FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

TÉCNICAS DE OPTIMIZACIÓN PARA PROGRAMACIÓN GENÉTICA APLICADAS AL DISEÑO DE FILTROS ANALÓGICOS PASIVOS

TESIS DE GRADO EN INGENIERÍA INFORMÁTICA

FACULTAD DE INGENIERÍA
UNIVERSIDAD DE BUENOS AIRES

TESISTA: SR. MARIANO MARTÍN CHOUZA

DIRECTORES: PROF. DR. RAMÓN GARCÍA-MARTÍNEZ
PROF. LIC. ING. OSVALDO CLÚA

LABORATORIO DE SISTEMAS INTELIGENTES

NOVIEMBRE 2008

A pesar de las inevitables omisiones, no puedo dejar de expresar mi agradecimiento:

A Ramón y Osvaldo, por acompañarme en este último tramo de mi carrera.

A Fer, Guille y otros muchos compañeros, por mostrarme que el trabajo en equipo puede funcionar.

A Fernando, mi hermano, por enseñarme el valor de una segunda opinión.

A mis padres, por su constante apoyo.

TABLA DE CONTENIDOS

Tabla de contenidos 5
1. Introducción 9
1.1. Generalidades 9
1.1.1. Definiciones 9
1.1.2. Aplicaciones de los filtros analógicos 10
1.1.3. Antecedentes 11
1.2. Objetivos 11
1.3. Plan de trabajo 12
1.4. Plan de estudio 12
1.5. Estructura de la tesis 12
2. Estado del arte 15
2.1. Diseño de filtros analógicos pasivos 15
2.1.1. Definición y clasificación de los filtros 15
2.1.2. Características propias de los filtros analógicos pasivos 27
2.1.3. Procedimientos manuales de diseño 28
2.1.4. Herramientas automatizadas no basadas en la programación genética 31
2.2. Programación genética 33
2.2.1. Definición y generalidades 33
2.2.2. Estructuras de los programas 37
2.2.3. Aplicaciones 40
2.3. Programación genética aplicada al diseño de filtros analógicos 43
2.3.1. Representaciones del circuito 43
2.3.2. Desarrollo embrionario 47
2.3.3. Resultados 49
3. Problemas Tratados 51
3.1. Generalidades 51
3.2. Crecimiento del Tamaño de los Individuos 52
3.2.1. Precisión en la replicación 52
3.2.2. Tendencia en la eliminación 52
3.2.3. Naturaleza de los espacios explorados 53
3.2.4. Correlación con la profundidad 53
3.3. Convergencia prematura 53
4. Solución propuesta 55
4.1. Generalidades 55
4.2. Técnicas de Control de poblaciones 56
4.2.1. Uso de plagas para controlar el esfuerzo computacional 56
4.2.2. Ajuste dinámico de la función de evaluación 58
4.3. Utilización de un caché de evaluación 59
4.3.1. Generalidades 60
4.3.2. Selección de una función de hash 62
4.3.3. Medición de rendimiento aplicado a la evaluación 64
5. Verificación experimental 67
5.1. Generalidades 67
5.2. Relación entre el tamaño de los individuos y el tiempo requerido para evaluarlos 68
5.3. Resultados de las técnicas de optimización 69
5.3.1. Caso base 70
5.3.2. Aplicando plagas para controlar el crecimiento del esfuerzo computacional 72
5.3.3. Realizando un ajuste dinámico de la función de evaluación 73

5.3.4. Aplicando un caché de evaluación 73
5.4. Comparación entre los resultados obtenidos al aplicar las distintas técnicas 79
5.5. Ejemplos de lOS resultados obtenidos y comparación con otros trabajos realizados 80
6. Conclusiones 85
7. Referencias 87
A. Manual de usuario 93
A.1. Generalidades 93
A.2. Sintaxis de la línea de comandos 93
A.3. Sintaxis del archivo de descripción de trabajos 93
A.4. Sintaxis de la salida 95
B. Estructura del código de prueba utilizado 97
B.1. Generalidades 97
B.2. Estructura general del sistema de prueba AFDGP 97
B.3. Estructura de los módulos integrantes del sistema 98
B.3.1. Módulo principal 98
B.3.2. Módulo de operaciones sobre el genoma 103
B.3.3. Módulo de evaluación 104
B.3.4. Módulo SPICE 106
B.4. ejecución del sistema 107
B.4.1. Flujo general del programa 107
B.4.2. Evaluación de un individuo 108
C. Descripción general del SPICE 113
C.1. Descripción general 113
C.2. Descripción de los módulos 114
C.2.1. Módulo de entrada/salida 114
C.2.2. Módulo de simulación y algoritmos numéricos 114
C.2.3. Módulo de dispositivos 115
C.2.4. Módulo de manejo de matrices 115
C.3. Flujo de ejecución 115
C.4. Formato de la entrada 115
D. Modificaciones Realizadas al SPICE 119
D.1. Motivos para las modificaciones realizadas 119
D.2. Modificaciones realizadas al módulo de entrada/salida 119
D.3. Modificaciones realizadas al manejo de memoria 121
D.4. Otras posibles modificaciones 121
E. Funciones utilizadas en el genotipo 123
E.1. Generalidades 123
E.2. Funciones que modifican la topología 123
E.2.1. SERIES 123
E.2.2. PARALLEL 124
E.2.3. TWO_GROUND 124
E.2.4. FLIP 124
E.2.5. END 125
E.2.6. SAFE_CUT 125
E.3. Funciones que crean componentes 125
E.3.1. TWO_LEAD 125
E.4. Funciones que devuelven valores 126
E.4.1. R 126
E.4.2. C 126
E.4.3. L 126
E.4.4. Valor de componente 126
F. Pruebas realizadas 129

F.1. Pruebas realizadas 129
F.2. Procesamiento de los Resultados obtenidos 131
F.2.1. Procedimiento de reducción de datos 131
F.2.2. Forma de los datos reducidos 132
G. Procedimientos estadísticos utilizados 135
G.1. Media geométrica 135
G.2. Intervalos de confianza 135

1. INTRODUCCIÓN

En este capítulo se presenta una introducción general a la temática de la tesis. Además se describe el contenido de los capítulos posteriores en los que se desarrollarán estos temas con mayor profundidad.

1.1. GENERALIDADES

1.1.1. DEFINICIONES

Se denomina programación genética a la aplicación de algoritmos genéticos [Mitchell, 1998] al problema de la programación automática, es decir al desarrollo de programas sin injerencia directa de seres humanos. La intervención humana se limita a la determinación de un objetivo para el programa. Esta técnica fue popularizada por John Koza en su trabajo pionero *Genetic Programming* [Koza, 1992]. Posteriormente se aplicó a diferentes problemas, obteniendo en años recientes resultados considerados competitivos con los de seres humanos [Koza *et al.*, 2003; Lohn, 2004].

Uno de los principales problemas que posee la programación genética es el crecimiento de los programas que integran la población, lo que conlleva un correspondiente incremento en los tiempos de simulación [Streeter, 2003]. Entre las técnicas propuestas para eliminar este inconveniente están:

- Plagas para mantener la diversidad genética, en conjunto con poblaciones de tamaño variable. [Fernandez *et al.*, 2003]
- Creación de "agujeros" en forma dinámica en la función de ajuste de los individuos [Poli, 2003].

Para los fines de este trabajo, se define como filtro a un sistema que realiza una función de procesamiento de señales, más específicamente removiendo y/o amplificando componentes de las mismas. [Paarmann, 2003]

Como puede suponerse a partir de la definición, existen muchas clases de filtros y muchas formas de clasificarlos. Debe diferenciarse entre el filtro "abstracto", o sea la entidad matemática que representa la relación deseada entre la entrada y la salida y el filtro "concreto", que es el objeto físico que realiza la transformación de la señal [Paarmann, 2003]. De la misma forma, debe diferenciarse entre las entradas y salidas "abstractas", que son funciones matemáticas y las entradas y salidas "concretas", que son magnitudes físicas; en el caso del presente trabajo, señales eléctricas.

El filtro "abstracto" suele caracterizarse por lo que se denomina *función de transferencia* [Oppenheim *et al.*, 1996], que es la relación entre la Transformada de Laplace de la entrada y la

Transformada de Laplace de la salida. Como es lógico, esta representación depende de que el filtro sea *lineal*, como es el caso de la gran mayoría de los filtros de aplicación práctica y, particularmente, en los filtros que se tratarán en el presente trabajo.

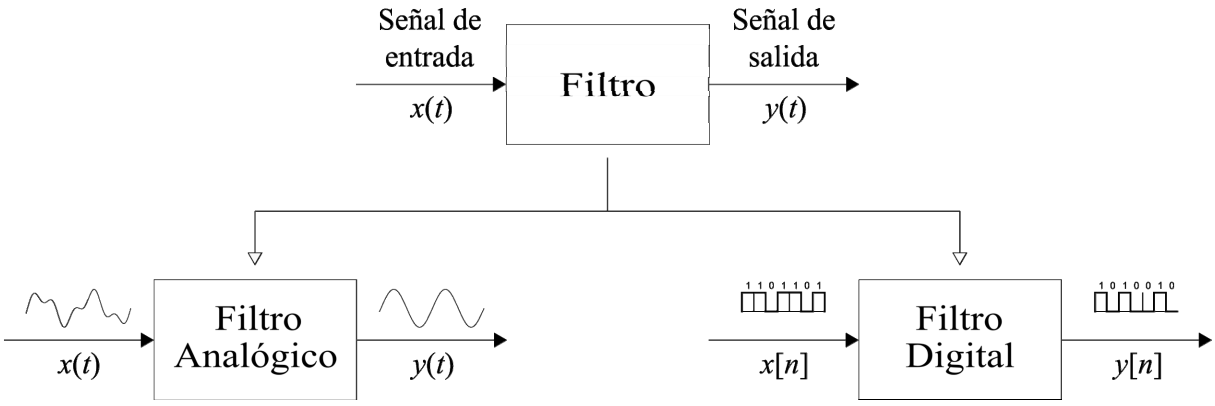


Figura 1.1. Filtro abstracto y sus implementaciones clasificadas según la representación de la señal.

Los filtros "concretos" pueden caracterizarse según varios aspectos. Para los fines de esta introducción se analizarán dos: según la codificación de señales con las que opera (analógico vs. digital) y según la clase de componentes con los que está formado (pasivo vs. activo).

La diferencia entre un filtro analógico y uno digital consiste en que, mientras el analógico opera sobre una señal eléctrica que se considera análoga a la señal de entrada "abstracta" (por ejemplo una tensión proporcional a la señal de entrada), el digital lo hace sobre una representación numérica, por lo general binaria, de la entrada (ver figura 1.1).

Los filtros pasivos y activos solo difieren en la clase de componentes que se utilizan: los activos utilizan semiconductores mientras que los pasivos no. Sin embargo, suele limitarse aun más la categoría de filtros pasivos, restringiéndola a los formados por una combinación de resistores, capacitores e inductores [Paarmann, 2003]. Esta definición más restrictiva es la que se utilizará para este trabajo.

1.1.2. APLICACIONES DE LOS FILTROS ANALÓGICOS

Los filtros analógicos siguen teniendo en la actualidad un amplio espectro de aplicaciones, a pesar del gran crecimiento en la utilización de filtros digitales ocurrido en las últimas décadas. Debido a la gran cantidad de dispositivos en los que se realiza alguna tarea de procesamiento de señales, es difícil enumerar exhaustivamente las aplicaciones de los filtros analógicos. Algunos ejemplos específicos donde se utilizan son:

- Procesamiento de señales de audio para mejorar la inteligibilidad de la voz humana.
- Separación de canales de audio en la recepción de radio estéreo.

- Decodificación de los tonos del sistema DTMF, correspondientes al marcado telefónico.
- Separación del tráfico de voz y datos en una conexión ADSL.

1.1.3. ANTECEDENTES

Los primeros trabajos sobre diseño automático de circuitos analógicos por técnicas no convencionales (es decir que no fueran una extensión de procedimientos manuales) fueron realizados por Richard Stallman y Gerald Sussman mientras trabajaban para el Laboratorio de Inteligencia Artificial del MIT [Sussman & Stallman, 1975]. Sin embargo este trabajo estaba en general orientado a la utilización de técnicas heurísticas para analizar "por inspección" circuitos complejos. Luego aparecieron diversas herramientas de síntesis de filtros en los que el usuario elegía una topología, aparte de la transferencia deseada, y el programa seleccionaba los valores de los componentes.

El primero en aplicar la técnica de programación genética al diseño de filtros analógicos fue el equipo de John Koza tal como se describe en [Koza *et al.*, 1997] y, con mayor detenimiento, en [Koza *et al.*, 2003]. En algunos trabajos posteriores se utilizaron representaciones menos convencionales de los circuitos (ver por ejemplo [Fan *et al.*, 2001]), pero sus ventajas no son aparentes y conllevan una transformación más compleja a la hora de la simulación, si esta se lleva a cabo con un sistema "estándar" como el SPICE [Kielkowski, 1998].

1.2. OBJETIVOS

- Desarrollar un sistema para desarrollar filtros analógicos utilizando programación genética. Si bien este trabajo solo utilizará filtros pasivos, debe ser fácilmente adaptable para una mayor generalidad. Los filtros desarrollados utilizarán componentes cuyos valores serán los normalizados, disponibles comercialmente.

- Estudiar la aplicabilidad a este problema de las estrategias mencionadas anteriormente para evitar el crecimiento del código y las diferencias entre la utilización de poblaciones estacionarias vs. la estrategia de recambio generacional.

- Comparar los resultados obtenidos por el sistema desarrollado con los obtenidos por los otros autores mencionados.

1.3. PLAN DE TRABAJO

- Estudio detallado del problema. Diseño de la arquitectura general del sistema. Estudio de las modificaciones requeridas al SPICE [Kielkowski, 1998; Quarles, 1989b] para utilizarlo en la evaluación. (3 meses)
- Implementación del sistema. (4 meses)
- Estudio del sistema desarrollado (2 meses). Comparación entre diferentes estrategias. Comparación con resultados de la literatura.
- Preparación del informe sobre los resultados obtenidos. (2 meses)

1.4. PLAN DE ESTUDIO

- Análisis de Circuitos.
- Introducción a los Sistemas Inteligentes.
- Sistemas de Programación No Convencional de Robots.

1.5. ESTRUCTURA DE LA TESIS

Esta tesis se encuentra estructurada en forma de seis capítulos principales y siete capítulos anexos.

En el capítulo 2 se presenta el estado del arte de las tecnologías utilizadas, incluyendo: una descripción general del área del diseño de filtros analógicos pasivos (sección 2.1), un estudio de los aspectos generales de la programación genética (sección 2.2) y un análisis de la aplicación de la programación genética al diseño de filtros analógicos (sección 2.3), mostrando tanto los aspectos generales como los resultados alcanzados por otros autores.

En el capítulo 3 se muestra una sinopsis de los problemas que se intentan resolver en el presente trabajo, incluyendo: una descripción general de los problemas que afectan el rendimiento de la programación genética (sección 3.1), una descripción detallada del problema del crecimiento del código (*bloat*) (sección 3.2) y un análisis del problema de la convergencia prematura (sección 3.3), vinculado a la calidad de los resultados obtenidos aplicando las técnicas de optimización.

En el capítulo 4 se explica la solución propuesta a los problemas descritos en el capítulo 3, o sea: una descripción general de las soluciones que se han propuesto para enfrentar estos problemas y las métricas para evaluar su desempeño (sección 4.1), un recorrido más detallado por las técnicas propuestas para resolver estos problemas mediante el control de poblaciones (sección 4.2) y una

descripción de la aplicación de un caché con objeto de acelerar la evaluación de las poblaciones (sección 4.3).

En el capítulo 5 se presentan los pasos seguidos para realizar la verificación experimental y los resultados obtenidos, más en detalle: una descripción general de la forma en que se llevaron a cabo los experimentos (sección 5.1), un análisis de la relación entre el tamaño de los individuos y el tiempo que se requiere para evaluarlos (sección 5.2), se estudian los resultados obtenidos al aplicar las técnicas de optimización propuestas (sección 5.3) y se concluye con un análisis comparativo de los resultados obtenidos (sección 5.3.4.4) en si mismos y comparados con otros trabajos (sección 5.5).

En el capítulo 6 se indican las conclusiones alcanzadas en base a los resultados experimentales obtenidos y se dan posibles líneas de investigación que podrían extender los resultados obtenidos en el presente trabajo.

En el capítulo anexo A, se describen en forma breve la forma de uso del software desarrollado para evaluar las técnicas de optimización propuestas en el capítulo 4. Esto incluye los aspectos generales de este software (sección A.1), la sintaxis de línea de comandos para su ejecución (sección A.2), la sintaxis del archivo mediante el cual se describe la tarea de diseño a realizar (sección A.3) y la sintaxis de la salida de este programa (sección A.4).

En el capítulo anexo B se muestran aspectos generales del sistema desarrollado (sección B.1), la estructura general de este sistema (sección B.2), incluyendo su arquitectura modular, la estructura de cada uno de los módulos que lo integran (sección B.3) y, para finalizar, se describe la ejecución del sistema (sección B.4).

En el capítulo anexo C se da una descripción general del simulador SPICE (sección C.1), una descripción de los módulos que integran este programa (sección C.2), el flujo de ejecución del programa (sección C.3) y, por último, una descripción del formato de las entradas en los aspectos utilizados en este trabajo (sección C.4).

En el capítulo anexo D se muestran los motivos para las modificaciones que se efectuaron al SPICE3F5 (sección D.1), una descripción de las modificaciones realizadas sobre el módulo de entrada/salida del SPICE (sección D.2), un detalle de las alteraciones efectuadas al manejo de memoria del SPICE (sección 0) y se concluye con algunas modificaciones no efectuadas, pero que podrían ser de interés para trabajos futuros (sección D.4).

En el capítulo anexo E se presentan aspectos generales respecto al conjunto de funciones seleccionado (sección E.1), una descripción de las funciones que modifican la topología (sección

E.2), una explicación del comportamiento de la única función de creación de componentes incluida (sección E.3) y se concluye con una análisis de las funciones que devuelven valores (sección E.4).

En el capítulo anexo F se detallan las pruebas realizadas con el software desarrollado (sección F.1) y el procesamiento de los resultados obtenidos (sección F.2).

En el capítulo anexo G se muestran los motivos por los que las evaluaciones de tiempos se promediaron en forma geométrica y la relación de esta media con la media aritmética de los logaritmos (sección G.1) y se explica la forma en que se determinaron los intervalos de confianza para los promedios de los tiempos de ejecución en la sección 5.3.4.4 (sección G.2).

2. ESTADO DEL ARTE

En este capítulo se presenta: una descripción general del área del diseño de filtros analógicos pasivos (sección 2.1), incluyendo la definición y clasificación de los filtros en general (sección 2.1.1), las características propias de los filtros analógicos pasivos (sección 2.1.2), los procedimientos manuales de diseño de estos filtros (sección 2.1.3) y las herramientas automatizadas para esta misma tarea (sección 2.1.4); se continua con un estudio de la programación genética (sección 2.2), abarcando su definición y aspectos generales (sección 2.2.1), las formas de representar a los programas que participan de este proceso (sección 2.2.2) y las aplicaciones en las que se ha utilizado esta técnica (sección 2.2.3); se concluye con una descripción de la aplicación de la programación genética al diseño de filtros analógicos (sección 2.3), donde se comienza mostrando las representaciones del circuito que pueden utilizarse (sección 2.3.1), se sigue con una resumen del proceso de desarrollo embrionario (sección 2.3.2) y se concluye mostrando algunos de los resultados obtenidos por otros autores (sección 2.3.3).

2.1. DISEÑO DE FILTROS ANALÓGICOS PASIVOS

En esta sección se describe el estado del arte en cuanto al diseño de los filtros analógicos pasivos, exceptuando el uso de la programación genética, descrito en la sección 2.3. Para ello, se detallan: la definición y clasificación de los filtros en general (sección 2.1.1), incluyendo tanto la definición abstracta de filtros y su descripción matemática (sección 2.1.1.1), como las clasificaciones en término de la representación utilizada para las señales (sección 2.1.1.2), según la forma de la transferencia (sección 2.1.1.3), de acuerdo a la forma de aproximar a la transferencia (sección 2.1.1.4) y en base a los componentes utilizados para construirlo físicamente (sección 2.1.1.5); luego se remarca en los aspectos propios de los filtros analógicos pasivos (sección 2.1.2), seleccionados como problema de aplicación para este trabajo, y se analizan los métodos disponibles para diseñar a esta clase de filtros, tanto manuales (sección 2.1.3) como automatizados (sección 2.1.4).

2.1.1. DEFINICIÓN Y CLASIFICACIÓN DE LOS FILTROS

En esta sección se analiza tanto la definición general de los filtros (sección 2.1.1.1), como las distintas clasificaciones que pueden realizarse sobre ellos, incluyendo: su clasificación en base a ala forma en que se representa a las señales que manejan (sección 2.1.1.2); de acuerdo a la forma general que toma su función de transferencia (sección 2.1.1.3); de acuerdo a como se aproximan a una transferencia ideal (sección 2.1.1.4) y según la clase de componentes utilizados para su construcción (sección 2.1.1.5)

2.1.1.1. Definición y generalidades

En muchos dominios distintos se encuentra la idea de *filtro*. Existen, por nombrar solo algunos, filtros fotográficos, cuyo objetivo es modificar la composición de la luz que ingresa a la cámara, filtros de agua, que remueve impurezas del agua mediante el uso de una barrera física, filtros de aire, que remueven partículas sólidas del mismo y filtros que operan con señales. Si bien los campos de aplicación de estos dispositivos mencionados son totalmente dispares, tienen una idea en común: dejar pasar a su salida *solo algunos componentes* de su entrada [Paarmann, 2003].

Todos los filtros que se encontrarán en este trabajo están dedicados a procesar señales que varían en el tiempo; un esquema de esta clase de filtro puede verse en la figura 2.1. Los filtros que trabajan con señales no constituyen un conjunto homogéneo; como se verá en las secciones posteriores, pueden clasificarse de acuerdo a la forma en que filtran, representación de las señales con que trabajan, componentes que los integran y otros aspectos adicionales.

Figura 2.1. Esquema representando a un filtro en el área de procesamiento de señales.

Los filtros encargados de procesar señales se utilizan en un gran número de aplicaciones. Estas van desde procesar señales de audio para eliminar ruido u obtener distintos efectos hasta separar el tráfico de datos de la voz en una conexión ADSL. Posteriormente serán analizadas las aplicaciones específicas de la clase de filtros de las que se ocupa este trabajo.

Se dice que un filtro es lineal cuando responde a una combinación lineal de entradas dando como salida esa misma combinación lineal de las salidas que corresponderían a cada entrada por separado. En otros términos:

$$T\left(\alpha\, x(t) + \beta\, y(t)\right) = \alpha\, T\left(x(t)\right) + \beta\, T\left(y(t)\right),$$

donde T representa la acción del filtro sobre la señal. Se refiere a un sistema como independiente del tiempo cuando su acción sobre una señal no depende en forma explícita del tiempo. Expresado matemáticamente:

$$T\left(x(t)\right) = y(t) \rightarrow T\left(x(t - t_0)\right) = y(t - t_0)\,.$$

Existe un importante conjunto de teoría dedicado a tratar con sistemas lineales e invariantes en el tiempo, denominados LTI por sus iniciales en inglés. Entre otras cosas, esta teoría permite aplicar los conceptos de transformada Z, en el caso de sistemas discretos, o los de transformadas de Fourier

y Laplace, para el caso de sistemas continuos, al análisis de los filtros en cuestión [Oppenheim *et al.*, 1996].

Aunque existen aplicaciones importantes para los filtros no lineales [Daum, 2005] y para los que poseen características dependientes del tiempo [Alvarez, 2002], la gran mayoría de los filtros utilizados en la práctica son lineales e independientes del tiempo. De este punto en adelante se supondrá implícitamente que todo filtro es lineal y presenta características invariantes en el tiempo.

Además de la linealidad y de la invariancia temporal, existen otras características que pueden utilizarse para clasificar a los filtros. En las siguientes secciones se verán algunas de estas clasificaciones, seleccionando las que presenten un especial interés para este trabajo.

2.1.1.2. Clasificación según la representación de la señal utilizada

La primera clasificación que puede hacerse es marcar una diferencia entre los filtros que operan con representaciones analógicas de una señal y aquellos que operan con representaciones digitales de la misma. De acuerdo a esto se divide de forma correspondiente a los filtros en analógicos y digitales (ver figura 2.2).

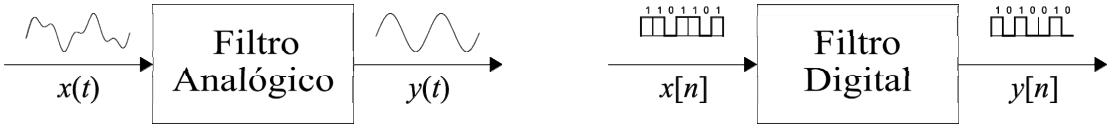


Figura 2.2. Un filtro analógico **(a)** y un filtro digital **(b)**.

En el caso de un filtro analógico, tanto la entrada como la salida toman la forma de funciones continuas del tiempo (o de funciones que pueden *tratarse* como "continuas", tales como la "función" delta). Si se asume que el filtro es lineal e invariante en el tiempo tal como se mencionó anteriormente, se puede representar el comportamiento del filtro por una función compleja, denominada *función de transferencia*.

Para ver esto, pueden realizarse las siguientes definiciones:

$$\begin{aligned} y(t) &= T\big(x(t)\big) \\ X(s) &= \mathcal{L}\{x(t)\} \\ Y(s) &= \mathcal{L}\{y(t)\} \\ h(t) &= T\big(\delta(t)\big) \\ H(s) &= \mathcal{L}\{h(t)\} \end{aligned}$$

Entonces, teniendo en cuenta que la convolución con la "función" delta es la identidad y las propiedades de la transformada de Laplace (en este trabajo se utiliza la transformada *bilateral* de Laplace salvo indicación en contrario) [Oppenheim *et al.*, 1996], puede apreciarse que:

$$
\begin{aligned}
Y(s) &= \mathcal{L}\{T(x(t)*\delta(t))\} \\
&= \mathcal{L}\left\{T\left(\int_{\Re} x(u)\delta(t-u)\,du\right)\right\} \\
&= \mathcal{L}\left\{\int_{\Re} T(x(u)\delta(t-u))du\right\} \\
&= \mathcal{L}\left\{\int_{\Re} x(u)T(\delta(t-u))du\right\} \\
&= \mathcal{L}\{x(t)*h(t)\} \\
&= \mathcal{L}\{x(t)\}\mathcal{L}\{h(t)\} \\
&= X(s)H(s).
\end{aligned}
$$

Esto indica que puede obtenerse la transformada de Laplace de la salida simplemente multiplicando la entrada por una función compleja *H(s)* que cuantifica como se "transfiere" la señal de entrada a la salida. Por este motivo, la función *H(s)* es conocida como función de transferencia y permite una descripción abstracta de como se comporta un filtro analógico dado.

Por otro lado, los filtros digitales operan con señales de tiempo discreto y con un conjunto discreto de valores. A primera vista podría parecer absurdo preferir una señal con un contenido claramente limitado de información a una señal capaz de variar en forma continua su valor entre un continuo de posibles valores. Sin embargo, si se consideran canales analógicos realistas, las limitaciones en cuanto a ancho de banda y relación señal-ruido imponen similares limitaciones a la capacidad de transmisión de estos canales [MacKay, 2002].

Con los filtros digitales puede realizarse un análisis similar al efectuado más arriba, sustituyendo la transformada Z [Oppenheim *et al.*, 1996] por la transformada de Laplace. Efectivamente, si realizamos las siguientes definiciones:

$$
\begin{aligned}
y[n] &= T(x[n]) \\
X(z) &= \mathcal{Z}\{x(t)\} \\
Y(z) &= \mathcal{Z}\{y(t)\} \\
h[n] &= T(\delta[n]) \\
H(z) &= \mathcal{Z}\{h[n]\},
\end{aligned}
$$

donde *n* representa el tiempo para hacer énfasis en su carácter discreto, podemos mostrar la existencia de una transferencia bien definida. Realizando una operación análoga a la efectuada en el caso continuo:

$$
\begin{aligned}
Y(z) &= \mathcal{I}\{T(x[n]*\delta[n])\} \\
&= \mathcal{I}\left\{T\left(\sum_{k=-\infty}^{+\infty} x[k]\delta[n-k]\right)\right\} \\
&= \mathcal{I}\left\{\sum_{k=-\infty}^{+\infty} T(x[k]\delta[n-k])\right\} \\
&= \mathcal{I}\left\{\sum_{k=-\infty}^{+\infty} x[k]T(\delta[n-k])\right\} \\
&= \mathcal{I}\{x[n]*h[n]\} \\
&= \mathcal{I}\{x[n]\}\mathcal{I}\{h[n]\} \\
&= X(z)\,H(z).
\end{aligned}
$$

Al igual que en el caso de los filtros analógicos, se observa la existencia de una función, en este caso *H(z)*, que describe el comportamiento del filtro ante una entrada arbitraria. Pueden verse más detalles respecto a estas descripciones en [Oppenheim *et al*., 1996], [Paarmann, 2003] y [Thede, 2004].

2.1.1.3. Clasificación según la forma de la función de transferencia

Otra clasificación muy empleada se basa en la relación entre la composición espectral de la salida y la composición espectral de la entrada. Esta relación puede establecerse en base a la forma de la función de transferencia ya que, como

$$Y(s) = H(s)X(s) \rightarrow |Y(s)| = |H(s)| \cdot |X(s)| \rightarrow |Y(j\omega)| = |H(j\omega)| \cdot |X(j\omega)|,$$

la forma del módulo de la función de transferencia indica qué componentes espectrales serán atenuadas y qué componentes serán reforzadas. Este análisis se realizó sobre el dominio de tiempo continuo, por ser el utilizado en este trabajo; sin embargo, es igualmente aplicable a los sistemas de tiempo discreto.

Si la relación entre el módulo de la función de transferencia y la frecuencia se expresa en un gráfico de forma similar al de la figura 2.3, indicaría la presencia de un filtro conocido como *pasabajos*. El nombre se refiere a que deja pasar bajas frecuencias y bloque frecuencias más elevadas. La frecuencia que marca la transición entre la banda pasante y la banda bloqueada se conoce como frecuencia de corte.

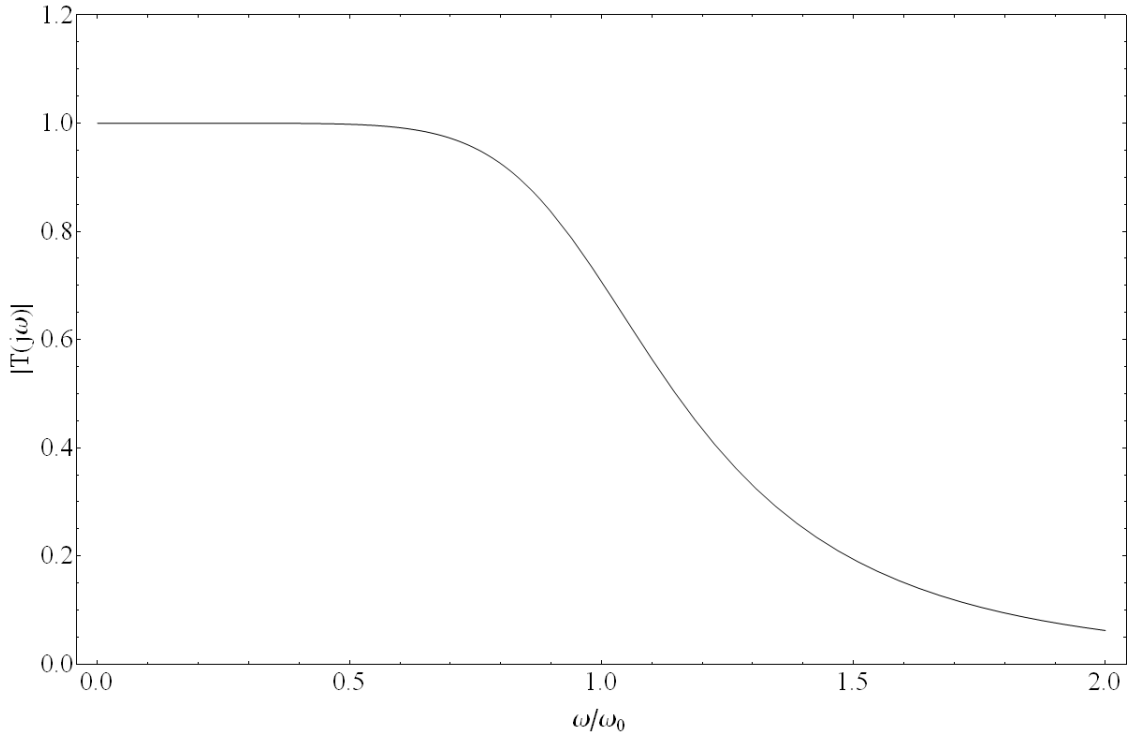


Figura 2.3. Transferencia típica de un filtro pasabajos.

En forma simétrica a los filtros pasabajos, se define a los filtros *pasaaltos*, que presentan una transferencia de la forma general visible en la figura 2.4.

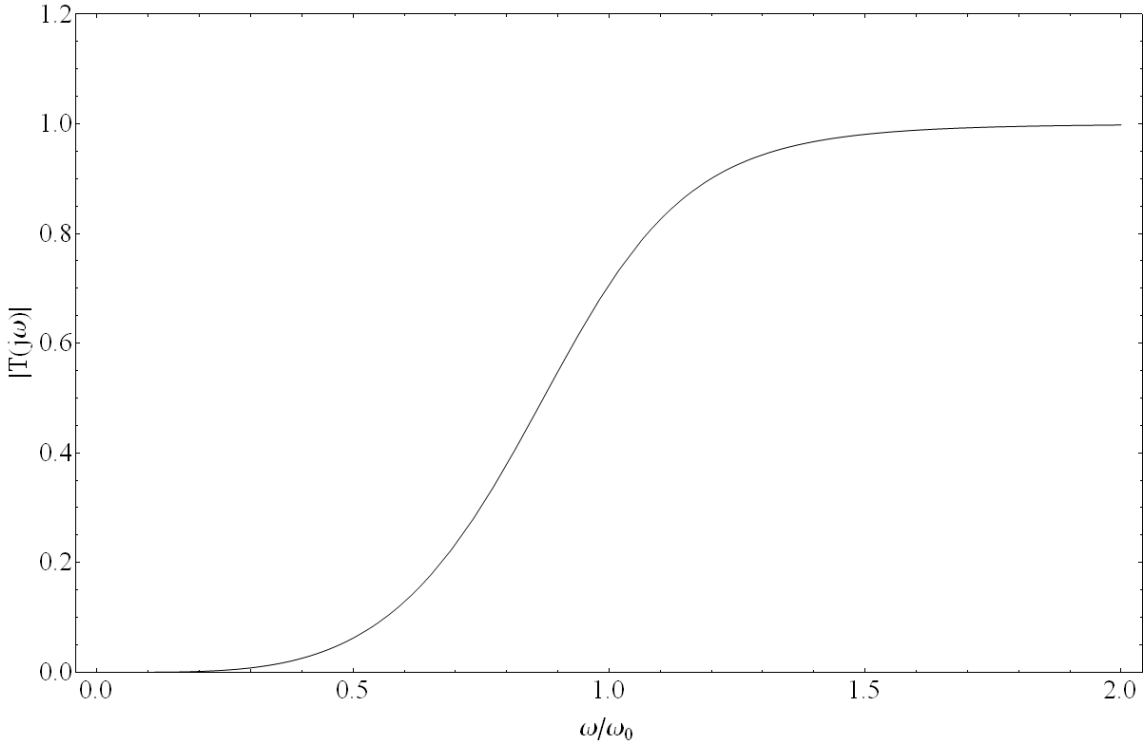


Figura 2.4. Transferencia típica de un filtro pasaaltos.

Combinando las características de los filtros pasabajos y de los filtros pasaaltos, pueden crearse los denominados filtros *pasabanda* (*bandpass filters*) y filtros *supresores de banda* (*bandstop filters*). Los filtros pasabanda, como su nombre lo indica, dejan pasar una banda dada de frecuencias (ver figura 2.5). Por otro lado, los filtros supresores de banda actúan en forma inversa a los filtros pasabanda, bloqueando una banda determinada de frecuencias (ver figura 2.6).

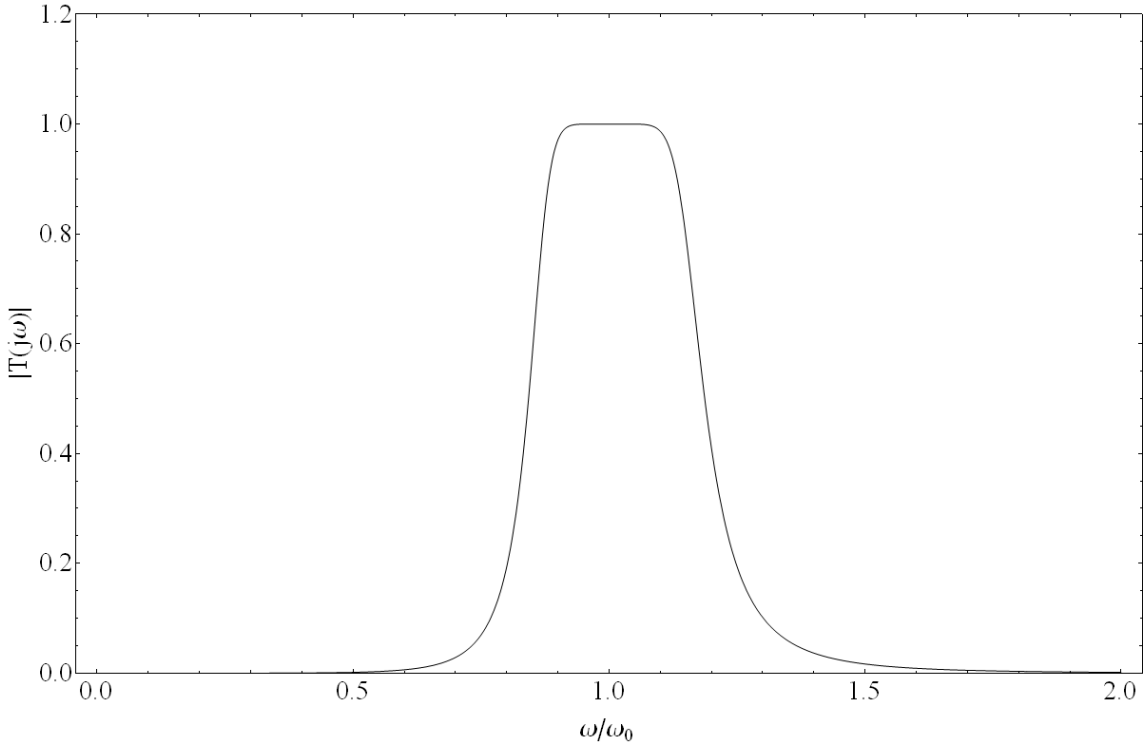


Figura 2.5. Transferencia típica de un filtro pasabanda.

La última clase de filtro que analizaremos son los denominados filtros *pasatodo* (*allpass filters*). A esta clase de filtro, como el módulo de la transferencia es la unidad, se los identifica según su repuesta en fase, o sea según el comportamiento del argumento de la función de transferencia.

De hecho, para obtener una salida que tenga la misma forma que la entrada no alcanza con requerir que el módulo de la función de transferencia sea constante, sino que la respuesta en fase debe ser lineal en la frecuencia ya que

$$\begin{aligned} y(t) &= Ax(t-a) \\ Y(s) &= Ae^{-as}X(s) \\ H(s) &= Ae^{-as} \\ H(j\omega) &= Ae^{-ja\omega} \\ |H(j\omega)| &= |A| \qquad \text{y} \quad \arg\big(H(j\omega)\big) = -a\omega. \end{aligned}$$

Por lo tanto, cualquier función de transferencia que no tenga una respuesta en fase lineal en la frecuencia distorsionará la forma de la salida, aunque mantenga la composición espectral. Pueden

verse más detalles respecto a esta clase de filtros y a las otras clases mencionadas en [Paarmann, 2003].

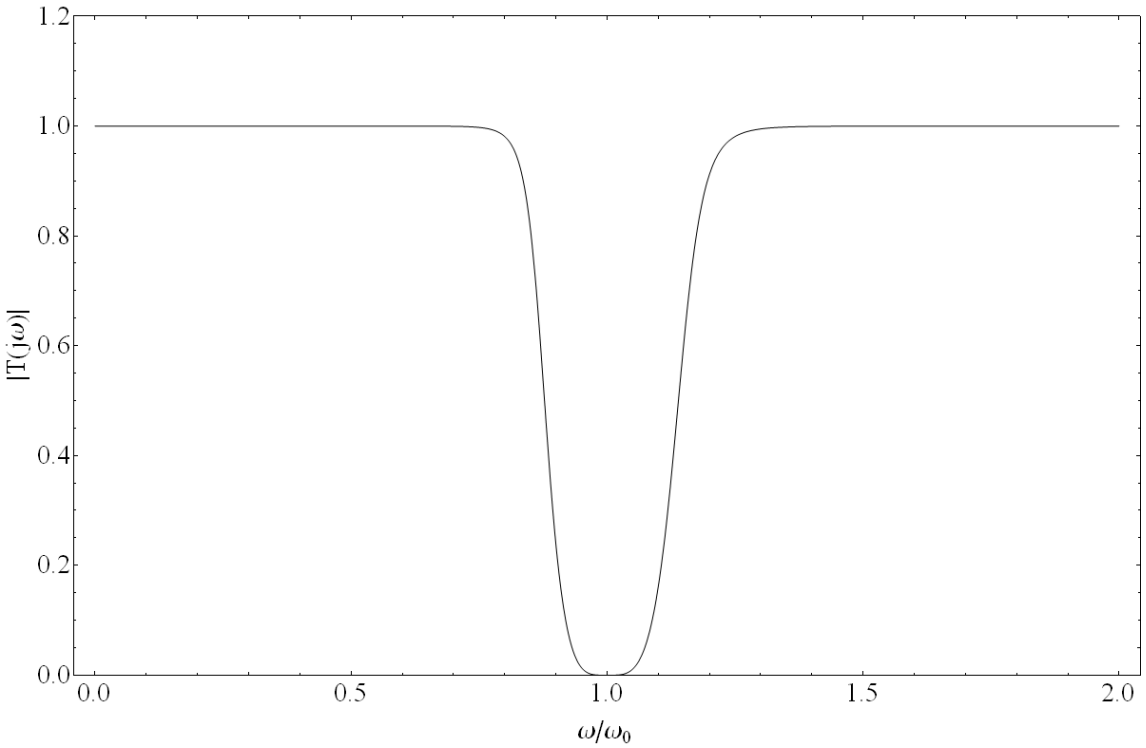


Figura 2.6. Transferencia típica de un filtro supresor de banda.

2.1.1.4. Clasificación según la forma de aproximación

Si se buscara eliminar las componentes de alta frecuencia de una señal podría parecer conveniente emplear un filtro con una transferencia como la que se observa en la figura 2.7. Sin embargo, un resultado conocido como Teorema de Paley-Wiener [Paarmann, 2003] impide bajo condiciones muy generales realizar esta clase de filtros.

Este teorema se basa en requerir que el filtro sea causal, o sea en demandar que la salida del filtro no dependa del comportamiento "futuro" de la entrada. Esto es obviamente necesario en los casos en que el procesamiento no se efectúe *offline*, como suele ser el caso con los filtros analógicos. El teorema establece que, dada una función de transferencia tal que el cuadrado de su módulo es integrable a lo largo del eje imaginario,

$$\int_{\Re} |H(j\omega)|^2 \, d\omega < +\infty,$$

es condición necesaria y suficiente para que sea realizable el exigir que se satisfaga la siguiente desigualdad:

$$\int_{\Re} \frac{\left|\ln\left|H(j\omega)\right|\right|}{1+\omega^2} d\omega < +\infty.$$

Esta desigualdad claramente no puede cumplirse en los casos en que haya segmentos del eje imaginario en los que *H(s)* sea nula ya que, si el numerador diverge en un segmento, puede verse que es imposible que la integral converja. Por ende, ninguna de las versiones ideales de los filtros mencionadas en la sección anterior es realizable. Pasa a ser necesario buscar como aproximarlos mediante transferencias realizables en la práctica.

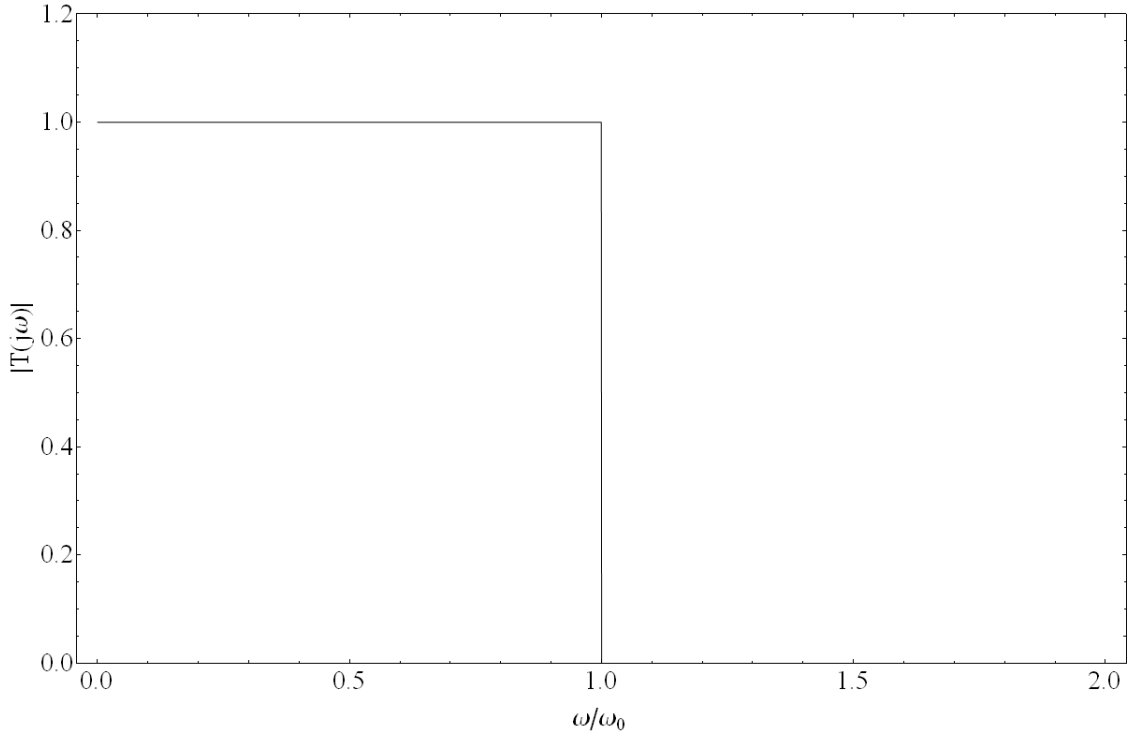


Figura 2.7. Transferencia de un filtro pasabajos ideal.

Como se verá en la sección 2.1.1.5, las funciones de transferencia realizables en los filtros analógicos de los que se ocupa este trabajo son funciones racionales. En general, cuanto mayor es el orden de la función racional utilizada para aproximar la transferencia, mayor será la precisión de la aproximación. Sin embargo, existen distintos métodos de aproximación que optimizan distintos aspectos de la transferencia para un orden fijo dado. Analizaremos cualitativamente tres de estos métodos que son los más aplicados en la práctica: Butterworth, Chebyshev y Cauer. Un análisis más riguroso de los mismos así como de otros métodos de aproximación escapa al objetivo del presente trabajo, aunque puede encontrarse en [Paarmann, 2003].

La aproximación Butterworth busca obtener una respuesta lo más plana posible en la banda pasante, evitando las oscilaciones características de otras aproximaciones. Como contrapartida, la transición

que presenta entre la banda pasante y la banda bloqueada es más gradual que en las otras aproximaciones.

La aproximación Cauer, también conocida como elíptica, no se preocupa por obtener una respuesta plana en frecuencias sino que admite oscilaciones tanto en la banda pasante como en la bloqueada. Pero, para un orden fijo y un valor dado de dichas oscilaciones, presenta la transición más rápida posible entre ambas bandas.

Las aproximaciones Chebyshev pueden verse como un compromiso entre las dos aproximaciones anteriores, presentando oscilaciones en solo una de las bandas junto con una transición más rápida que la de los filtros Butterworth. Los filtros Chebyshev tipo I exhiben oscilaciones en la banda pasante, mientras que los filtros Chebyshev tipo II tienen oscilaciones en la banda bloqueada.

Estas aproximaciones pueden observarse comparadas entre si en la figura 2.8.

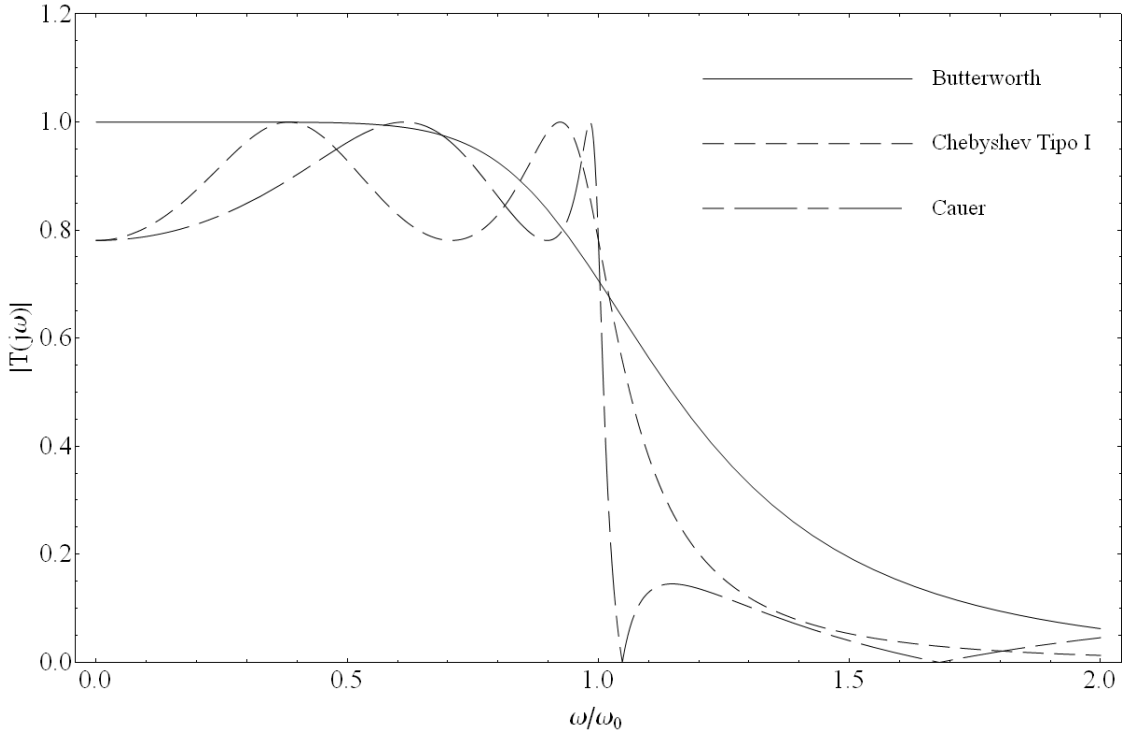


Figura 2.8. Comparación entre las aproximaciones de orden 4 efectuadas con los métodos Butterworth, Chebyshev y Cauer. Observar que la transición más abrupta corresponde al filtro Cauer, seguido por el Chebyshev.

2.1.1.5. Clasificación según los componentes utilizados y sus efectos en la realización

Hasta este momento se han tratado los filtros como entidades abstractas, sin preocuparse de su implementación física más allá de asumir que respetan la causalidad. Como este trabajo se limitará a tratar con filtros eléctricos, esto introduce la posibilidad de una clasificación adicional de acuerdo a la construcción de estos filtros.

Todos los filtros que se tratan en este trabajo operan en el régimen de bajas frecuencias, donde la longitud de onda de las señales es mucho mayor que las dimensiones del circuito. En este caso puede aplicarse el modelo de parámetros concentrados [Agarwal & Lang, 2005], en donde la propiedades eléctricas se consideran concentradas en un conjunto discreto de elementos denominados componentes. La construcción será clasificada entonces de acuerdo a la clase de componentes presentes en el circuito.

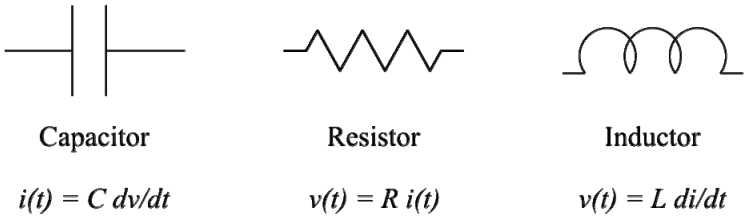


Figura 2.9. Componentes pasivos con sus relaciones constitutivas.

Para este trabajo se usa la definición de pasividad dada en [Paarmann, 2003], es decir se consideran componentes pasivos solamente a los resistores, capacitores e inductores (ver figura 2.9). Por otro lado, transistores y amplificadores operacionales serían ejemplos de componentes considerados como activos (ver figura 2.10). Un filtro será denominado como *pasivo* si está *íntegramente* constituido por componentes pasivos. En caso contrario, será denominado como *activo*.

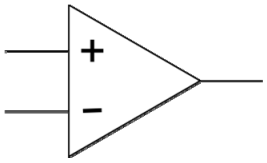


Figura 2.10. El "componente" activo *lineal* más utilizado, el amplificador operacional. En la práctica se construye utilizando transistores y componentes pasivos.

Al ser todos los componentes de un filtro pasivo elementos de dos terminales, un circuito puede verse entonces como un grafo dirigido (ver figura 2.11). En este, las aristas representan a los componentes (con la dirección de la arista indicando la polaridad del mismo) y los vértices las conexiones de los mismos. Si se aplican las leyes de Kirchhoff en los nodos y ciclos independientes del grafo y las relaciones constitutivas, se llega a un sistema de ecuaciones que describe el comportamiento del filtro [Wellstead, 2000].

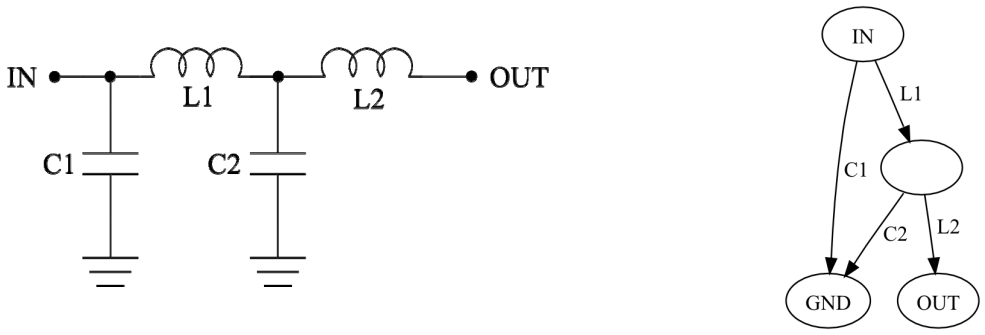
Al ser ecuaciones diferenciales las relaciones constitutivas de cada uno de los componentes, en general el sistema asociado al filtro será un sistema de ecuaciones diferenciales ordinarias. Pero trabajando en el dominio de la frecuencia compleja, las relaciones constitutivas pasan a ser algebraicas:

$$\text{Resistor}: \quad v(t) = Ri(t) \rightarrow V(s) = RI(s)$$

$$\text{Capacitor}: \quad i(t) = C\frac{dv(t)}{dt} \rightarrow I(s) = (sC)V(s)$$

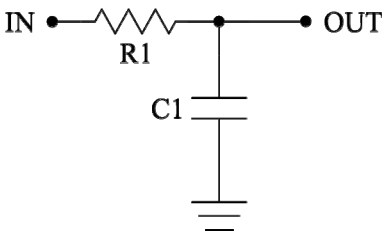
$$\text{Inductor}: \quad v(t) = L\frac{di(t)}{dt} \rightarrow V(s) = (sL)I(s).$$

Por lo tanto el problema se reducirá a la resolución de un sistema en el que tanto los coeficientes como las variables toman valores dentro del conjunto de las funciones racionales de *s*. Como puede verse mediante la aplicación de la regla de Cramer, las variables en cualquier punto serán una función racional de los coeficientes y de los términos independientes. Si hacemos la entrada *X(s)* igual a 1, la salida *Y(s)*, que será igual a la transferencia del filtro *H(s)*, será una función racional de funciones racionales de *s* y, por consiguiente, una función racional de *s* [Paarmann, 2003] (puede verse un ejemplo muy simple en la figura 2.12).



(a) **(b)**

Figura 2.11. Ejemplo de filtro analógico pasivo **(a)** y su grafo asociado **(b)**.



$$I_{R1} + I_{C1} = 0$$

$$\frac{1}{R_1}(V_{OUT}(s) - V_{IN}(s)) + (sC_1)V_{OUT}(s) = 0$$

$$\frac{1}{R_1}(T(s) - 1) + sC_1T(s) = 0$$

$$T(s) - 1 + sR_1C_1T(s) = 0$$

$$T(s) + sR_1C_1T(s) = 1$$

$$T(s) = \frac{1}{1 + sR_1C_1}$$

Figura 2.12. Filtro pasabajos de 1er orden y la determinación de su transferencia.

2.1.2. CARACTERÍSTICAS PROPIAS DE LOS FILTROS ANALÓGICOS PASIVOS

Los filtros pasivos tienen una serie de características que les otorgan distintas ventajas y desventajas, cuya relevancia dependerá de la aplicación específica para la que se consideren. Las ventajas más apreciables de los filtros pasivos son:

- no requieren una fuente de energía adicional;
- son compactos en el régimen de frecuencias relativamente elevadas;
- muy robustos;
- bajo costo, sobre todo a frecuencias de trabajo elevadas, donde los componentes activos se tornan más costosos;
- en caso de estar constituidos solo por elementos reactivos (capacitores e inductores), no disipan energía.

Las desventajas principales de los filtros pasivos son:

- para el régimen de bajas frecuencias requieren valores irrazonablemente elevados de los componentes;
- presentan una impedancia de salida relativamente elevada y una impedancia de entrada relativamente baja;
- como consecuencia de lo anterior, no pueden combinarse en múltiples etapas, tal como es posible con los filtros activos;
- al utilizar inductores, son especialmente vulnerables a las interferencias electromagnéticas.

Debido a que los filtros pasivos tienen una impedancia de salida relativamente alta, debe considerarse la carga que se le va a aplicar al filtro antes de diseñar la implementación. De forma inversa, debido a que la impedancia de entrada puede ser relativamente baja, es imperativo considerar la impedancia de salida de la fuente de señal. Otra forma de ver estas restricciones consiste en apreciar que toda la energía de la señal que se obtenga a la salida deberá provenir de la entrada, por lo cual existe una relación directa entre la impedancia de entrada y la impedancia de salida que es posible conseguir.

2.1.3. PROCEDIMIENTOS MANUALES DE DISEÑO

Inicialmente la gran mayoría de los filtros utilizados eran filtros analógicos pasivos, por el elevado costo e inconveniencia de uso de los componentes activos disponibles en aquel entonces (válvulas electrónicas). La ausencia de computadoras requería también el desarrollo de métodos manuales relativamente simples de ejecutar para poder efectuar el proceso de diseño en forma efectiva.

Estos métodos asumen una topología determinada, en general la topología *ladder* (así denominada por su semejanza con una escalera, tal como puede verse en la figura 2.13 si se imaginan las tierras unidas por un conductor), y emplean un conjunto de tablas y cálculos simples para determinar los valores requeridos de los componentes. En el proceso 2.1 se describen en forma exhaustiva los pasos que deben llevarse a cabo para diseñar un filtro manualmente [Paarmann, 2003].

Para clarificar la ejecución de este procedimiento de diseño, se mostrará un ejemplo : se realizará el diseño manual de un filtro pasaaltos Butterworth con frecuencia de corte 10 kHz, orden 4 y con impedancias de fuente y de carga iguales a 50 Ω.

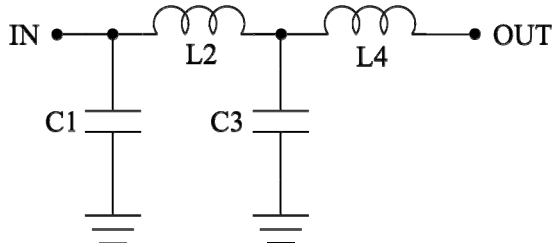


Figura 2.13. Ejemplo de filtro pasabajos *ladder* de orden 4.

Por motivos de practicidad, las tablas indican solo los valores de componentes para construir un filtro pasabajos con ciertos valores estándar, dejando al usuario las modificaciones que deban realizarse. En este caso, por tratarse de un filtro de orden 4, debe elegirse la fila con N = 4 de la tabla 2.1.

N	C1	L2	C3	L4	C5
2	1.4142	1.4142			
3	1.0000	2.0000	1.0000		
4	0.7654	1.8478	1.8478	0.7654	
5	0.6180	1.6180	2.0000	1.6180	0.6180

Tabla 2.1. Valores de componentes (en farads y henries) para construir un filtro pasabajos Butterworth con topología *ladder*, ω_{corte} = 1 (rad)/s e impedancias de fuente y carga iguales a 1 Ω (adaptado de [Paarmann, 2003]). Se resalta la fila utilizada en el ejemplo.

PROCESO: Diseño manual de un filtro ladder pasivo

ENTRADAS: Tipo de transferencia buscada, frecuencia de corte, ancho de banda (si el tipo de transferencia buscado es pasabanda o supresor de banda), orden de aproximación, forma de aproximación, impedancia de carga, parámetros adicionales de la aproximación

SALIDA: Valores de los componentes del filtro pasivo

1. Elegir la tabla adecuada en base a la *forma de aproximación* y a los *parámetros adicionales de la aproximación*.
2. Si el *tipo de transferencia* es *pasabanda* o *supresor de banda*:
   a. $n \leftarrow$ *orden de aproximación* / 2 (el orden debe ser par)

   En caso contrario:

   a. $n \leftarrow$ *orden de aproximación*
3. Tomar del renglón de la tabla correspondiente a n los valores de los componentes:
   L_i y C_j (i toma valores en un conjunto disjunto a el de los valores de j).
4. $k_f \leftarrow$ *frecuencia de corte* * 2 * π
5. $B \leftarrow$ *ancho de banda*
6. Si el *tipo de transferencia buscada* es *pasaaltos*:
   a. $C_i' \leftarrow 1 / (k_f * L_i)$
   b. $L_j' \leftarrow 1 / (k_f * C_j)$
7. Si el *tipo de transferencia buscada* es *pasabajos*:
   a. $C_j' \leftarrow C_j / k_f$
   b. $L_i' \leftarrow L_i / k_f$
8. Si el *tipo de transferencia buscada* es *pasabanda*:
   a. $C_i' \leftarrow B / (k_f^2 * L_i)$
   b. $L_i' \leftarrow L_i / B$
   c. $C_j' \leftarrow C_j / B$
   d. $L_j' \leftarrow B / (k_f^2 * C_j)$
9. Si el *tipo de transferencia buscada* es *supresor de banda*:
   a. $L_i' \leftarrow (B * L_i) / k_f^2$
   b. $C_i' \leftarrow 1 / (B * L_i)$
   c. $C_j' \leftarrow (B * C_j) / k_f^2$
   d. $L_j' \leftarrow 1 / (B * C_j)$
10. Si la impedancia de carga es distinta a 1 Ω:
    a. $k_{imp} \leftarrow$ *impedancia de carga* (supuesta puramente resistiva)
    b. $C_i'' \leftarrow C_i' / k_{imp}$
    c. $C_j'' \leftarrow C_j' / k_{imp}$
    d. $L_i'' \leftarrow L_i' * k_{imp}$
    e. $L_j'' \leftarrow L_j' * k_{imp}$
11. Devolver L_i'', L_j'', C_i'' y C_j'' como salida.

Proceso 2.1. Procedimiento manual para diseñar un filtro *ladder*. El procedimiento asume tablas construidas para un filtro pasabajos con frecuencia angular de corte 1 (rad)/s e impedancia de carga 1 Ω. Las conexiones entre componentes deben determinarse de acuerdo a un modelo de circuito adjunto a las tablas.

Como no se desea construir un filtro pasabajos con ω_{corte} = 1 (rad)/s, sino un pasaaltos con f_{corte} = 1 kHz (equivalente a $\omega_{corte} = 20\pi$ (krad)/s), es necesario aplicar la transformación correspondiente:

$$L_1' = \frac{1}{k_f\, C_1} = \frac{1}{62830 \cdot 0.7654} = 2.0794 \cdot 10^{-5}$$

$$C_2' = \frac{1}{k_f\, L_2} = \frac{1}{62830 \cdot 1.8478} = 8.6135 \cdot 10^{-6}$$

$$L_3' = C_2' = 8.6135 \cdot 10^{-6} \quad \text{(por simetría)}$$

$$C_4' = L_1' = 2.0794 \cdot 10^{-5} \quad \text{(por simetría)}$$

Con esto ya se tendría un filtro pasaaltos con frecuencia de corte 10 kHz. Sin embargo, faltaría adaptar la impedancia del filtro, lo que es de gran importancia en el caso de los filtros pasivos (ver sección 2.1.2).

En este caso el factor de escala de impedancias k_{imp} es igual a 50, por lo tanto:

$$L_1'' = k_i L_1' = 50 \cdot 2.0794 \cdot 10^{-5} = 1.0397 \cdot 10^{-3}$$

$$C_2'' = \frac{C_2'}{k_i} = \frac{8.6135 \cdot 10^{-6}}{50} = 1.7227 \cdot 10^{-7}$$

$$L_3'' = k_i L_3' = 50 \cdot 8.6135 \cdot 10^{-6} = 4.3068 \cdot 10^{-4}$$

$$C_4'' = \frac{C_4'}{k_i} = \frac{2.0794 \cdot 10^{-5}}{50} = 4.1588 \cdot 10^{-7}$$

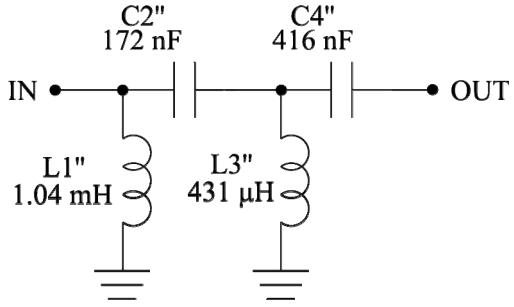


Figura 2.14. Resultado del proceso de diseño manual.

En la figura 2.14 puede verse el esquema resultante y en la figura 2.15, la comparación de la transferencia simulada utilizando el simulador SPICE [Kielkowski, 1998] con la aproximación correspondiente. Se observa que la transferencia tiene un máximo de 0.5; debido a la presencia de una impedancia de salida en la fuente igual a la impedancia de carga, el caso límite corresponde a iguales caídas de tensión en ambas impedancias. El conjunto de datos proveniente de la simulación es discreto, ya que el análisis se realiza para valores particulares de la frecuencia de entrada.

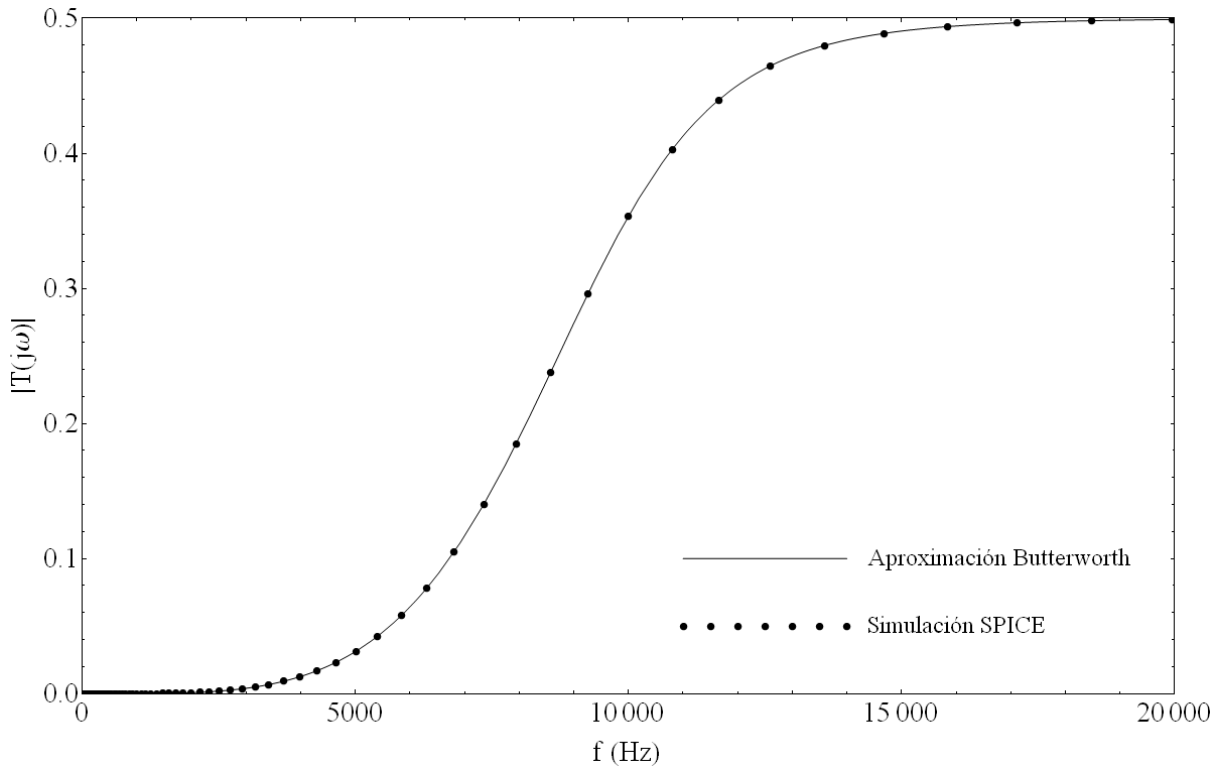


Figura 2.15. Transferencia simulada mediante SPICE del filtro que se muestra en la figura 2.14 comparado con la aproximación Butterworth de cuarto orden.

Aunque la disponibilidad de herramientas automatizadas ha disminuido en gran medida la utilización de estos procedimientos en la práctica, estos siguen utilizándose en la enseñanza por su carácter pedagógico [Koller & Wilamowski, 1996].

2.1.4. HERRAMIENTAS AUTOMATIZADAS NO BASADAS EN LA PROGRAMACIÓN GENÉTICA

La gran mayoría de las herramientas convencionales son desarrollos más o menos directos de los procedimientos "históricos" [Szentirmai, 1977; Koller & Wilamowski, 1996; Nuhertz, 2008] (ver figuras 2.16 y 2.17). Las entradas son el orden de aproximación, la forma de aproximación y la topología a utilizar (donde "ladder" es una de las opciones). La salida es el diseño completo del circuito, incluyendo los valores de los componentes. Una contribución de esta clase de herramientas es que permiten realizar un análisis de sensibilidad respecto a variaciones en los valores de los componentes, ayudando a detectar diseños no satisfactorios, pero sus limitaciones respecto a la incapacidad de generar nuevas topologías persisten incluso en sistemas sofisticados [Koza *et al.*, 1997].

```
IS THIS A COMPLETE(C) OR A PARTIAL(P) REMOVAL ? P
ENTER TWO INDICATORS OF POLES TO BE REMOVED: (Z),(I) OR (=)
THEY MAY NOT BE THE SAME AND EXTREME ONE ENTERED FIRST ? Z I
ENTER #'S OF FINITE TRANSMISSION ZEROS YOU WANT TO CREATE ? 1 2
 1.6841394E+01      .        L       1.3401956E+00
                    .                .
 8.9298823E-02      .        C       1.9739374E-08
                    .                .
WISH TO SEE RESULTS SO FAR? (Y/N) ? N
ENTER 'IMP' FOR SERIES BRANCH, 'ADM' FOR SHUNT BR.
# OF STEPS TO BE SCRATCHED FOR BACKUP, OR 'END' ? ADM
WISH TO SEE POLE-ZERO PATTERN? (Y/N) ? Y

PATTERN OF CUT-OFFS,IMMITTANCE SINGULARITIES AND REMAINING TRANSMISSION ZEROS
 1      I            I      1
 0  X       0  X  0      X  0
    1                   2
SINGLE(S) OR DOUBLE(D) POLE REMOVAL NEXT ? S
COMPLETE(C), PARTIAL(P) OR VALUE(V) REMOVAL ? C
IS POLE AT ZERO(Z), INFINITE(I) OR FINITE FREQUENCY(=) ? 1
 1.5530754E+01      ...L.C...       1.2358981E+00      ZERO FREQUENCY
 1.7051543E-01      .        .      3.7692185E-08       7.3740000E+02
                    .                .
WISH TO SEE RESULTS SO FAR? (Y/N) ? N
```

Figura 2.16. Imagen de FILSYN, uno de los primeros sistemas prácticos de diseño de filtros analógicos [Szentirmai, 1977].

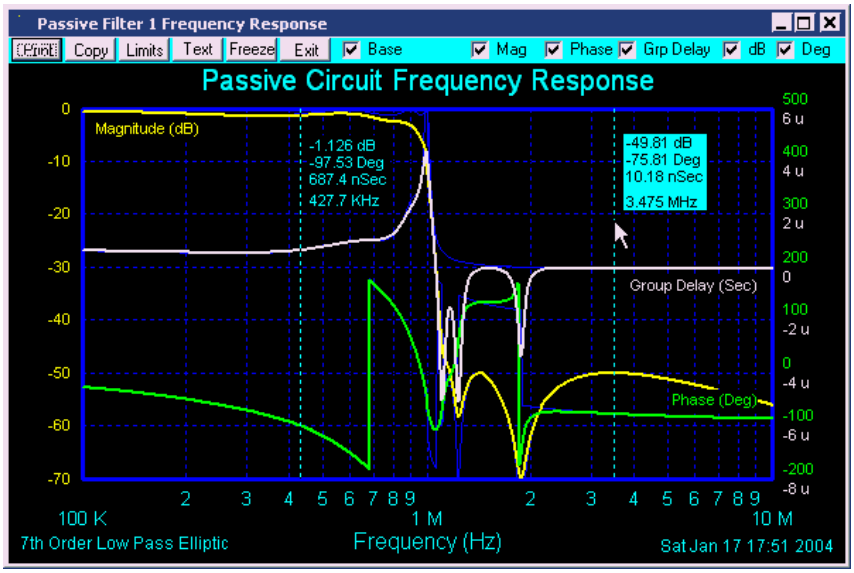


Figura 2.17. Captura de pantalla de una herramienta moderna para el diseño de filtros analógicos [Nuhertz, 2008].

Se han hecho numerosos esfuerzos para aplicar técnicas desarrolladas en el área de la inteligencia artificial al diseño de filtros analógicos. Uno de los primeros trabajos de esta clase, utilizando propagación de restricciones, fue realizado a mediados de la década de 1970 [Sussman & Stallman, 1975], aunque más orientado al análisis de circuitos que a su síntesis. Posteriormente se desarrollaron algunas herramientas tales como OASYS [Harjani, 1989] que trataba un diseño de circuito como una composición jerárquica de bloques funcionales abstractos con conocimiento asociado a cada uno de ellos. Esto le permitía buscar la forma de cumplir con una serie de requerimientos funcionales a través de la descomposición de los mismos en base al conocimiento del sistema en tareas capaces de ser cubiertas por los bloques disponibles.

El uso de algoritmos genéticos se limitó en general a la optimización de un conjunto de parámetros dentro de topologías preexistentes [Koza *et al.*, 1997], ya que el formato fijo de los cromosomas empleados impide la generación de topologías arbitrarias a partir de los mismos.

2.2. PROGRAMACIÓN GENÉTICA

En esta sección se describe a la programación genética: se comienza explicando la definición de esta técnica (sección 2.2.1), incluyendo una descripción resumida de los algoritmos genéticos en los que esta técnica se basa (sección 2.2.1.1) y una de la programación genética en sí (sección 2.2.1.2); se continúa con las estructuras de los programas sobre los que trabaja la programación genética (sección 2.2.2), incluyendo las estructuras en forma de árbol (sección 2.2.2.1), las lineales (sección 2.2.2.2) y las estructuradas como grafos más generales (sección 2.2.2.3); por último se concluye con una sinopsis de las aplicaciones de la programación genética (sección 2.2.3), exceptuando su aplicación al diseño de filtros analógicos, tratado en la sección siguiente.

2.2.1. DEFINICIÓN Y GENERALIDADES

En esta sección se analiza como se define a la programación genética y algunos aspectos generales de la misma: este análisis comienza con una breve reseña de los algoritmos genéticos (sección 2.2.1), que constituyen la base histórica de la programación genética; se continúa luego con un rápido estudio, tanto en términos de su definición como históricos, de la programación genética en si (sección 2.2.1.2)

2.2.1.1. Algoritmos genéticos

Históricamente, los algoritmos genéticos fueron la primera técnica evolutiva utilizada. Esta técnica fue creada por John Holland y descrita por él mismo en su libro *Adaptation in Natural and Artificial Systems* [Holland, 1992]. Un algoritmo genético [Koza, 1995] transforma una población integrada por un cierto número de individuos, cada uno de los cuales tienen un cierto valor de adaptación, mediante la utilización de análogos a las operaciones naturales de cruza y mutación.

Los individuos son en este caso análogos a lo que se denomina el genotipo de un individuo en biología y representan posibles soluciones al problema. El efecto indirecto que tiene normalmente el genotipo del individuo sobre su constitución (fenotipo) se reemplaza en esta técnica por el cálculo del valor de adaptación de los individuos.

Para representar el efecto que tiene el proceso de selección natural sobre una población, se emplean un gran número de distintas técnicas [Mitchell, 1998] de selección de población. Sin embargo todas tienen en común el seleccionar preferencialmente a los individuos con una mejor adaptación, diferenciando a un algoritmo genético de una simple búsqueda aleatoria [Spall, 2003; Banzhaf *et*

al., 1998]. También existen distintas formas de generar la población inicial y de manejar la selección de individuos a los que aplicarle las operaciones de cruza y mutación, pero no afectan la clasificación de una técnica como algoritmo genético.

PROCESO:	Algoritmo genético
ENTRADAS:	Criterio de optimalidad, función de evaluación (*Evaluar*()), criterio de detención, operadores genéticos con sus probabilidades asociadas.
SALIDA:	La "mejor" solución obtenida, de acuerdo al *criterio de optimalidad*.

1. $G \leftarrow 0$
2. $P[G] \leftarrow$ Población generada aleatoriamente (hay otras posibilidades).
3. $N \leftarrow Tamaño(P[G])$
4. $S \leftarrow Evaluar(P[G])$
5. Mientras no se cumpla el *criterio de detención*:
   a. $G \leftarrow G + 1$
   b. Repetir N veces:
      1. Seleccionar un *operador genético* $OG()$ en forma aleatoria.
      2. Seleccionar individuos I_j e I_k de acuerdo al vector de puntajes S.
      3. Agregar $OG(I_j, I_k)$ a $P[G]$.
   c. $S \leftarrow Evaluar(P[G])$
6. Devolver la solución correspondiente al "mejor" individuo, de acuerdo al *criterio de optimalidad*.

Proceso 2.2. Descripción del funcionamiento de un algoritmo genético.

Lo que se ha mencionado hasta el momento es aplicable tanto para los algoritmos genéticos como para la técnica de programación genética que veremos en la próxima sección. Lo que diferencia a los algoritmos genéticos es la utilización de una asignación fija de las distintas partes del cromosoma (genes) a características del fenotipo. El esquema de representación es quien indica como se realiza esta asignación.

2.2.1.2. Programación genética

La representación de los datos es clave en los algoritmos genéticos, ya que manejan directamente una representación codificada de las soluciones factibles al problema. El esquema de representación representa la "ventana" mediante la que el sistema mira al problema y por lo tanto puede limitar severamente el alcance del mismo. Aunque la representación de un problema mediante cadenas de longitud fija permite resolver una amplia variedad de problemas, tiene importantes limitaciones.

Una de estas es la relación directa entre el tamaño de las cadenas utilizadas y el tamaño de la solución factible representada. Otra es la relativa rigidez de los sistemas de codificación utilizables en algoritmos genéticos.

La programación genética busca eliminar estos problemas a expensas de eliminar la relación directa entre las partes del genoma y las partes de la solución. En lugar de ello se utiliza una representación indirecta donde las cadenas pasan a ser de longitud variable e interpretarse como programas. El efecto de cada instrucción consiste en alterar la forma de la solución o cambiar el estado del sistema encargado de interpretar a los individuos. Esto permite una representación mucho más compacta, donde la longitud de la cadena que representa un individuo puede acercarse a la complejidad de Kolmogorov del mismo [Li & Vitányi, 1997].

Puede verse una cierta similitud entre esta codificación y el funcionamiento de la trascripción del ADN en una célula normal. Si bien la trascripción desde la secuencia de bases en el ADN a la secuencia de aminoácidos en las proteínas es relativamente directa, la complejidad e indirección de la acción de éstas para formar un individuo es similar a la forma en que se produce la interpretación de un programa en la programación genética [Dawkins, 1986].

Puede observarse que la descripción del funcionamiento de la programación genética (ver proceso 2.3) es muy similar a la de los algoritmos genéticos (ver proceso 2.2). La diferencia esencial radica en el proceso de evaluación: mientras los algoritmos genéticos evalúan directamente el genotipo del individuo, la programación genética ejecuta al genotipo como un programa tomando el resultado como el fenotipo a evaluar. La frontera exacta no es clara, ya que podría tomarse a la ejecución de un programa como una forma sofisticada de evaluación [McCarthy, 1960], pero es esta la característica distintiva de la programación genética.

También existen diferencias menores dadas por el hecho de que los programas tienen en general estructuras complejas y de tamaño variable. Esto fuerza el desarrollo de operadores genéticos relativamente complejos ya que, en caso contrario, habría un número excesivo de individuos incapaces de ser ejecutados.

Desde los comienzos mismos del uso de las computadoras se ha pensado en automatizar el proceso mismo de programación, por lo que en general se entiende expresar los problemas en un lenguaje de mayor nivel de abstracción que los existentes [Novak, 2005]. Limitándose a la aplicación de operadores genéticos al desarrollo de programas, puede señalarse a [Smith, 1980] como el primer trabajo donde se realizó un desarrollo evolutivo de *programas*, en lugar de solo parámetros de los mismos. Sin embargo, el origen de un interés más amplio en la programación genética estuvo dado por la publicación por parte de John Koza de varios trabajos y un libro describiendo este paradigma [Koza, 1989; Koza, 1990; Koza, 1992].

PROCESO: Programación genética

ENTRADAS: Criterio de optimalidad, función de evaluación (*Evaluar*()) aplicables sobre los fenotipos, criterio de detención, operadores genéticos con sus probabilidades asociadas.

SALIDA: La "mejor" solución obtenida, de acuerdo al *criterio de optimalidad*.

1. $G \leftarrow 0$
2. $P[G] \leftarrow$ Población generada aleatoriamente (hay otras posibilidades).
3. $N \leftarrow Tamaño(P[G])$
4. Para cada individuo I_j en $P[G]$:
   a. $F \leftarrow Ejecutar(I_j)$
   b. $S_j \leftarrow Evaluar(F)$
   c. Agregar S_j al vector de puntajes S.
5. Mientras no se cumpla el *criterio de detención*:
   a. $G \leftarrow G + 1$
   b. Repetir N veces:
      1. Seleccionar un *operador genético* $OG()$ en forma aleatoria.
      2. Seleccionar individuos I_j e I_k de acuerdo al vector de puntajes S.
      3. Agregar $OG(I_j, I_k)$ a $P[G]$.
   a. Para cada individuo I_j en $P[G]$:
      1. $F \leftarrow Ejecutar(I_j)$
      2. $S_j \leftarrow Evaluar(F)$
      3. Agregar S_j al vector de puntajes S.
6. Devolver la solución correspondiente al "mejor" individuo, de acuerdo al *criterio de optimalidad*.

Proceso 2.3. Descripción del funcionamiento general de la programación genética.

En los trabajos mencionados, Koza aplicó la programación genética a la resolución de numerosos problemas, entre ellos:

- Control óptimo de la posición de un carro.
- Creación de un plan de acción para controlar el movimiento de una "hormiga artificial".
- Regresión simbólica.
- Síntesis de un multiplexor digital.

No obstante esta demostración de la versatilidad de la programación genética, la relativamente elevada demanda de recursos computacionales restringió su aplicabilidad a problemas sencillos. No fue sino hasta fines de la década de 1990 que comenzó a aplicarse la programación genética a un

número más amplio de problemas reales. Se darán más detalles acerca de las aplicaciones históricas y actuales de la programación genética en la sección 2.2.3.

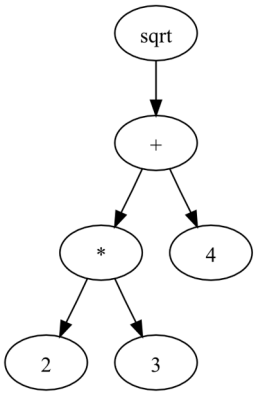
2.2.2. ESTRUCTURAS DE LOS PROGRAMAS

En esta sección se muestran las representaciones utilizadas para los genotipos en la programación genética: se comienza por la representación más utilizada, la estructurada en forma de árbol (sección 2.2.2.1); se continúa con la representación lineal (sección 2.2.2.2) y se finaliza con un tratamiento breve de las representaciones estructuradas en forma de grafos más generales (sección 2.2.2.3).

2.2.2.1. Estructura en forma de árbol

Para poder almacenar un programa es necesario elegir una forma de representarlo. La forma más comúnmente elegida [Brameier, 2004] es la representación en forma de árbol, inspirada en la utilizada por el lenguaje de programación Lisp y empleada por John Koza en algunos de los trabajos pioneros del área [Koza, 1989; Koza, 1992].

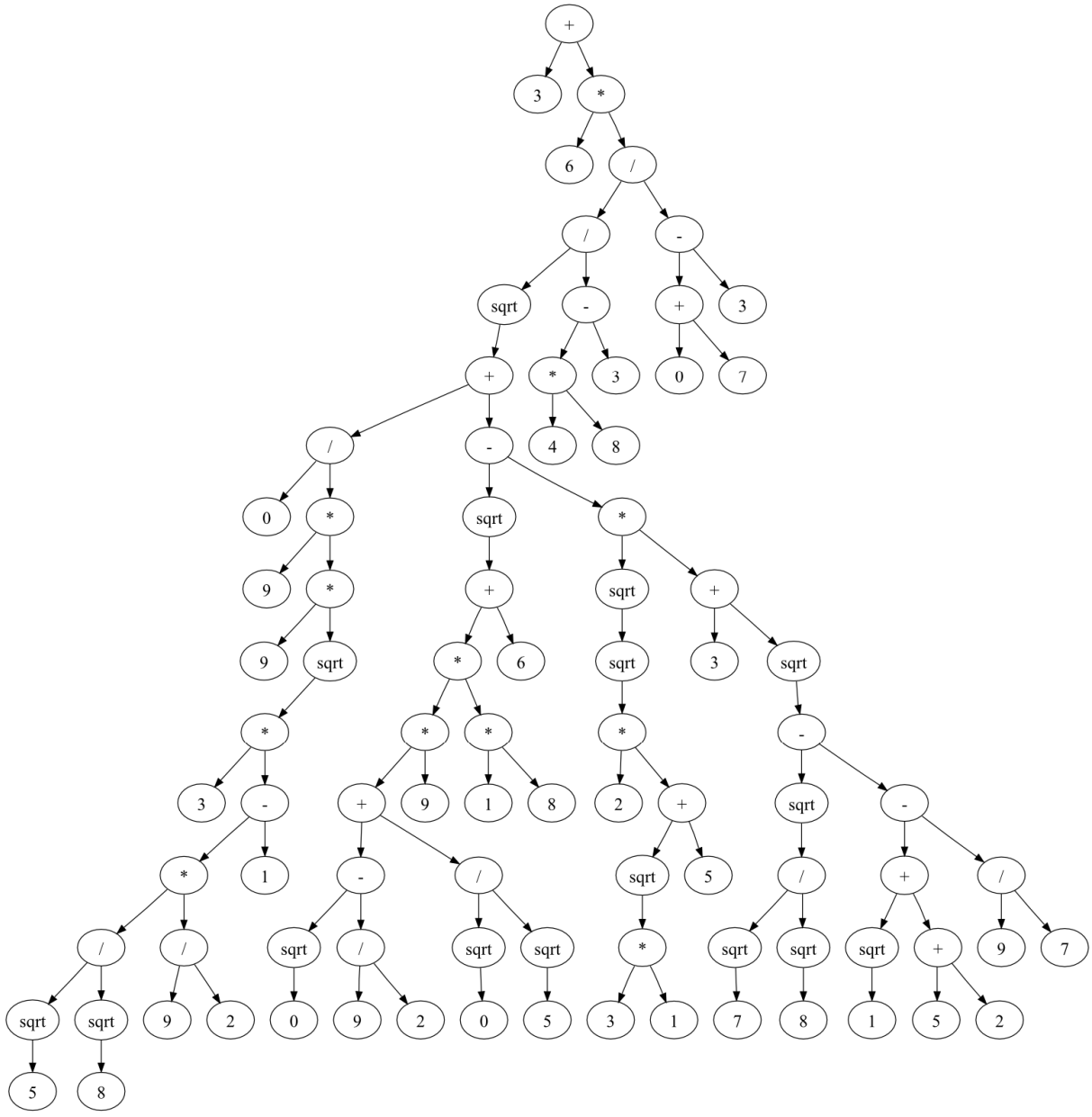
En esta representación, un programa se almacena como un árbol en el que los nodos terminales representan valores o funciones sin argumentos, mientras que los nodos internos representan funciones que toman tantos argumentos como descendientes tienen. De esta forma, la ejecución se realiza en forma recursiva desde el nodo raíz, recibiendo cada nodo los valores producto de la ejecución de los subárboles descendientes, procesándolos con la función que corresponda y devolviendo el resultado a su nodo padre. En la figura 2.18 puede verse como se representaría una expresión aritmética en forma de árbol y la *s-expression* [Rivest, 1997] asociada a esa representación.




```
(sqrt (+ (* 2 3) 4))
```

Figura 2.18. Representación de una expresión aritmética en forma de árbol junto a la *s-expression* asociada.

En los casos en que los operadores tengan una cantidad de argumentos definida es posible evitar representar los árboles utilizando punteros y emplear una representación prefija. Esto permite un uso más eficiente de la memoria y una mayor velocidad de ejecución ya que, al mantenerse los datos en un conjunto compacto, es más "amigable" con los cachés. Un ejemplo de esta representación aplicada sobre una expresión más compleja puede verse en la figura 2.19.



```
+ 3 * 6 / / sqrt + / 0 * 9 * 9 sqrt * 3 - * / sqrt 5 sqrt 8 / 9 2 1 - sqrt + * *
+ - sqrt 0 / 9 2 / sqrt 0 sqrt 5 9 * 1 8 6 * sqrt sqrt * 2 + sqrt * 3 1 5 + 3
sqrt - sqrt / sqrt 7 sqrt 8 - + sqrt 1 + 5 2 / 9 7 - * 4 8 3 - + 0 7 3
```

Figura 2.19. Representación en forma de árbol de una expresión más compleja. Observar que la cantidad fija de argumentos por operador permite un representación prefija libre de ambigüedades, sin requerir el uso de paréntesis (datos obtenidos a partir del ejemplo en [Chouza, 2007]).

En la práctica pueden utilizarse no solo los valores devueltos por los operadores sino que también pueden emplearse efectos secundarios de la ejecución (*side effects*). Esta clase de representación es la que se utiliza en el diseño de filtros analógicos, tal como se verá en la sección 2.3. Es importante tener en cuenta que, para que esta representación esté libre de ambigüedades, es necesario especificar estrictamente el orden de ejecución.

2.2.2.2. Estructura lineal

La gran mayoría de las computadoras existentes ejecutan programas estructurados linealmente, utilizando instrucciones especiales para implementar estructuras de control. Esto, junto a la mejora de rendimiento posible en caso de utilizar instrucciones nativas, motivó el desarrollo de representaciones lineales [Poli *et al.*, 2008] (ver figura 2.20).

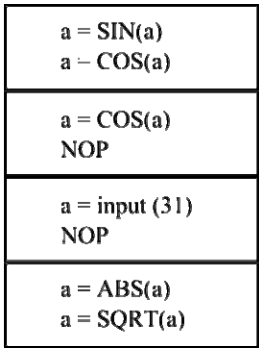


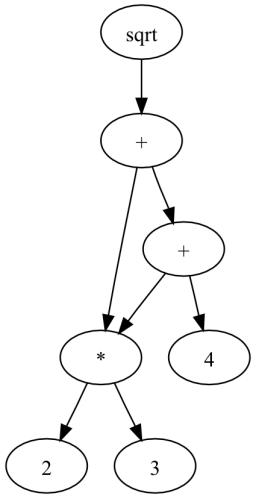
Figura 2.20. Ejemplo de una representación lineal (basado en un ejemplo de [Nordin *et al.*, 1999]). Las instrucciones NOP se utilizan con propósitos de alineación.

Obviamente no es necesario que las instrucciones sean nativas, sino que pueden corresponder a una máquina virtual con un conjunto arbitrario de instrucciones. Una de las mayores dificultades de utilizar instrucciones nativas, además de la inherente falta de portabilidad, radica en que la familia de microprocesadores x86, dominante en el mercado, presenta un formato de instrucciones de notable complejidad. Algunas de sus características, como la longitud variable, pueden ser compensadas mediante la utilización de instrucciones NOP (nulas) con propósitos de alineamiento.

En general, al trabajar con representaciones lineales se suele ignorar la estructura detallada de los individuos sobre los que se opera, en forma similar a como se procede en los algoritmos genéticos. Si bien esto trae a una mayor simplicidad, suele aumentar la proporción de mutaciones y cruzas destructivas respecto a representaciones más elaboradas.

2.2.2.3. Estructura en forma de grafo

Siendo los árboles una clase particular de grafos, es natural evaluar el uso de representaciones mediante el uso de grafos más generales. Si bien, estas aproximaciones no han alcanzado una difusión comparable a las dos anteriores, tienen posibles beneficios.




```
(sqrt (+ (* 2 3) (+ 4 (* 2 3))))
```

Figura 2.21. Ejemplo de representación de una expresión mediante un grafo acíclico no dirigido. Se resalta la subexpresión reutilizada.

Una posible aplicación [Poli *et al.*, 2008] es la reutilización de los resultados de ciertas funciones. De esta forma, un grafo acíclico dirigido como el de la figura 2.21 puede representar la reutilización de la subexpresión que se resalta. La utilización de un caché de expresiones previamente evaluadas disminuye la utilidad de esta aproximación, ya que disminuye los costos de evaluar repetidamente las mismas subexpresiones.

2.2.3. APLICACIONES

Como todas las técnicas de resolución de problemas existentes, la programación genética se destaca particularmente en la solución de ciertas clases de problemas. A continuación se mencionarán algunas de las características de los problemas que han sido atacados con éxito por la programación genética, siguiendo lo expresado en [Poli *et al.*, 2008]:

- **La interrelación de las variables no es conocida en una forma satisfactoria:** Cuando la forma en que se relacionan las variables es conocida con un cierto grado de detalle, pueden desarrollarse procedimientos analíticos con sus inherentes ventajas. Los procedimientos generales como la programación genética brillan en áreas en las que el conocimiento de las interrelaciones es incompleto.

- **Existe una forma rápida de evaluar soluciones tentativas al problema:** Como la programación genética debe evaluar un gran número de candidatos para poder obtener una solución razonablemente efectiva, debe existir un procedimiento para evaluar soluciones parciales que pueda ejecutarse un gran número de veces con un costo computacional razonable.

- **Son aceptables soluciones parciales:** Al ser un procedimiento de búsqueda estocástico operando en general en un espacio enorme de posibles soluciones, la programación genética no puede garantizar encontrar un óptimo global para el problema. Por lo tanto, el problema debe poder aceptar soluciones no óptimas ya que, de lo contrario, deberán emplearse en general técnicas analíticas para obtener la solución.

- **No es necesario que los resultados tengan una justificación comprensible:** Como es normal en los procesos evolutivos, pueden producirse resultados que difieren notablemente de los que produciría un ingeniero humano en la misma situación. Si bien en algunos casos se han obtenido resultados muy similares a los encontrados previamente por seres humanos [Koza *et al.*, 2003], en otros los resultados del proceso evolutivo pueden resultar difíciles de comprender.

Una de las primeras aplicaciones para las que se utilizó la programación genética fue para efectuar una regresión simbólica [Koza, 1989], es decir aproximar funciones de un conjunto de variables x_1, x_2, …, x_n mediante expresiones que las conecten utilizando operadores de un conjunto predefinido. Este problema se adapta directamente a la utilización de representaciones en forma de árbol (ver sección 2.2.2.1) ya que puede utilizarse el mismo programa (o sea el genotipo) como representación de la solución a desarrollar (fenotipo). Un ejemplo de una representación similar, empleada para aproximar números reales con enteros en lugar de una función, puede verse en la figura 2.19.

Otro problema similar en que se pudo aplicar la programación genética es el desarrollo de un controlador para un péndulo invertido [Shimooka & Fujimoto, 2000] como el que puede verse en la figura 2.22. En este caso, el programa no toma un conjunto de variables genéricas sino que toma los parámetros del péndulo que pueden ser medidos y desarrolla una expresión que operando con ellos permita cerrar el lazo de control.

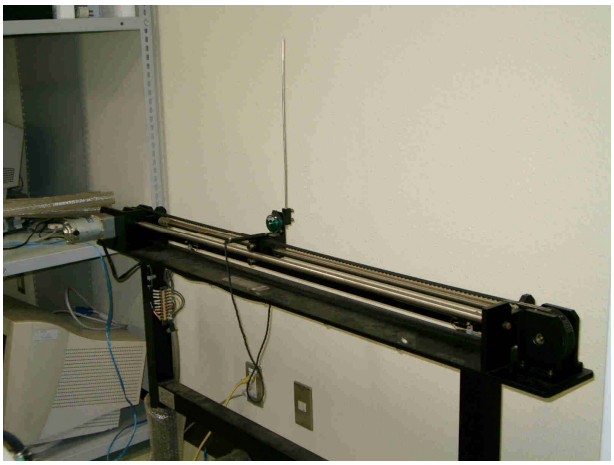


Figura 2.22. Péndulo invertido: su control constituye un problema clásico [Oita, 2008].

Si bien la resolución de los problemas mencionados anteriormente resultó de interés, no representaba el aporte de soluciones novedosas, ya que estos habían sido resueltos de forma satisfactoria utilizando otros métodos con anterioridad a la aplicación de la programación genética a su solución.

Con el aumento del poder computacional disponible y el perfeccionamiento de los algoritmos empleados, en esta última década fue posible obtener resultados competitivos con los obtenidos por seres humanos. Como ejemplo, en [Koza *et al.*, 2003] pueden observarse múltiples desarrollos considerados competitivos con los desarrollados por seres humanos, tales como:

- Síntesis de topologías para controladores PID.
- Creación de reglas para ajustar un controlador PID.
- Creación de circuitos mixtos digital-analógicos.
- Creación de generadores de señal cúbicos.
- Diseño de la topología como de la ubicación física de los componentes de un circuito integrado.

Otros resultados obtenidos de gran interés son: la creación de un nuevo algoritmo cuántico que presenta un mejor desempeño que cualquier otro algoritmo clásico y el desarrollo de una antena para la misión espacial *Space Technology 5* [Hornby *et al.*, 2006].

El algoritmo cuántico desarrollado busca resolver un problema conocido como "*depth two and-or tree problem*", que consiste en determinar con la menor probabilidad de error posible la salida de una función booleana conocida particular aplicada sobre todas las posibles salidas de una función booleana desconocida de dos variables, utilizando una sola evaluación de esta última función. Si bien el problema carece de interés práctico en su restricción a dos variables, el algoritmo descubierto mediante el uso de programación genética obtuvo un resultado superior a los mejores algoritmos clásicos conocidos.

La antena desarrollada para la misión espacial *Space Technology 5* demostró que el uso de programación genética permitía encontrar un diseño competitivo con los desarrollados por expertos humanos, en un tiempo inferior y con prestaciones superiores en algunos aspectos (ver figura 2.23). Se ha continuado trabajando en el desarrollo de antenas mediante procesos evolutivos con vistas a su posible aplicación a futuras misiones de la NASA.

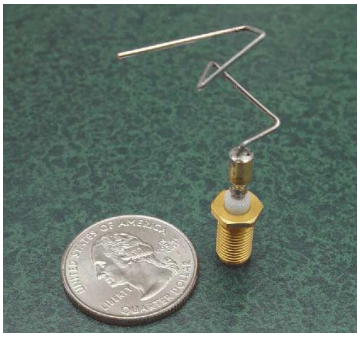


Figura 2.23. Antena desarrollada mediante programación genética para la misión espacial *Space Technology 5* [Hornby *et al.*, 2006].

2.3. PROGRAMACIÓN GENÉTICA APLICADA AL DISEÑO DE FILTROS ANALÓGICOS

En esta sección se analiza la aplicación de la programación genética al diseño de filtros analógicos: en primer lugar se analizan dos representaciones posibles para los circuitos desarrollados (sección 2.3.1), la representación en forma de *netlist* (sección 2.3.1.1) y la representación utilizando *bond graphs* (sección 2.3.1.2); luego se analiza el proceso de desarrollo embrionario (sección 2.3.2), mediante el cual se construye el circuito; por último, se muestran algunos de los resultados obtenidos mediante el uso de la técnica (sección 2.3.3).

2.3.1. REPRESENTACIONES DEL CIRCUITO

En esta sección se comparan dos posibles representaciones para el fenotipo de los individuos en la resolución de problemas de diseño de filtros analógicos: la representación en forma de *netlist* (sección 2.3.1.1) y la representación utilizando *bond graphs* (sección 2.3.1.2).

2.3.1.1. Representación en forma de *netlist*

La distinción marcada que presenta la programación genética entre los aspectos fenotípicos y genotípicos del individuo impide ocuparse solo de la representación del genotipo. En los casos en que el fenotipo sea trivial, por ejemplo un número real, puede prescindirse de especificar de una representación exacta. Sin embargo, en este caso, se trata de un fenotipo mucho más complejo ya que tiene que representar todo un circuito electrónico de tamaño arbitrario. Debido a la necesidad de interactuar con herramientas de simulación para evaluar los filtros obtenidos, es deseable que la representación a utilizar sea compatible con alguna de estas herramientas.

Una de las representaciones más utilizadas para los circuitos electrónicos es la forma textual conocida como *netlist*. Desde los comienzos mismos del estudio de los circuitos eléctricos se trabajó con representaciones de estos como grafos [Cederbaum, 1984]. Esta representación no hace más que expresar el grafo asociado al circuito como una lista de aristas con atributos.

En este trabajo se eligió utilizar la representación empleada por SPICE, el programa de simulación de circuitos electrónicos estándar en la industria. A continuación se efectuará una breve descripción general del formato. Una descripción más detallada, aunque limitándose a los aspectos utilizados en este trabajo, puede encontrarse en la sección anexa C.4. Para conocer más detalles de esta representación, incluyendo el tratamiento de componentes activos, pueden consultarse referencias tales como [Tuinenga, 1988] o [Quarles, 1989b].

Las *netlists* de SPICE modelan al circuito (mientras contenga solo elementos de dos terminales) como un grafo dirigido que abstrae la conectividad eléctrica de los componentes. De la misma forma que se indicó en la sección 2.1.1.5, se representa a los componentes electrónicos como aristas y a las conexiones entre estos componentes como vértices. La direccionalidad de las aristas sirve para dar una convención respecto a la polaridad de los componentes, permitiendo interpretar los valores de corriente y tensión de una forma libre de ambigüedades.

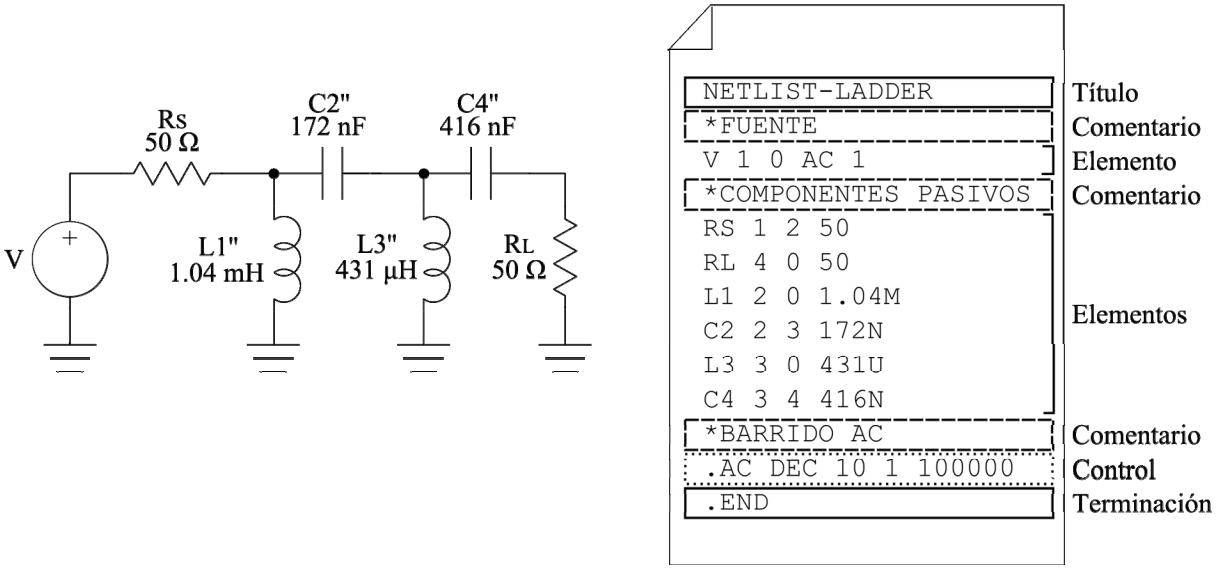


Figura 2.24. Circuito electrónico junto a su *netlist* correspondiente.

El formato de estos archivos es orientado a líneas, teniendo cada una de éstas un significado particular. En la figura 2.24 puede verse una *netlist* donde se aprecian claramente los bloques que la integran. La primera línea se dedica al título del circuito mientras que el resto de las líneas pueden dividirse en 4 clases: comentarios, elementos, control y terminación.

Las líneas de comentario constituyen simplemente una indicación para las personas que puedan leer estos archivos y se marcan con un `'*'` inicial. Son completamente ignoradas por SPICE y pueden aparecer en cualquier posición. También pueden incluirse comentarios en una línea cualquiera, si se preceden con un carácter `';'`.

La línea de terminación indica el punto en que SPICE debe dejar de procesar la *netlist*. Es un residuo de los tiempos en que la entrada al simulador se daba por medio de tarjetas perforadas y este comando permitía separar los distintos circuitos a simular.

Las líneas de elementos son el núcleo de la descripción, ya que son las que describen la composición del circuito. Cada una de ellas representa un componente del circuito. Si observamos su formato, para el caso de los elementos de dos terminales utilizados en este trabajo, podemos observar que todos contienen al menos cuatro campos. El primer campo es el nombre del elemento, donde el primer carácter indica el tipo de elemento del que se trata. Los dos campos siguientes indican a cual nodo se conecta el terminal positivo y a cual el negativo. El cuarto campo es el que indica el valor del elemento en unidades SI. En algunos casos puede requerirse el uso de campos adicionales, como puede verse en las referencias mencionadas anteriormente.

Los nodos corresponden a las interconexiones de los componentes y se identifican mediante una cadena de texto, que por convención suele ser un número. La asignación de identificación a los nodos es arbitraria a excepción del nodo de tierra, que debe estar presente y denominarse “0”.

Las líneas de control determinan las clases de análisis a realizar y las opciones con las que realizar el proceso de simulación. Comienzan con el carácter `'.'`, que va seguido del nombre del comando de simulación y un número variable de parámetros.

2.3.1.2. Representación en forma de *bond graph*

Una forma alternativa de representar a un circuito es la que los modela utilizando *bond graphs* [Wellstead, 2000]. Esta técnica sigue los flujos de energía en el circuito en lugar de abstraer las interconexiones físicas, como es el caso de la aproximación anterior. De este modo se consigue representar un sistema eléctrico arbitrariamente complejo con un árbol, en lugar de la clase más general de grafos utilizados por la representación convencional.

Para representar los flujos de energía, esta representación emplea el hecho de que en una amplia variedad de campos puede verse cada flujo de potencia como el producto de dos cantidades: esfuerzo y flujo. Estas cantidades se indican en cada una de las flechas que indican transferencias de energía. Dependiendo del sistema del que se trate, esfuerzo y flujo se refieren a cantidades diferentes; por ejemplo, en un sistema neumático, el esfuerzo sería la presión y el flujo sería el caudal de aire. En el caso que nos ocupa, el de los circuitos eléctricos, se identifica al esfuerzo con la tensión y al flujo con la corriente.

Si se desea abstraer el flujo de energía en un sistema complejo, será necesario indicar poder simbolizar las uniones donde el flujo de energía se separa hacia diferentes destinos. El formalismo de *bond graphs* incluye dos clases de uniones: uniones "0" y uniones "1". En las uniones "0" los flujos suman 0 y los esfuerzos son iguales, mientras que en las uniones "1" son los esfuerzos los que suman 0, mientras que los flujos son iguales. Si bien existen otras clases de nodos, no serán analizados en este trabajo. En la figura 2.25 puede verse la simbología básica utilizada, mientras que en la figura 2.26 puede apreciarse como puede ser expresado un filtro analógico pasivo en esta notación.

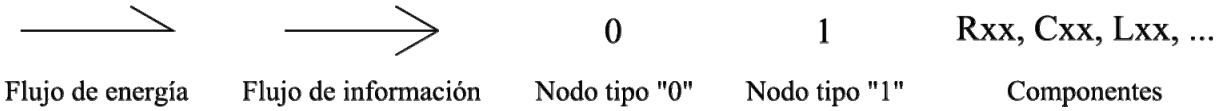


Figura 2.25. Simbología utilizada en el formalismo de *bond graphs*.

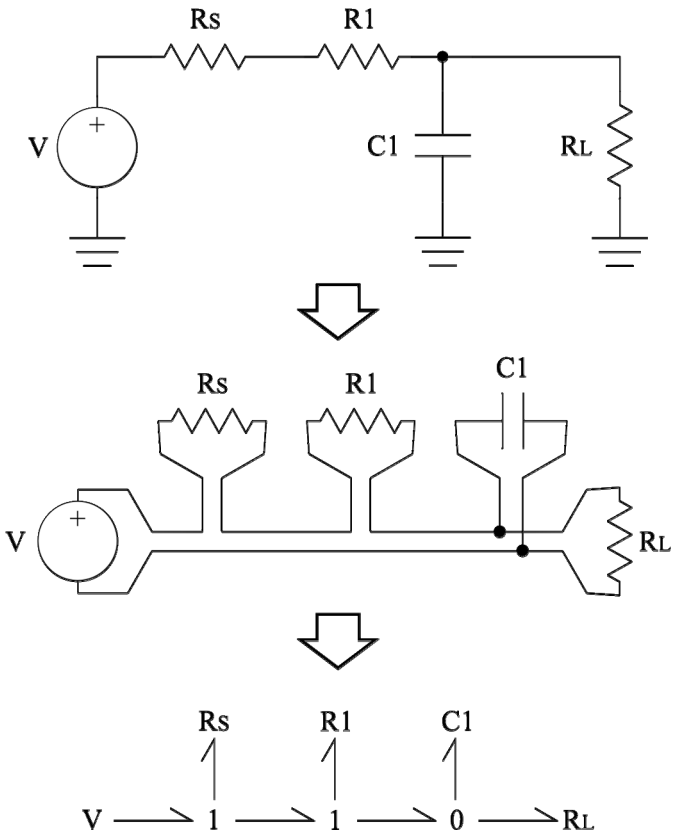


Figura 2.26. Visualización del proceso de transformación desde una representación convencional de un filtro RC hasta su representación como *bond graph*.

Aunque la gran mayoría del trabajo en el área de filtros se lleva a cabo utilizando las representaciones convencionales mencionadas en la sección anterior, los *bond graphs* tienen la ventaja de permitir una aproximación uniforme a un grupo variado de problemas, ya que toman un principio general como es la conservación de la energía como base para su resolución. Más

específicamente, en el campo del diseño de filtros analógicos usando programación genética, se han realizado algunos trabajos como [Fan *et al.*, 2001; Hu *et al.*, 2005] utilizando esta aproximación.

Sin embargo, para este trabajo se ha preferido utilizar la representación convencional por presentar herramientas de simulación más maduras y una mayor facilidad de expansión; como el simulador permite utilizar componentes activos, sería relativamente simple extender el trabajo para considerar el caso más general de filtros analógicos activos.

2.3.2. DESARROLLO EMBRIONARIO

Utilizando la representación en forma de árbol de los programas, estos consistes en árboles con ramas ordenadas y etiquetas en los nodos, mientras que los circuitos, usando la representación en forma de *netlists*, consisten en multigrafos cíclicos. Esto lleva a la necesidad de decidir como realizar la conexión entre ambas representaciones.

Siguiendo a [Koza *et al.*, 1997], puede utilizarse un proceso análogo al desarrollo embrionario, en el cual cada instrucción ejecutada tiene el efecto de alterar una parte de la solución (ver sección 2.2.1.2). Se denomina embrión al circuito inicial del que se parte para construir la solución.

Las funciones utilizadas para construir el programa (que serían las "etiquetas" de los nodos del árbol, ver figura 2.27) se pueden dividir en tres clases: funciones que modifican la topología, funciones que crean componentes y funciones que devuelven valores (no se tratará el uso de funciones definidas automáticamente, utilizadas en [Koza *et al.*, 2007], porque no son empleadas en este trabajo). Todas estas funciones actúan sobre lo que se denomina "cabezas de escritura" situadas sobre distintas aristas del multigrafo que representa al circuito en construcción.

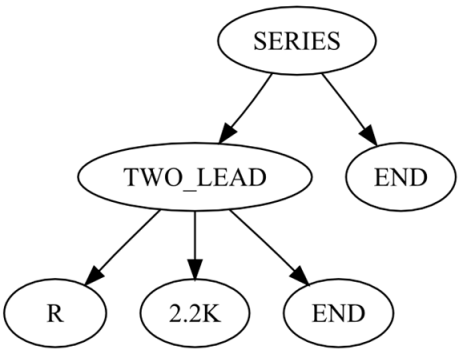


Figura 2.27. Ejemplo de programa para la construcción de un circuito.

Tal como puede verse en la figura 2.28, las funciones que modifican la topología toman la arista del grafo sobre la que actúa la cabeza de escritura correspondiente y realizan la operación asociada, creando nuevas cabezas de escritura en el caso de que la operación involucre la creación de nuevas

aristas. Las cabezas de escritura que resulten de la operación son pasadas a las funciones descendientes para que puedan continuar el proceso de desarrollo evolutivo.

Las funciones de creación de componentes simplemente toman la cabeza de escritura, modifican la arista asociada, y pasan la cabeza de escritura a sus descendientes. Para asignar el tipo y valor correspondientes al componente creado, deben tener como descendiente a un subárbol encargado de devolver el tipo del componente y a otro encargado de construir dicho valor.

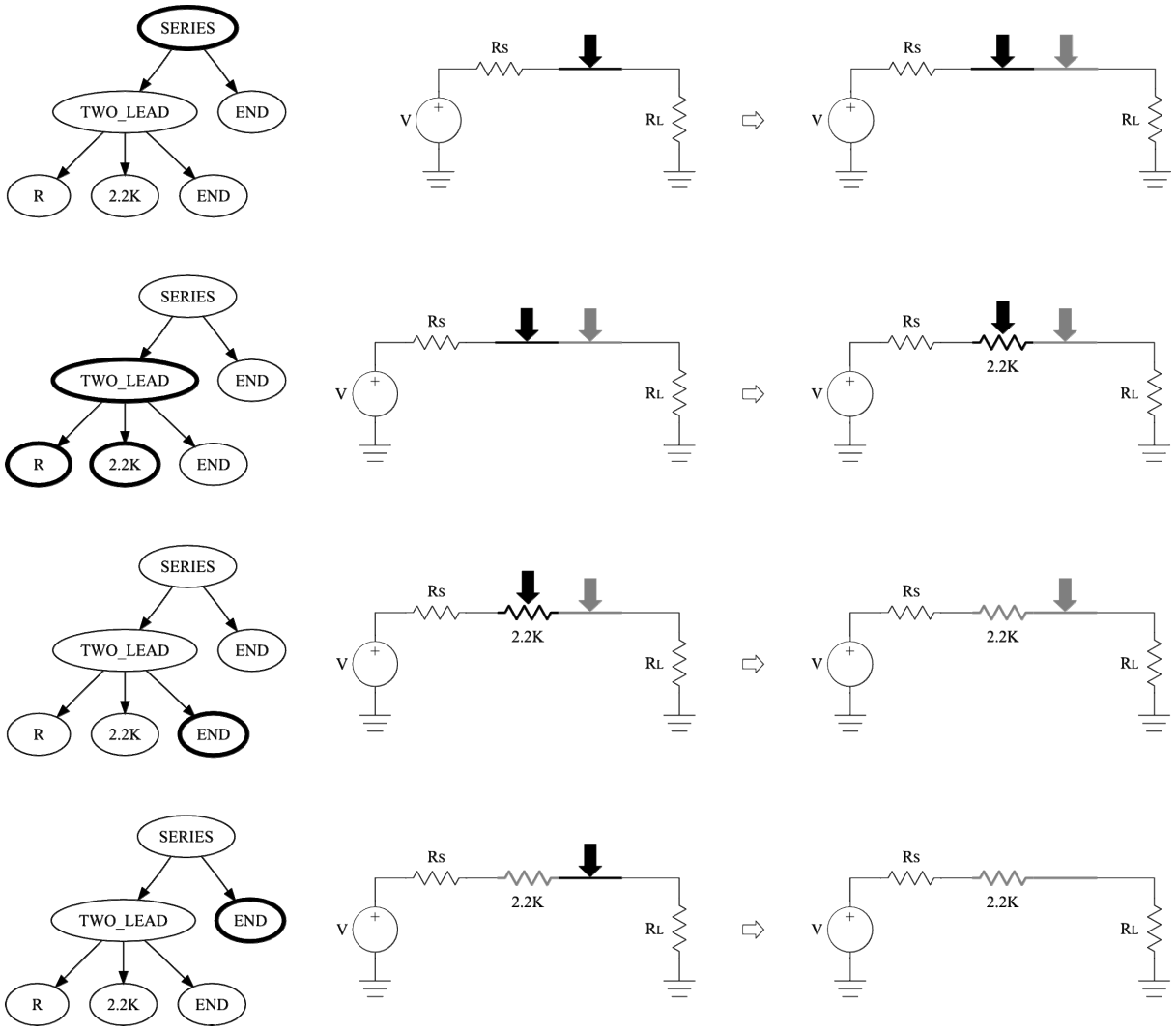


Figura 2.28. Ejemplo de la construcción del circuito correspondiente al programa de la figura 2.27. En cada "línea" de la figura se resalta la porción del programa que se está ejecutando y se marca con flechas la posición de las cabezas de escritura, resaltando la que se encuentra activa.

Dentro de cada una de las categorías mencionadas, existe un número de funciones. Por ejemplo, dentro de las funciones que modifican la topología, existen algunas como `SERIES`, que duplican el componente sobre el que se encuentre la cabeza de escritura creando una nueva cabeza de escritura y `END`, que destruye la cabeza de escritura. Si se observan las funciones de creación de componentes, solo se utiliza `TWO_LEAD`, encargada de crear componentes de dos terminales. Esto es

debido a que en los filtros analógicos pasivos solo se utilizan componentes de esta clase. Una descripción más completa de las funciones asociadas al proceso de desarrollo embrionario puede encontrarse en el capítulo anexo E.

Para poder obtener un resultado bien definido de la ejecución de un programa, puede ser necesario (dependiendo del conjunto de funciones utilizado) definir un orden de ejecución global (esto no afecta la posibilidad de aplicar paralelismo, ya que pueden construirse distintos individuos en paralelo). La elección más común es procesar "*depth first*" el programa, eligiendo los subárboles "de izquierda a derecha".

Otra decisión a tomar es como representar los árboles que constituyen a los individuos. Si bien una representación basada en punteros tiene como ventaja el disminuir el consumo de memoria en el caso de subárboles duplicados y de simplificar el proceso de cruza, tiene baja coherencia de accesos, reduciendo la utilidad de los cachés del procesador. Una representación prefija de los árboles, posible porque la cantidad de descendientes de un nodo queda determinada pro el tipo de éste, permite almacenar un individuo en un bloque contiguo de memoria, disminuyendo así el número de accesos a memoria.

2.3.3. RESULTADOS

Si se limita la discusión a filtros analógicos pasivos, pueden destacarse en primer lugar los resultados obtenidos por John Koza y su equipo. En [Koza *et al.*, 1997] aplicó al programación genética al diseño de múltiples circuitos, incluyendo un filtro pasabajos con una banda pasante de 0 a 1 kHz y una banda bloqueada arriba de 2 kHz. Obtuvo resultados satisfactorios en 32 generaciones utilizando una población de 640000 individuos, o sea una carga computacional total de unas 20 millones de evaluaciones de circuitos (ver figura 2.29).

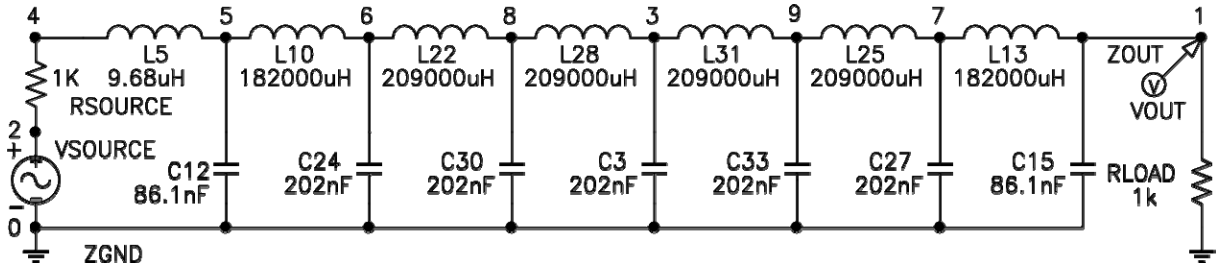


Figura 2.29. Filtro pasabajos obtenido mediante programación genética [Koza *et al.*, 1997].

Otro diseño interesante, que puede verse en [Koza *et al.*, 2003], es el de un circuito con una ganancia de tensión mayor que 2, realizado utilizando solo resistores y capacitores (lo que no era considerado posible por varios ingenieros electrónicos consultados por los autores). Este resultado, que puede observarse en la figura 2.30, demandó 927 generaciones utilizando una población de 660000 individuos, o sea una carga computacional de unas 600 millones de evaluaciones. En este

caso, los autores también incluyen una indicación aproximada del número de ciclos consumido, que es aproximadamente $3 \cdot 10^{15}$ ciclos.

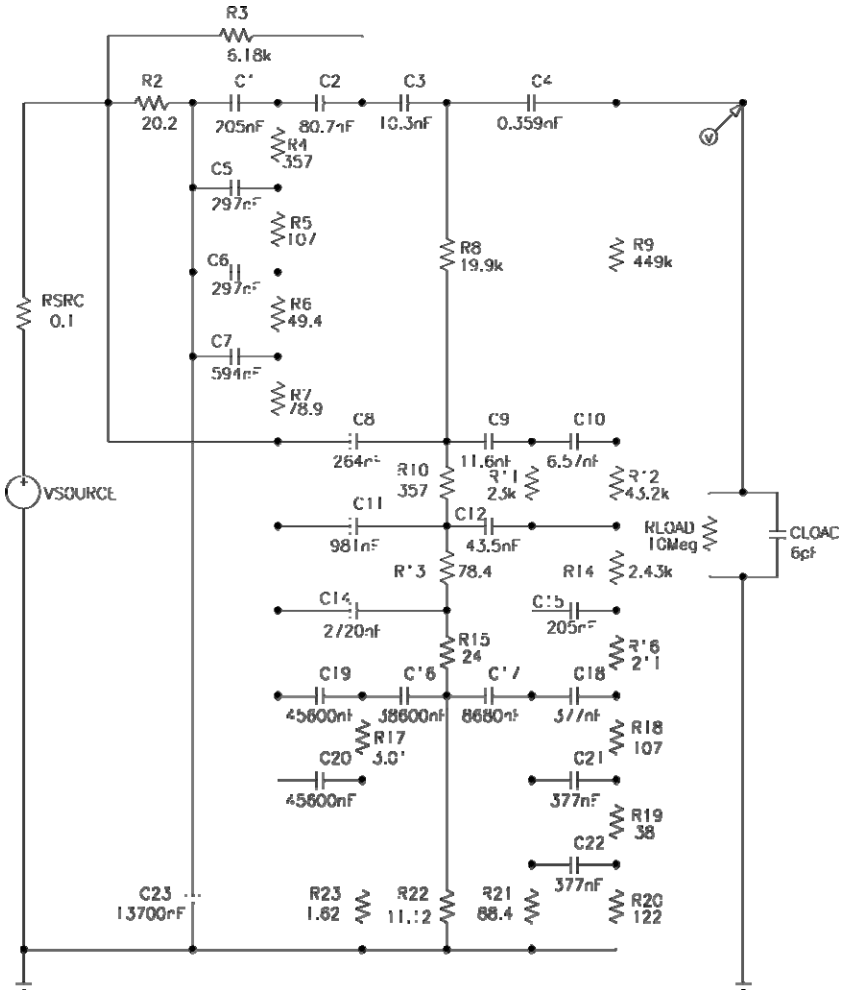


Figura 2.30. Circuito con ganancia mayor que 2, realizado solo con resistores y capacitores [Koza *et al.*, 2003].

Dentro de los trabajos realizados por otros autores, puede destacarse el trabajo utilizando representaciones lineales que se encuentra en [Lohn & Colombano, 1998], en el que se desarrolló un filtro pasabajos aceptable en 22 generaciones de unos 18000 individuos, o sea aproximadamente 400000 evaluaciones. También es de interés el trabajo utilizando *bond graphs* que puede verse en [Hu *et al.*, 2005], en el que se desarrollaron filtros pasabajos satisfactorios utilizando alrededor de 10^6 evaluaciones. Estos resultados son comparados con los del presente trabajo en la sección 5.5.

3. PROBLEMAS TRATADOS

En este capítulo se presenta: una descripción general de los problemas que afectan el rendimiento de la programación genética (sección 3.1); una descripción detallada del problema del crecimiento del código (*bloat*) (sección 3.2), analizando algunas de las causas propuestas para explicar este problema, tales como: precisión en la replicación (sección 3.2.1), tendencia en la eliminación (sección 3.2.2), la naturaleza de los espacios explorados (sección 3.2.3) y la correlación con la profundidad (sección 3.2.4); se concluye analizando el problema de la convergencia prematura (sección 3.3), vinculado a la calidad de los resultados obtenidos aplicando las técnicas de optimización.

3.1. GENERALIDADES

El principal obstáculo que se presenta para la aplicación en la práctica de la programación genética es el elevado uso de recursos computacionales que requiere. Este es debido principalmente a los elevados costos que se incurren al evaluar un individuo en un dominio de aplicación relativamente complejo. Por ejemplo, en el caso que se trata en este trabajo, la evaluación de cada individuo requiere una ejecución del programa SPICE tomando como entrada la representación del circuito.

Con el progresivo cambio actual hacia las arquitecturas paralelas [Asanovic *et al*., 2006], cobra especial importancia la facilidad con que aprovechan el paralelismo los algoritmos utilizados. Una de las ventajas que posee la programación genética (pero que es compartida otras técnicas de búsqueda similares como los algoritmos genéticos) consiste en ser fácilmente paralelizable en distintos grados. Puede irse desde un caso extremo, en el cual podría distribuirse la evaluación de cada individuo de la población a un distinto hilo de ejecución, hasta simplemente realizar en paralelo las distintas ejecuciones del proceso evolutivo.

Naturalmente estas ventajas no invalidan la importancia de la utilización eficiente de los recursos computacionales. Una mayor eficiencia constituye una ventaja comparativa respecto a otras implementaciones no optimizadas, permitiendo la obtención de una mejor solución para un conjunto dado de recursos. De aquí la importancia de evaluar distintas técnicas de optimización que posibiliten un aprovechamiento más completo de los recursos disponibles.

Analizando más en detalle el aspecto de los recursos demandados, se observa que cuando se trabaja con dominios complejos, el desempeño de la programación genética suele estar limitado por los requerimientos de procesamiento y no por el consumo de memoria. Como la complejidad de evaluar a los individuos suele ser superlineal respecto al tamaño, los tiempos de evaluación crecen en comparación con el espacio ocupado al aumentar el tamaño promedio de los individuos. Por

ejemplo, una población que ocupa menos de 8 MB de memoria puede llegar a demorar más de 10 minutos en evaluarse.

3.2. CRECIMIENTO DEL TAMAÑO DE LOS INDIVIDUOS

Una de las experiencias más frecuentes en la aplicación de la programación genética es el crecimiento del tamaño de los individuos, conocido como *bloat*. No es un fenómeno que se produzca en forma gradual, sino que se manifiesta como un rápido crecimiento del tamaño de algunos individuos en término de unas pocas generaciones, sin que vaya acompañado por una mejora asociada en los puntajes. Este fenómeno tampoco se produce en forma consistente en distintas ejecuciones del proceso evolutivo; por el contrario, es habitual que múltiples ejecuciones se completen sin problemas antes de que se haga ostensible.

El principal inconveniente que conlleva este crecimiento es el aumento ocasionado en los tiempos de evaluación ya que, por los motivos explicados en la sección anterior, el incremento en el uso de memoria es un problema menor en comparación. La disminución del progreso del proceso evolutivo puede llegar a extremo de constituir una "detención" para fines prácticos, forzando su reinicialización.

Este fenómeno ha sido estudiado en gran detalle [Poli *et al.*, 2008; Streeter, 2003] y se han formulado múltiples teorías respecto a sus causas, en su mayoría aplicables a las representaciones estructuradas en forma de árbol. Algunas de ellas serán descritas brevemente a continuación: precisión en la replicación (sección 3.2.1), tendencia en la eliminación (sección 3.2.2), naturaleza de los espacios explorados (sección 3.2.3) y correlación con la profundidad (sección 3.2.4).

3.2.1. PRECISIÓN EN LA REPLICACIÓN

Esta teoría explica el crecimiento al indicar que, al ser destructiva la mayoría de las operaciones de cruza y de las mutaciones, obtendrán una ventaja evolutiva aquellos individuos resistentes a los efectos de estas operaciones. Una de las formas en las que un individuo puede protegerse de estos operadores genéticos es la inclusión de grandes cantidades de código no funcional. De ese modo, la mayor parte de las operaciones no originarán alteraciones funcionales.

3.2.2. TENDENCIA EN LA ELIMINACIÓN

La mayor parte del código no funcional de los individuos suele encontrarse cerca de las "hojas" del cromosoma. Por ello, la mayor parte de las cruzas que no afectan al individuo modificarán subárboles pequeños y los reemplazarán por otros más grandes. Si se analizan los efectos de esta selección, se verá que los individuos con cruzas neutrales (que constituirán la mayoría de los resultados de cruzas seleccionados) tenderán a crecer a lo largo del tiempo.

3.2.3. NATURALEZA DE LOS ESPACIOS EXPLORADOS

Esta teoría predice que, por arriba de un cierto tamaño mínimo, la distribución de aptitudes no varía con el tamaño. Como existen muchos más programas grandes que pequeños, al transcurrir el proceso evolutivo pasaría a ser más probable que los individuos con un puntaje dado sean grandes que pequeños. O sea que, con el paso de las generaciones, los individuos se harían más grandes porque tendrían más formas posibles de ser grandes.

3.2.4. CORRELACIÓN CON LA PROFUNDIDAD

Esta teoría se basa en la observación de la distinta importancia de los nodos de los individuos: los nodos más alejados de la raíz tienen una influencia mucho menor en el puntaje del individuo que los que se encuentran más próximos al nodo raíz. Esta diferencia originaría dos efectos: promover que los padres sean más grandes, ya que por su profundidad resultarían menos afectados por las cruzas en general, y aumentar el tamaño promedio de los individuos ya que subárboles relativamente grandes podrían "adosarse" en un nodo distante de la raíz sin afectar demasiado el puntaje del individuo (este efecto constituiría una variación de la teoría presentada en la sección 3.2.2).

3.3. CONVERGENCIA PREMATURA

La gran mayoría de los problemas de optimización global son clasificables como NP-duros [Neumaier, 2004], lo que muy probablemente haga imposible [Aaronson, 2004] el desarrollo de algoritmos que los resuelvan *garantizando* resultados óptimos y tiempos de ejecución razonables (polinómicos). Por consiguiente, los algoritmos prácticos para resolver esta clase de problemas deben abandonar alguna o ambas de estas garantías. En el caso de la programación genética se abandonan las garantías de optimalidad, preservando la posibilidad de controlar el tiempo de ejecución del proceso.

Muchas técnicas generales de optimización, incluyendo la programación genética, obtienen la información de la función a optimizar evaluándola en múltiples puntos. Como esto no puede proporcionar un conocimiento completo del comportamiento de esta función, existe la posibilidad de que converjan a un óptimo local en lugar del óptimo global deseado. La programación genética no otorga garantías de optimalidad aunque, tal como se vio en la sección 2.2.3, es capaz de obtener resultados competitivos en problemas intratables con técnicas convencionales.

El fenómeno de convergencia se aprecia como una reducción en la diversidad de la población. Esta reducción se da tanto si se trata de una convergencia a un óptimo local como el caso de un óptimo global, pero en caso de existir suficiente diversidad genética en la población, la acción de la cruza impide que el proceso quede atrapado en óptimos locales.

En los casos en que la diversidad de la población no resulte suficiente, se observará convergencia a un resultado no óptimo. Esa eventualidad motiva el uso de múltiples ejecuciones para aumentar la confianza en la calidad del resultado obtenido.

El principal interés de este problema desde el punto de vista de las técnicas de optimización analizadas en el capítulo 4 viene dado por lo efectos nocivos sobre la diversidad genética que tienen algunas de éstas. En efecto, la eliminación de individuos de la población o la alteración de su aptitud interfieren con el normal desarrollo del proceso evolutivo, aumentando en consecuencia la probabilidad de convergencia prematura.

4. SOLUCIÓN PROPUESTA

En este capítulo se presenta: una descripción general de las soluciones que se han propuesto para enfrentar los problemas descritos en el capítulo anterior y las métricas para evaluar su desempeño (sección 4.1); un recorrido más detallado por las técnicas propuestas para resolver estos problemas mediante el control de poblaciones (sección 4.2), en particular el uso de plagas para controlar el esfuerzo computacional (sección 4.2.1) y el ajuste dinámico de la función de evaluación (sección 4.2.2); una descripción de la aplicación de un caché con objeto de acelerar la evaluación de las poblaciones (sección 4.3), incluyendo aspectos generales (sección 4.3.1), la selección de una función de hash adecuada para este caché (sección 4.3.2) y la medición del efecto en particular sobre la evaluación de individuos (sección 4.3.3).

4.1. GENERALIDADES

Métricas tales como mejora del puntaje en función del número de generaciones resultan inadecuadas para evaluar la aplicabilidad práctica de la programación genética y de sus técnicas de optimización. Para poder lograr una apreciación de la efectividad práctica de estas técnicas y de sus posibles optimizaciones es necesario reemplazarlas por métricas que incorporen los aspectos que limitan el desarrollo del proceso evolutivo en problemas reales.

En general el proceso evolutivo se ve limitado en primer lugar por el tiempo de procesamiento disponible para efectuar el desarrollo evolutivo y en segundo lugar por la memoria disponible para almacenar la población. El peso relativo entre estos dos factores varía de acuerdo a la naturaleza del dominio en cuestión. Por ejemplo, en el diseño de filtros analógicos pasivos, que es el caso de interés para este trabajo, los costos de evaluación dominan ampliamente sobre el costo de las operaciones genéticas.

Esto lleva a la necesidad de emplear métricas que tomen estos factores en consideración de acuerdo a su peso relativo. Para el caso de este trabajo se eligió representar la relación entre el puntaje obtenido y el tiempo transcurrido, ya que el uso de memoria es menor en comparación, como se argumenta en el párrafo anterior.

El desarrollo del proceso evolutivo sigue generalmente un camino irregular, ya que suelen concentrarse los mayores avances en las secciones iniciales del proceso. Luego la velocidad de mejora suele disminuir en gran medida debido a la acción de dos factores: la dificultad de escapar de los óptimos locales y el *bloat*.

Como se indicó en el capítulo anterior, uno de los problemas más importantes que se enfrentan al aplicar la programación genética a problemas reales es el crecimiento desmedido del tamaño de los individuos, que no viene acompañado por un aumento relacionado en los puntajes correspondientes. Este fenómeno, conocido como *bloat*, origina un progresivo aumento del costo computacional del proceso evolutivo, disminuyendo la escalabilidad del proceso a mayores tiempos de ejecución. Si bien la programación genética es una de las técnicas en la que este problema se manifiesta en forma más marcada. También se encuentra presente en otras técnicas evolutivas en las que se trabaja con individuos de tamaño variable.

Por otro lado, la dificultad de escapar desde los óptimos locales es hasta cierto punto intrínseca al problema a resolver: el algoritmo debe basar su búsqueda en los resultados obtenidos en puntos ya explorados del espacio de posibles diseños. Refinar su conocimiento acerca de este espacio requiere necesariamente incurrir en costos de evaluación adicionales. Aunque la programación genética es especialmente eficaz en explorar dominios con multitud de óptimos locales, requiere para ello evitar la pérdida prematura de la diversidad genética. Esto pone condiciones a la forma que pueden tomar las técnicas de optimización a emplear.

Se han propuesto múltiples formas de reducir los costos asociados al desarrollo del proceso evolutivo [Poli et al., 2008; Ryan et al., 2003].

Las técnicas de optimización evaluadas en este trabajo toman dos formas distintas: en primer lugar se tratan las denominadas como técnicas de control de poblaciones, que buscan optimizar el desarrollo del proceso evolutivo a través de modificar la conformación de la población; en segundo lugar se trata con una técnica que (idealmente) no modifica el desarrollo del proceso evolutivo sino que acelera su ejecución en la práctica, intercambiando un mayor uso de memoria pro esa disminución en el tiempo de procesamiento.

4.2. TÉCNICAS DE CONTROL DE POBLACIONES

En esta sección se describen las técnicas propuestas que involucran el control de la composición de las poblaciones: en primer lugar se analiza la utilización de plagas para reducir el esfuerzo computacional propio de realizar el proceso evolutivo (sección 4.2.1); en segundo lugar se analiza la posibilidad de efectuar un ajuste dinámico de la función de evaluación (sección 4.2.2).

4.2.1. USO DE PLAGAS PARA CONTROLAR EL ESFUERZO COMPUTACIONAL

Esta propuesta se basa en la observación de que, utilizando métricas adecuadas como las descritas en la sección 4.1, la eliminación de un cierto número de los individuos que presentan peor puntaje en cada generación conduce a la obtención de mejores resultados. La reducción en el esfuerzo

computacional originada por la disminución del número de individuos a evaluar supera los efectos negativos propios de la reducción en diversidad genética.

Este proceso ha sido denominado por sus autores [Fernández *et al.*, 2003] como plaga, por analogía con las plagas naturales en las que el aumento de la mortalidad puede llevar a una disminución progresiva del tamaño de la población. Puede verse esta técnica expresada con más detalle en el listado 4.1.

PROCESO: Programación genética utilizando plagas para reducir el esfuerzo computacional.

ENTRADAS: Criterio de optimalidad, función de evaluación (*Evaluar*()) aplicables sobre los fenotipos, criterio de detención, operadores genéticos con sus probabilidades asociadas, **cantidad de individuos a eliminar por generación.**

SALIDA: La "mejor" solución obtenida, de acuerdo al *criterio de optimalidad*.

1. $G \leftarrow 0$
2. $P[G]$ ← Población generada aleatoriamente (hay otras posibilidades).
3. $N \leftarrow$ *Tamaño*($P[G]$)
4. ***D*** ← **Cantidad de individuos a eliminar por generación.**
5. Para cada individuo I_j en $P[G]$:
   a. $F \leftarrow$ *Ejecutar*(I_j)
   b. $S_j \leftarrow$ *Evaluar*(F)
   c. Agregar S_j al vector de puntajes S.
6. Mientras no se cumpla el *criterio de detención*:
   c. $G \leftarrow G + 1$
   d. Repetir *D* veces:
      1. Eliminar al individuo de menor puntaje de acuerdo al vector de puntajes *S*.
   e. Repetir N veces:
      1. Seleccionar un *operador genético* *OG*() en forma aleatoria.
      2. Seleccionar individuos I_j e I_k de acuerdo al vector de puntajes S.
      3. Agregar *OG*(I_j,I_k) a $P[G]$.
   d. Para cada individuo I_j en $P[G]$:
      1. $F \leftarrow$ *Ejecutar*(I_j)
      2. $S_j \leftarrow$ *Evaluar*(F)
      3. Agregar S_j al vector de puntajes S.
7. Devolver la solución correspondiente al "mejor" individuo, de acuerdo al *criterio de optimalidad*.

Proceso 4.1. Descripción en pseudocódigo de la utilización de plagas para reducir el esfuerzo computacional en la programación genética (se resaltan las diferencias con la programación genética sin optimizar).

Un problema que va inexorablemente ligado a la aplicación de esta técnica es la reducción de la diversidad. Al eliminar a los individuos que presentan baja aptitud, elimina el principal mecanismo mediante el cual se evita la convergencia prematura. Aunque este riesgo no puede eliminarse, la selección cuidadosa del número de individuos a eliminar en cada generación permite minimizar su probabilidad de ocurrencia.

Los autores también indican otras posibles variantes de este procedimiento para mitigar el riesgo de convergencia prematura, incluyendo la eliminación de individuos redundantes, lo que preserva la diversidad genética, y la eliminación selectiva de individuos grandes, que podría contribuir a la reducción más efectiva del *bloat*.

4.2.2. AJUSTE DINÁMICO DE LA FUNCIÓN DE EVALUACIÓN

Una de las formas más simples en las que podría intentarse reducir el *bloat* es mediante la aplicación de un corte neto al tamaño de los individuos o de aplicarles una penalización según su tamaño. En caso de realizarse un corte neto de acuerdo a la profundidad o al tamaño, suele encontrarse el problema de que los individuos crezcan "justo hasta el límite", que no suele ser lo deseado. Otras penalizaciones más sofisticadas suelen requerir realizar estimaciones previas sobre el tamaño del individuo óptimo, lo que es difícil de realizar en general [Poli *et al.*, 2008].

En [Poli, 2003] se describe un método que busca evitar estos problemas actuando en forma probabilística. El autor denomina a este método como un ajuste dinámico de la función de evaluación, ya que modifica la función de evaluación creando agujeros en los que individuos que normalmente serían aptos "caen" fuera de la población. El método se califica como dinámico porque estos agujeros son creados en forma dinámica y no determinista, con objeto de evitar los problemas de los métodos anteriores.

La forma de operación consiste en seleccionar en forma aleatoria un conjunto de individuos entre los que poseen un tamaño mayor al promedio y penalizarlos en forma infinita, tal como puede verse en el listado 4.2. De este modo se permite el crecimiento en caso de ser necesario, pero la penalidad probabilística actúa como freno al crecimiento que no conlleve mejorías apreciables en el puntaje.

De todos modos, aunque la penalidad sea probabilística, si se elimina a una fracción muy elevada de los individuos de tamaño mayor al promedio se estará efectivamente en el caso de las técnicas descritas inicialmente. Por consiguiente, esta técnica sigue requiriendo un cierto ajuste en la probabilidad de selección de los individuos con tamaño superior al promedio.

Otras variantes propuestas por el autor incluyen el hacer dependiente del puntaje a la probabilidad de eliminación, de modo de no eliminar individuos prometedores, y realizar la selección automática de individuos pequeños. Esta última aproximación puede considerarse en cierto modo como una

técnica complementaria a la descrita anteriormente, ya que en lugar de eliminar a los individuos grandes busca quedarse con los de menor tamaño.

PROCESO:	Programación genética aplicando ajuste dinámico de la función de evaluación.
ENTRADAS:	Criterio de optimalidad, función de evaluación (*Evaluar*()) aplicables sobre los fenotipos, criterio de detención, operadores genéticos con sus probabilidades asociadas, **probabilidad de ejecución para los individuos "grandes".**
SALIDA:	La "mejor" solución obtenida, de acuerdo al *criterio de optimalidad*.

7. $G \leftarrow 0$
8. $P[G]$ ← Población generada aleatoriamente (hay otras posibilidades).
9. **T ← Probabilidad de ejecución para individuos "grandes".**
10. $N \leftarrow$ *Tamaño*($P[G]$)
11. Para cada individuo I_j en $P[G]$:
    a. $F \leftarrow$ *Ejecutar*(I_j)
    b. $S_j \leftarrow$ *Evaluar*(F)
    c. Agregar S_j al vector de puntajes S.
12. Mientras no se cumpla el *criterio de detención*:
    a. $G \leftarrow G + 1$
    b. Repetir N veces:
        1. Seleccionar un *operador genético* *OG*() en forma aleatoria.
        2. Seleccionar individuos I_j e I_k de acuerdo al vector de puntajes S.
        3. Agregar *OG*(I_j,I_k) a $P[G]$.
    c. $TP \leftarrow$ Tamaño promedio de los individuos de $P[G]$.
    d. Para cada individuo I_j en $P[G]$:
        1. $F \leftarrow$ *Ejecutar*(I_j)
        2. Si *Tamaño*(I_j) > *TP*, con probabilidad *T*:
            a. $S_j \leftarrow +\infty$
            Caso contrario:
            a. $S_j \leftarrow$ *Evaluar*(F)
        3. Agregar S_j al vector de puntajes S.
13. Devolver la solución correspondiente al "mejor" individuo, de acuerdo al *criterio de optimalidad*.

Proceso 4.2. Aplicación del ajuste dinámico de la función de evaluación (se resaltan las diferencias con la programación genética sin optimizar).

4.3. UTILIZACIÓN DE UN CACHÉ DE EVALUACIÓN

En esta sección se describe como puede utilizarse un caché para disminuir el tiempo promedio de evaluación de la población en sus distintos aspectos: en primer lugar, se analizan las características generales del algoritmo de evaluación utilizando el caché (sección 4.3.1); en segundo lugar, se

comparan distintas funciones de hash en el rol de identificar un individuo dentro del caché (sección 4.3.2); por último se miden los efectos que tiene el caché al evaluar una población generada en forma aleatoria (sección 4.3.3).

4.3.1. GENERALIDADES

En esta sección se propondrá una técnica que buscará mejorar el rendimiento del proceso d evaluación exclusivamente. A diferencia de las técnicas de control de poblaciones descritas en la sección 4.2, que buscan alterar el desarrollo evolutivo en una forma "favorable", la técnica descrita en esta sección no modificará (idealmente) el desarrollo del proceso evolutivo.

La utilización de un caché de evaluación consiste en almacenar el puntaje resultante de aplicar el proceso normal de evaluación a cada individuo, de modo que las posteriores evaluaciones de ese mismo individuo puedan resolverse mediante la simple lectura del puntaje almacenado previamente. Esta técnica, también denominada "memoización", ha sido aplicada previamente a los algoritmos genéticos [Povinelli & Feng, 1999] y en general a un gran número de problemas donde la evaluación de una función es costosa y pueden repetirse los valores a evaluar.

Para implementar un caché es necesario encontrar una forma de asociar a los resultados del proceso de evaluación con los individuos evaluados. Esto se realiza utilizando una estructura de datos que actúe como "diccionario", que asocian un valor dado a un clave en particular.

Existen múltiples estructuras de datos que pueden servir para este fin, pero una de las más eficientes para este propósito son las tablas *hash* [Knuth, 1998]. Normalmente las tablas hash le aplican una función (denominada función de *hash*) a la clave del dato a almacenar para obtener la posición donde será almacenado el dato correspondiente. Como en general el número de claves posibles es mucho mayor que el tamaño de la tabla, se suele almacenar la clave junto al dato asociado para evitar ambigüedades. En caso de que dos claves de datos a almacenar tengan el mismo hash, se aplica algún método para resolver este fenómeno denominado como "colisión".

El conjunto de claves suele tener una distribución poco uniforme, por lo que la función de hash debe "uniformizar" esta distribución para evitar un número excesivo de colisiones, con el consiguiente deterioro del rendimiento. Para ello debe ser sensible a pequeñas variaciones en las claves de modo que un conjunto de claves muy similar termine en posiciones muy distintas de la tabla. Sin embargo, no es necesario que se trate de un *hash* criptográfico ya que no se está frente a un oponente humano que *busque* colisiones.

Para esta aplicación la clave debería ser el individuo y el valor almacenado su correspondiente puntaje. Pero debido a que los individuos suelen ocupar más de 100 bytes, se decidió *utilizar como clave* a un hash (resultado de aplicar una función de hash) de 64 bits del individuo.

```
PROCESO:    Evaluación utilizando un caché hash

ENTRADAS:   Caché C, función de evaluación Eval(), función de
            hash Hash(), individuo I.

SALIDA:     Puntaje.

1.  N ← Tamaño(C)
2.  H ← Hash(I)
3.  E ← C[H Mod N]
4.  Si E.hash = H:
    a.  Devolver E.score
    Caso contrario:
    a.  S ← Eval(I)
    b.  E.hash ← H
    c.  E.score ← S
    d.  C[H Mod N] ← E
    e.  Devolver S
```

Proceso 4.3. Utilización de un caché de evaluación. El procedimiento descrito reemplaza a la evaluación normal.

Como la distribución de los *hashes* es bastante uniforme, puede prescindirse de la aplicación de una segunda función de hash para ubicar al par clave-valor en la tabla. Es suficiente con aplicar la operación módulo para restringir las ubicaciones a posiciones válidas dentro de la tabla.

Al tratarse de un caché, si no se encuentra un valor puede simplemente recalcularse afectando solo al rendimiento, no a la corrección de los valores. Por lo tanto, en caso de que se produzca una colisión puede prescindirse de utilizar una estrategia para ubicar los nuevos valores en otra posición. Simplemente se sobrescriben los valores en esa posición. Puede verse el funcionamiento de esta técnica con mayor detalle en el listado 4.3.

La única forma en que podría producirse un error en el resultado obtenido del caché sería en el caso de que se produjera una colisión en el hash de 64 bits utilizado para identificar al individuo evaluado. Si bien esto es más probable de lo que podría parecer debido a la "paradoja del cumpleaños" [Weisstein, 2008], sigue siendo extremadamente improbable. Si suponemos la utilización de una función de hash de 64 bits razonablemente efectiva, esta probabilidad está dada por

$$p(n) \approx 1 - \exp\left(-\frac{n^2}{2 \cdot 2^{64}}\right),$$

donde n representa a la cantidad de pares clave-valor insertados [van Oorschot & Wiener, 1994]. Como n es muy inferior a 2^{32}, la probabilidad de colisión también será notablemente inferior a la unidad.

4.3.2. SELECCIÓN DE UNA FUNCIÓN DE HASH

Al seleccionar una función de hash para este propósito se busca realizar un compromiso entre una buena distribución de sus salidas y una buena velocidad de cálculo. Si se buscara uniformizar los resultados "a ultranza" podría utilizarse una función de hash criptográfica tal como SHA-1 [NIST, 2002] o MD-5 [Rivest, 1992], que poseen muy buenas distribuciones de los valores de salida para una amplia gama de posibles entradas. Pero para el caso del presente trabajo, en el que los datos son individuos que participan en un proceso evolutivo, no es necesario tener una garantía tan fuerte. Alcanza con poseer una razonable confianza en la ausencia de colisiones entre los *hashes* producidos en una iteración del proceso evolutivo.

Con este fin se eligieron tres funciones de hash relativamente eficientes para realizar una evaluación comparativa, a saber: *FNV-1a hash* [Noll, 2003], *SuperFastHash* [Hsieh, 2007] y *MurmurHash* 2.0 [Appleby, 2008]. Como tanto *MurmurHash* 2.0 como *SuperFastHash* poseían solo versiones de 32 bits, se debió adaptarlas para que produjeran un resultado de 64 bits. Con un hash de solo 32 bits hubiera sido imposible garantizar estar libre de colisiones por la mencionada "paradoja del cumpleaños".

Para cada una de las tres funciones se evaluaron tres distintos parámetros: la velocidad de evaluación, por ser el objeto optimizar el tiempo de procesamiento, la ausencia de colisiones, para evitar confusiones entre la identidad de distintos individuos, y la distribución, para tener una idea acerca del riesgo de posibles colisiones.

Para medir la velocidad de evaluación se realizaron dos pruebas diferentes: en primer lugar se evaluaron las funciones sobre un bloque de memoria de 128 MB con un contenido aleatorio; en segundo lugar se realizó esta evaluación sobre 2^{20} bloques de memoria, cada uno de los cuales tenía un tamaño aleatorio elegido sobre una distribución uniforme entre 0 y 200 bytes y estaba relleno también con bytes aleatorios. Los tiempos resultantes de estas evaluaciones pueden observarse en la figura 4.1.

Los resultados obtenidos al evaluar las funciones de hash sobre los 2^{20} vectores también fueron utilizados para comprobar la ausencia de colisiones (o sea que distintos contenidos impliquen distintos *hashes*) y para obtener una representación de la distribución de las salidas.

Se observa claramente que *FNV-1a* es la función más lenta de las evaluadas, obteniendo *SuperFastHash* y *MurmurHash 2.0* tiempos muy similares entre sí. Los tiempos requeridos para calcular el hash del bloque de 128 MB y de los múltiples bloques pequeños fueron similares en ambos casos, siendo la diferencia más marcada en el caso de *FNV-1a*.

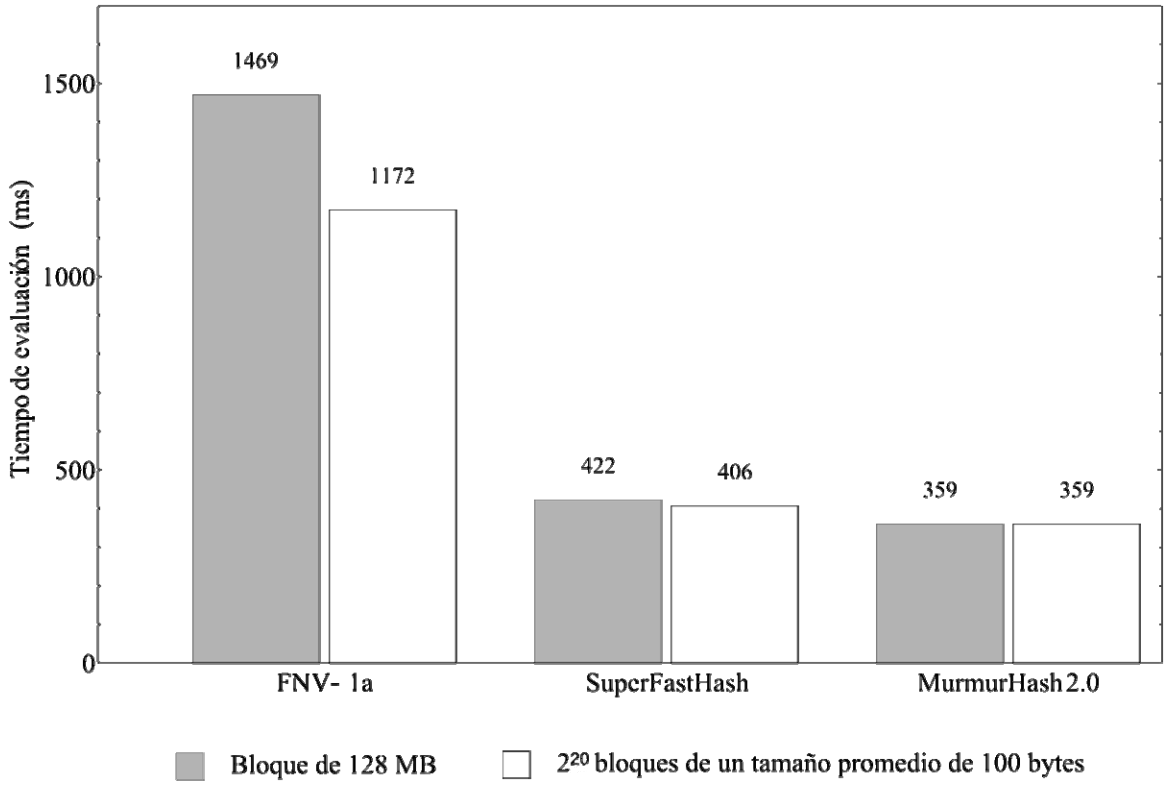


Figura 4.1. Comparación entre los tiempos requeridos para evaluar las tres funciones de hash en cuestión sobre dos distintos conjuntos de datos.

No fueron observadas colisiones en ninguno de los casos, aunque ello requirió realizar una pequeña modificación a MurmurHash 2.0, ya que originalmente presentaba colisiones en el caso de vectores muy pequeños. Esto es lo que cabría esperar desde un punto de vista estadístico. Si suponemos que la salida de la función de *hash* estudiada es una variable aleatoria en el rango de 0 a 2^{64}-1, tendríamos como probabilidad de colisión:

$$p(2^{20}) \approx 1 - \exp\left(-\frac{(2^{20})^2}{2 \cdot 2^{64}}\right) = 1 - \exp\left(-\frac{2^{40}}{2^{65}}\right) = 1 - \exp\left(-2^{-25}\right) \approx 1 - \left(1 - 2^{-25}\right) = 2^{-25} << 1$$

Si hubiéramos utilizado, en cambio, funciones de hash de 32 bits tendríamos incluso en el caso ideal:

$$p'(2^{20}) \approx 1 - \exp\left(-\frac{(2^{20})^2}{2 \cdot 2^{32}}\right) = 1 - \exp\left(-\frac{2^{40}}{2^{33}}\right) = 1 - \exp\left(-2^{7}\right) \approx 1$$

En la figura 4.2 puede verse la distribución de los *hashes* obtenidos al evaluar los 2^{20} bloques con cada una de las tres funciones de *hash*. Para clasificar las salidas se definieron 100 clases de igual tamaño cubriendo el rango [2, 2^{64}-1].

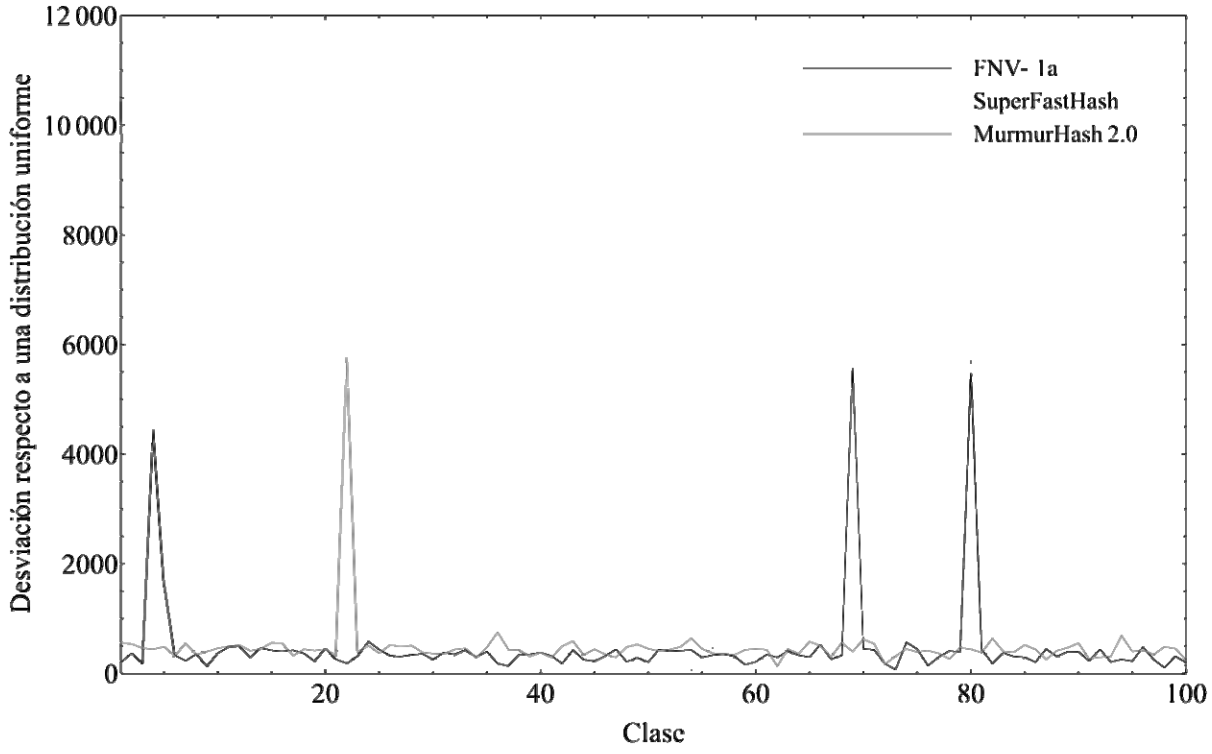


Figura 4.2. Distribución de los resultados de evaluar las tres funciones de hash sobre 2^{20} vectores aleatorios. Se cubrió el rango de posibles valores de salida de la función con 100 clases.

Si bien se observan algunas irregularidades, no tienen mayor importancia por las dimensiones del codominio de las funciones de hash en cuestión. La más significativa es la que se observa en *SuperFastHash*, en la que la clase 0 presenta casi el doble de individuos que otras clases, probablemente por un defecto propio de las modificaciones realizadas para adaptarla a trabajar con 64 bits.

En base a estos resultados se decidió seleccionar a *MurmurHash 2.0* para utilizarla en el caché d evaluación, ya que presenta la mejor velocidad de evaluación, es simple y posee una buena distribución de salida, junto a una ausencia de colisiones en las pruebas realizadas.

4.3.3. MEDICIÓN DE RENDIMIENTO APLICADO A LA EVALUACIÓN

Las poblaciones que se producen durante le proceso evolutivo tienen una distribución de tamaños que dista de ser uniforme (ver figura 4.3), lo que se ve acompañado por el hecho de que los valores que integran a los individuos suelen también ser marcadamente no aleatorios, por las restricciones que impone la sintaxis de los programas. Para observar si la performance de la evaluación mejoraba en forma real al combinarla con un caché implementado de acuerdo a los lineamientos de la sección 4.3.1 y con la función de hash seleccionada en la sección 4.3.2, se efectuó una serie de pruebas.

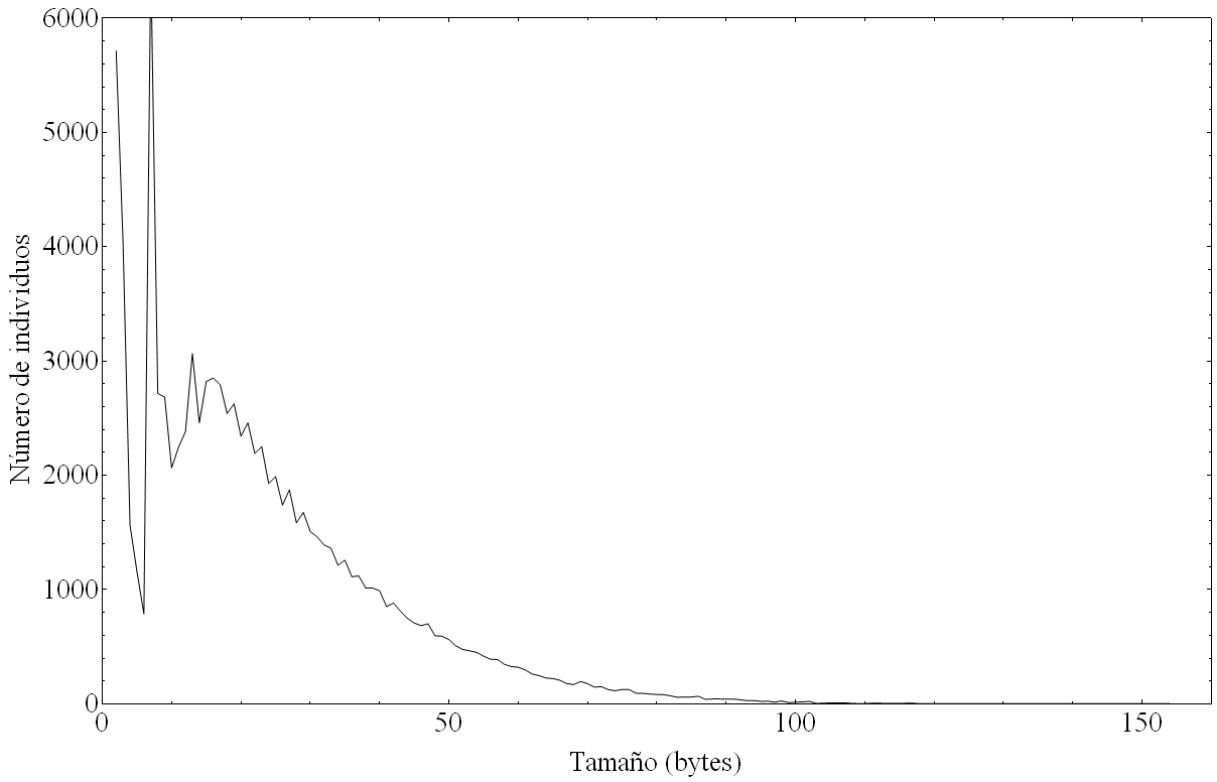


Figura 4.3. Cantidad de individuos según el tamaño.

Se evaluó una población de 100000 individuos generados aleatoriamente, utilizando un caché con 2^{20} elementos y empleando la función de hash *MurmurHash 2.0*, seleccionada anteriormente. La evaluación utilizando el caché se repitió tres veces de modo de poder comparar el tiempo de ejecución durante el proceso de carga del caché en la evaluación inicial con las ejecuciones posteriores. Se realizó también una evaluación sin utilizar el caché. Los resultados obtenidos pueden observarse en la figura 4.4.

Tal como podía esperarse, el tiempo de la primera evaluación fue notablemente superior al de las posteriores, debido a la necesidad de evaluar a cada individuo. La segunda y tercera evaluación no presentaron mayores diferencias porque el caché ya se encontraba completamente cargado.

A pesar de tener la sobrecarga adicional de tener que cargar el caché, la primera evaluación fue más rápida que la evaluación sin utilizar el caché. Probablemente esto se deba a que aprovecha la redundancia existente en la población, evitando evaluar dos veces al mismo individuo.

Estos resultados no se traducen directamente al proceso evolutivo, ya que la misma esencia de este proceso es la aparición de nuevos individuos, no presentes en el caché. No obstante ello, cabe esperar que esta técnica reduzca en forma notable los tiempos de evaluación en poblaciones poco diversas, “saltando sobre” las partes menos variadas del proceso evolutivo. En el capítulo 5 se

analizará el impacto del caché de evaluación propuesto sobre el desarrollo del proceso evolutivo en su conjunto.

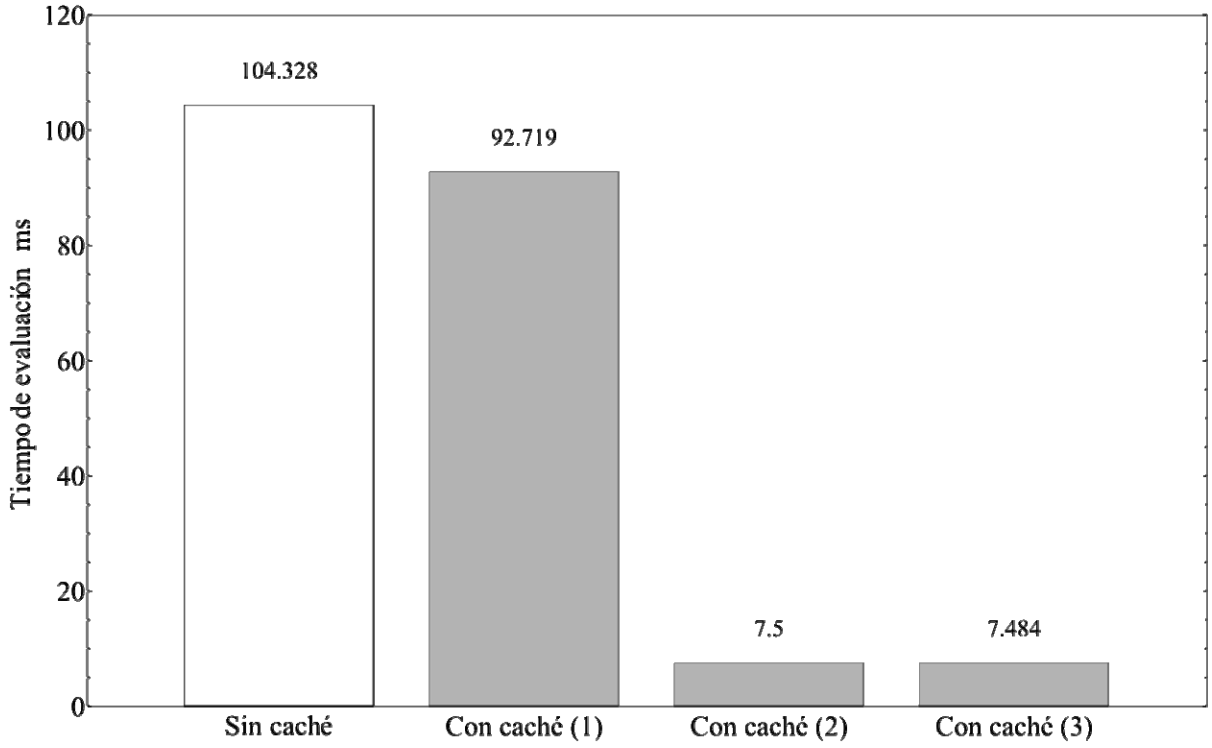


Figura 4.4. Tiempos de la evaluación sin utilizar caché y de las tres evaluaciones consecutivas utilizándolo.

5. VERIFICACIÓN EXPERIMENTAL

En este capítulo se presenta: una descripción general de la forma en que se llevaron a cabo los experimentos (sección 5.1); se analiza la relación entre el tamaño de los individuos y el tiempo que se requiere para evaluarlos (sección 5.2); se estudian los resultados obtenidos al aplicar las técnicas de optimización propuestas (sección 5.3), analizando como referencia al caso donde no se aplica ninguna de estas técnicas (sección 5.3.1) y aplicando a continuación cada una de ellas: plagas para disminuir el esfuerzo computacional (sección 5.3.2), ajuste dinámico de la función de evaluación (sección 5.3.3) y utilización de un caché de evaluación (sección 5.3.4); esta última técnica se analiza en general (sección 5.3.4.1) y aplicándola por separado (sección 5.3.4.2), así como combinándola con la utilización de plagas (sección 5.3.4.2) y el ajuste dinámico de la función de evaluación (sección 5.3.4.4); se concluye con un análisis comparativo de los resultados obtenidos (sección 5.3.4.4).

5.1. GENERALIDADES

Con el objeto de poder determinar la efectividad de las distintas técnicas mencionadas en el capítulo anterior, se realizó el diseño de un filtro analógico pasivo utilizando todas las posibles combinaciones de las técnicas anteriormente mencionadas.

En todos los casos se realizaron 10 ejecuciones independientes con una población de 1000 individuos, limitando el desarrollo del proceso evolutivo a 100 generaciones. Se utilizó una selección por ranking elitista, en la que el individuo con mejor puntaje es copiado en forma automática a la nueva generación. Para el resto de las operaciones se seleccionaron individuos con una distribución de probabilidad exponencial en base a su ranking, o sea la probabilidad de seleccionar al individuo *i*-ésimo en el ranking fue tomada como $P(i) \approx \lambda e^{-\lambda i}$ con $\lambda = 0.002$ (la probabilidad real de selección era algo mayor que la dada por esa fórmula, ya que los individuos seleccionados tenían valores de *i* entre 1 y 999). Las probabilidades de cruza, mutación, alteración de valores y copia directa fueron optimizadas empíricamente, obteniéndose valores de 0.1, 0.2, 0.2 y 0.5, respectivamente.

El problema que se eligió para probar el desempeño de las distintas técnicas fue el diseño de un filtro pasabajos con una frecuencia de corte de 10 kHz. Al tratarse de un filtro pasivo, donde los efectos de carga son especialmente importantes, toma especial importancia la especificación de la impedancia de la fuente de señal y de la impedancia de carga. Como el propósito de este diseño era solo comparar el desempeño de las distintas técnicas de optimización, se asignó en forma arbitraria un valor puramente resistivo de 1 kΩ a dichas impedancias.

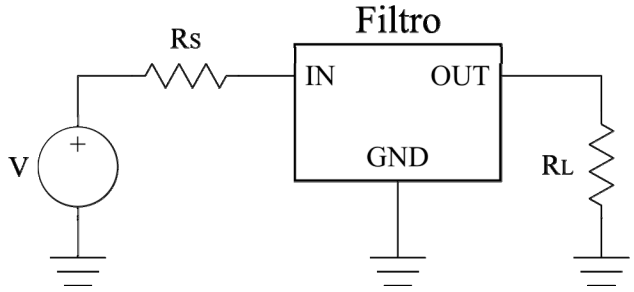


Figura 5.1. Esquema de ubicación del filtro a diseñar, marcando la impedancia de la fuente de señal (R_S) y la impedancia de carga (R_L).

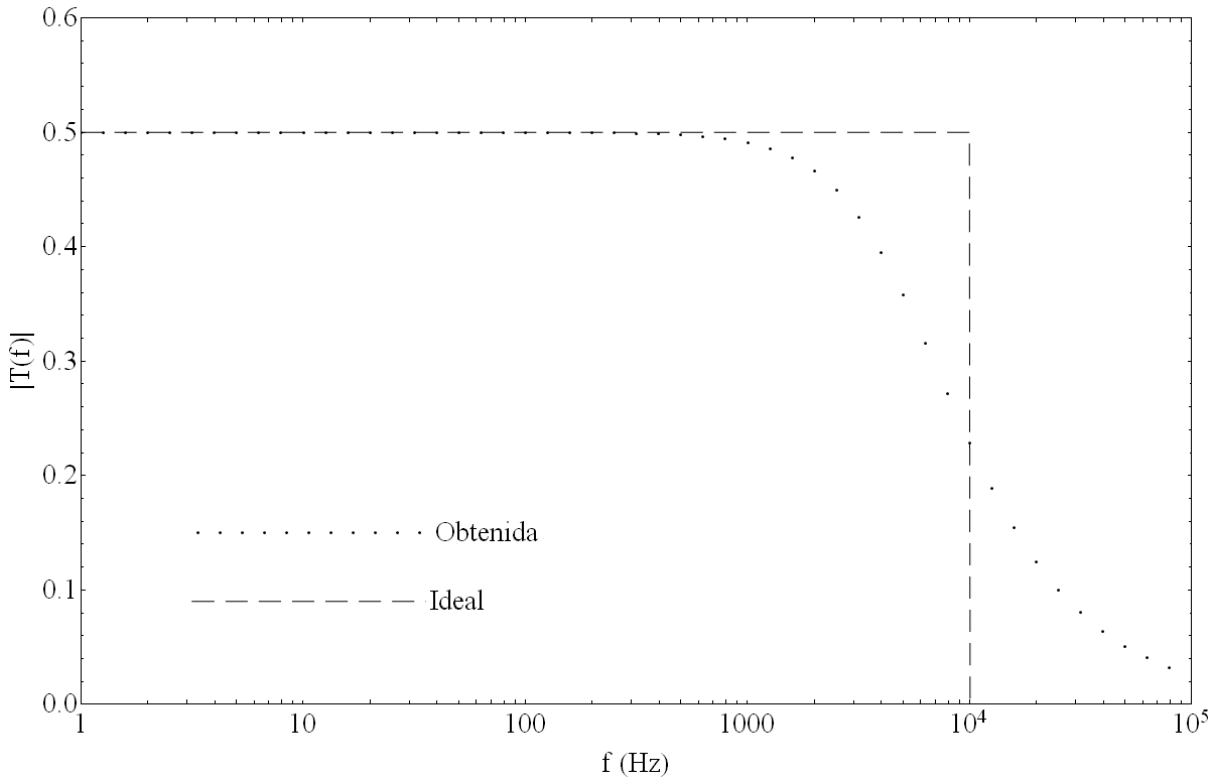


Figura 5.2. Comparación entre la función de transferencia ideal buscada y los 50 puntos de la transferencia "real" de uno de los individuos evaluados,

La correspondiente función de evaluación seleccionada, consistió en efectuar la suma de un puntaje base de 1 con las diferencias al cuadrado entre la transferencia real y la ideal en un total de 50 puntos, distribuidos logarítmicamente entre 1 Hz y 100 kHz. Por lo tanto, los menores puntajes corresponden a los diseños que mejor se ajustan a la especificación y los puntajes no pueden tomar valores inferiores a uno.

5.2. RELACIÓN ENTRE EL TAMAÑO DE LOS INDIVIDUOS Y EL TIEMPO REQUERIDO PARA EVALUARLOS

El primer análisis realizado fue con el objetivo de determinar la relación entre el tiempo empleado en evaluar cada generación y el tamaño promedio de sus individuos. Fue efectuado realizando 20

ejecuciones, sin aplicar ninguna de las técnicas de optimización. No obstante ello, no cabría esperar un cambio en los resultados en caso de aplicar las técnicas de control del tamaño de la población, ya que solo deberían afectar el tamaño de los individuos y no la correlación de este con el tiempo de evaluación. Por otro lado, el uso del caché de evaluación afecta los tiempos de evaluación de los individuos en una forma que no depende esencialmente de su tamaño, tal como se analizará con mayor detalle en la sección 5.3.4.1.

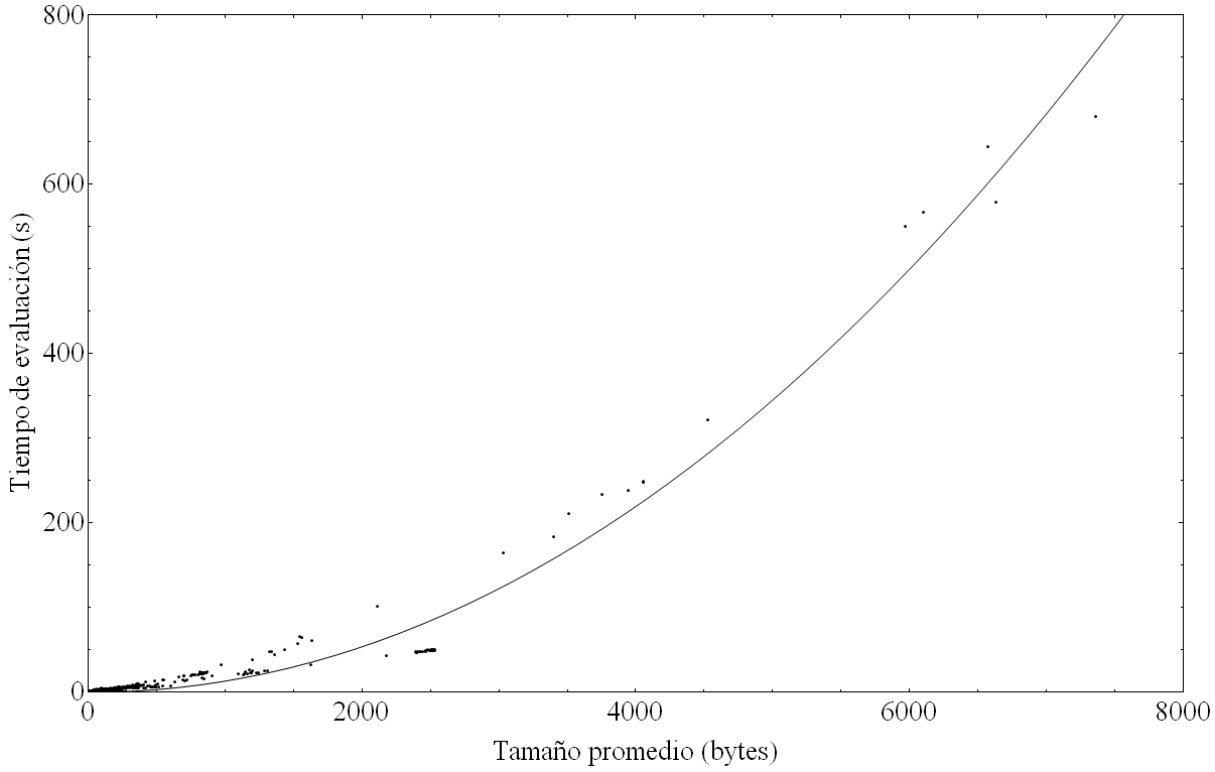


Figura 5.3. Relación entre el tamaño promedio de los individuos de una generación y el tiempo requerido para evaluarlos.

En la figura 5.3 podemos observar la relación entre el tamaño promedio de los individuos evaluados en una generación y el tiempo demorado en evaluar toda la población.

Realizando un ajuste a los datos con una expresión de la forma $t_{eval} = A \cdot (tam_{indiv})^B$, se obtuvo como resultado un exponente B de 2.03, lo que nos indica que el tiempo de evaluación depende en forma aproximadamente cuadrática del tamaño de los individuos. Esto, junto a la observación de que algunas generaciones demoraron más de 600 segundos en evaluarse, subraya la importancia de controlar el tamaño de los individuos.

5.3. RESULTADOS DE LAS TÉCNICAS DE OPTIMIZACIÓN

En esta sección se describen y analizan los resultados experimentales obtenidos aplicando las técnicas de optimización propuestas: en primer lugar se analizan los resultados obtenidos sin aplicar

ninguna de las técnicas, para formar una marco comparativo (sección 5.3.1); luego se estudian los resultados obtenidos al aplicar plagas para reducir la carga computacional (sección 5.3.2); a continuación se observan los resultados de realizar un ajuste dinámico de la función de evaluación (sección 5.3.3) y por último se analiza la aplicación de un caché de evaluación (sección 5.3.4), tanto en términos generales (sección 5.3.4.1), como por separado (sección 5.3.4.2) y en combinación con las técnicas previamente evaluadas de aplicación de plagas para reducir el esfuerzo computacional (sección 5.3.4.2) y de ajuste dinámico de la función de evaluación (sección 5.3.4.4).

5.3.1. CASO BASE

Con objeto de tener una referencia respecto a la cual comparar la efectividad de las distintas técnicas descritas en el capítulo anterior, se realizaron diez ejecuciones del proceso sin aplicar ninguna de ellas. Los resultados obtenidos se muestran en las figuras 5.4 y 5.5. Adicionalmente, estas ejecuciones permiten apreciar las características del proceso evolutivo antes de ser modificadas por la aplicación de las distintas técnicas.

En la figura 5.4 se muestra como fue variando el puntaje del “mejor” individuo a lo largo de las generaciones para cada una de las ejecuciones. Además de las ejecuciones individuales, se incluyó una “ejecución promedio” tomando como puntaje para cada generación el promedio de los valores obtenidos en las distintas ejecuciones en esa misma generación. También se destacó particularmente la ejecución que terminó con el menor puntaje final.

En la figura 5.5 se muestra la misma evolución, pero en relación al tiempo transcurrido desde el inicio de cada ejecución (observar que la escala temporal es logarítmica). Para el caso de la “ejecución promedio” se realizó también el promedio de los tiempos transcurridos.

Puede observarse que el desarrollo del proceso evolutivo de cada ejecución suele alternar periodos de cambio lento o nulo con variaciones relativamente abruptas. Los periodos de cambio nulo corresponden a aquellos en los que no se producen alteraciones en el mejor individuo, mientras que los de cambio abrupto representan la aparición de un nuevo diseño. Los periodos de cambio lento suelen representar un progresivo ajuste de los valores de los componentes, especialmente significativo en los casos en que el individuo es relativamente complejo.

En el comienzo del proceso evolutivo suele producirse un cambio abrupto, producto de la relativa facilidad con la que puede mejorarse una población generada en forma aleatoria.

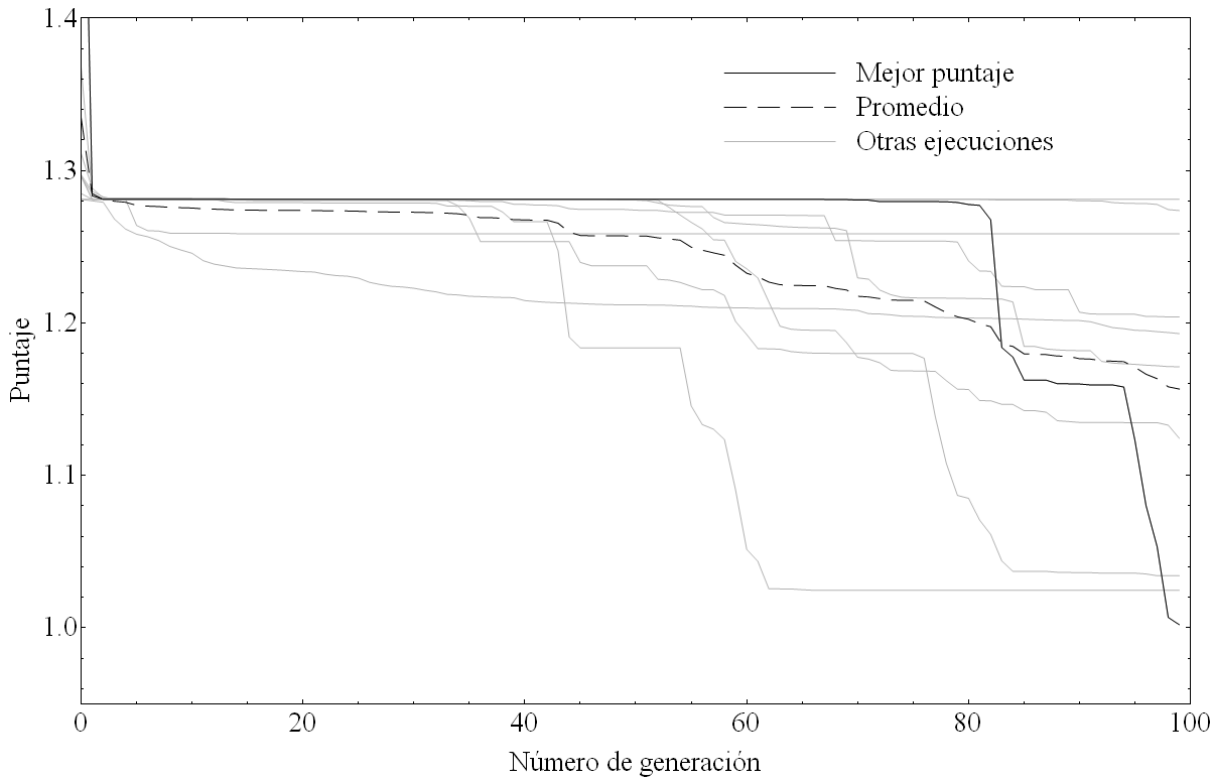


Figura 5.4. Evolución del puntaje a lo largo del proceso evolutivo en las distintas ejecuciones del caso base.

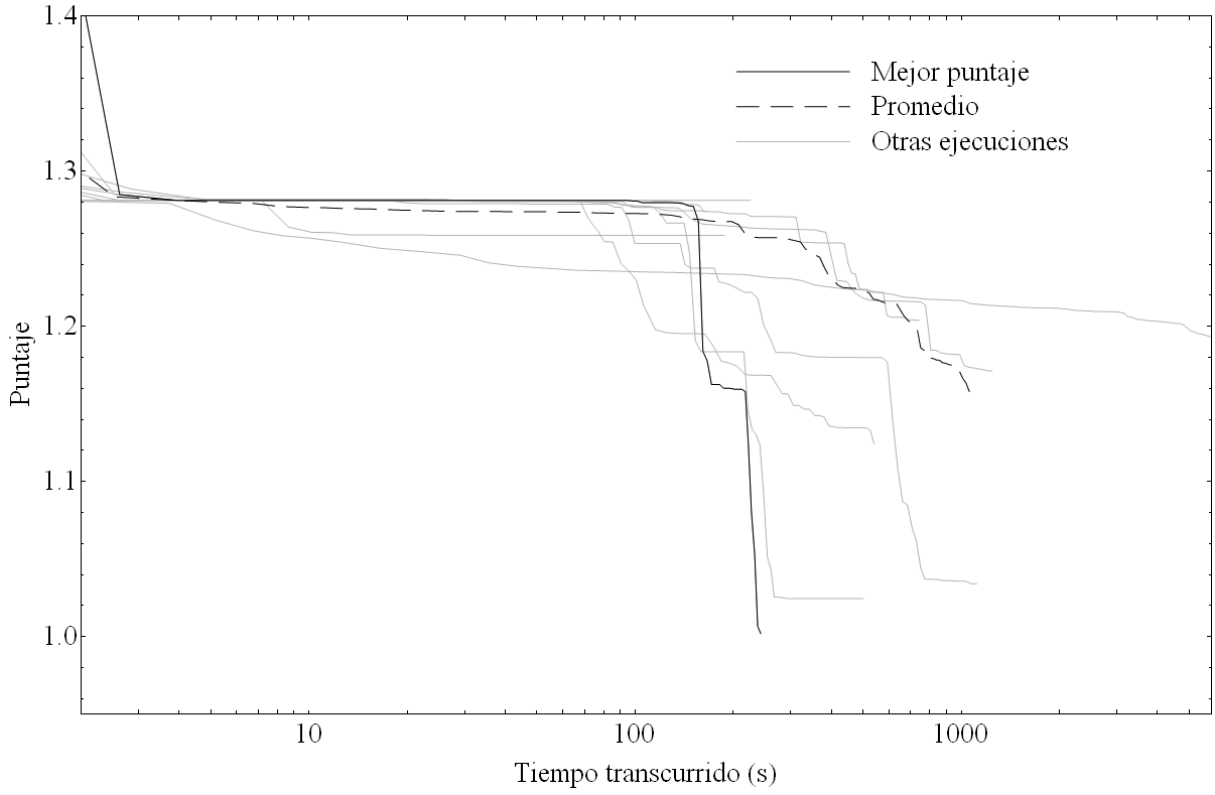


Figura 5.5. Evolución del puntaje respecto al tiempo en las distintas ejecuciones del caso base.

5.3.2. APLICANDO PLAGAS PARA CONTROLAR EL CRECIMIENTO DEL ESFUERZO COMPUTACIONAL

Para la aplicación de plagas se eligió en forma empírica eliminar a los 10 individuos de menor puntaje en cada generación, manteniendo un mínimo de un individuo. Puede observarse que esto detiene completamente el proceso evolutivo al realizar 100 recambios generacionales, ya que el único individuo restante es seleccionado automáticamente en los recambios posteriores (ver sección 5.1).

En las figuras 5.6 y 5.7 se observa la variación del puntaje respecto al número de generación y al tiempo transcurrido. Puede apreciarse la detención casi completa del proceso evolutivo luego de la generación 50 y la concentración de la reducción del puntaje en las primeras generaciones, en marcado contraste con lo que sucede en el caso base. Esto es debido a que la disminución en el tamaño de las poblaciones, si bien acelera la evaluación de las mismas, termina reduciendo la diversidad genética e impidiendo el surgimiento de nuevos diseños.

No obstante este problema, la disminución del esfuerzo computacional compensa con sobras esta pérdida de diversidad (ver sección 5.3.4.4).

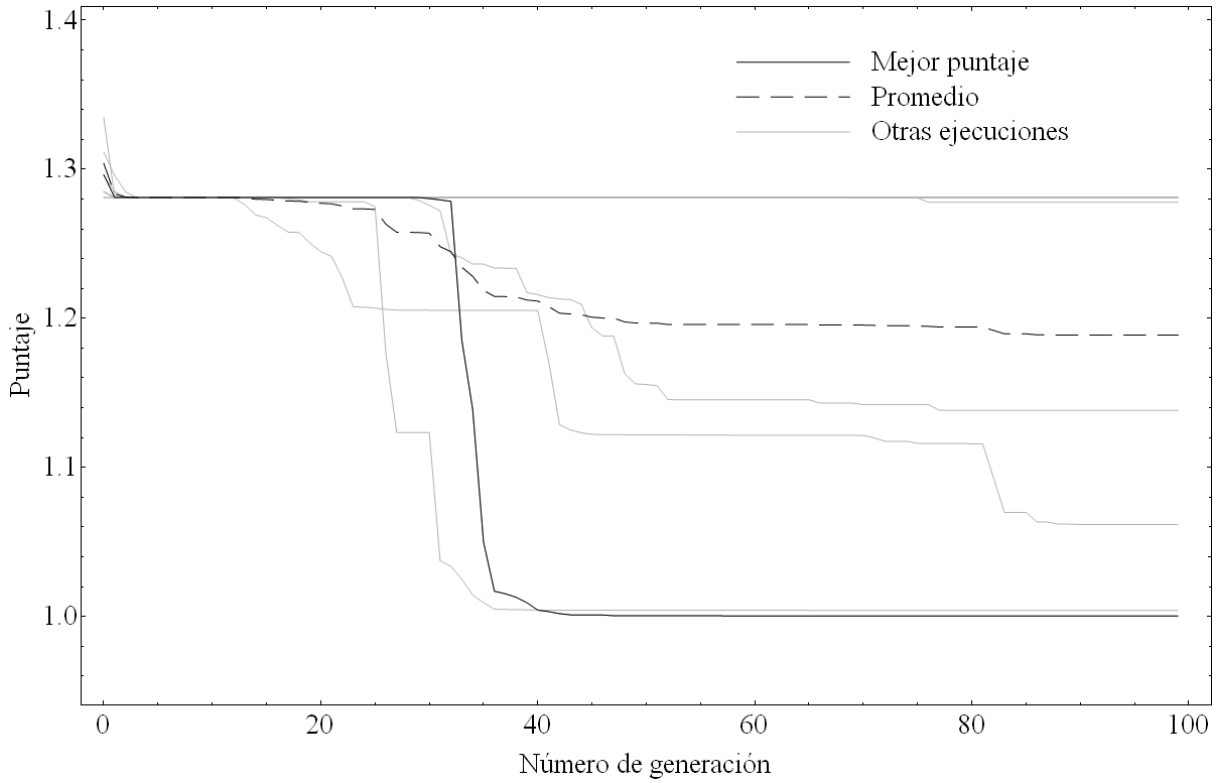


Figura 5.6. Evolución del puntaje a lo largo del proceso evolutivo en las distintas ejecuciones realizadas aplicando plagas.

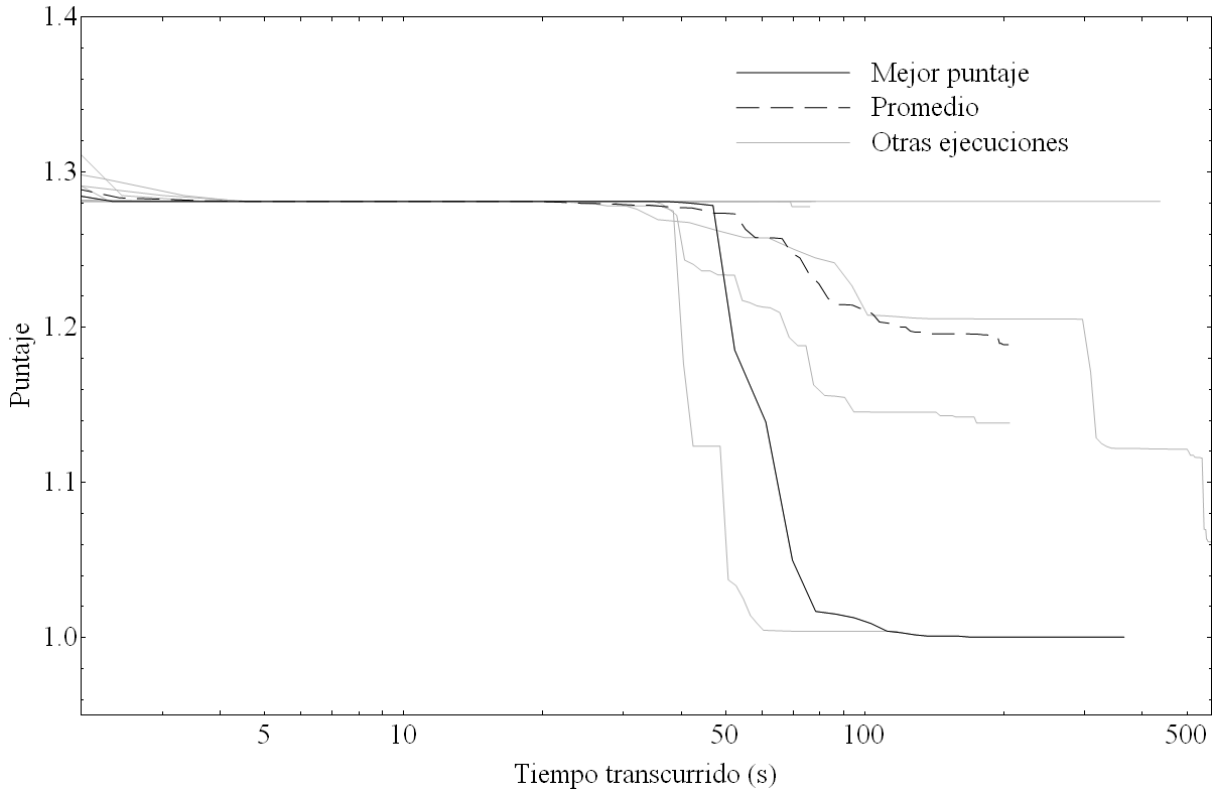


Figura 5.7. Evolución del puntaje respecto al tiempo en las distintas ejecuciones realizadas aplicando plagas.

5.3.3. REALIZANDO UN AJUSTE DINÁMICO DE LA FUNCIÓN DE EVALUACIÓN

Para esta técnica se determinó en forma empírica la utilización de una probabilidad de ejecución de 0.2 para los individuos de tamaño mayor que el promedio. La utilización de probabilidades menores producía efectos menos observables en el desarrollo, mientras que el uso de probabilidades más grandes tendía a impedir el desarrollo del proceso evolutivo, dejándolo atrapado en un mínimo local.

En las figuras 5.8 y 5.9 puede apreciarse la evolución del puntaje aplicando esta técnica, que se realiza en forma bastante similar a la observada en el caso base.

5.3.4. APLICANDO UN CACHÉ DE EVALUACIÓN

En esta sección se analizan los resultados de aplicar un caché de evaluación: tanto dando una descripción general de los parámetros utilizados (sección 5.3.4.1); como aplicándolo por separado sobre el caso base (sección 5.3.4.2) y utilizándolo en combinación con las técnicas de plagas para reducir el esfuerzo computacional (sección 5.3.4.2) y del ajuste dinámico de la función de evaluación (sección 5.3.4.4).

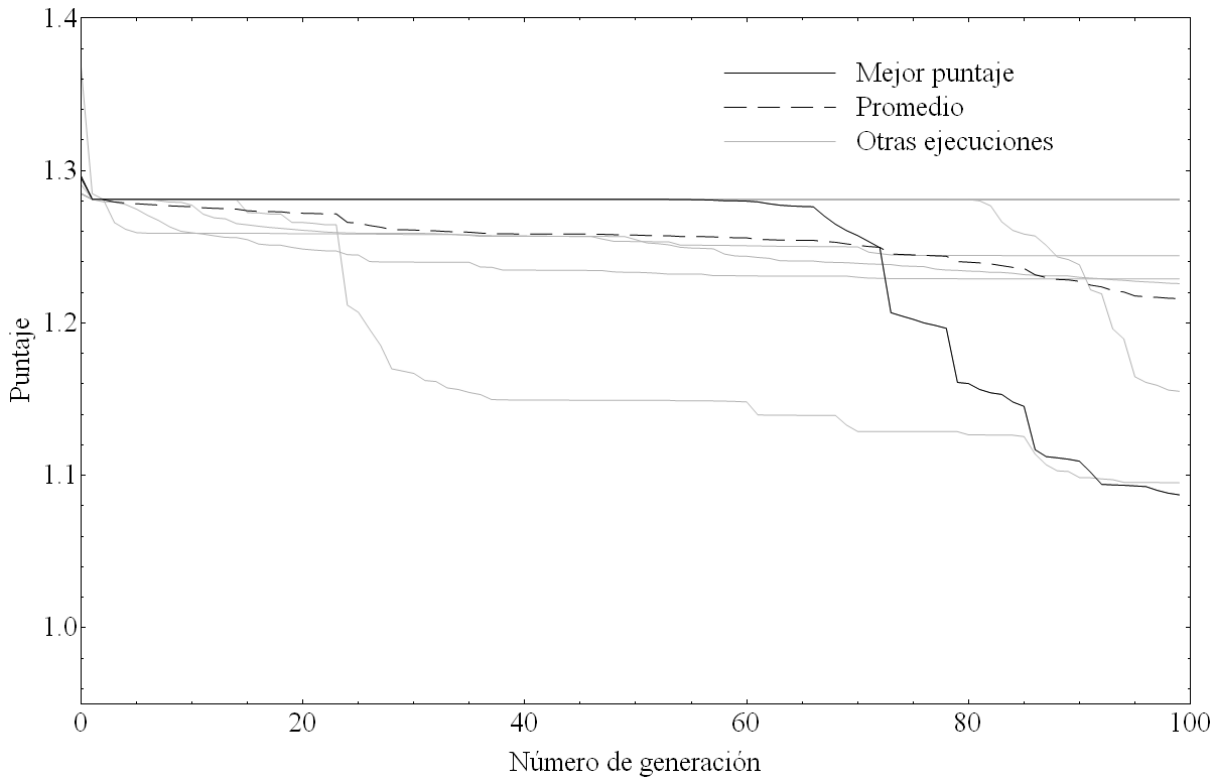


Figura 5.8. Evolución del puntaje a lo largo del proceso evolutivo en las distintas ejecuciones realizadas aplicando el ajuste dinámico de la función de evaluación.

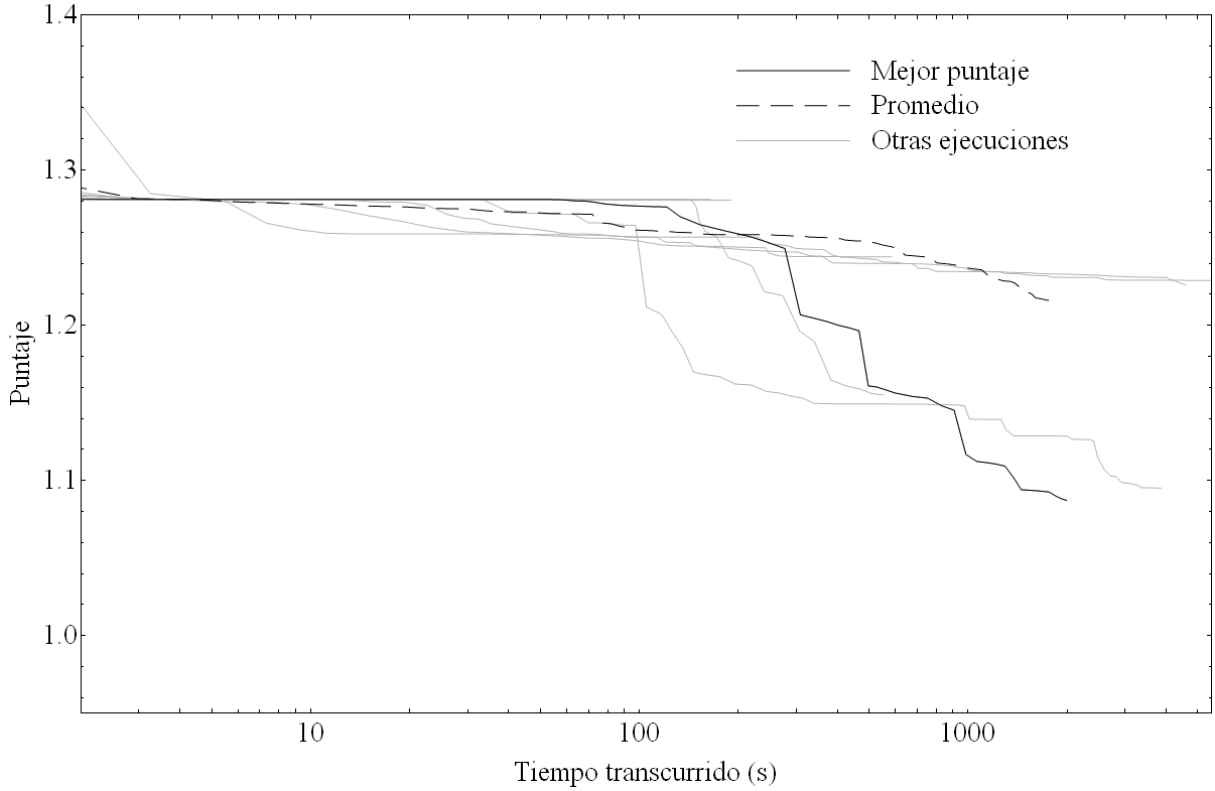


Figura 5.9. Evolución del puntaje respecto al tiempo en las distintas ejecuciones realizadas aplicando el ajuste dinámico de la función de evaluación.

5.3.4.1. Generalidades

Para todas las pruebas se utilizó un caché con 2^{20} elementos que, considerando que se utilizan 8 bytes para el hash del individuo y otros 8 bytes para el puntaje, representan un consumo de 16 MiB [IEC, 2008] de RAM para este fin. Este valor fue seleccionado debido a que, al trabajar con un máximo de 100000 individuos distintos, no existían ventajas significativas en utilizar un caché de tamaño mayor.

La utilización de un caché de evaluación no afecta todas las evaluaciones por igual, solo reduciendo los tiempos de evaluación de aquellos individuos que ya han sido evaluados anteriormente. Esto acelera especialmente los "puntos muertos" del desarrollo evolutivo, ya que en estos casos la población suele presentar poca variación. No cabe esperar que tenga un efecto significativo en la optimalidad de los resultados obtenidos en un número dado de generaciones pero, al disminuir los tiempos de evaluación, se posibilita la utilización de mayores poblaciones o un mayor número de generaciones con la misma carga computacional.

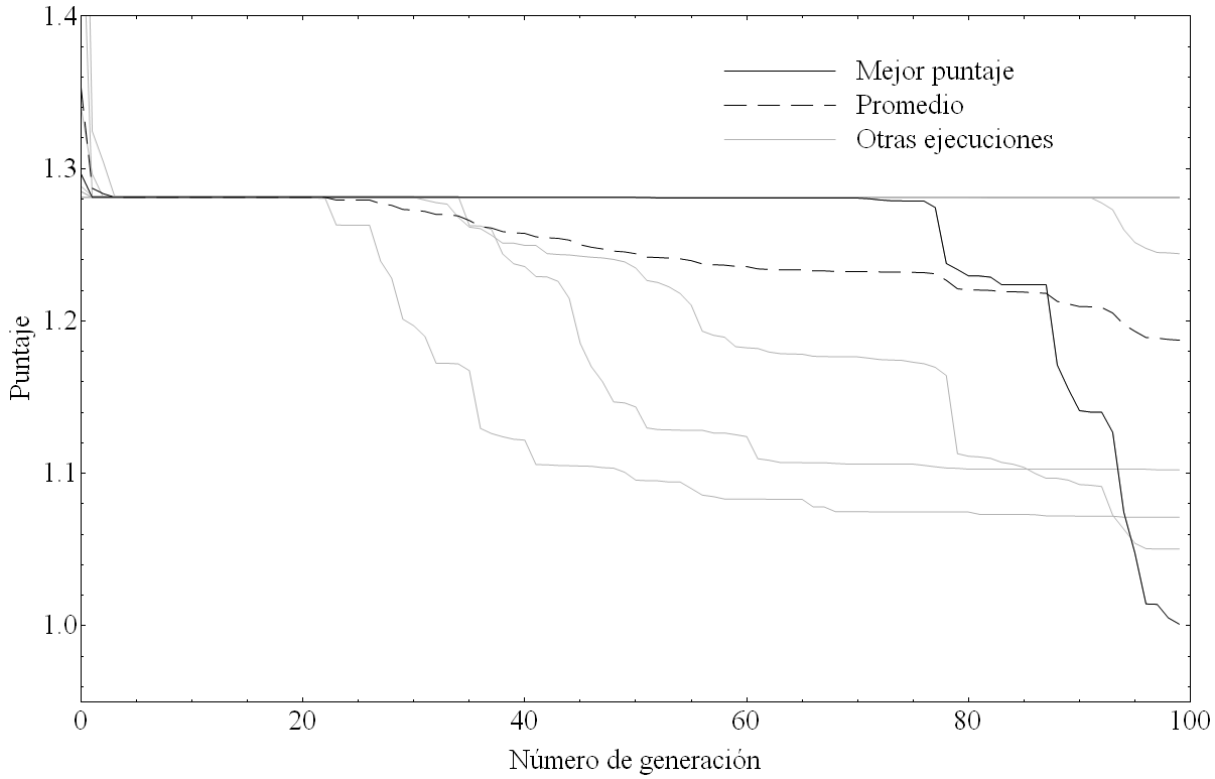


Figura 5.10. Evolución del puntaje a lo largo del proceso evolutivo en las distintas ejecuciones del caso base que se realizaron aplicando solo el caché de evaluación.

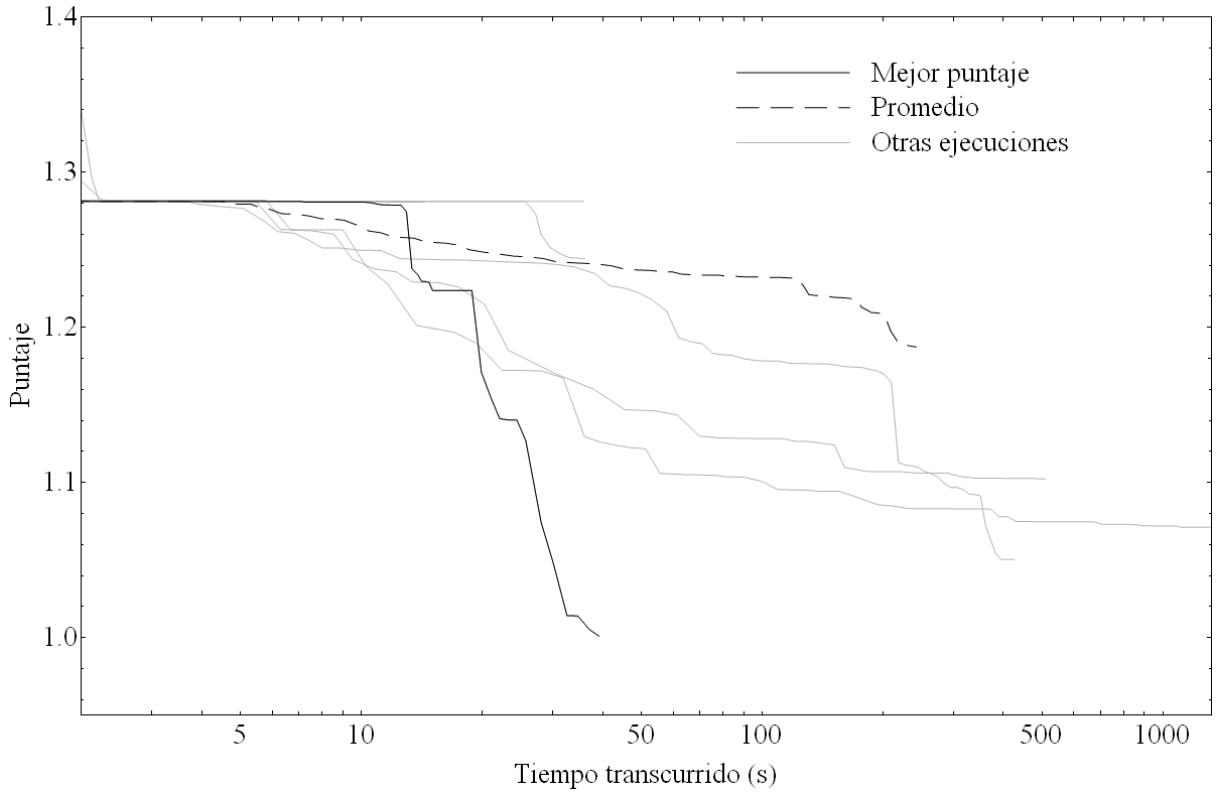


Figura 5.11. Evolución del puntaje respecto al tiempo en las distintas ejecuciones del caso base que se realizaron aplicando solo el caché de evaluación.

5.3.4.2. Sobre el caso base

En las figuras 5.10 y 5.11 puede apreciarse un comportamiento muy similar en apariencia al que se observó sin aplicar el caché de evaluación. La diferencia notable se presenta en la escala de tiempos: mientras la ejecución promedio sin aplicar el caché demoraba unos 1000 segundos, aplicando el caché de evaluación esto se reduce a unos 300, una reducción del 70%. Los puntajes evolucionan de manera similar en ambos casos, sin que se aprecie ninguna reducción en la calidad de los resultados obtenidos.

5.3.4.3. En combinación con el uso de plagas

En las figuras 5.12 y 5.13 puede observarse el desempeño de la aplicación de plagas en conjunto con el caché de evaluación. Al igual que cuando se aplican solamente las plagas, se observa el patrón de un rápido cambio inicial, seguido por un estancamiento al reducirse la población a un número insuficiente para proseguir el proceso evolutivo. Se observan tiempos de ejecución extremadamente cortos, alcanzándose el menor puntaje en una ejecución de menos de 40 segundos.

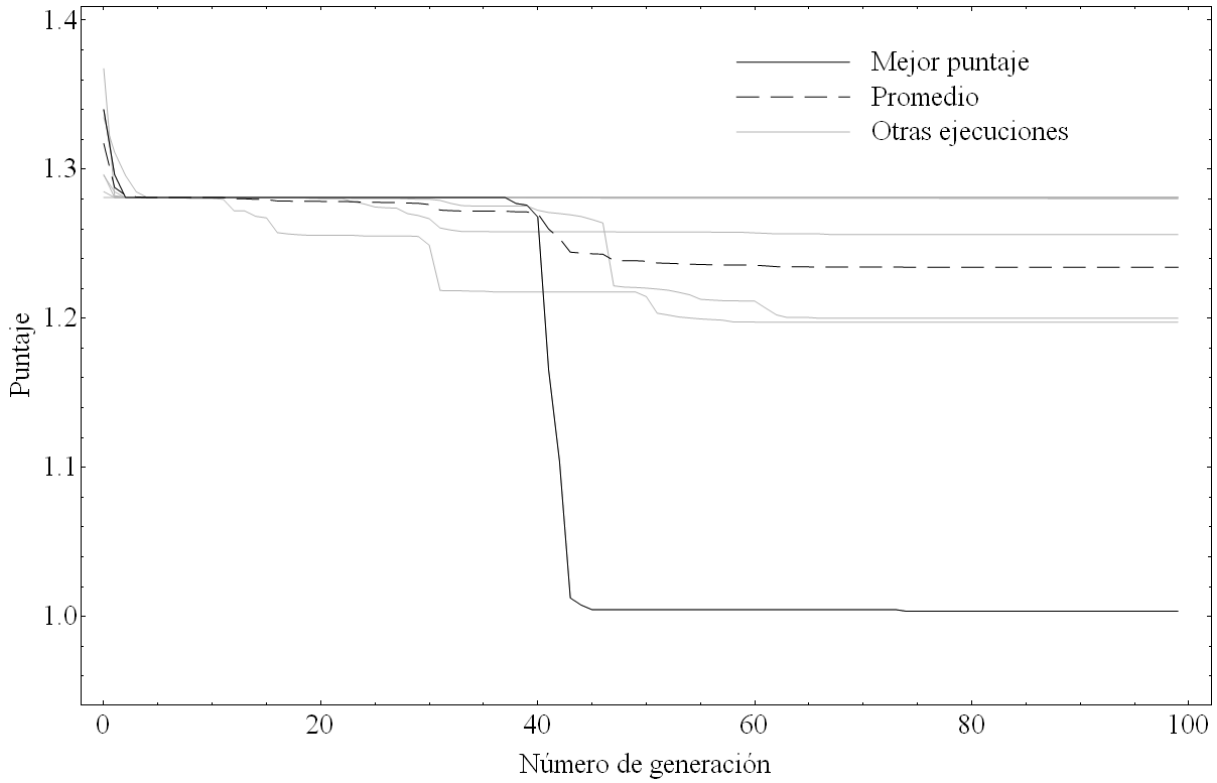


Figura 5.12. Evolución del puntaje a lo largo del proceso evolutivo en las distintas ejecuciones que se realizaron aplicando en forma conjunta el uso de plagas y el caché de evaluación.

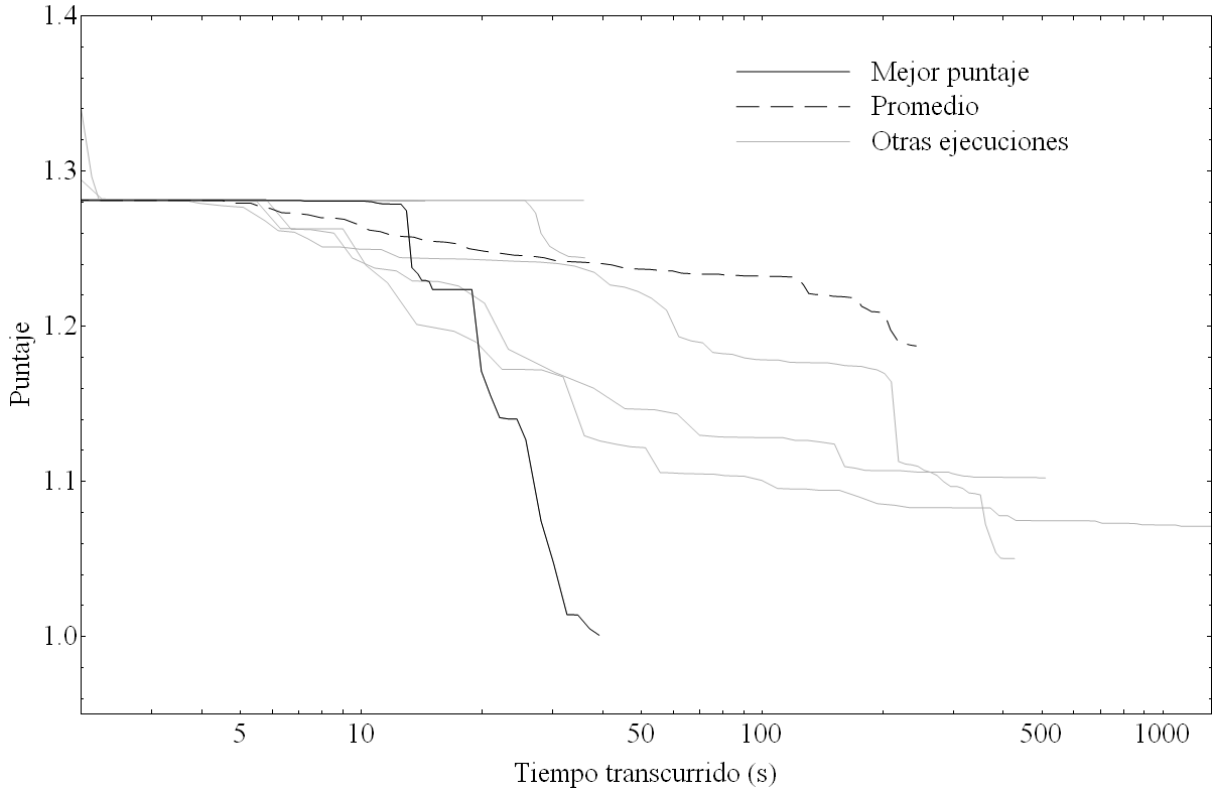


Figura 5.13. Evolución del puntaje respecto al tiempo en las distintas ejecuciones que se realizaron aplicando en forma conjunta el uso de plagas y el caché de evaluación.

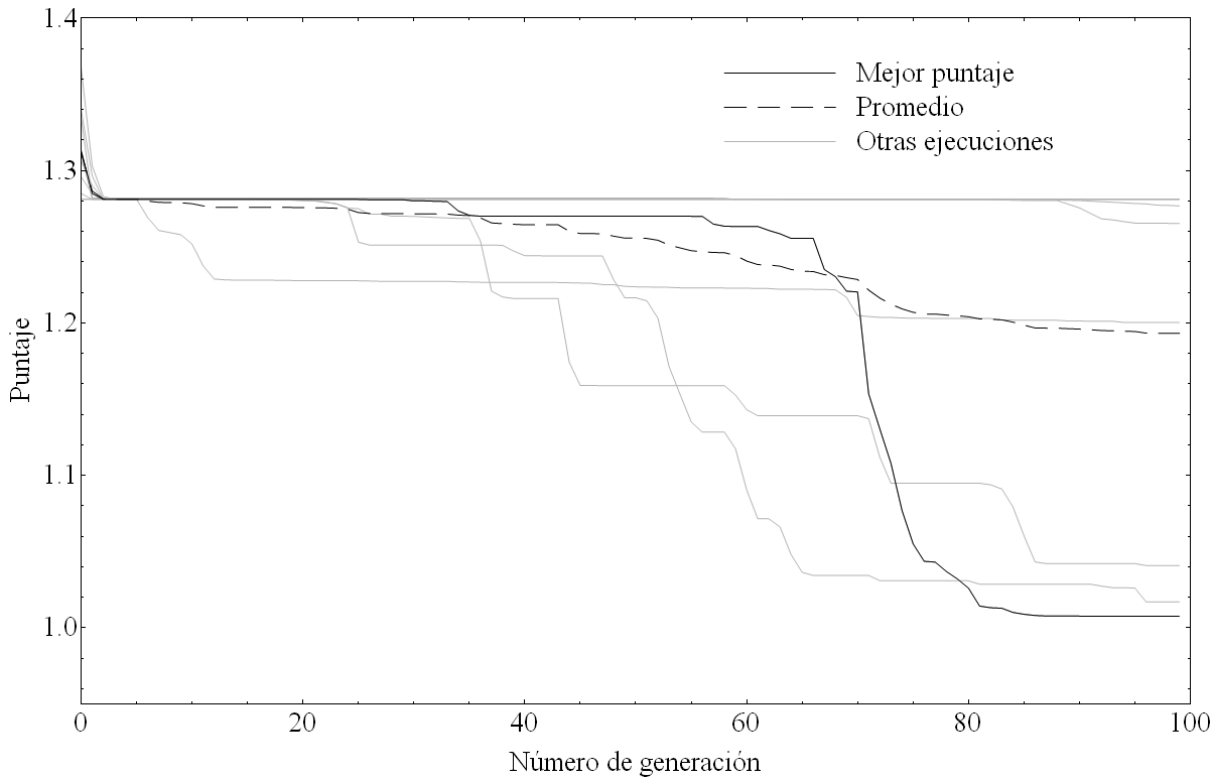


Figura 5.14. Evolución del puntaje a lo largo del proceso evolutivo en las distintas ejecuciones que se realizaron aplicando en forma conjunta el ajuste dinámico de la función de evaluación y el caché de esta misma función.

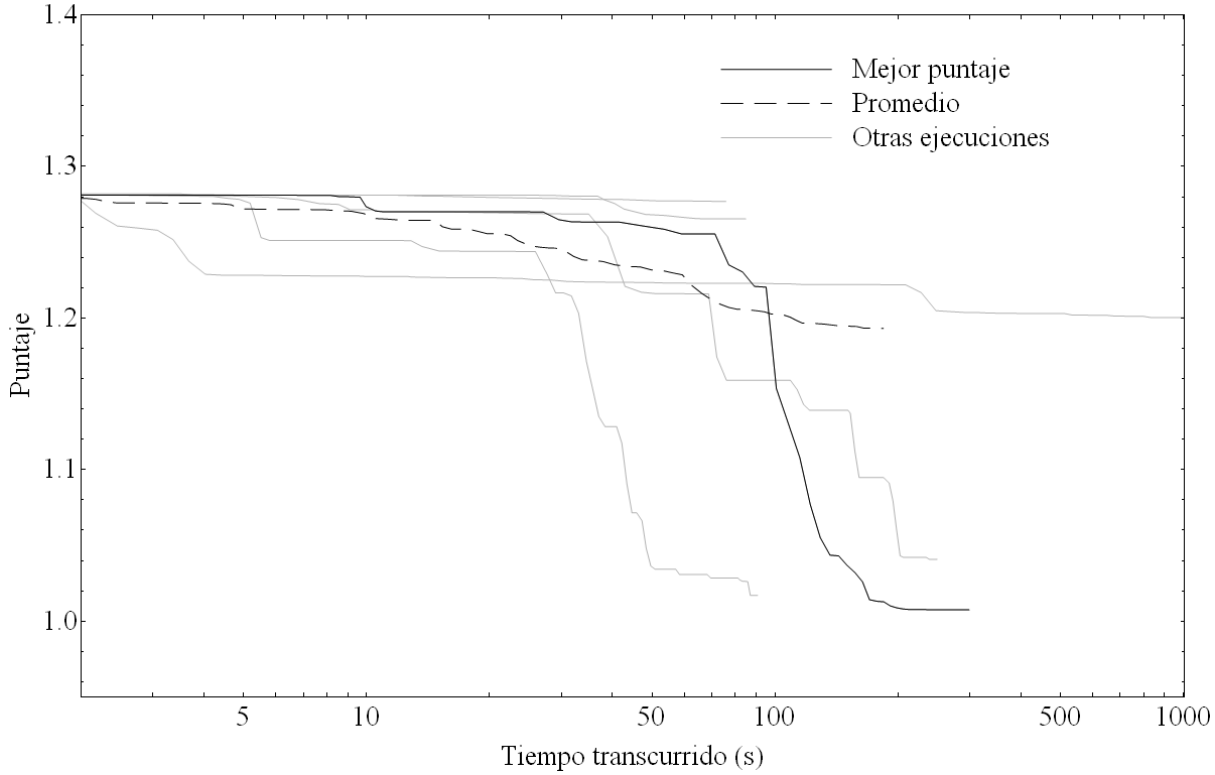


Figura 5.15. Evolución del puntaje respecto al tiempo en las distintas ejecuciones que se realizaron aplicando en forma conjunta el ajuste dinámico de la función de evaluación y el caché de esta misma función.

5.3.4.4. En combinación con el ajuste dinámico de la función de evaluación

En las figuras 5.14 y 5.15 puede observarse el desarrollo del proceso evolutivo. Al igual que en el caso base, la utilización de un caché de evaluación solo produce cambios significativos en los tiempos de ejecución, presentando la evolución de los puntajes una forma similar a la obtenida sin el caché.

Si bien los resultados obtenidos en este caso son algo superiores a los obtenidos por el caso base utilizando el caché, la diferencia no es muy significativa como se verá con mayor detalle en la sección 5.3.4.4.

5.4. COMPARACIÓN ENTRE LOS RESULTADOS OBTENIDOS AL APLICAR LAS DISTINTAS TÉCNICAS

La figura 5.16 resume los resultados expuestos en la sección anterior. Se marcan en gris los resultados obtenidos sin utilizar el caché de evaluación y en negro los obtenidos aplicando el mismo. Los símbolos se ubican en los valores promedio, tanto de la duración de una ejecución como del puntaje obtenidos al aplicar esa técnica o combinación de técnicas. Para el caso de los tiempos se utilizó una media geométrica, en lugar de la simple media aritmética, por los motivos indicados con mayor detalle en la sección anexa G.1. Las líneas indican la distancia desde los valores promedios hasta los valores extremos observados en cada una de las dimensiones.

En ninguno de los casos se observó una variación significativa en los puntajes obtenidos; la característica más visible en la figura 5.16, es la gran influencia que presenta el uso del caché de evaluación en los resultados. En el caso de la técnica de plagas, que presenta el mejor resultado sin utilizar el caché, la aplicación del mismo reduce tanto el tiempo promedio como el mínimo en casi un orden de magnitud. Si combinamos esto con la notable diferencia ya existente entre la técnica de utilización de plagas para reducir el esfuerzo computacional y el caso base, tenemos una diferencia de un factor 30 en el tiempo de ejecución entre la combinación más efectiva de técnicas estudiada y el caso base. El único costo extra que se incurre en cuanto a recursos computacionales es el almacenamiento para el caché, pero al presentar un costo de solo 8 bytes por resultado almacenado, no constituye un factor limitante del desempeño.

La técnica de ajuste dinámico de la función de evaluación presentó un desempeño muy similar al del caso base, siendo algo superior cuando se utilizó junto al caché y algo inferior cuando se utilizó sin el mismo. Estas variaciones no son estadísticamente significativas, como se indica más adelante.

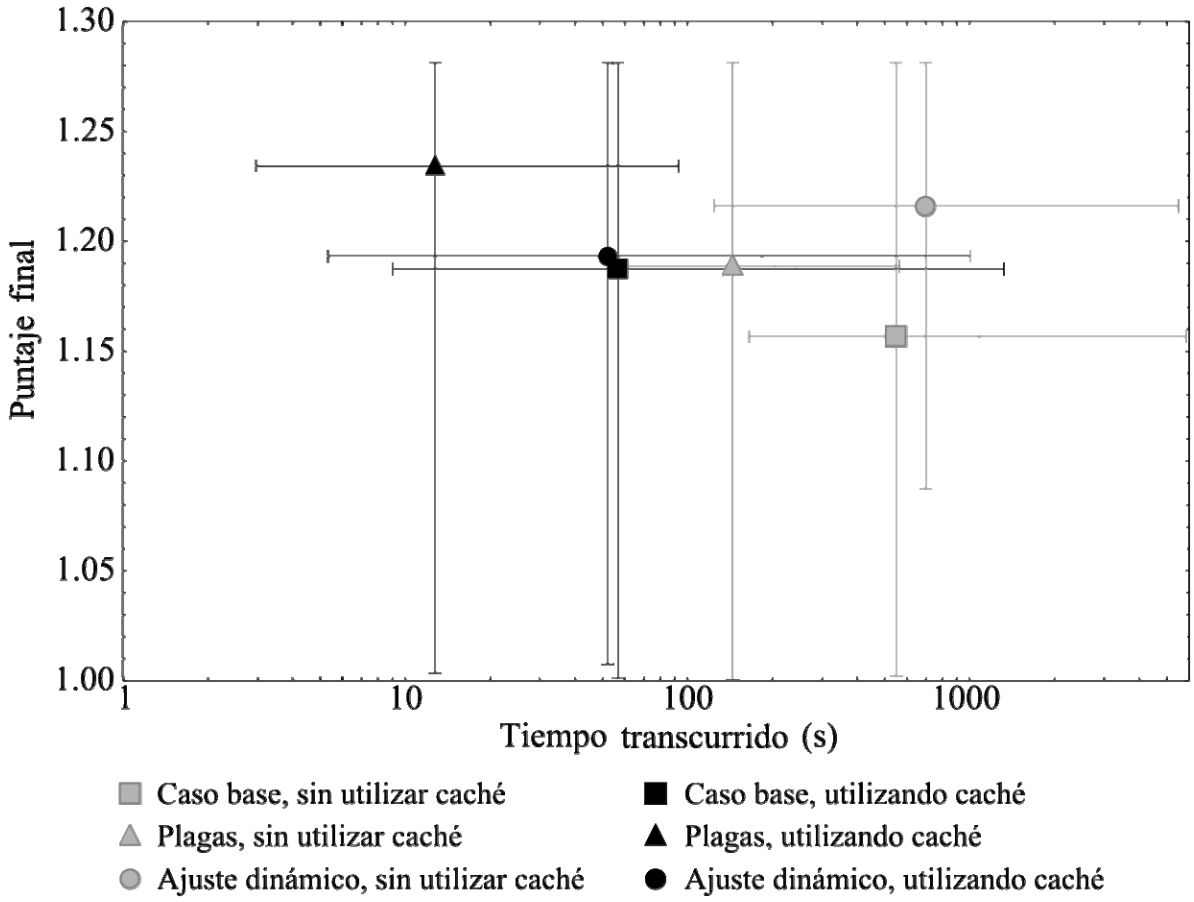


Figura 5.16. Gráfico comparativo mostrando los resultados de la aplicación de cada una de las técnicas.

Para poder apreciar si los resultados son estadísticamente significativos, se determinaron intervalos de confianza para los resultados obtenidos (ver sección anexa G.2), que pueden apreciarse en la figura 5.17. Como las diferencias en cuestión están dadas por factores multiplicativos, estos intervalos se determinaron sobre los logaritmos de los tiempos en lugar de los tiempos en si.

En primer lugar puede observarse que la técnica de ajuste dinámico no presentó en este caso una ventaja significativa sobre el caso base, tanto utilizando como no el caché de evaluación. Mientras que en todos los casos la utilización del caché de evaluación representó una disminución significativa de los tiempos de evaluación. Solo puede decirse que la mejora observada al utilizar plagas es estadísticamente significativa si se opera con un nivel de confianza del 90 % y aun así solo en el caso en que no se trabaja con el caché de evaluación.

5.5. EJEMPLOS DE LOS RESULTADOS OBTENIDOS Y COMPARACIÓN CON OTROS TRABAJOS REALIZADOS

En la figura 5.18 puede observarse el filtro obtenido durante el caso de prueba de las ejecuciones utilizando plagas para reducir el esfuerzo computacional en combinación con el caché de

evaluación. En la figura 5.19 puede observarse la transferencia de este mismo filtro y, si se compara con la transferencia del pasabajos ideal buscada, se llega a un puntaje de 1.00349.

Este filtro fue obtenido en la séptima de las diez ejecuciones realizadas con esta combinación de técnicas. Esa ejecución en particular demandó 12.4 segundos, mientras que las diez ejecuciones en total demandaron 280 segundos. Con objeto de dar una idea muy aproximada del rendimiento comparativo, puede indicarse que las ejecuciones se llevaron a cabo en una computadora con un microprocesador AMD Sempron 2800+ a 1.6 GHz con 512 MB de RAM.

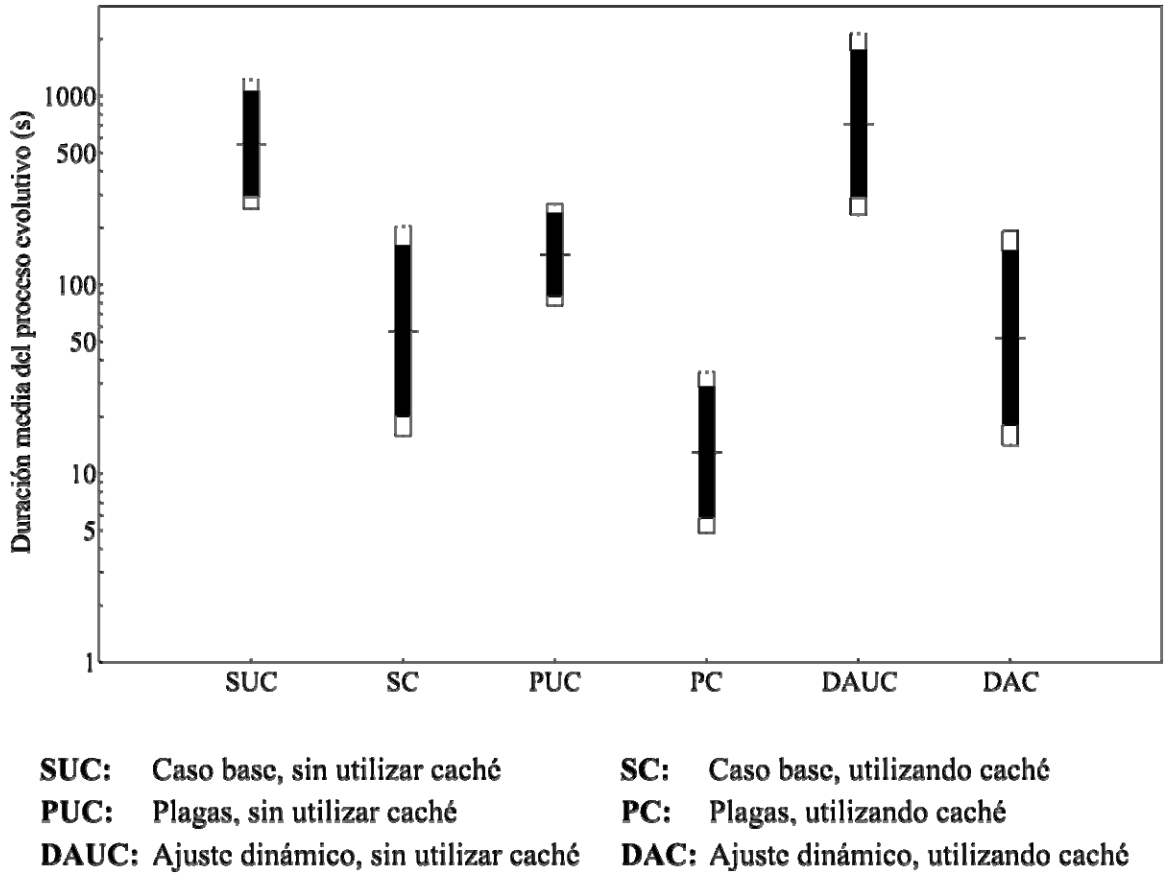


Figura 5.17. Intervalos de confianza para las duraciones medias (geométricas) de los procesos evolutivos, obtenidos aplicando distintas combinaciones de las técnicas de optimización. Los rectángulos blancos representan intervalos de 95 % de confianza, mientras que los negros representan intervalos de 90 % de confianza.

Es difícil comparar los resultados obtenidos directamente con los que se encuentran en [Koza *et al.*, 2003] por la disparidad de recursos computacionales empleados. Pero si podrían efectuarse comparaciones con uno de los resultados que aparecen en el trabajo original presentado en [Koza *et al.*, 1997], que no utiliza componentes activos.

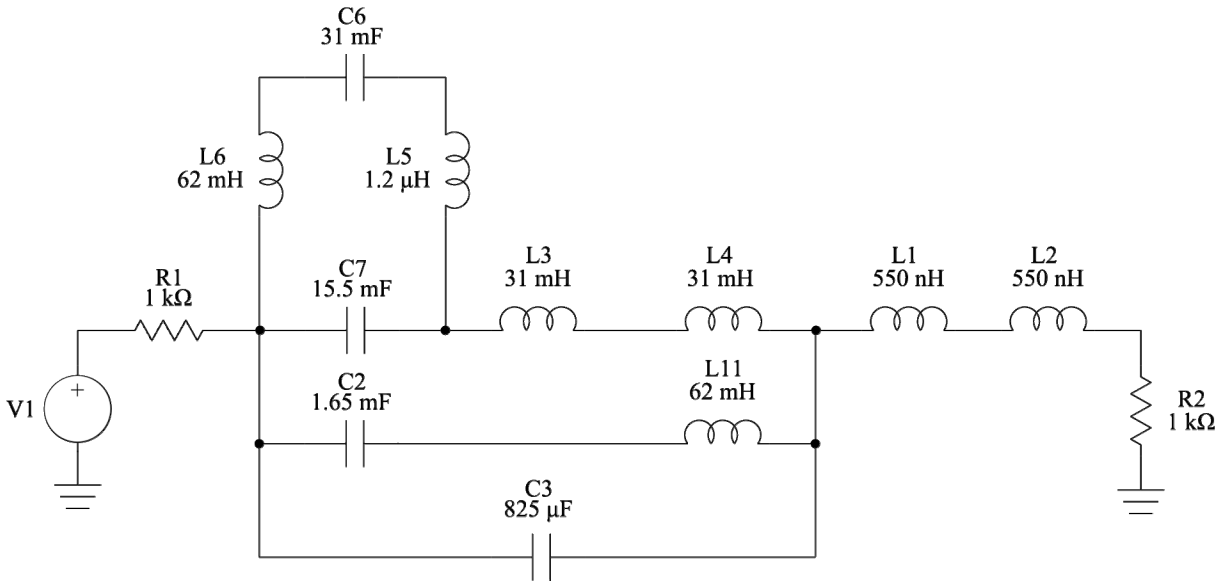


Figura 5.18. Filtro obtenido durante las ejecuciones que se realizaron aplicando plagas para reducir el esfuerzo computacional.

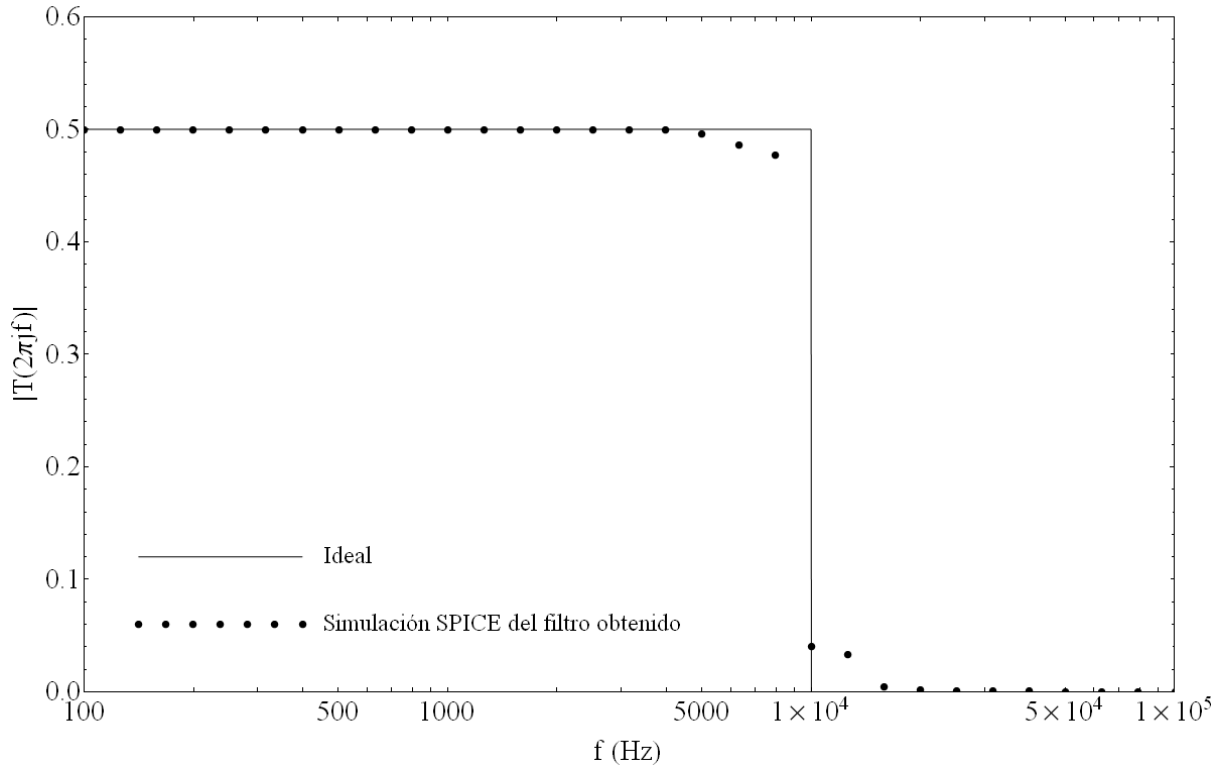


Figura 5.19. Transferencia del filtro de la figura 5.18.

Si, utilizando el método estimativo de Koza, se mide el esfuerzo computacional realizado en ciclos, puede estimarse que el resultado de las figuras 5.18 y 5.19 demandó unos $4.48 \cdot 10^{11}$ ciclos de reloj del microprocesador, o sea un valor varios órdenes de magnitud inferior al requerido por los circuitos desarrollados en este trabajo mencionado (que demandaron casi un petaflop de

procesamiento cada uno). Esto sugiere que (a pesar de posibles diferencias en la calidad del proceso de evaluación realizado y, por consiguiente, en los resultados obtenidos) existe una mejora notable en la eficiencia computacional al aplicar las técnicas de optimización propuestas.

6. CONCLUSIONES

Se observó una disminución significativa de los tiempos de ejecución mediante el uso de un caché *hash* para acelerar la evaluación. La utilización de plagas para reducir el esfuerzo computacional también determinó reducciones significativas aunque inferiores a las observadas con el uso de un caché. En conjunto, ambas técnicas aceleraron los tiempos medios de ejecución observados en más de un orden de magnitud, sin que la calidad de los resultados obtenidos disminuyera en forma apreciable.

Los resultados obtenidos también demostraron ser claramente competitivos con los observados en otros trabajos realizados en el área, aunque limitaciones computacionales y en el conjunto de funciones de construcción implementados impidieron una comparación directa con los circuitos utilizando componentes activos descritos en [Koza *et al.*, 2003].

No obstante los resultados obtenidos, existen muchas posibles líneas de investigación abiertas: por ejemplo, podría evaluarse el impacto de utilizar un conjunto más completo de funciones de construcción, permitiendo la creación de circuitos activos. Asimismo, podría explorarse con mayor detalle el impacto de variaciones en las probabilidades de mutación y cruza, ya que solo fueron seleccionados para el caso de la ejecución realizada sin aplicar técnicas de optimización.

Una limitación a tener en cuenta de los resultados obtenidos es que fueron obtenidos en base a la evaluación en un número finito de puntos, distribuidos logarítmicamente en el intervalo en cuestión. Para tener resultados más confiables, sería deseable modificar el simulador SPICE para que permita realizar la evaluación del circuito en un conjunto de puntos dado por el usuario. Esto permitiría concentrar la evaluación en los puntos de rápido cambio, evitando malgastar recursos computacionales en áreas “poco interesantes” de la transferencia. Otra modificación al simulador que podría realizarse consistiría en hacer que acepte directamente una representación binaria del circuito, lo que probablemente aceleraría notablemente el proceso de evaluación.

Otra variante más en el nivel de la implementación que podría explorarse, es el impacto de estas optimizaciones en los sistemas distribuidos de programación genética, esenciales para asegurar la escalabilidad de esta técnica. Muchas de las técnicas desarrolladas para poblaciones únicas podrían requerir adaptaciones para aplicarse más efectivamente en este entorno.

7. REFERENCIAS

Aaronson, S. (2004). *Limits on Efficient Computation in the Physical World.* Tesis doctoral. University of California, Berkeley.

Agarwal, A., Lang, J. (2005). *Foundations of Analog and Digital Electronic Circuits*. Morgan Kauffman.

Alvarez, G. (2002). *Implementing non-stationary filtering in time and in Fourier domain.* http://sepwww.stanford.edu/public/docs/sep111/gabriel1/paper_html/. Último acceso: 29 de julio de 2008.

Appleby, A. (2008). *MurmurHash 2.0*. http://murmurhash.googlepages.com/. Último acceso: 10 de junio de 2008.

Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J. , Williams, S. W., Yelick, K. A. (2006). *The Landscape of Parallel Computing Research: A View from Berkeley*. Informe técnico UCB/EECS-2006-183. EECS Department, University of California, Berkeley.

Banzhaf, W., Francone, F., Keller, R., Nordin, P. (1998). *Genetic programming: an introduction: on the automatic evolution of computer programs and its applications.* Morgan Kaufmann Publishers.

Brameier, M. (2004) *On Linear Genetic Programming*. Tesis doctoral. Universität Dortmund, Dortmund, Germany.

Cederbaum, I. (1984). *Some Applications of Graph Theory to Network Analysis and Síntesis*. IEEE Transactions on Circuits and Systems 31(1): 64-68.

Chouza, M. (2007). *Aproximación de números reales utilizando programación genética*. http://www.chouza.com.ar/gpjs_ej1. Último acceso: 20 de julio de 2008.

Chouza, M. (2008). *Creación de factories con registro automático en C++*. http://www.chouza.com.ar/auto_reg_cpp. Último acceso: 18 de agosto de 2008.

Daum, F. (2005). *Nonlinear Filters: beyond the Kalman filter*. IEEE Aerospace and Electronics Systems Magazine 20(8): 57-69.

Dawkins, R. (1986). *The Blind Watchmaker*. Penguin.

Fan, Z., Hu, J., Seo, K., Goodman, E., Rosenberg, R., Zhang, B. (2001). *Bond Graph Representation and GP for Automated Analog Filter Design*. 2001 Genetic and Evolutionary Computation Conference Late-Breaking Papers. ISGEC Press.

Fernandez, F., Vanneschi, L., Tomassini, M. (2003). *The Effect of Plagues in Genetic Programming: A Study of Variable-Size Populations.* Lectures Notes in Computer Science 2610: 317-326.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.

Harjani, R. (1989). *OASYS: a framework for analog circuit synthesis*. Proceedings of the Second Annual IEEE ASIC Seminar and Exhibit: P13-1/1-4.

Holland, J. (1992). *Adaptation in Natural and Artificial Systems.* MIT Press.

Hornby, G., Globus, A., Linden, D., Lohn, J. (2006). *Automated Antenna Design with Evolutionary Algorithms*. Proceedings of the 2006 AIAA Space Conference.

Hsieh, P. (2007). *Hash functions*. http://www.azillionmonkeys.com/qed/hash.html. Último acceso: 10 de junio de 2008.

Hu, J., Zhong, X., Goodman, E. (2005). *Open-ended Robust Design of Analog Filters Using Genetic Programming*. Proc. 2005 Genetic and Evolutionary Computation Conference. ACM.

IEC (2008). *Prefixes for binary multiples*. http://www.iec.ch/zone/si/si_bytes.htm. Último acceso: 10 de junio de 2008.

ISO/IEC 14977 (1996). *Information technology – Syntactic metalanguage - Extended BNF*. International Organization for Standardization.

Karlsson, B. (2005). *Beyond the C++ Standard Library: An Introduction to Boost*. Addison Wesley.

Kielkowski, R. (1998). *Inside SPICE* (2nd ed.). McGraw-Hill.

Knuth, D. (1998). *The Art of Computer Programming*. Vol. 3: Sorting and Searching. Addison Wesley.

Koller, R., Wilamowski, B. (1996). *LADDER. A Microcomputer Tool for Passive Filter Design and Simulation*. IEEE Transactions on Education 39(4): 478-487.

Koza, J. (1989). *Hierarchical genetic algorithms operating on populations of computer programs*. En Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89: 768-774.

Koza, J. (1990). *Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems*. Reporte STAN-CS-90-1314. Department of Computer Science, Stanford University.

Koza, J. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press.

Koza, J. (1995). *Survey of genetic algorithms and genetic programming.* Proceedings of 1995 WESCON Conference: 589-594.

Koza, J., Bennett, F., Andre, D., Keane, M., Dunlap, F. (1997). *Automated Synthesis of Analog Electrical Circuits by Means of Genetic Programming*. IEEE Transactions on Evolutionary Computations 1(2): 109-128.

Koza, J., Keane, M., Streeter, M., Mydlowec, W., Yu, J., Lanza, G. (2003). *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Springer.

Li, M., Vitányi, P. (1997). *An introduction to Kolmogorov complexity and its applications.* Springer-Verlag.

Lohn, J., Colombano, S. (1998). *Automated Analog Circuit Synthesis using a Linear Representation*. Proc. of the Second Int'l Conf on Evolvable Systems: 125-133.

Luke, S. (2000). *Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat*. Tesis doctoral. University of Maryland.

MacKay, D. (2002). *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press.

Mason, R., Gunst, R., Hess, J. (2003). *Statistical Design and Analysis of Experiments.* John Wiley & Sons.

McCarthy, J. (1960). *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I*. Communications of the ACM 3(4): 184-195.

Mitchell, M. (1998). *An Introduction to Genetic Algorithms.* MIT Press.

Neumaier, A. (2004). *Complete Search in Continuous Global Optimization and Constraint Satisfaction*. Acta Numerica 2004. Cambridge University Press.

NIST (2002). *Secure Hash Standard. FIPS 180-2.* http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf. Último acceso: 10 de julio de 2008.

Noll, L. (2003). *Fowler / Noll / Vo (FNV) Hash*. http://www.isthe.com/chongo/tech/comp/fnv/. Último acceso: 10 de junio de 2008.

Nordin, P., Banzhaf, W., Francone, F. (1999). *Efficient Evolution of Machine Code for CISC Architectures using Blocks and Homologous Crossover.* En *Advances in Genetic Programming III*: 275-299. MIT Press.

Novak, G. (2005). *CS 394P: Automatic Programming: Lecture Notes*. University of Texas. http://www.cs.utexas.edu/~novak/cs394pcontents.html. Último acceso: 29 de julio de 2008.

Nuhertz Technologies. (2008). *Passive Filters Solutions*. http://www.filter-solutions.com/passive.html. Último acceso: 20 de abril de 2008.

van Oorschot, P., Wiener, M. (1994). *Parallel collision search with application to hash functions and discrete logarithms.* Proceedings of the 2nd ACM Conference on Computer and communications security: 210-218.

Oita University. (2008). *Facilities*. http://www.hwe.oita-u.ac.jp/mc/eng/facilities.htm. Último acceso: 30 de julio de 2008.

Oppenheim, A., Willsky, A., Nawab, S. H. (1996). *Signals and Systems* (2nd ed). Prentice-Hall.

Paarmann, L. (2003). *Design and Analysis of Analog Filters*. Kluwer Academic Publishers.

Poli, R. (2003). *A Simple but Theoretically-Motivated Method to Control Bloat in Genetic Programming.* Lectures Notes in Computer Science 2610: 204-217.

Poli, R., Langdon, W., McPhee, N. (2008). *A Field Guide to Genetic Programming*. http://www.gp-field-guide.org.uk/ Último acceso: 23 de junio de 2008.

Povinelli, R., Feng, X. (1999). *Improving Genetic Algorithms Performance by Hashing Fitness Values.* Proceedings of the Artificial Neural Networks in Engineering Conference (ANNIE) '99: 399-404

Quarles, T. (1989a). *Analysis of Performance and Convergence Issues for Circuit Simulation.* Memorandum Nro. UCB/ERL M89/42. Electronics Research Laboratory. University of California at Berkeley.

Quarles, T. (1989b). *SPICE3 Version 3C1 Users Guide*. Memorandum Nro. UCB/ERL M89/46. Electronics Research Laboratory. University of California at Berkeley.

Rabaey, J. (2008). The Spice Page. http://bwrc.eecs.berkeley.edu/Classes/IcBook/SPICE/. Último acceso: 10 de agosto de 2008.

Rivest, R. (1992). *The MD5 Message-Digest Algorithm*. Request for Comments 1321. http://tools.ietf.org/html/rfc1321. Último acceso: 10 de julio de 2008.

Rivest, R. (1997). *S-Expressions*. Internet Draft. http://people.csail.mit.edu/rivest/Sexp.txt. Último acceso: 21 de julio de 2008.

Ryan, C., Soule, T., Keijzer, M., Tsang, E., Poli, R., Costa, E. (editores) (2003). *Proceedings Euro Genetic Programming 2003 Euro GP 2003.* Lectures Notes in Computer Science 2610.

Shafranovitch, Y. (2005). *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. Request for Comments 4180. http://tools.ietf.org/html/rfc4180. Último acceso: 11 de agosto de 2008.

Shimooka, H., Fujimoto, Y. (2000). *Generating Robust Control Equations with Genetic Programming*. Proc. 2000 Genetic and Evolutionary Computation Conference: 491-495.

Smith, S. (1980). *A Learning System Based on Genetic Adaptive Algorithms.* Tesis doctoral. University of Pittsburgh.

Spall, J. (2003). *Introduction to stochastic search and optimization.* John Wiley and Sons.

Sun Microsystems. (2006). *Properties (Java Platform SE 6).* http://java.sun.com/javase/6/docs/api/java/util/Properties.html. Último acceso: 11 de agosto de 2008.

Sussman, G., Stallman, R. (1975). *Heuristic Techniques in Computer Aided Circuit Analysis*. IEEE Transactions on Circuits and Systems 22(11): 857-865.

Sutter, H. (1999). *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison Wesley.

Szentirmai, G. (1977). *FILSYN - A General Purpose Filter Synthesis Program*. Proceedings of the IEEE 65(10): 1443-1458.

Streeter, M. (2003). *The Root Causes of Code Growth in Genetic Programming*. Lectures Notes in Computer Science 2610: 443-454.

Thede, L. (2004). *Practical Analog and Digital Filter Design.* Artech House.

Tuinenga, P. (1988). *SPICE: A Guide to Circuit Simulation and Analysis Using PSpice.* Prentice Hall.

Wedepohl, L., Jackson, L. (2002). *Modified nodal analysis: an essential addition to electrical circuit theory and analysis.* Engineering Science and Education Journal 11(3): 84-92.

Weisstein, E. (2008). *Birthday Problem*. Mathworld: A Wolfram Web Resource. http://mathworld.wolfram.com/BirthdayProblem.html. Último acceso: 5 de julio de 2008.

Wellstead, P. (2000). *Introduction to Physical System Modelling.* Control Systems Principles.

Wolfram Research. (2008). *MeanCI*. Wolfram Mathematica Documentation Center. http://reference.wolfram.com/mathematica/HypothesisTesting/ref/MeanCI.html. Último acceso: 11 de agosto de 2008.

A. MANUAL DE USUARIO

En este capítulo anexo se describen en forma breve la forma de uso del software desarrollado par avaluar las técnicas de optimización propuestas en el capítulo 4. Los contenidos tratados son: los aspectos generales de este software (sección A.1); la sintaxis de línea de comandos para su ejecución (sección A.2); la sintaxis del archivo mediante el cual se describe la tarea de diseño a realizar (sección A.3); y, por último, la sintaxis de la salida de este programa (sección A.4).

A.1. GENERALIDADES

El programa que se ha construido para realizar la verificación experimental del funcionamiento de las técnicas de optimización se denomina *afdgp* y debe encontrarse acompañado por los módulos de operaciones con el genoma y de evaluación. Además debe adjuntarse el módulo con el SPICE, ya que es utilizado por el módulo de evaluación.

Para realizar el diseño, debe ejecutarse este programa pasándole como parámetro un archivo describiendo el problema a resolver y distintos parámetros del proceso de resolución. Por salida estándar se devuelve una indicación del avance del proceso evolutivo y se termina la ejecución mostrando al mejor individuo encontrado, en forma de *netlist*, junto a su puntaje.

A.2. SINTAXIS DE LA LÍNEA DE COMANDOS

La sintaxis de la línea de comandos es simple ya que solo se invoca a *afdgp* pasándole como parámetro opcional el nombre del archivo que describe el trabajo a realizar. En otras palabras:

```
afdgp [job_file]
```

donde el parámetro opcional `job_file` representa el nombre de un archivo *.properties* describiendo el trabajo a realizar. En caso de omitirse el nombre de este archivo, los mismos datos serán tomados de la entrada estándar.

A.3. SINTAXIS DEL ARCHIVO DE DESCRIPCIÓN DE TRABAJOS

La sintaxis básica de este archivo es la de un archivo *.properties* [Sun, 2006], en donde las claves utilizadas toman los sentidos indicados en la tabla A.1. Si bien no todas las claves son requeridas en todos los casos, la presencia de claves adicionales no constituye un error, ya que son ignoradas. Un ejemplo de esto puede verse en la sección anexa F.1, donde el archivo incluye el tamaño del caché a utilizar independientemente de si esté este activado o no.

Clave	Descripción
ID	Identifica al trabajo realizado (no utilizado en la presente versión).
PopulationSize	Determina el tamaño de población con el que se trabajará. Para los casos en los que la población es variable, indica la población inicial.
NumberOfGens	Indica el número de generaciones que incluirá cada una de las ejecuciones independientes.
NumberOfRuns	Indica el número de ejecuciones independientes que se realizarán.
PopStrategy.Name	Indica el nombre de la técnica de control de poblaciones a utilizar.
PopStrategy.MutRate	Indica la probabilidad de que un individuo seleccionado sea mutado.
PopStrategy.CrossRate	Indica la probabilidad de que un individuo seleccionado sea cruzado con otro igualmente seleccionado.
PopStrategy.ArchModRate	Indica la probabilidad de que se le realice una operación de alteración de arquitectura a un individuo seleccionado.
PopStrategy.FitnessSelLambda	Es un factor que determina la distribución de probabilidades con las que se selecciona a los individuos (ver sección 5.1).
PopStrategy.PlagueDeathsByGen	En caso de utilizarse plagas (ver sección 4.2.1), determina la cantidad de individuos que se eliminan por generación.
PopStrategy.TarpeianExecProb	Indica la probabilidad de que se eliminen individuos con tamaño superior al promedio, en caso de utilizarse la técnica de ajuste dinámico (ver sección 4.2.2).
EvalModule.Name	Indica el nombre del módulo de evaluación a utilizar.
EvalModule.UseCache	Indica si se utiliza el caché de evaluación (ver sección 4.3).
EvalModule.CacheSize	Expresa el tamaño en cantidad de individuos del caché.
EvalModule.InitialEmbryo	Indica el embrión inicial del que se parte.
EvalModule.Tolerance	Indica la tolerancia de los componentes con que se trabaja. En esta versión solo es utilizada para determinar los valores permitidos de los componentes.
EvalModule.ProbedNode	Indica el nodo en que se mide la salida.
EvalModule.DesiredTransfer	Indica la transferencia deseada como una serie de pares frecuencia valor separados por espacios.
EvalModule.PenaltyZero EvalModule.PenaltyMultiplier	Parámetros de una penalización según el tamaño del genotipo. Esta penalización opera de acuerdo a la fórmula: $pen = e^{(tam - pz)\, pm}$
EvalModule.StartFreq EvalModule.EndFreq EvalModule.PointsPerDec	Indican los parámetros del análisis AC que llevará a cabo el SPICE (ver sección C.4).
OpsModule.Name	Indica el nombre del módulo de operaciones genéticas.

Tabla A.1. Contenido asociado a cada una de las claves del archivo de trabajos.

A.4. SINTAXIS DE LA SALIDA

La salida consta en su mayoría de líneas indicando estadísticas de la población por cada generación con el objeto adicional de proveer una realimentación visual indicando el avance del proceso evolutivo. A continuación se indica la sintaxis EBNF [ISO/IEC 14977] de esta salida:

```
OUTPUT = DESC_LINE, {GEN_LINE}-, BEST_INDIV_LINE, BEST_SCORE_LINE;

DESC_LINE = "Run", TAB, "Gen", TAB, "MaxLen", TAB, "MinLen", TAB, "AvgLen",
            "MaxScore", TAB, "MinScore", TAB, "AvgScore", TAB, "TimeToNext",
            EOL;

GEN_LINE = RUN, TAB, GEN, TAB, MAX_LEN, TAB, MIN_LEN, TAB, AVG_LEN,
           MAX_SCORE, TAB, MIN_SCORE, TAB, AVG_SCORE, TAB, TIME_TO_NEXT,
           EOL;

BEST_INDIV_LINE = "BestIndividual: ", ESC_NETLIST, EOL;

BEST_SCORE_SO_FAR = "BestScoreSoFar: ", REAL_NUMBER, EOL;

RUN = INT_NUMBER;

GEN = INT_NUMBER;

MAX_LEN = INT_NUMBER;

MIN_LEN = INT_NUMBER;

AVG_LEN = REAL_NUMBER;

MAX_SCORE = INT_NUMBER;

MIN_SCORE = INT_NUMBER;

AVG_SCORE = REAL_NUMBER;

TIME_TO_NEXT = REAL_NUMBER;

INT_NUMBER = {DIGIT}-;

REAL_NUMBER = {DIGIT}-, ["." , {DIGIT}-];
```

TAB = *carácter de tabulación;*

EOL = *carácter de fin de línea;*

DIGIT = *carácter de dígito numérico;*

ESC_NETLIST = *netlist con los caracteres de fin de línea reemplazados por* `"\n";`

Pueden verse ejemplos de salidas en la sección F.2.1.

B. ESTRUCTURA DEL CÓDIGO DE PRUEBA UTILIZADO

En el presente capítulo anexo se muestran: aspectos generales del sistema desarrollado (sección B.1); estructura general de este sistema (sección B.2), incluyendo su arquitectura modular; estructura de cada uno de los módulos que lo integran (sección B.3), es decir el módulo principal (sección B.3.1), el módulo de operaciones sobre el genoma (sección B.3.2), el módulo de evaluación (sección B.3.3) y el módulo auxiliar SPICE (sección B.3.4); para finalizar, se describe la ejecución del sistema (sección B.4), tanto en una perspectiva general (sección B.4.1), como haciendo un detalle del proceso de construcción del fenotipo y su evaluación (sección B.4.2).

B.1. GENERALIDADES

El código desarrollado para este trabajo puede dividirse en dos partes: código desarrollado para el sistema de prueba en sí y código desarrollado para adaptar al SPICE con objeto de poder ser utilizado pro el sistema desarrollado. El sistema de prueba, denominado AFDGP por las iniciales de "*Analog Filter Design by Genetic Programming*", fue desarrollado en el lenguaje C++; por otro lado, las adaptaciones al SPICE fueron desarrolladas en una combinación de C y C++ para acomodarse al código existente de este programa.

El sistema fe desarrollado haciendo énfasis en la portabilidad y encapsulando toda dependencia específica del sistema operativo en un *namespace* denominado `OSDep`. Sin embargo, la única implementación que se desarrolló fue la correspondiente al sistema operativo Windows.

B.2. ESTRUCTURA GENERAL DEL SISTEMA DE PRUEBA AFDGP

Puede dividirse al sistema de pruebas en sí en tres partes principales: el módulo principal, encargado de mantener las poblaciones e implementar las estrategias mencionadas; el módulo de operaciones genéticas, encargado de operar con el genotipo de los individuos, tanto realizando la cruza como las mutaciones, y el módulo de evaluación, encargado de devolver el puntaje asociado al genotipo de un individuo. Como auxiliar del módulo de evaluación se tiene a un módulo SPICE, encargado de encapsular al SPICE en una interfaz que simplifique la obtención de la respuesta asociada a un filtro analógico dado. Un diagrama de colaboración mostrando estos módulos con sus interrelaciones puede observarse en la figura B.1.

[Figura]

Figura B.1. Diagrama de colaboración del sistema de prueba.

Cada uno de los módulos mencionados (a excepción del módulo principal) fue desarrollado como una biblioteca separada, de modo de simplificar su posible cambio en caso de pretender evaluar en forma comparativa distintas alternativas. Los módulos a utilizar, así como los parámetros a emplear, son tomados de un archivo de trabajo que sigue el formato descrito en la sección A.3.

B.3. ESTRUCTURA DE LOS MÓDULOS INTEGRANTES DEL SISTEMA

En esta sección se describe la estructura interna de cada uno de los módulos integrantes del sistema de pruebas desarrollado. Los módulos descritos son: el principal (sección B.3.1); el módulo de operaciones genéticas (sección B.3.2); el módulo de evaluación de individuos (sección B.3.3) y un módulo auxiliar SPICE (sección B.3.4).

B.3.1. MÓDULO PRINCIPAL

Este módulo está integrado fundamentalmente por unas 18 clases, de las cuales una es abstracta (no puede instanciarse, pero contiene funcionalidad) y 5 son interfaces (carecen de funcionalidad importante). En la figura B.2 puede verse un diagrama de clases simplificado que muestra las clases con sus métodos más importantes e interrelaciones.

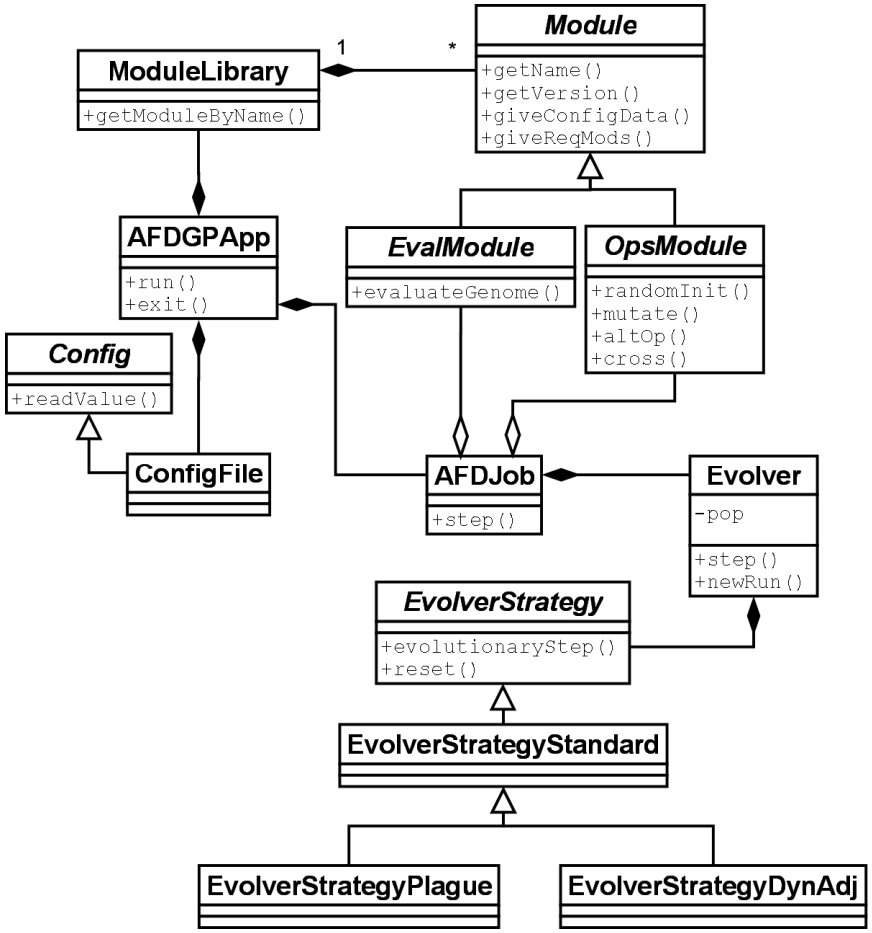


Figura B.2. Diagrama de clases simplificado del módulo principal.

En las secciones a continuación se describe la funcionalidad de las clases más importantes que integran este módulo.

B.3.1.1. AFDJob

Esta clase representa un trabajo de diseño de un filtro analógico. Contiene una referencia a la configuración que utiliza para realizar el trabajo y al objeto que realiza el proceso evolutivo, que es una instancia de la clase `Evolver`. Su método más importante, que es el utilizado por la clase de la aplicación `AFDGPApp` para efectuar el proceso de diseño, es el denominado `step()`. Su propósito es avanzar el proceso evolutivo en una unidad mínima dada.

El objeto de proveer este método, en lugar de uno que realice el proceso evolutivo completo, es simplificar la implementación de la creación de puntos de restauración del proceso evolutivo. Estos puntos no fueron implementados en el presente trabajo debido a que su objetivo era evaluar el rendimiento del proceso evolutivo y no los detalles de su implementación con fines prácticos.

B.3.1.2. AFDGPApp

Esta clase representa a toda la aplicación. Utiliza la técnica conocida como "uso de punteros opacos" [Sutter, 1999] para ocultar su implementación, pero dentro de ésta contiene fundamentalmente a su configuración, trabajo a realizar y biblioteca de módulos a utilizar. Su método principal es uno denominado `run()`, que implementa las llamadas requeridas para desarrollar el proceso de diseño. Incluye también un método denominado `exit()`, que permite solicitar en forma asincrónica la terminación de la ejecución.

B.3.1.3. CachedEvalModule

Esta clase realiza una implementación del patrón *Decorator* [Gamma *et al.*, 1995], agregando la funcionalidad del caché de evaluación a un módulo cualquiera que respete la interfaz `EvalModule`. Como es de esperar, respeta en si mismo la misma interfaz, y tiene como su atributo más importante a un `boost::scoped_array` [Karlsson, 2005] denominado `cache_`.

B.3.1.4. Config (abstracta)

Esta clase actúa como interfaz de las clases de configuración utilizadas en el programa. Pero no es propiamente una interfaz, ya que incluye una cantidad sustancial de código en si misma, en los métodos `getView()` y `getJoinedConfig()`. Estos métodos crean objetos que responden a la interfaz `Config` y permiten representar partes de la configuración o combinación de múltiples configuraciones, manteniendo un orden de prioridad en la búsqueda de valores.

Junto a los dos métodos citados, los métodos más importantes incluidos en esta interfaz son los dedicados a leer valores (`readValue()`) y a consultar si una clave se encuentra presente (`hasKey()`).

B.3.1.5. ConfigFile

Esta clase implementa la interfaz `Config` y permite cargar valores de configuración a partir de un archivo *.properties*. Es empleada en este sistema para obtener la configuración de un trabajo a partir de un archivo que siga el formato especificado en la sección A.3.

B.3.1.6. EvalModule (interfaz)

El objeto de esta clase es definir la interfaz a la que deben adecuarse los distintos módulos de evaluación. Como los módulos de evaluación siguen siendo módulos, esta interfaz refina a la que presenta la clase `Module`.

De los métodos que le pertenecen específicamente, los más importantes son `evaluateGenome()`, encargado de obtener el puntaje asociado al genotipo de un individuo, y `showIndiv()`, que se ocupa de mostrar a un individuo en una forma que sea útil para el propósito del proceso de diseño. Este último método tiene gran importancia, ya que es el encargado de mostrar el resultado del proceso evolutivo.

B.3.1.7. Evolver

Esta clase se encarga de encapsular el proceso evolutivo en sus aspectos independientes del manejo de poblaciones utilizado. Es la clase a la que `AFDJob` le delega la responsabilidad por ejecutar el proceso evolutivo en sí.

Contiene a la población, compuesta por vectores de bytes representando a los genotipos y también referencias a los módulos de evaluación y operaciones genéticas, los que deben implementar las interfaces `EvalModule` y `OpsModule`, respectivamente. También contiene una instancia con la interfaz `EvolverStrategy`, que se encargará de efectuar las partes del proceso evolutivo que dependen de las técnicas de optimización particulares aplicadas.

Los dos métodos más importantes que presenta son `step()`, encargado de implementar un "paso evolutivo", en este caso equivalente a una generación, y `newRun()`, que permite empezar una nueva ejecución independiente de las anteriores (esencialmente inicializa la población y vuelve a cero varios contadores estadísticos).

B.3.1.8. EvolverStrategy (interfaz)

Esta interfaz encapsula las secciones del proceso evolutivo dependientes de las técnicas de optimización utilizadas, lo que permite incluir distintas técnicas de optimización que se involucren con el proceso evolutivo en si. Otras técnicas, tales como el uso de un caché de evaluación, operan "decorando" a un módulo existente, en lugar de afectar al proceso evolutivo.

Los dos métodos más importantes que presenta son `evolutionaryStep()`, encargado de avanzar una generación, y `reset()`, que maneja la "vuelta a cero" requerida para manejar una nueva ejecución independiente.

B.3.1.9. EvolverStrategyDynAdj

Es simplemente una implementación de la interfaz `EvolverStrategy` que permite aplicar la técnica de ajuste dinámico de la función de evaluación descrita en la sección 4.2.2. Reutiliza partes sustanciales de la clase `EvolverStrategyStandard`.

B.3.1.10. EvolverStrategyFactory

Esta clase actúa como *factory* [Gamma *et al.*, 1995] de las estrategias utilizadas, permitiendo crear a la implementación de EvolverStrategy que haya sido requerida en el archivo que define al trabajo de diseño a ejecutar. Para manejar el proceso de registro de las clases a crear, utiliza un sistema de registro automático similar al descrito en [Chouza, 2008].

B.3.1.11. EvolverStrategyPlague

Es simplemente una implementación de la interfaz `EvolverStrategy` que permite aplicar la técnica de uso de plagas para reducir la carga computacional descrita en la sección 4.2.1. Reutiliza partes sustanciales de la clase `EvolverStrategyStandard`.

B.3.1.12. EvolverStrategyStandard

Es la implementación básica de la interfaz `EvolverStrategy`, que no utiliza ninguna de las técnicas de optimización descritas (a excepción del caché de evaluación, que opera en forma esencialmente ortogonal al proceso evolutivo en si). Algunas secciones importantes de esta clase son reutilizadas por las clases `EvolverStrategyDynAdj` y `EvolverStrategyPlague`.

B.3.1.13. ISharedLibrary (interfaz)

Los módulos de este sistema, a excepción del módulo principal que es un ejecutable, fueron realizados como bibliotecas compartidas, enlazada dinámicamente. Estas son conocidas como *Dynamic Link Libraries* (DLLs) en ambientes Windows y como *Shared Objects* (SOs) en ambientes Unix.

Para abstraer las diferencias entre estas plataformas, se creó esta interfaz cuyos métodos más significativos son `makeFromPath()`, que construye un objeto de esta clase en base a su nombre de archivo y `getSymbolAddress()`, que devuelve un puntero a la dirección de un símbolo dado en base a su nombre. La conversión de este puntero al tipo adecuado queda, como es habitual, a criterio del desarrollador.

B.3.1.14. Module (interfaz)

Es la interfaz común a todas las clases de módulos utilizados, tanto módulos de evaluación, como módulos de operación y como módulos auxiliares de otros tipos no reconocidos por el módulo principal. Estos módulos se obtienen a partir de bibliotecas que responden a la interfaz `ISharedLibrary` mediante la ejecución de la correspondiente función `getModule()`.

Todos los módulos incluyen en su interfaz métodos para obtener su nombre (`getName()`), versión (`getVersion()`) y otros módulos requeridos (`getReqMods()`). También incluyen en su interfaz métodos para darles los módulos requeridos (`giveReqMods()`) y para configurarlos (`giveConfigData()`).

B.3.1.15. ModuleLibrary

Esta clase se ocupa de cargar todos los módulos pertenecientes al sistema y de satisfacer las dependencias entre los mismos. También permite al resto del sistema abstraerse por completo de estos problemas y solo requerir un módulo en base a su nombre.

Su método principal es `getModuleByName()` que, como su nombre los indica, devuelve una interfaz `Module` correspondiente al módulo con el nombre pedido.

B.3.1.16. OpsModule (interfaz)

El propósito de esta clase es definir una interfaz a la que deban adecuarse los distintos módulos encargados de realizar operaciones sobre el genotipo de un individuo, tales como la mutación o la cruza. Al igual que en el caso de la clase `EvalModule`, esta interfaz refina a la interfaz `Module`.

Enfocándose en los métodos específicos de esta interfaz, presenta cuatro que pueden considerarse importantes: `randomInit()`, que carga un individuo con un programa aleatorio (pero sintácticamente válido); `mutate()`, que realiza una operación de mutación sobre el genotipo; `altOp()`, que realiza una operación de alteración de valores sobre el genotipo; y, finalmente, `cross()`, que realiza una cruza entre dos genotipos.

B.3.1.17. Registrator

Esta clase colabora en el proceso de registro de estrategias en la clase *factory* `EvolverStrategyFactory`. Son sus instancias las que realizan este proceso. Pueden verse más detalles de este proceso en [Chouza, 2008].

B.3.1.18. WinSharedLibrary

Esta clase no es más que la implementación específica al sistema operativo Windows de la interfaz `ISharedLibrary`. Es la única implementación que se realizó efectivamente.

B.3.2. MÓDULO DE OPERACIONES SOBRE EL GENOMA

A este módulo lo componen principalmente 4 clases, de las cuales una actúa de interfaz. Se muestra un diagrama de clases simplificado de este módulo en la figura B.3.

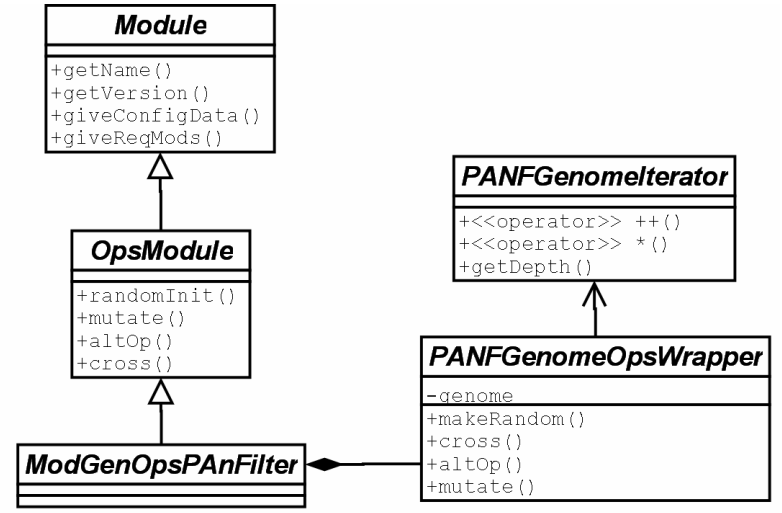


Figura B.3. Diagrama de clases del módulo de operaciones genéticas.

B.3.2.1. ModGenOpsPAnFilter

Es la implementación específica que realiza este módulo de operaciones de la correspondiente interfaz `OpsModule`.

B.3.2.2. OpsModule (interfaz)

Es la misma clase que fue descrita en la sección B.3.1.16.

B.3.2.3. PANFGenomeIterator

El genotipo de los individuos está constituido por una secuencia de bytes que se interpretan como funciones de diversos tipos o como valores. El propósito de esta clase es proveer una forma de recorrer a estos genomas como si se tratase de un árbol, pero permitiendo su representación prefija que es más eficiente.

Su principal método es el operador ++ prefijo, que realiza un recorrido en preorden sobre el árbol representado en el genotipo. Para poder efectuar este recorrido en forma eficiente, utiliza una estructura auxiliar `ancestors_` que le permite conocer la posición de los ancestros del lugar en que se encuentra.

También tiene otros métodos, tales como el operador de desreferencia, que devuelve el valor o la función en la que se encuentra y `getDepth()`, que indica la profundidad en el árbol en la que se encuentra posicionado.

B.3.2.4. PANFGenomeOpsWrapper

Esta clase actúa como un *wrapper* del genoma, permitiendo efectuar las operaciones de mutación, cruza, construcción aleatoria y modificación de valores con simples llamadas a métodos.

B.3.3. MÓDULO DE EVALUACIÓN

Este módulo está integrado principalmente por unas 11 clases, de las que solo una es una interfaz. En la figura B.4 puede verse un diagrama de clases simplificado de este módulo, indicando las relaciones entre sus componentes más importantes.

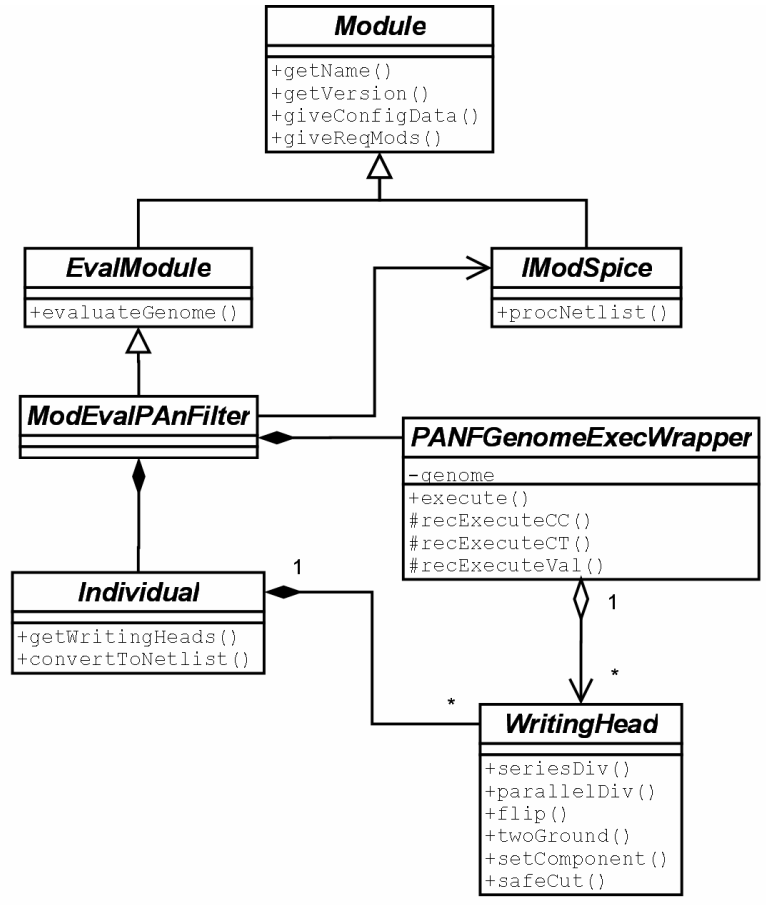


Figura B.4. Diagrama de clases del módulo de evaluación.

A continuación se describen algunos detalles acerca de la funcionalidad y responsabilidades de cada una de estas clases.

B.3.3.1. Component

Representa a un componente electrónico. En el caso que se está tratando, el de los filtros analógicos pasivos, puede ser de cinco tipos: resistor, capacitor, inductor, cable o punta de medición.

B.3.3.2. ComponentNamer

Es la clase encargada de darle nombre a los componentes que se crean al ejecutar al genotipo. Sus dos métodos más importantes son `advance()`, encargado de evitar colisiones con los nombres existentes en el embrión, y `getComponentName()`, que devuelve un nombre a asignar en base al tipo de componente.

B.3.3.3. ComponentValueTransl

En la práctica, los componentes electrónicos no toman un continuo de valores sino que toman sus valores de una tabla de valores estandarizada. El objetivo de esta clase es obtener un valor para el componente en base al entero utilizado para representarlo en el genotipo. El método que realiza este trabajo es denominado `getComponentValue()`.

B.3.3.4. EvalModule (interfaz)

Es la misma clase que fue descrita en la sección B.3.1.6.

B.3.3.5. IModSpice

Esta clase da una interfaz al módulo que encapsula al SPICE, permitiendo la fácil evaluación de los fenotipos construidos. El único método que la caracteriza respecto a la interfaz `Module` a la que refina es el denominado como `procNetlist()`, que toma una *netlist* como una cadena en memoria y devuelve los resultados binarios del proceso de evaluación.

B.3.3.6. Individual

Es una representación del fenotipo del individuo en desarrollo. Sus principales métodos son `getWritingHeads()`, que obtiene las cabezas de escritura requeridas para construirlo, y `convertToNetlist()`, que obtiene la representación del fenotipo en la forma de netlist para ser evaluada.

B.3.3.7. ModEvalPAnFilter

Es la implementación específica que realiza este módulo de evaluación de la correspondiente interfaz `EvalModule`.

B.3.3.8. PANFGenomeExecWrapper

En forma similar a la clase `PANFGenomeOpsWrapper`, descrita en la sección B.3.2.4, esta clase actúa como un *wrapper* del genoma. Pero a diferencia de la clase anteriormente analizada, en este caso provee métodos destinados a ejecutar al genotipo.

El único método público para esto es `execute()`, que toma una serie de cabezas de escritura (instancias de `WritingHead`) y algunas instancias de clases auxiliares (`ComponentNamer`, `ComponentValueTransl`) para construir un individuo. Este método se ve asistido por una serie de métodos protegidos asociado a cada tipo de elemento del genotipo que son los que realizan el trabajo en forma efectiva (`recExecuteCC()`, `recExecuteCT()`, `recExecuteVal()`).

B.3.3.9. SpiceACOut

Esta clase se ocupa de interpretar la salida binaria del SPICE en una forma que pueda ser utilizada por el resto del módulo de evaluación. Su principales métodos son `getNodeVoltages()`, que entregan el vector de valores complejos de tensión en un nodo dado, y `getFreqs()`, que entregan el correspondiente vector paralelo de frecuencias, que le da sentido al anterior.

B.3.3.10. TransferSpec

Es la clase que, instanciándola a partir de una cadena de texto conteniendo pares frecuencia – módulo de transferencia, permite obtener valores del módulo de transferencia interpolados linealmente para cualquier valor de frecuencia que se encuentre entre los dados al momento de la construcción. Este proceso de interpolación se realiza mediante el operador “llamada a función”.

B.3.3.11. WritingHead

Esta clase representa una cabeza de escritura sobre el fenotipo en construcción. Tiene un conjunto de métodos que forman un paralelo con el conjunto de funciones de construcción soportadas y que toman una cabeza de escritura sobre la que realizan la operación devolviendo la cantidad que corresponda de funciones de escritura.

B.3.4. MÓDULO SPICE

En esta sección se describen brevemente en forma estructural las modificaciones realizadas al SPICE. Solo se incluyó una clase específicamente en este módulo, que consta primordialmente del código preexistente del sistema SPICE3F5 (ver capítulo anexo C). Pueden verse más detalles al respecto en el capítulo anexo D.

B.3.4.1. IModSpice (interfaz)

Es la misma clase que fue descrita en la sección B.3.3.5.

B.3.4.2. ModSpice

Es la implementación específica que realiza este módulo SPICE de la correspondiente interfaz `IModSpice`.

B.3.4.3. Manejo de memoria

Para resolver problemas de pérdida de memoria que se presentaban con el SPICE, se reemplazaron las funciones de reserva de memoria que utilizaba este programa con versiones propias. Pueden verse más detalles acerca de los cambios realizados en la sección D.3.

B.3.4.4. Entrada/Salida

Para acelerar el procesamiento de la entrada y la salida, se reemplazaron las funciones de manejo de archivos de la biblioteca estándar de C con versiones propias operando en memoria RAM. Pueden verse más detalles acerca de los cambios realizados en la sección D.2.

B.4. EJECUCIÓN DEL SISTEMA

En esta sección se analiza en una forma más dinámica la interacción que se desarrolla al ejecutar el sistema desarrollado. Para esto se estudian dos casos: en primer lugar, el desarrollo general de la ejecución del sistema (sección B.4.1); en segundo lugar, el proceso de evaluación de un individuo (sección B.4.2), centrándose especialmente en la construcción del fenotipo, pro ser de especial dificultad.

B.4.1. FLUJO GENERAL DEL PROGRAMA

En primer lugar el programa construye una instancia de la clase de la aplicación `AFDGPApp`. Al construirse la clase de aplicación, realiza una serie de tareas incluyendo:

- Lectura de la configuración general
- Carga de los módulos disponibles y satisfacción de sus dependencias.
- Construcción del trabajo de diseño a realizar, que será una instancia de la clase `AFDJob`, pasándole el conjunto de módulos cargados así como un archivo de configuración especificando los parámetros del trabajo a realizar.

La construcción del trabajo en si involucra también una serie de pasos, pero especialmente se centra en la construcción de una instancia de la clase `Evolver` incorporando a los módulos de operaciones genéticas y de evaluación que correspondan en base al archivo *.properties* que describe al trabajo.

La construcción de la instancia de la clase `Evolver` también implica la construcción de la estrategia evolutiva apropiada y la inicialización aleatoria de la población, contenida en esta misma instancia.

Una vez terminado el proceso de construcción, se pasa a la fase de ejecución que, desde el punto de vista de las clases de aplicación, del trabajo de diseño y de la instancia de `Evolver`, involucra poco más que realizar "*steps*" de avance del proceso e imprimir información de progreso por la salida estándar. Pero desde el punto de vista de la instancia de la estrategia evolutiva elegida es distinto, ya que en la clase base concreta `EvolutionaryStrategyStandard`, el método `evolutionaryStep()` descompone la ejecución en una serie de pasos, cada uno de los cuales puede ser implementado de distintas formas en las estrategias derivadas. En caso de que se requirieran modificaciones a la estrategia estándar más profundas, el uso de la interfaz abstracta `EvolutionaryStrategy` da la posibilidad de efectuar un reemplazo de estrategia más fundamental, sin depender de la división en secciones de un paso evolutivo realizada en la clase `EvolutionaryStrategyStandard`.

Si bien cada estrategia puede efectuar la secuencia de tareas dentro de un paso evolutivo de distintas formas, todas ellas involucran el uso de los módulos de evaluación y de operaciones genéticas en algún grado. Las secuencias de operaciones realizadas dentro de cada estrategia son similares a las indicadas en el pseudocódigo de los procesos 2.3, 4.1 y 4.2. La técnica de uso de un caché de evaluación se realiza a nivel de la instancia de `Evolver` que puede elegir en base a la configuración entre utilizar un módulo de evaluación normal o "envolverlo" utilizando el *decorator* `CachedEvalModule`. El funcionamiento general de este módulo de caché está descrito en el proceso 4.3.

Una vez terminado el trabajo de diseño, la instancia correspondiente de `AFDJob` se encarga de mostrar por salida estándar una representación del mejor individuo encontrado de modo de que este pueda ser utilizado, junto con información estadística adicional sobre el desarrollo del proceso de diseño.

B.4.2. EVALUACIÓN DE UN INDIVIDUO

En la sección anterior se describió que todas las técnicas de optimización requerían en última instancia la evaluación de los individuos, ya que este es el proceso que determina la utilidad de los mismos para satisfacer los requerimientos de diseño. Como es procedimiento habitual dentro de la programación genética, esta evaluación se realiza en dos etapas: ejecución del genotipo para obtener el fenotipo y evaluación del fenotipo en términos de los requerimientos.

De estas dos partes es la construcción del fenotipo la más compleja y es realizada en forma similar a la que puede observarse en la sección 2.3.2 por una instancia de la clase `PANFGenomeExecWrapper`, al llamarse su método `execute()`. Este método toma a una serie de cabezas de escritura, apuntando

a distintas aristas del grafo que representa al circuito embrionario en desarrollo, y al genoma que codifica en forma prefija el árbol constituido por las funciones que se describen en la sección anexa E. Mediante una serie de métodos auxiliares que se llaman en forma recursiva, este método explora al árbol de funciones presente en el genotipo, alterando al embrión por medio de los métodos presentes en las cabezas de escritura de acuerdo a la naturaleza de cada función detectada en el genotipo.

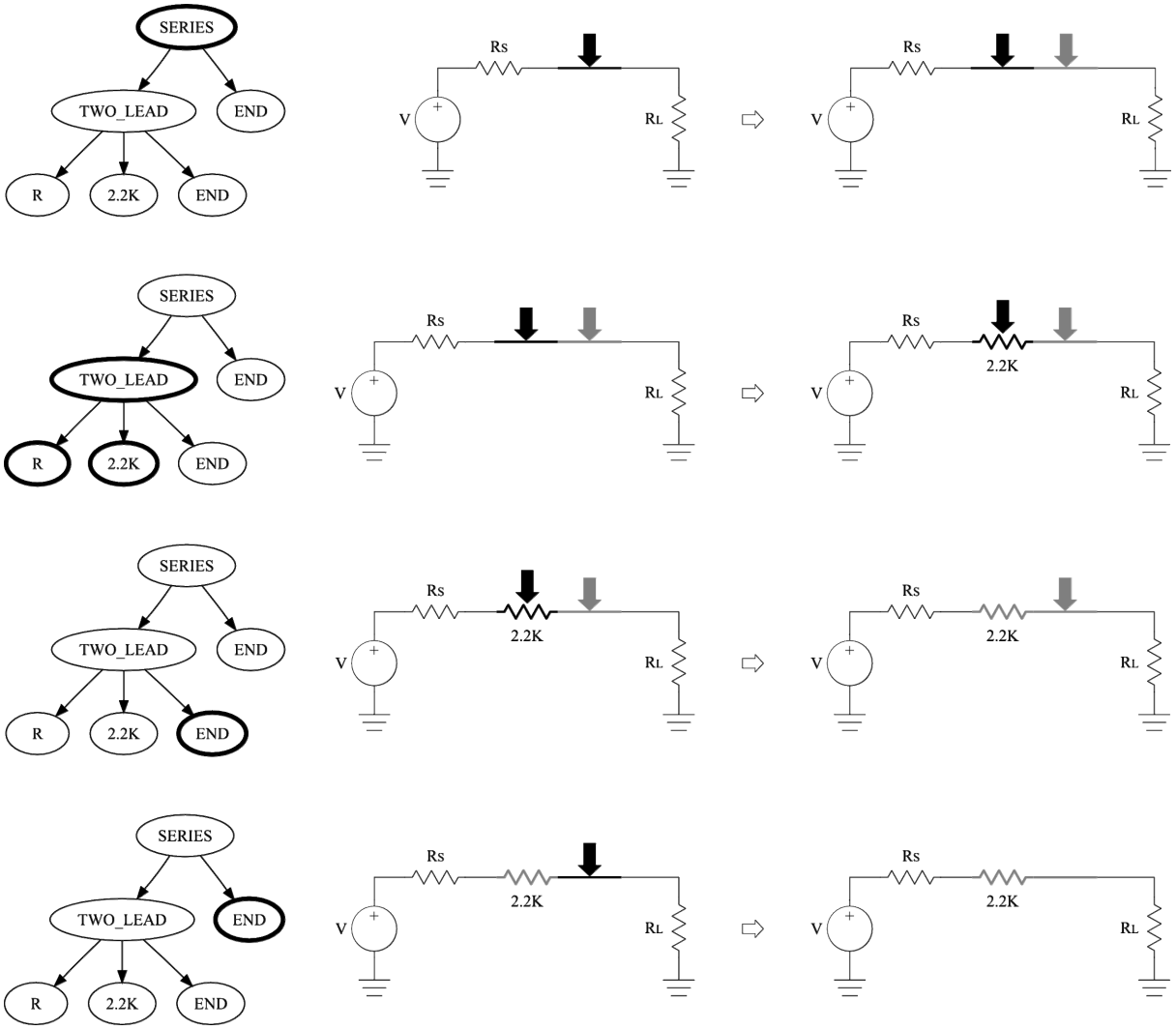


Figura B.5. Reproducción de la figura 2.28, mostrando la ejecución de un genotipo en particular.

Por ejemplo, suponiendo que el genotipo en cuestión es el que se observa en la figura B.5, que expresándolo en forma hexadecimal en base a la codificación empleada por el sistema sería `04 00 01 38 00 00 00 09 09`, tendríamos la secuencia de ejecución que se observa en el proceso B.1.

PROCESO: Construcción del fenotipo de un individuo

ENTRADAS: Genotipo de un individuo *(G)*.

SALIDA: Fenotipo correspondiente a ese individuo.

1. Se llama al método *execute()* de la instancia de la clase *PANFGenomeExecWrapper* correspondiente al genotipo *G*, pasándole como parámetro en este caso una única apuntando al "cable" libre del embrión.
2. Este llama al método *recExecuteCC()* sobre esa cabeza de escritura, pasándole un iterador sobre el genotipo *G*.
3. A su vez, el método *recExecuteCC()* lee el opcode 0x04, correspondiente a la función *SERIES*. Por lo tanto ejecuta el método *seriesDiv()* de la cabeza de escritura, modificando al embrión y obteniendo dos cabezas de escritura, que pueden llamarse *wh1* y *wh2*.
4. Llama al método *recExecuteCC()* sobre la cabeza de escritura *wh1*.
    a. Al ejecutarse el método *recExecuteCC()* sobre la cabeza de escritura *wh1*, lee el opcode 0x00 correspondiente a la función *TWO_LEAD*. Por consiguiente, llama al método *recExecuteCT()* para determinar el tipo de componente a construir.
        i. Al ejecutarse el método *recExecuteCT()*, lee el opcode 0x01, correspondiente al tipo de componente "resistor".
    b. Luego llama al método *recExecuteVal()* para determinar el valor del componente a construir.
        i. Al ejecutarse el método *recExecuteVal()*, lee 4 bytes para dteerminar, de acuerdo con una tabla, el valor del componente a construir. En este caso es de "0x38 0x00 0x00 0x00", correspondiente a 2.2K.
    c. Llama al método *setComponent()* de la cabeza *wh1* para construir un resistor de 2.2K en la arista apuntada.
    d. Para finalizar, llama al método *recExecuteCC()* sobre la cabeza de escritura *wh1*.
        i. Al ejecutarse el método *recExecuteCC()* sobre la cabeza de escritura *wh1*, lee el opcode 0x09 correspondiente a la función *END*. Por lo tanto, se destruye la cabeza *wh1* y se termina esta rama de la recursión.
5. Se ejecuta al método *recExecuteCC()* sobre la cabeza de escritura *wh2*.
    a. Al ejecutarse el método *recExecuteCC()* sobre la cabeza de escritura *wh1*, lee el opcode 0x09 correspondiente a la función *END*. Por lo tanto, se destruye la cabeza *wh1* y se termina esta rama de la recursión.

Proceso B.1. Secuencia de ejecución resultante de la construcción del fenotipo que se observa en la figura B.5.

Una vez obtenido el grafo describiendo el circuito, este se representa en el formato de *netlist* descrito en la sección C.4 y es posteriormente analizado utilizando el módulo auxiliar SPICE. Los resultados de este análisis se comparan con una descripción de la transferencia deseada, en este caso sumando los cuadrados de las diferencias entre módulos en una serie de puntos, y el resultado de esta comparación es asignado como puntaje a cada individuo. Al ser una medida de la discrepancia entre el valor deseado de la transferencia y el obtenido, un mayor puntaje indicará un "peor" individuo.

C. DESCRIPCIÓN GENERAL DEL SPICE

En este capítulo anexo se presentan: una descripción general del simulador SPICE (sección C.1); una descripción de los módulos que integran este programa (sección C.2), incluyendo el módulo de entrada/salida (sección C.2.1), el módulo de simulación y algoritmos numéricos (sección C.2.2), el módulo de dispositivos (sección C.2.3) y el módulo de manejo de matrices (sección C.2.4); el flujo de ejecución del programa (sección C.3) y, por último, una descripción del formato de las entradas en los aspectos utilizados en este trabajo (sección C.4).

C.1. DESCRIPCIÓN GENERAL

El SPICE es un programa de simulación de circuitos utilizado ampliamente en una gran variedad de campos, constituyendo el estándar en ese campo. Este programa toma como entrada una descripción textual de un circuito (cuyo formato se analiza en la sección C.4) y realiza diversas clases de simulaciones, dependiendo de los detalles de la entrada.

La versión utilizada en el desarrollo de este trabajo es la denominada SPICE3F5, que fue la última versión desarrollada por la Universidad de California, Berkeley [Rabaey, 2008]. Al igual que las otras versiones 3, denominadas colectivamente SPICE3, está escrita en el lenguaje de programación C y consta de alrededor de 157000 líneas. Para facilitar la adición de nuevos componentes, tiene una estructura modular, que puede observarse en una forma simplificada en la figura C.1.

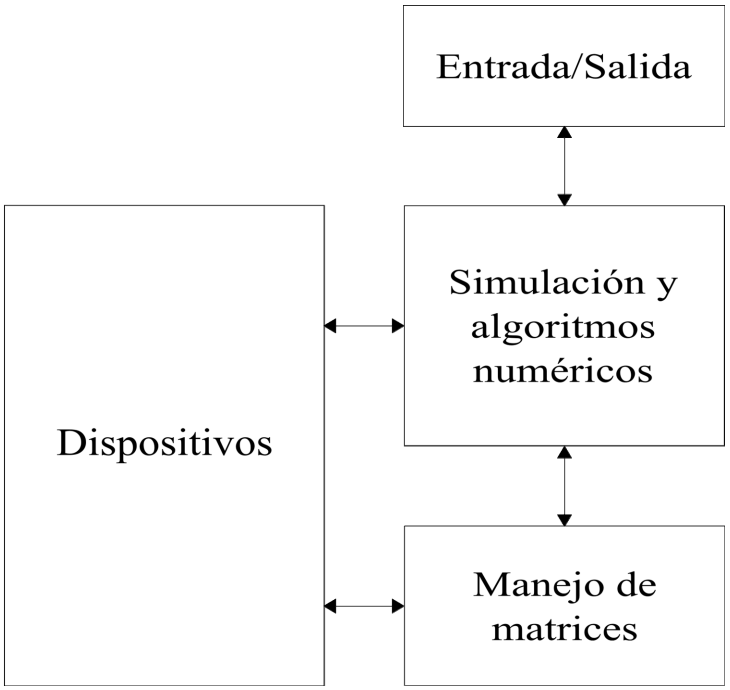


Figura C.1. Diagrama mostrando los módulos que componen al simulador SPICE3.

C.2. DESCRIPCIÓN DE LOS MÓDULOS

En esta sección se realiza una breve descripción de los módulos que constituyen el SPICE, es decir: el módulo de entrada/salida (sección C.2.1), el módulo de simulación y algoritmos numéricos (sección C.2.2), el módulo de dispositivos (sección C.2.3) y el módulo de manejo de matrices (sección C.2.4).

C.2.1. MÓDULO DE ENTRADA/SALIDA

Este módulo se encarga de manejar el procesamiento de los datos de entrada y de representar la salida de la simulación. La entrada se realiza en el formato *netlist*, explicado en detalle en la sección C.4, y la salida se realiza en distintos formatos de acuerdo a los análisis realizados y a los parámetros enviados por línea de comandos.

Este módulo realiza el proceso de *parsing* de la *netlist*, transformándola en una representación interna en memoria. Esta representación captura las distintas instancias de los componentes y los conecta con los modelos que serán utilizados en al etapa de simulación. Luego de efectuada la simulación, los datos de salida son procesados por este módulo, que representa estos datos en un formato textual o binario de acuerdo a las opciones previamente establecidas.

C.2.2. MÓDULO DE SIMULACIÓN Y ALGORITMOS NUMÉRICOS

Este módulo contiene las funciones encargadas de efectuar la simulación del circuito. Permite realizar distintas clases de análisis, incluyendo análisis con fuentes sinusoidales de frecuencias dadas y análisis de la respuesta en función del tiempo.

El método básico utilizado es el denominado *análisis nodal modificado* o MNA, por sus iniciales en inglés [Quarles, 1989a; Wedepohl & Jackson, 2002]. Este método se basa en aplicar las leyes de Kirchhoff a un circuito en una forma especifican con objeto de obtener un sistema de ecuaciones. Una vez resuelto este sistema, las variables contendrán los valores de las tensiones de los nodos y las corrientes de las ramas del circuito.

Los casos que involucren elementos lineales pueden tratarse mediante el uso de aritmética compleja, pero los casos más generales comprendiendo elementos no lineales requieren el uso de métodos iterativos para resolver los sistemas no lineales resultantes. Asimismo la utilización de señales de entrada no periódicas requiere la realización de análisis llamados “transitorios” (*transient*) con su correspondiente necesidad de utilizar métodos de integración numérica.

C.2.3. MÓDULO DE DISPOSITIVOS

Este módulo contiene los modelos de los componentes electrónicos utilizados en SPICE. Si bien en el caso de los componentes utilizados en este trabajo los modelos son bastante triviales, en el caso más general de componentes no lineales, los modelos de los componentes pueden ser notablemente más complejos. Además, los detalles de su comportamiento serán críticos para conseguir la convergencia de la simulación, ya que modelos inadecuados pueden introducir discontinuidades que dificultan notablemente la convergencia de los procesos iterativos.

Uno de requisitos claves de un simulador es poder agregar nuevos dispositivos, por lo que toma especial importancia poder realizar esta tarea en forma relativamente simple, sin que sea necesario realizar modificaciones en múltiples lugares del código. Para esto, SPICE concentra todos los datos del dispositivo en estructuras con una misma interfaz que, a su vez, maneja a través de este módulo, controlando aspectos tales como que inicial corresponde a cada dispositivo.

C.2.4. MÓDULO DE MANEJO DE MATRICES

La simulación involucra operar con grandes matrices donde la mayoría de sus elementos son nulos. Este módulo permite operar eficientemente con esta clase de matrices denominadas como ralas (*sparse matrices*), lo cual disminuye en gran medida los requisitos de memoria respecto al uso de matrices normales y, en algunos casos, también disminuye los tiempos de procesamiento.

C.3. FLUJO DE EJECUCIÓN

El flujo de ejecución del SPICE respeta lo que podría esperarse en base a las estructuras de datos que utiliza. En primer lugar, el módulo de entrada/salida toma el texto de la *netlist* desde la entrada, realiza el proceso de *parsing* y construye una estructura representando el circuito y los análisis que se le deben realizar. Luego el módulo de simulación y algoritmos numéricos realiza las operaciones necesarias para efectuar los análisis requeridos, delegando las operaciones con matrices al módulo creado para tal fin. Por último, la salida que se ha almacenado junto a la estructura del circuito es representada por el módulo de entrada/salida.

C.4. FORMATO DE LA ENTRADA

En esta sección se da una descripción detallada del formato de los aspectos de las *netlists* utilizadas en el presente trabajo. Una descripción introductoria del formato de las *netlists* puede encontrarse en la sección 2.3.1.1, mientras que una descripción más completa puede encontrarse en [Quarles, 1989b].

La gramática de las *netlists* en general puede describirse en notación EBNF [ISO/IEC 14977] de la siguiente forma:

```
NETLIST = TITLE-LINE, {CONTENT-LINE};

TITLE-LINE = ANY-TEXT, EOL;

CONTENT-LINE = {WS}, COMMENT-LINE | TERM-LINE | ELEM-LINE | CONTROL-LINE, {WS},
               [';', ANY-TEXT], EOL;

COMMENT-LINE = '*', ANY-TEXT;

TERM-LINE = '.END';

ELEM-LINE = ELEM-NAME, {WS}-, FIRST-NODE, {WS}-, SECOND-NODE, {WS}-,
            {PARAM, {WS}-}, PARAM;

CONTROL-LINE = '.', COMMAND-NAME, {{WS}-, PARAM};

ANY-TEXT = {PRINTABLE-CHAR | WS}-;

ELEM-NAME = ELEM-TYPE, {PRINTABLE-CHAR};

FIRST-NODE = {PRINTABLE-CHAR}-;

SECOND-NODE = {PRINTABLE-CHAR}-;

PARAM = {PRINTABLE-CHAR}-;

COMMAND-NAME = {PRINTABLE-CHAR}-;

ELEM-TYPE = ALPHA-CHAR;

EOL = carácter o secuencia de fin de línea;

WS = carácter cualquiera de espacio en blanco, sin incluir el fin de línea;

PRINTABLE-CHAR = carácter imprimible, no incluye a los carácteres de espacio en blanco;

ALPHA-CHAR = carácter alfabético;
```

Las funciones generales de los distintos tipos de líneas ya ha sido tratados en el cuerpo de la tesis, pro lo que no se describirán nuevamente. En cuanto a los tipos de componentes, tenemos en este trabajo 4 tipos, a saber:

- **Resistor.** Identificado por el uso de una letra `'R'` como tipo de elemento, representa al elemento del mismo nombre. Tiene un solo parámetro, que identifica su resistencia eléctrica en ohms.

- **Capacitor.** Se identifica por tener la letra `'C'` como tipo de elemento. Representa a este elemento eléctrico y tiene un único parámetro que mide su capacidad en farads.

- **Inductor.** Puede reconocerse por el uso de la inicial `'L'` como tipo de elemento. Al igual que los dos elementos anteriores, representa al mismo elemento eléctrico y toma un parámetro indicando el valor de su inductancia en henries.

- **Fuente de tensión.** Se identifica por utilizar a la letra `'V'` como tipo de elemento. Para el caso de este trabajo, representa a la señal de entrada y toma dos parámetros: el primero describe el tipo de fuente y es siempre `'AC'`, ya que se trabaja con señales sinusoidales; el segundo es `'1'`, ya que, como se busca la transferencia del filtro, se utiliza una entrada unitaria.

Solo se usa una línea de comando, `'.AC'`, que indica al SPICE realizar un barrido de frecuencias en las fuentes de AC que se encuentren conectadas al circuito. Este comando toma cuatro parámetros que fueron tomados como fijos para los análisis realizados. El primero indica que medida de intervalo de frecuencias se utiliza, el segundo indica cuantos puntos se analizarán pro cada intervalo de frecuencias; los dos últimos parámetros expresan la frecuencia inicial y final del barrido de frecuencias a realizar.

Por ejemplo, `'.AC DEC 10 1 100000'` indica barrer frecuencias desde 1 Hz hasta 100 kHz, analizando 10 puntos por década. Esto daría un total de 51 puntos, si consideramos que se incluyen los extremos, como es el caso en el SPICE.

D. MODIFICACIONES REALIZADAS AL SPICE

En este capítulo anexo se muestran: los motivos para las modificaciones que se efectuaron al SPICE3F5 (sección D.1), una descripción de las modificaciones realizadas sobre el módulo de entrada/salida del SPICE (sección D.2), un detalle de las alteraciones efectuadas al manejo de memoria del SPICE (sección 0) y se concluye con algunas modificaciones no efectuadas, pero que podrían ser de interés para trabajos futuros (sección D.4).

D.1. MOTIVOS PARA LAS MODIFICACIONES REALIZADAS

El programa desarrollado para probar experimentalmente el desempeño de las distintas técnicas de optimización requiere ejecutar cientos de miles de veces al simulador SPICE. Por lo tanto, si bien este simulador se encuentra altamente optimizado en lo que se refiere a *simular* los circuitos, fue necesario modificar el funcionamiento del módulo de entrada/salida (ver capítulo anexo C) de este programa para obtener un desempeño aceptable.

Otra modificación requerida fue la implementación de un control sobre la memoria reservada. Si bien en general SPICE no pierde grandes cantidades de memoria al ejecutarse, pérdidas de memoria que resultan insignificantes en una sola ejecución del programa se vuelven significativas cuando esta ejecución se realiza un gran número de veces. Si no se hubieran corregido estas pérdidas de memoria, hubiera sido imposible desarrollar el proceso evolutivo, ya que, antes de efectuar las correcciones, el "uso" de memoria llegaba en unas decenas de segundos a cientos de megabytes.

D.2. MODIFICACIONES REALIZADAS AL MÓDULO DE ENTRADA/SALIDA

El módulo de entrada/salida del SPICE está diseñado para operar con archivos. Sin embargo, en su nuevo papel de auxiliar del módulo de evaluación del sistema de prueba, pasa a tener un rol mucho más importante la eficiencia de la carga de datos.

Para resolver este problema con un impacto mínimo en la base de código del SPICE, se creó un reemplazo simple de los archivos que trabaje con bloques de memoria denominado MEMFILE. La creación de un reemplazo, en lugar de limitarse a utilizar alternativas existentes, se debió a que el estudio de los requisitos específicos de SPICE permitía desarrollar un sustituto con un funcionamiento significativamente más eficiente en esta aplicación. Para una mayor compatibilidad con SPICE, MEMFILE desarrolló íntegramente en el lenguaje de programación C.

A continuación se muestra la estructura MEMFILE:

```
typedef struct MEMFILE_T
{
        /* El buffer */
        unsigned char* buffer;
        /* La cantidad de elementos válidos que tiene */
        size_t cant_elem;
        /* La cantidad de elementos reservados que tiene */
        size_t cant_reserved;
        /* Cursor (apunta al próximo lugar a leer o escribir) */
        size_t pos;
        /* El nombre */
        unsigned char* name;
        /* Está abierto? */
        int is_open;
} MEMFILE;
```

Como puede verse, incluye el un buffer de datos y marca explícitamente cuantos datos son utilizados dentro del mismo. Esto permitiría implementar un manejo más eficiente de memoria, en caso de que fuera requerido. Se implementaron primitivas análogas a las de la biblioteca estándar de C, de modo que las funciones de SPICE pudieran adaptarse agregando simplemente el prefijo "mem" a las funciones de manejo de archivos. No se implementaron todas, sino las utilizadas por SPICE:

```
/* Abre un MEMFILE. No necesito modo. */
MEMFILE* memfopen(const char* name, const char* mode);

/* Cierra un MEMFILE */
void memfclose(MEMFILE* mfp);

/* Escribe en un MEMFILE */
size_t memfwrite(const void* buffer, size_t size, size_t count,
                                MEMFILE* mfp);

/* Lee desde un MEMFILE */
size_t memfread(void* buffer, size_t size, size_t count, MEMFILE* mfp);

/* Escribe una cadena en un MEMFILE */
int memfputs(const char* str, MEMFILE* mfp);

/* Escribe con formato en un MEMFILE */
int memfprintf(MEMFILE* mfp, const char* fmt, ...);

/* Hace seek en MEMFILE */
int memfseek(MEMFILE* mfp, long offset, int base);

/* Devuelve la posición en un MEMFILE */
size_t memftell(MEMFILE* mfp);
```

También se incluyeron algunas funciones de limpieza para la biblioteca en general:

```
/* Limpia todos los MEMFILEs creados */
void memfgencleanup(void);

/* Limpia un MEMFILE en particular */
void memfcleanup(const char* name);

/* Obtiene el tamaño de los datos */
size_t memfgetsize(MEMFILE* mfp);
```

Estas funciones son requeridas, ya que cerrar un MEMFILE no libera su memoria, permitiendo su acceso posterior para recuperar los datos.

D.3. MODIFICACIONES REALIZADAS AL MANEJO DE MEMORIA

Como se explicó anteriormente, el SPICE3F5 utilizado originalmente presentaba pérdidas de memoria que, al acumularse, alcanzaban proporciones inadmisibles. Para corregir este problema, se modificaron las funciones que el SPICE utilizaba para reservar y liberar memoria. Además de reservar y liberar memoria, almacenaban y eliminaban los punteros a la memoria reservada en un *hash set*.

Al terminar la ejecución del SPICE, podía entonces liberarse toda la memoria perdida, evitando los problemas mencionados anteriormente. Como beneficio adicional, esto permitía centralizar los pedidos de memoria, posibilitando la fácil obtención de estadísticas respecto a las reservas de memoria.

D.4. OTRAS POSIBLES MODIFICACIONES

Además de las modificaciones realizadas, existen otras modificaciones que podrían realizarse sobre el SPICE para hacerlo más adecuado al uso dentro de la programación genética. Estas modificaciones no fueron realizadas por no ser la optimización del SPICE el foco del presente trabajo, sino la optimización dentro de la programación genética en si. Algunas de estas posibles mejoras fueron descritas brevemente en el capítulo 6, pero en este capítulo anexo serán tratadas con algo más de detalle.

Uno de los aspectos donde podrían darse mejoras es en el módulo de entrada/salida. Si bien este módulo fue modificado para aceptar los datos a través de la memoria (ver sección D.2), sigue realizando procesos relativamente costosos como el *parsing* del texto de las *netlists*. Si se utilizaran para la entrada/salida las representaciones internas del SPICE, adaptadas como es práctica habitual para no depender de posiciones fijas de memoria, podrían eliminarse estos costos.

Otra limitación es el manejo de memoria “simple” descrito en la sección 0. La utilización de *pools* de memoria organizadas según tamaño, junto a la aceptación de una reserva de memoria relativamente ineficiente en cuanto a espacio (por ejemplo con bloques cuyos tamaños sean potencias de dos), podría permitir una reserva de memoria más eficiente. Estas técnicas no fueron consideradas en detalle, ya que el foco del manejo de la memoria estaba en simplemente permitir el desarrollo del proceso evolutivo.

Una funcionalidad que podría ayudar a obtener resultados más confiables es la implementación de una evaluación eficiente del circuito en un conjunto de frecuencias definidas por el usuario. Esto hubiera requerido modificaciones relativamente importantes al SPICE, pero habría posibilitado concentrar las evaluaciones en los puntos "más interesantes" de la transferencia, optimizando el uso de los recursos computacionales.

Por último, el SPICE fue desarrollado en una época en la que el uso de *caches* era poco significativo. Por ende, hace un uso extensivo de estructuras tales como listas enlazadas, con la consiguiente falta de localidad en los accesos a memoria. Una forma de mejorar la *performance* del SPICE sin modificar sus algoritmos, sería cambiar las estructuras utilizadas a otras que presenten mayor localidad de acceso, tales como vectores.

Es importante destacar que estas modificaciones no fueron evaluadas en detalle, sino solo en forma preliminar. Antes de efectuar cualquiera de estas modificaciones, sería imperativo efectuar mediciones que demuestren concluyentemente que producirían una mejora significativa en el rendimiento.

E. FUNCIONES UTILIZADAS EN EL GENOTIPO

En este capítulo anexo se muestran: aspectos generales respecto al conjunto de funciones seleccionado (sección E.1); una descripción de las funciones que modifican la topología (sección E.2), incluyendo la función `SERIES` (sección E.2.1), la función `PARALLEL` (sección E.2.2), la función `TWO_GROUND` (sección E.2.3), la función `FLIP` (sección E.2.4), la función `END` (sección E.2.5) y la función `SAFE_CUT` (sección E.2.6); una explicación del comportamiento de la única función de creación de componentes incluida (sección E.3), la función `TWO_LEAD` (sección E.3.1); por último, se concluye con una análisis de las funciones que devuelven valores (sección E.4), incluyendo tanto las que devuelven valores de componentes, como `R` (sección E.4.1), `C` (sección E.4.2) y `L` (sección E.4.3), como la "función" que devuelve los valores de los componentes (sección E.4.4).

E.1. GENERALIDADES

El conjunto de funciones utilizado es una versión restringida del empleado por el equipo de John Koza en sus trabajos originales en el área [Koza *et al.*, 1997]. Las características omitidas más significativas son el empleo de funciones de conexión no locales y las funciones relativas a componentes activos o de más de 2 terminales.

Por razones de eficiencia, decodifican las funciones en forma prefija, ya que se conoce el tipo y tamaño de los argumentos que toma cada función. Todas las funciones se identifican mediante un código de un byte, a excepción de los valores que utilizan 4 bytes.

E.2. FUNCIONES QUE MODIFICAN LA TOPOLOGÍA

En esta sección se describen las funciones que modifican la topología del circuito, es decir: la función `SERIES` (sección E.2.1), que permite introducir componentes en serie, la función `PARALLEL` (sección E.2.2), que permite hacerlo en paralelo, la función `TWO_GROUND` (sección E.2.3), que permite agregar tierras en el medio de un circuito, la función `FLIP` (sección E.2.4), que invierte el sentido de un componente, la función `END` (sección E.2.5), que termina el proceso de construcción y la función `SAFE_CUT` (sección E.2.6), que elimina un componente sin dejar a otros desconectados.

E.2.1. SERIES

Esta función toma el elemento que esté bajo la cabeza de escritura y crea otro elemento del mismo tipo con su correspondiente cabeza de escritura en serie. Las dos cabezas de escritura son pasadas a las funciones descendientes (ver figura E.1).

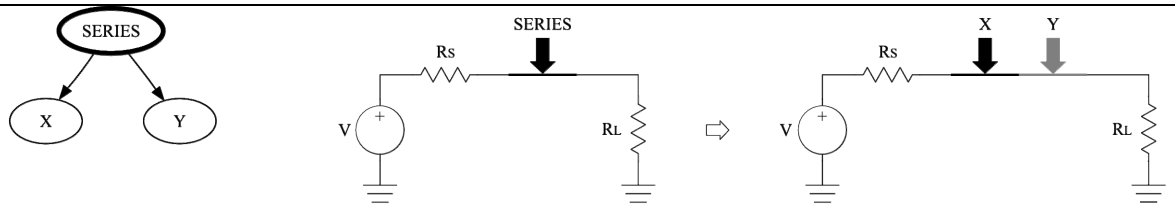


Figura E.1. Efecto de la función `SERIES` en la topología del circuito.

E.2.2. PARALLEL

Esta función toma el elemento que esté bajo la cabeza de escritura y crea otro elemento del mismo tipo con su correspondiente cabeza de escritura en paralelo con el anterior. Al elemento adicionado lo conecta utilizando dos "cables" con sendas cabezas de escritura. Las cuatro cabezas de escritura son pasadas a las funciones descendientes (ver figura E.2).

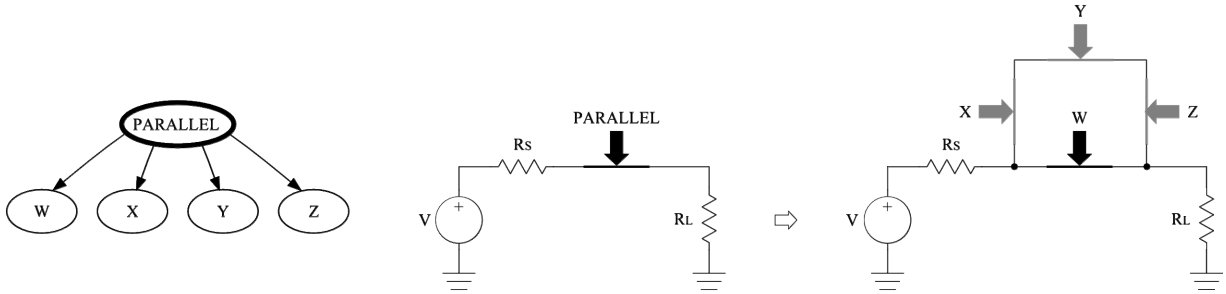


Figura E.2. Efecto de la función `PARALLEL` en la topología del circuito.

E.2.3. TWO_GROUND

Esta función opera en forma casi idéntica a la función SERIES. La única diferencia radica en que conecta a tierra el nodo creado uniendo ambos componentes (ver figura E.3).

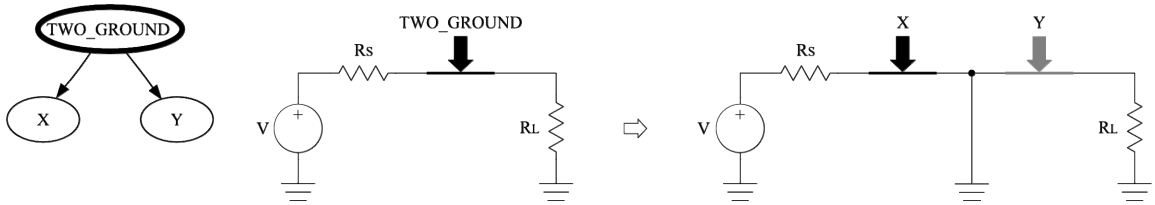


Figura E.3. Efecto de la función `TWO_GROUND` en la topología del circuito.

E.2.4. FLIP

Esta función simplemente invierte la dirección del elemento que se encuentre bajo la cabeza de escritura, pasando la misma a su descendiente. Se introdujo por completitud y por ser de fácil implementación, pero no origina efectos a trabajar con componentes lineales.

E.2.5. END

El objeto de esta función es proveer una forma de terminar el proceso de desarrollo. Su único efecto consiste en eliminar la cabeza de escritura que recibe, sin efectuar ningún cambio en el circuito (ver figura E.4).

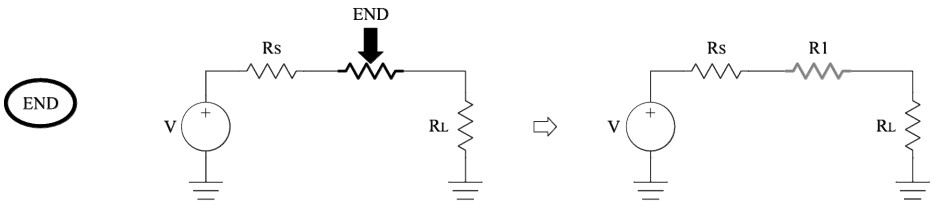


Figura E.4. Efecto de la función `END` en la topología del circuito.

E.2.6. SAFE_CUT

El propósito de esta función consiste en proveer otra forma de terminar el proceso de desarrollo, eliminando al componente apuntado por la cabeza de escritura (ver figura E.5). Para evitar dejar componentes con terminales desconectados, solo se elimina el componente si el grado de ambos nodos terminales es mayor o igual que 3. Puede verse fácilmente que la eliminación del componente solo reducirá este grado en uno y que, por lo tanto, la aplicación de la función `SAFE_CUT` no puede originar que un nodo quede conectado a un solo componente. Sin embargo, no puede impedir la pérdida de conectividad de circuito ya que determinar si una arista es o no una arista de corte no es un problema que pueda resolverse solo con datos locales.

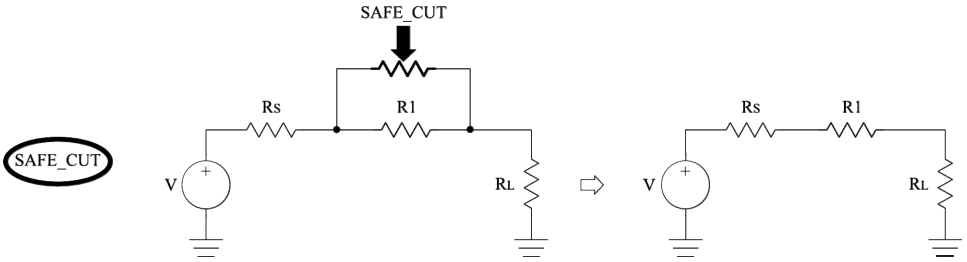


Figura E.5. Efecto de la función `SAFE_CUT` en la topología del circuito.

E.3. FUNCIONES QUE CREAN COMPONENTES

Esta sección contiene a la única función que se utiliza en este trabajo para crear componentes, ya que solo se emplean elementos de dos terminales.

E.3.1. TWO_LEAD

Esta función toma 3 parámetros: el primero, le indica un tipo de componente; el segundo, un valor para el componente y el tercero, una función descendiente. El trabajo que realiza esta función

consiste en introducir un componente del tipo y valor indicados en el lugar apuntado por la cabeza de escritura y pasar esta cabeza de escritura a su función descendiente (ver figura E.5).

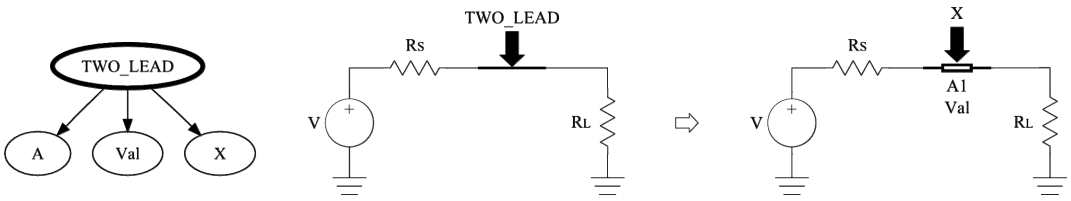


Figura E.5. Efecto de la función `TWO_LEAD` en el circuito.

E.4. FUNCIONES QUE DEVUELVEN VALORES

En esta sección se detallan las funciones que devuelven valores: esto incluye tanto las funciones que devuelven tipos de componente como `R` (sección E.4.1), `C` (sección E.4.2) y `L` (sección E.4.3), como la función que devuelve los valores que adoptarán los componentes (sección E.4.4).

E.4.1. R

Esta es una función que no toma argumentos y devuelve un valor de tipo de componente, `'R'`, que es aceptado por la función `TWO_LEAD` para determinar el tipo de componente a insertar, en este caso un resistor.

E.4.2. C

Esta es otra función que no toma argumentos y devuelve un valor de tipo de componente, `'C'`, que es aceptado por la función `TWO_LEAD` para determinar el tipo de componente a insertar, en este caso un capacitor.

E.4.3. L

Esta es una función que no toma argumentos y devuelve un valor de tipo de componente, `'C'`, que es aceptado por la función `TWO_LEAD` para determinar el tipo de componente a insertar, en este caso un inductor.

E.4.4. VALOR DE COMPONENTE

Esta función no toma argumentos y devuelve un valor de componente. Una característica particular de esta función es que no se identifica mediante un código sino por contexto, ya que siempre se al encuentra en una posición fija respecto a la función de construcción de componentes `TWO_LEAD`.

Los valores de componente que devuelve no son números reales arbitrarios, sino que son índices en una tabla de valores de componentes. Esto es debido a que en la práctica se utiliza un conjunto discreto de valores de componentes.

F. PRUEBAS REALIZADAS

En este capítulo anexo se detallan: las pruebas realizadas con el software desarrollado (sección F.1) y el procesamiento de los resultados obtenidos (sección F.2), incluyendo el proceso de reducción de datos (sección F.2.1) y la forma que toman los datos una vez reducidos (sección F.2.2).

F.1. PRUEBAS REALIZADAS

Con objeto de reunir los datos utilizados en las verificaciones experimentales del capítulo 5, se realizaron con ayuda de un *script* un total de 6 ejecuciones del software de prueba. En todas ellas se utilizó como base el siguiente *template* para el archivo de trabajo:

```
#
# Configuración general del trabajo
#

# Identificador único
ID = WvZT1DF3

# Tamaño
PopulationSize = $populationSize

# Cantidad de generaciones
NumberOfGens = $numberOfGens

# Número de ejecuciones
NumberOfRuns = $numberOfRuns

#
# Configuración de la estrategia de manejo de la población
#

# Estrategia para manejar la población
PopStrategy.Name = $popStrategy

# Tasa de mutación
PopStrategy.MutRate = $mutRate

# Tasa de cruza
PopStrategy.CrossRate = $crossRate

# Tasa de operaciones de alteración de la arquitectura
PopStrategy.ArchModRate = $archModRate

# Lambda para la selección de los individuos (relativo al tamaño de la
población)
PopStrategy.FitnessSelLambda = $fitnessSelLambda

# Cantidad que mueren por generación (Estrategia = Plague)
PopStrategy.PlagueDeathsByGen = $plagueDeathsByGen

# Probabilidad de ejecución (Estrategia = DynAdj)
PopStrategy.TarpeianExecProb = $tarpeianExecProb

#
# Configuración del módulo de evaluación
```

```
#

# Nombre del módulo
EvalModule.Name = EvalPassiveAnalogFilter

# Uso caché
EvalModule.UseCache = $useCache

# Tamaño 1 MB
EvalModule.CacheSize = 1048576

# Es el embrión sobre el que el genoma debe aplicarse
EvalModule.InitialEmbryo = TITLE\nV1 0 1 1\nR1 1 2 1K\nZ1 2 3 X\nR2 3 0 1K\nP1 3
0 X\n

# Indica la tolerancia de los componentes con los que trabajamos
EvalModule.Tolerance = 50

# Indica el nodo sobre el que se mide la transferencia
EvalModule.ProbedNode = 3

# Indica la transferencia deseada (en este caso se trata de un pasabajos)
EvalModule.DesiredTransfer = $desiredTransfer

# Los sigiuientes valores indican la penalización por excesivo tamaño. Opera en
base a la fórmula
# Penalidad = Exp((tamaño - PenaltyZero) * PenaltyMultiplier)

EvalModule.PenaltyZero = $penaltyZero
EvalModule.PenaltyMultiplier = $penaltyMultiplier

# Los siguientes valores indican los parámetros del barrido en SPICE que se
realiza en el análisis AC

# Frecuencia de inicio
EvalModule.StartFreq = $startFreq

# Frecuencia de finalización
EvalModule.EndFreq = $endFreq

# Puntos por década
EvalModule.PointsPerDec = $pointsPerDec

#
# Configuración del módulo de operaciones
#

OpsModule.Name = GenOpsPassiveAnalogFilter
```

Las sustituciones reemplazan los valores precedidos por "$" con un valor dado en el *script*. A este *template* se le realizaron una serie de sustituciones fijas y otra serie de sustituciones variables de acuerdo a las técnicas que se estuvieran planeando.

Las sustituciones fijas realizadas fueron:

```
$populationSize → "1000"
$numberOfGens → "100"
$numberOfRuns → "10"
$mutRate → "0.1"
$crossRate → "0.2"
$archModRate → "0.2"
```

```
$fitnessSelLambda → "2.0"
$desiredTransfer → "0 0.5 10000 0.5 10001 0 1000000 0"
$penaltyZero → "1000"
$penaltyMultiplier → "0"
$startFreq → "1"
$endFreq → "100000"
$pointsPerDec → "10"
$plagueDeathsByGen → "10"
$tarpeianExecProb → "0.2"
```

La transferencia indicada, `"0 0.5 10000 0.5 10001 0 1000000 0"`, significa transferencia igual a 0.5 desde frecuencia 0 hasta frecuencia 10000 Hz y transferencia igual a 0 desde 10001 Hz hasta 1 MHz. Entre cada uno de los puntos dados se realiza una interpolación lineal, aunque esto no es especialmente significativo en este caso, ya que solo ocurre esta interpolación en forma notable entre 10000 y 10001 Hz.

Las sustituciones a las "variables" `$popStrategy` y `$useCache` se realizaron de acuerdo al caso de prueba de acuerdo a la siguiente tabla:

Nombre del caso de prueba	Abreviatura	Valores de las variables
Caso base sin caché	SUC	`$popStrategy → "Standard"` `$useCache → "0"`
Caso base con caché	SC	`$popStrategy → "Standard"` `$useCache → "1"`
Plagas sin caché	PUC	`$popStrategy → "Plague"` `$useCache → "0"`
Plagas con caché	PC	`$popStrategy → "Plague"` `$useCache → "1"`
Ajuste dinámico sin caché	DAUC	`$popStrategy → "DynAdj"` `$useCache → "0"`
Ajuste dinámico con caché	DAC	`$popStrategy → "DynAdj"` `$useCache → "1"`

Tabla F.1. Sustituciones realizadas según cada caso de prueba.

Toda la salida procedente de las múltiples ejecuciones del software de prueba fue grabada en forma consecutiva en un archivo de texto para su posterior análisis.

F.2. PROCESAMIENTO DE LOS RESULTADOS OBTENIDOS

En esta sección se detalla el procesamiento realizado a los resultados obtenidos, incluyendo el procedimiento de reducción de datos en si (sección F.2.1) y algunas de las formas que toman estos datos reducidos (sección F.2.2).

F.2.1. PROCEDIMIENTO DE REDUCCIÓN DE DATOS

Los datos obtenidos en dicho archivo de texto tenían la forma siguiente:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Log iniciado -  Sat May 24 20:31:16 2008
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Run     Gen     MaxLen  MinLen  AvgLen  MaxScore        MinScore        AvgScore        TimeToNext
0       0       121     2       24.485858       1.#INF  1.2849828       1.#INF  0.984
0       1       85      1       19.528571       1.#INF  1.2811304       1.#INF  1.36
0       2       64      1       19.387628       1.#INF  1.281130        1.#INF  1.39
```

<líneas omitidas>

```
9       98      19      13      13.6    1.#INF  1.281130        1.#INF  0.032
9       99      35      13      15.2    16346.5367      1.2811304       1635.806693385764       0.015
BestIndividual: NETLIST\nV1 0 1  AC 1\nR1 1 18 1000
```

<resto de la netlist omitido>

```
BestScoreSoFar: 1.000405370296121
------------------------------------------------------------------------------
Run     Gen     MaxLen  MinLen  AvgLen  MaxScore        MinScore        AvgScore        TimeToNext
0       0       121     2       24.48585858585859       1.#INF  1.284982815312502       1.#INF  0.906
0       1       113     1       19.62448979591837       1.#INF  1.281130495565606       1.#INF  0.172
```

<resto de las líneas omitidas>

Con objeto de permitir su graficación, los datos fueron procesados de la siguiente forma:

- En primer lugar se importaron a una hoja de *Microsoft Excel* en las que se los llevó a un formato estándar, combinando los datos buscados en paralelo en una única hoja. Esta hoja fue exportada a un formato CSV [Shafranovitch, 2005].

- Luego estos datos fueron procesados utilizando el software *Mathematica* de *Wolfram Research* para obtener una versión resumida de los mismos y para graficarlos. Algunos de estos gráficos fueron editados utilizando *Inkscape*, un programa *open source* de edición vectorial.

F.2.2. FORMA DE LOS DATOS REDUCIDOS

Los datos referentes a la relación entre los tiempos de ejecución y los puntajes obtenidos fueron reducidos para obtener un conjunto de datos capaz de ser graficado en forma razonable. Por ejemplo, el conjunto de datos reducidos para efectuar el gráfico de las figura 5.4 tomaba la siguiente forma:

```
{{Gen,SUC Score},
{0,1.28498},{1,1.28113},{2,1.28113},{3,1.28113},{4,1.28113},{5,1.28113},{6,1.28113},{7,1.28113},
{8,1.28113},{9,1.28113},{10,1.28113},{11,1.28113},{12,1.28113},{13,1.28113},{14,1.28113},
{15,1.28113},{16,1.28113},{17,1.28113},{18,1.28113},{19,1.28113},{20,1.28113},{21,1.28113},
{22,1.28113},{23,1.28113},{24,1.28113},{25,1.28113},{26,1.28113},{27,1.28113},{28,1.28113},
```

<líneas omitidas>

```
{89,1.28062},{90,1.28013},{91,1.28013},{92,1.28009},{93,1.27901},{94,1.27864},{95,1.27848},
{96,1.27842},{97,1.27782},{98,1.27444},{99,1.2738},{0,1.6346},{1,1.2849},{2,1.28113},{3,1.28113},
{4,1.28113},{5,1.28113},{6,1.28113},{7,1.28113},{8,1.28113},{9,1.28113},{10,1.28113},
```

<líneas omitidas>

```
{79,1.25315},{80,1.24074},{81,1.23407},{82,1.23397},{83,1.22413},{84,1.22387},{85,1.22385},
{86,1.22191},{87,1.22189},{88,1.22189},{89,1.22189},{90,1.20708},{91,1.20593},{92,1.20586},
{93,1.20581},{94,1.20577},{95,1.20572},{96,1.20431},{97,1.20426},{98,1.20398},{99,1.20392}}
```

Por otro lado, el conjunto de datos utilizado para efectuar el análisis comparativo de la figura 5.16 tenía el siguiente aspecto:

```
{{Base sin caché,{{540.187,1.12433},{1244.08,1.17129},{225.5,1.28113},{187.986,1.2585},
{5849.09,1.19306},{1116.28,1.03415},{165.594,1.2738},{242.734,1.002},{500.814,1.02459},
{742.796,1.20392}}},
{Base con caché,{{426.673,1.05035},{9.014,1.28113},{39.235,1.00104},{13.828,1.28113},
{509.28,1.10227},{14.422,1.2807},{35.876,1.28112},{13.484,1.28113},{36.125,1.24417},
{1322.19,1.07125}}},
{Plaga sin caché,{{206.157,1.13831},{437.171,1.28112},{58.829,1.28113},{564.61,1.06175},
```

```
{76.031,1.27779},{65.422,1.28113},{117.921,1.00414},{365.218,1.00041},{78.25,1.28113},
{76.562,1.28113}}},
{Plaga con caché,{{33.157,1.25619},{3.141,1.28113},{46.828,1.1975},{74.015,1.28025},{6.984,1.28097},
{92.89,1.20004},{12.391,1.00349},{3.953,1.28113},{3.796,1.28113},{2.968,1.28113}}},
{Ajuste dinámico sin caché,{{124.732,1.28113},{1995.74,1.08722},{131.139,1.28113},{4610.31,1.22586},
{5499.36,1.22897},{551.577,1.15519},{164.812,1.28113},{189.861,1.2807},{585.764,1.24408},
{3868.06,1.09512}}},
{Ajuste dinámico con caché,{{6.094,1.28113},{75.889,1.27689},{84.86,1.2654},{1008.02,1.2003},
{90.671,1.01705},{5.345,1.28113},{249.64,1.04085},{10.296,1.28112},{10.173,1.28113},
{299.046,1.0075}}}}
```


En este caso, los datos se componían de un indicador de las técnicas aplicadas y de una lista con los diez pares tiempo – puntaje que componían resumían los datos obtenidos.

Estos datos reducidos fueron luego procesados con el mismo software para producir los gráficos correspondientes.

G. PROCEDIMIENTOS ESTADÍSTICOS UTILIZADOS

En este capítulo se presentan: los motivos por los que las evaluaciones de tiempos se promediaron en forma geométrica y la relación de esta media con la media aritmética de los logaritmos (sección G.1); la forma en que se determinaron los intervalos de confianza para los promedios de los tiempos de ejecución en la sección 5.3.4.4 (sección G.2).

G.1. MEDIA GEOMÉTRICA

En la obtención de valores promedio de tiempos en la sección 5.3.4.4 se utilizó el valor promedio de los logaritmos de los tiempos, que es equivalente a tomar la media geométrica de los tiempos, ya que

$$G(a_1, a_2, \ldots, a_n) = \left(\prod_{i=1}^{n} a_i \right)^{\frac{1}{n}}$$

$$\begin{aligned} \log G(a_1, a_2, \ldots, a_n) &= \log \left(\prod_{i=1}^{n} a_i \right)^{\frac{1}{n}} \\ &= \frac{1}{n} \log \prod_{i=1}^{n} a_i \\ &= \frac{1}{n} \sum_{i=1}^{n} \log a_i . \end{aligned}$$

El motivo por el que se eligió tomar medias geométricas de los tiempos es que reflejan adecuadamente la naturaleza multiplicativa de las mejoras efectuadas en los rendimientos y los pesos relativos de los tiempos, evitando que estos se vean excesivamente sesgados por algunos valores extremos. Otra ventaja de trabajar en una escala logarítmica es que evita el sinsentido de que los intervalos de confianza de los tiempos de ejecución se extiendan a valores negativos, cuando estos tiempos son positivos por definición.

G.2. INTERVALOS DE CONFIANZA

Los intervalos de confianza dan límites dentro de los cuales un parámetro se encuentra con una cierta probabilidad. Para el caso de este trabajo, los valores en cuestión eran las medias geométricas de los tiempos observados al realizar un proceso de diseño aplicando distintas combinaciones de las técnicas de optimización.

La ventaja de poseer intervalos de confianza es que permiten estimar si las diferencias observadas entre dos parámetros son significativas o no. Por ejemplo, si los intervalos de confianza, con un nivel de 0.95, de dos parámetros no se superponen, podemos decir que los parámetros son distintos con una probabilidad mayor que $0.95^2 \approx 0.9$. La probabilidad asignada a los intervalos de confianza se denomina "nivel de confianza".

Para el caso del presente trabajo, los intervalos de confianza de las medias geométricas fueron determinados a partir de los logaritmos de los tiempos utilizando el software *Mathematica* y su función `MeanCI[]`. Esta función opera mediante el uso de una distribución t de Student con n – 1 grados de libertad [Wolfram Research, 2008], o sea 9 en el presente caso, ya que debe estimar también la varianza de los datos. Puede encontrarse más información sobre el uso de esta distribución para estimar intervalos de confianza en [Mason *et al.*, 2003].

Debe destacarse que esta función asume que los datos están distribuidos normalmente, cosa que no puede asegurarse en el presente caso. Por lo tanto, los niveles de confianza no pueden tomarse en forma absoluta sino simplemente como una guía respecto a la confianza que puede tenerse respecto a los resultados obtenidos.