717123823, 38150.4812813617, NaN,
    37348.8195884455, 36763.3731934749, 37335.5139999544,
    38150.4812813617, NaN, NaN, 36992.8947829573,
    37418.6739906941, 37069.4019794514, 38073.9740848676,
    42015.7589775572, NaN, 36763.3731934749,
    29555.0633935706, 39537.5893692485, 37069.4019794514,
    38150.4812813617, 37069.4019794514, 37069.4019794514,
    37328.8612057088, NaN, NaN, 36763.3731934749,
    42015.7589775572, 37767.9452988911, 36992.8947829573,
    39155.0533867779, 36992.8947829573, 37308.9021199264,
    38150.4812813617, 36763.3731934749, NaN,
    39537.5893692485, NaN, 37259.0068034602,
    29555.0633935706, 28517.2273169264, NaN,
    36992.8947829573, 38150.4812813617, 37308.9021199264,
    36992.8947829573, 42365.0309888, NaN, NaN, NaN;
4 37069.4019794514, 36763.3731934749, 29555.0633935706,
    38456.5100673382, 37155.8877269382, 76616.9698996926,
    39155.0533867779, 37155.8877269382, 36992.8947829573,
    45415.3405980691, 38073.9740848676, 29555.0633935706, NaN
    , NaN, NaN, 36992.8947829573, 38207.0300951191,
    37418.6739906941, 44440.7038451338, 37538.4237094088,
    37538.4237094088, 37418.6739906941, NaN,
    29555.0633935706, 37418.6739906941, 39537.5893692485,
    37767.9452988911, 39537.5893692485, 36992.8947829573,
    37767.9452988911, 38150.4812813617, 29555.0633935706,
    38073.9740848676, 37069.4019794514, 37747.9869161544,
    36763.3731934749, 38073.9740848676, 42714.3030000426, NaN
```

```
    , 37747.9869161544, 39537.5893692485, 39537.5893692485,
    37335.5139999544, 38150.4812813617, 37418.6739906941,
    37308.9021199264, 38150.4812813617, 54040.6938383411, NaN
    , 37335.5139999544, NaN, 54246.9328695882,
    42015.7589775572, 36024.9131085617, 43605.7774779441, NaN
    , 29555.0633935706, 36763.3731934749, 37418.6739906941,
    37538.4237094088, 38150.4812813617, 37069.4019794514, NaN
    , 37348.8195884455, 47171.6783768897, 37418.6739906941,
    NaN, 39537.5893692485, NaN, 28517.2273169264,
    33393.7299127838, 29555.0633935706, 42365.0309888,
    37335.5139999544, 47903.4856675573, 37767.9452988911,
    38073.9740848676, 37767.9452988911, 45804.5286717396,
    38073.9740848676, 37348.8195884455, 37418.6739906941,
    46070.6406922426, 36763.3731934749, NaN,
    38150.4812813617, 37418.6739906941, 45205.7773913235,
    42015.7589775572, 52806.6001434426, 37418.6739906941,
    45970.8486532191, 42295.1765865514, 37747.9869161544,
    43675.6318801926, 37335.5139999544, 29555.0633935706, NaN
    , 36992.8947829573, 29555.0633935706];
boxplot(transpose(datos),nombres)
ax = gca
ax.FontSize = 20
title('Resultados de area para error optimo')
```

```
1 %datos para boxplot de area para error ruleta fc=0.1
2 area = [29555.0633935706, 37069.4019794514,
    42714.3030000426, 19190.001435422, 36763.3731934749,
    36763.3731934749, 37418.6739906941, 24562.1376412647,
    45205.7773913235, 37069.4019794514, 37308.9021199264,
    33393.7299127838, 47567.5199478514, 36736.7620164926,
    37308.9021199264, 36763.3731934749, 37418.6739906941,
    55913.4578634235, 36763.3731934749, 24562.1376412647,
    51313.0472640117, 37418.6739906941, 37767.9452988911,
    29555.0633935706, 42015.7589775572, 39537.5893692485,
    42365.0309888, 24562.1376412647, 36763.3731934749,
    42365.0309888, 19190.001435422, 28014.9409126955,
    29555.0633935706, 30636.1426954809, 37335.5139999544,
    37538.4237094088, 39537.5893692485, 37418.6739906941,
    42714.3030000426, 29555.0633935706, 36763.3731934749,
    38150.4812813617, 37348.8195884455, 38150.4812813617,
    37418.6739906941, 26704.3400213029, 33393.7299127838,
    37308.9021199264, 37418.6739906941, 28846.5402423867,
    28091.4481091897, 38187.0717123823, 38150.4812813617,
    24562.1376412647, 37348.8195884455, 36763.3731934749,
    37335.5139999544, 38150.4812813617, 29555.0633935706,
    28014.9409126955, 36992.8947829573, 37418.6739906941,
    37069.4019794514, 38073.9740848676, 42015.7589775572,
    18883.9726494455, 36763.3731934749, 29555.0633935706,
    39537.5893692485, 37069.4019794514, 38150.4812813617,
    37069.4019794514, 37069.4019794514, 37328.8612057088,
    19190.001435422, 24562.1376412647, 36763.3731934749,
    42015.7589775572, 37767.9452988911, 36992.8947829573,
    39155.0533867779, 36992.8947829573, 37308.9021199264,
    38150.4812813617, 36763.3731934749, 36826.5748014779,
    39537.5893692485, 43053.5950959264, 37259.0068034602,
    29555.0633935706, 28517.2273169264, 37747.9869161544,
    36992.8947829573, 38150.4812813617, 37308.9021199264,
    36992.8947829573, 42365.0309888, 33696.4323224588,
    27861.9265197073, 24562.1376412647];
3 error = [0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00633566695599004,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00633566695599004,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.0117187499999998,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
```

```
    0.00633566695599004, 0.00633566695599004,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00633566695599004,
    0.00390624999999984, 0.00390624999999984,
    0.00633566695599004, 0.00781249999999984,
    0.00390624999999984, 0.0117187499999998,
    0.00390624999999984, 0.00781249999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00538308304400996, 0.01414816695599,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.0117187499999998,
    0.00633566695599004, 0.00390624999999984,
    0.00390624999999984, 0.01414816695599,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.0117187499999998, 0.00781249999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.01414816695599,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00633566695599004, 0.00633566695599004,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00781249999999984,
    0.00390624999999984, 0.0117187499999998,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00538308304400996,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00390624999999984,
    0.00390624999999984, 0.00633566695599004,
    0.00538308304400996, 0.00633566695599004];
error = round(error, 4);
boxplot(transpose(area),error)
ax = gca
```

```
7  ax.FontSize = 20
8  title('Resultados de area en funcion del error (
       configuracion ruleta)')
9  xlabel('error');
10 ylabel('area');

1  %datos para boxplot de area para error torneo fc=0.1
2  area = [37069.4019794514, 36763.3731934749,
       29555.0633935706, 38456.5100673382, 37155.8877269382,
       76616.9698996926, 39155.0533867779, 37155.8877269382,
       36992.8947829573, 45415.3405980691, 38073.9740848676,
       29555.0633935706, 29515.146628097, 36763.3731934749,
       24562.1376412647, 36992.8947829573, 38207.0300951191,
       37418.6739906941, 44440.7038451338, 37538.4237094088,
       37538.4237094088, 37418.6739906941, 24562.1376412647,
       29555.0633935706, 37418.6739906941, 39537.5893692485,
       37767.9452988911, 39537.5893692485, 36992.8947829573,
       37767.9452988911, 38150.4812813617, 29555.0633935706,
       38073.9740848676, 37069.4019794514, 37747.9869161544,
       36763.3731934749, 38073.9740848676, 42714.3030000426,
       28434.0673261867, 37747.9869161544, 39537.5893692485,
       39537.5893692485, 37335.5139999544, 38150.4812813617,
       37418.6739906941, 37308.9021199264, 38150.4812813617,
       54040.6938383411, 33463.5843150323, 37335.5139999544,
       45139.2471645735, 54246.9328695882, 42015.7589775572,
       36024.9131085617, 43605.7774779441, 19190.001435422,
       29555.0633935706, 36763.3731934749, 37418.6739906941,
       37538.4237094088, 38150.4812813617, 37069.4019794514,
       19190.001435422, 37348.8195884455, 47171.6783768897,
       37418.6739906941, 45950.8902704823, 39537.5893692485,
       34774.185206425, 28517.2273169264, 33393.7299127838,
       29555.0633935706, 42365.0309888, 37335.5139999544,
       47903.4856675573, 37767.9452988911, 38073.9740848676,
       37767.9452988911, 45804.5286717396, 38073.9740848676,
       37348.8195884455, 37418.6739906941, 46070.6406922426,
       36763.3731934749, 45275.631793572, 38150.4812813617,
       37418.6739906941, 45205.7773913235, 42015.7589775572,
       52806.6001434426, 37418.6739906941, 45970.8486532191,
       42295.1765865514, 37747.9869161544, 43675.6318801926,
       37335.5139999544, 29555.0633935706, 38849.0246008014,
       36992.8947829573, 29555.0633935706];
3  error = [0.00390624999999984, 0.00390624999999984,
       0.00390624999999984, 0.00390624999999984,
       0.00390624999999984, 0.00390624999999984,
       0.00390624999999984, 0.00390624999999984,
```

```
0.00390624999999984, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.0117187499999998, 0.00633566695599004,
0.00633566695599004, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.00633566695599004, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.0117187499999998, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.0117187499999998, 0.00390624999999984,
0.0117187499999998, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.00390624999999984, 0.00633566695599004,
0.00390624999999984, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.00633566695599004, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.00538308304400996, 0.00390624999999984,
0.00633566695599004, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.00781249999999984, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
0.00390624999999984, 0.00390624999999984,
```

```
    0.00390624999999984, 0.00781249999999984,
    0.00390624999999984, 0.00390624999999984];
error = round(error, 4);
boxplot(transpose(area),error)
ax = gca
ax.FontSize = 20
title('Resultados de area en funcion del error (
    configuracion torneo)')
xlabel('error');
ylabel('area');

%datos de area fc=0.4
nombres = {'Ruleta', 'Torneo'};
datos = [28943.0058216176, 30177.0995165161,
    28636.9770356411, 28636.9770356411, 18654.4510599632,
    29076.0618318691, 28943.0058216176, 28636.9770356411,
    19093.5358561911, 35955.0587063132, 29076.0618318691,
    28636.9770356411, 28636.9770356411, 24562.1376412647,
    29076.0618318691, 21568.3775487985, 28636.9770356411,
    28636.9770356411, 35396.2213293706, 34754.2268236882,
    27861.9265197073, 28636.9770356411, 24562.1376412647,
    29076.0618318691, 29305.5834213514, 28636.9770356411,
    36826.5748014779, 28636.9770356411, 24562.1376412647,
    34754.2268236882, 36597.0532119955, 20194.5735408382,
    37538.4237094088, 28943.0058216176, 29076.0618318691,
    29259.0131585867, 24562.1376412647, 38389.9814218367,
    24562.1376412647, 34754.2268236882, 36763.3731934749,
    28636.9770356411, 28636.9770356411, 28636.9770356411,
    44247.772686672, 39624.0751167353, 29947.5779270338,
    36763.3731934749, 24562.1376412647, 37036.1380082235,
    29076.0618318691, 28636.9770356411, 39155.0533867779,
    28636.9770356411, 24562.1376412647, 35396.2213293706,
    28636.9770356411, 29259.0131585867, 28943.0058216176,
    30177.0995165161, 18654.4510599632, 28943.0058216176,
    20194.5735408382, 28943.0058216176, 34621.1708134367,
    28636.9770356411, 28636.9770356411, 28943.0058216176,
    36457.3444074985, 29076.0618318691, 34315.1420274603,
    9982.52597567794, 29076.0618318691, 28636.9770356411,
    28636.9770356411, 35745.4954995676, 64073.1158335308,
    20194.5735408382, 28636.9770356411, 30177.0995165161,
    25260.6809607044, 28636.9770356411, 29076.0618318691,
    28636.9770356411, 28943.0058216176, 19093.5358561911,
    29076.0618318691, 27994.9825299588, 28636.9770356411,
    18654.4510599632, 27555.8977337308, 35080.2139924014,
    28636.9770356411, 30177.0995165161, 29076.0618318691,
```

```
    29076.0618318691, 42504.739793297, 30177.0995165161,
    19093.5358561911, 28943.0058216176;
4 20194.5735408382, 28636.9770356411, 29259.0131585867,
    19093.5358561911, 29076.0618318691, 34621.1708134367,
    28636.9770356411, 18654.4510599632, 28154.6497171926,
    29076.0618318691, 54037.3696628426, 30177.0995165161,
    29076.0618318691, 28636.9770356411, 28636.9770356411,
    29076.0618318691, 38137.1756928705, 29076.0618318691,
    28636.9770356411, 28636.9770356411, 35396.2213293706,
    20194.5735408382, 29076.0618318691, 29947.5779270338,
    28636.9770356411, 20194.5735408382, 30177.0995165161,
    52969.5930874235, 37538.4237094088, 18960.4798459397,
    30177.0995165161, 27994.9825299588, 28636.9770356411,
    30177.0995165161, 30177.0995165161, 30177.0995165161,
    30177.0995165161, 24562.1376412647, 29076.0618318691,
    20194.5735408382, 40758.3789842205, 30177.0995165161,
    28636.9770356411, 30177.0995165161, 35885.2043040647,
    19093.5358561911, 43216.5887012279, 29076.0618318691,
    28636.9770356411, 28943.0058216176, 29076.0618318691,
    28636.9770356411, 27555.8977337308, 38466.4893213764,
    30177.0995165161, 29076.0618318691, 29076.0618318691,
    28636.9770356411, 35855.2645083353, 50295.1658608603,
    28636.9770356411, 48276.0418182838, 30177.0995165161,
    37601.6253174117, 28636.9770356411, 30177.0995165161,
    29076.0618318691, 30177.0995165161, 28943.0058216176,
    28483.9633456985, 28636.9770356411, 29076.0618318691,
    65583.302961925, 35156.7211888955, 37601.6253174117,
    27994.9825299588, 30177.0995165161, 29076.0618318691,
    29076.0618318691, 63464.3852990764, 28636.9770356411,
    29076.0618318691, 64209.5004625293, 28636.9770356411,
    45378.7473050485, 44068.1464136559, 28943.0058216176,
    20194.5735408382, 54968.7587472632, 18654.4510599632,
    30177.0995165161, 29076.0618318691, 30177.0995165161,
    43263.1583027132, 30177.0995165161, 36686.8659969808,
    29259.0131585867, 29076.0618318691, 30177.0995165161,
    34315.1420274603];
5 boxplot(transpose(datos),nombres)
6 ax = gca
7 ax.FontSize = 20
8 title('Resultados de area')
```

Referencias

[1] Bäck T., Hammel U. & Schwefel H-P. (1997) *Evolutionary Computation: Comments on the History and Current State*. In *IEEE Transactions on Evolutionary Computation* 1(1), 3-17.

[2] Beyer H.G., & Schwefel H-P. (2002) *Evolution strategies - A comprehensive introduction*. Natural Computing 1(1), 3-52. March 2002. Department of Computer Science XI, Technische Universität Dortmund, Joseph-von-Fraunhoferstr. 20, D-44221. Dortmund, Germany.

[3] Brabazon A, O'Neill M, McGarraghy S (2015) *Natural Computing Algorithms*. Natural Computing Series. Springer, Berlin, Heidelberg

[4] Bull D. & Horrocks D. (1991) *Primitive operator digital filters*. In *IEE Proceedings G - Circuits, Devices and Systems* 138(3), 401–412.

[5] Cappello P. R. & Steiglitz K. (1984) *Some complexity issues in digital signal processing*. In *IEEE Transactions on Acoustics Speech and Signal Processing* 32(5), 1037-1041.

[6] Chellapilla K. (1997) *Evolving computer programs without subtree crossover*. In *IEEE Transactions on Evolutionary Computation* 1(3), 209–216.

[7] Cramer N.L. (1985) *A Representation for the Adaptive Generation of Simple Sequential Programs*. In *Proceedings of an International Conference on Genetic Algorithms and their applications* 183-187. Carnegie-Mellon University, Pittsburgh, Pennsylvania.

[8] Couchet J., Manrique D., Ríos J., Rodríguez-Patón A. (2007) *Crossover and mutation operators for grammar-guided genetic programming. Soft Computing 10* (2007): 943-955.

[9] Crawford-Marks R, Spector L (2002) *Size control via size fair genetic operators in the PushGP genetic programming system*. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation* 733-739. Morgan Kaufmann Publishers Inc.

[10] Dempster A. G. & Macleod M. (1994). *Constant integer multiplication using minimum adders*. In *IEE Proceedings - Circuits, Devices and Systems* 141(5), 407–413.

[11] Dempster A. G. & Macleod M. D. (1995) *Use of minimum-adder multiplier blocks in FIR digital filters*. In *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 42(9), 569-577. Orlando, Florida.

[12] D'haeseleer P. (1994) *Context preserving crossover in genetic programming*. In *Proceedings of the First IEEE Conference on Evolutionary Computation*. IEEE World Congress on Computational Intelligence. 256-261

[13] Fogel L.J., Angeline P.J., Fogel D.B. (1995) *An Evolutionary Programming Approach to Self-Adaptation on Finite State Machines*. In *Proceedings of the Fourth Annual Conference on Evolutionary Programming*, 355-365. MIT Press.

[14] Fogel D.B. (1999) *An Overview of Evolutionary Programming*. In: Davis L.D., De Jong K., Vose M.D., Whitley L.D. (eds) Evolutionary Algorithms. The IMA Volumes in Mathematics and its Applications, vol 111. Springer, New York, NY.

[15] Forsyth R. (1981) *BEAGLE - A Darwinian Approach to Pattern Recognition*, In Kybernetes, Vol. 10 Issue: 3, pp.159-166, https://doi.org/10.1108/eb005587

[16] García-Arnau M., Manrique D., Ríos J., Rodríguez-Patón A. (2007) *Initialization method for grammar-guided genetic programming. Knowledge-Based Systems 20* (2007) 127–133. Departamento de Inteligencia Artificial. Universidad Politécnica de Madrid.

[17] Goldberg D.E. (1983) *Computer-aided gas pipeline operation using genetic algorithms and rule learning*. University of Michigan. PhD Thesis. University Microfilms International.

[18] Graphviz - Graph Visualization Software. `https://www.graphviz.org/`. 04/06/2019.

[19] Hausser R. (2014) *Foundations of Computational Linguistics. Human-Computer Communication in Natural Language*. Third Edition. Springer. Friedrich-Alexander-Universität Erlangen-Nürnberg. Erlangen, Germany.

[20] Hoai, N. X., McKay, R. I., & Abbass, H. A. (2003). *Tree adjoining grammars, language bias, and genetic programming*. In *European Conference on Genetic Programming*. pp. 335-344. Springer, Berlin, Heidelberg.

[21] Holland J. H. (1975) *Adaptation in Natural and Artificial Systems*. University of Michigan Press. (Second edition: MIT Press, 1992.)

[22] Iba H. (1996) *Random Tree Generation for Genetic Programming*. In *International Conference on Parallel Problem Solving from Nature IV*. pp 144-153. Lecture Notes in Computer Science, vol 1141. Springer, Berlin, Heidelberg.

[23] Konak A., Coit D.W., Smith A.E. (2006) *Multi-objective optimization using genetic algorithms: A tutorial*. In *Reliability Engineering & System Safety*. 91(9), 992-1007. Elsevier.

[24] Koza J.R. (1988) *Non-Linear Genetic Algorithms for Solving Problems*. USA Patent 4935877, June 1990.

[25] Koza J.R. (1989) *Hierarchical Genetic Algorithms operating on populations of computer programs*. In *11^{th} International Joint Conference on Artificial Intelligence* 1, 768-774. Detroit, Michigan.

[26] Koza J.R. (1990) *Genetic Programming: A Paradigm for genetically breeding populations of computer programs to solve problems*. Computer Science Department. Stanford University.

[27] Koza J.R. (1992) *Genetic programming: On the programming of computers by means of natural selection*. Complex Adaptive Systems. MIT Press edition. Sixth printing, 1998. ISBN 0-262-11170-5

[28] Langdon W. (1998) *The Evolution of Size in Variable Length Representations*. In *1998 IEEE International Conference on Evolutionary Computation*. IEEE World Congress on Computational Intelligence pp. 633-638. University of Birmingham.

[29] Langdon W. (1999) Size Fair and Homologous Tree Genetic Programming Crossovers. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation*. Volume 2. pp. 1092-1097. Morgan Kaufmann Publishers Inc.

[30] Luke S. (2000) *Two Fast Tree-Creation Algorithms for Genetic Programming*. In *IEEE Transactions on Evolutionary Computation*. 4(3), 274-283.

[31] Luke S. & Panait L. (2001) *A Survey and Comparison of Tree Generation Algorithms*. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation* 81-88. San Francisco, California.

[32] Luke S. (2015) *Essentials of metaheuristics: a set of undergraduate lecture notes*. Department of Computer Science: George Mason University. Online Version 2.2. `http://cs.gmu.edu/~sean/book/metaheuristics/`.

[33] Manrique D, Márquez F, Ríos J, Rodríguez-Patón A. (2005) *Grammar Based Crossover Operator in Genetic Programming*. In *Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach*. IWINAC 2005. Lecture Notes in Computer Science, vol 3562. Springer, Berlin, Heidelberg

[34] Manrique D, Ríos J, Rodríguez-Patón A. (2009) *Grammar-Guided Genetic Programming*. Departamento de Inteligencia Artificial. Universidad Politécnica de Madrid. In *Encyclopedia of Artificial Intelligence*. Universidade da Coruña.

[35] Martínez Sober M, Gómez Chova L, Gómez Sanchis J, Serrano López A.J, Vila Francés J. (2009-2010) *Filtros Digitales 2009/2010*. Departamento de Enginyeria Electrònica: Universitat de València. `http://ocw.uv.es/ingenieria-y-arquitectura/filtros-digitales/temario/`.

[36] Melián Batista B., Moreno Pérez J.A., Moreno Vega J.M. (2009) *Introducción a la Computación Evolutiva*. Universidad de La Laguna. En *NÚMEROS, Revista de Didáctica de las Matemáticas*. Volumen 71 (Agosto 2009), 21-27. Sociedad Canaria Isaac Newton de Profesores de Matemáticas.

[37] Mitchell, M. (1996) *An introduction to genetic algorithms*. Cambridge, Mass: MIT Press. ISBN: 978-0-262-63185-3

[38] Proakis J., & Manolakis D. (2007). *Tratamiento digital de señales*. 4ª Edición. Traducción de *Digital Signal Processing*. Pearson-Prentice Hall. ISBN: 978-84-8322-347-5

[39] Püschel M, Franchetti F, & Voronenko Y. (2011) *Spiral*. In *Encyclopedia of Parallel Computing* pp. 1920-1933. Department of Electrical and Computer Engineering. Carnegie Mellon University. Pittsburgh, PA

[40] Rabanillo García J. (2013) *Detección de Soft Errors en Bloques Multiplicadores de Filtros FIR*. Trabajo Fin de Máster. Máster en Investigación Informática. Departamento de Arquitectura de Computadores y Automática, Universidad Complutense de Madrid.

[41] Reeves C. (2003) *Genetic algorithms*. In *Handbook of Metaheuristics*. Kluwer Academics. 55-82.

[42] Ryan C, Collins J.J, O'Neill M. (1998) *Grammatical evolution: Evolving programs for an arbitrary language*. In: *Genetic Programming*. EuroGP 1998. Lecture Notes in Computer Science, vol 1391. Springer, Berlin, Heidelberg

[43] Spiral project. *Multiplier Block Generator*. `http://spiral.ece.cmu.edu/mcm/gen.html`. 09/03/2019

[44] Spiral project. *Finite/Infinite Impulse Response Filter Generator*. `http://spiral.net/hardware/filter.html`. 29/05/2019

[45] The MathWorks, Inc. *MATLAB fir1() function*. `https://es.mathworks.com/help/signal/ref/fir1.html?lang=en`. 09/03/2019

[46] Turing A.M. (1950) *Computing Machinery and Intelligence* In *Mind* 49: 433-460

[47] Voronenko, Y. & Püschel, M. (2007) *Multiplierless Multiple Constant Multiplication. ACM Transactions on Algorithms (TALG)* 3(2), 11. Carnegie Mellon University.

[48] Whigham P.A. (1995) *Grammatically-based genetic programming*. In *Proceedings of the Workshop on Genetic Programming: from theory to real-world applications* 33-41. University of Otago. New Zealand.

[49] White, D. (2009) *An overview of Schema Theory*. arXiv:1401.2651. `https://arxiv.org/abs/1401.2651`.