Universidad de Murcia — Facultad de Informática

Máster en Informática y Matemáticas Aplicadas en Ciencias e Ingeniería

Tesis de Máster

Técnicas Heurísticas para Problemas de Diseño en Telecomunicaciones

Autor:
José Ceferino Ortega Carretero

Directores:
Domingo Giménez Cánovas[1]
Fernando Daniel Quesada Pereira[2]

9 de septiembre de 2008

[1]Dpto. de Informática y Sistemas, *Universidad de Murcia*
[2]Dpto. de Tecnologías de la Información y las Comunicaciones, *Universidad Politécnica de Cartagena*

Índice general

1. Introducción

En el mundo del diseño de dispositivos de telecomunicaciones, existe una gran cantidad de problemas que requieren de la aplicación de técnicas de optimización para hallar los mejores valores para diversos parámetros, pudiendo ser éstos de naturaleza discreta (un conjunto finito de valores posibles, como por ejemplo un subconjunto de los números enteros) o continua (un conjunto potencialmente infinito de valores posibles, usualmente números reales) [8, 17]. Generalmente esta optimización no es abordable mediante técnicas analíticas, y por tanto debe enfocarse mediante el empleo de técnicas de optimización heurísticas [21]. Estas técnicas de optimización heurísticas pueden ser técnicas exhaustivas (o completas) de resolución de problemas de optimización, tales como los esquemas algorítmicos *Backtracking* (vuelta atrás) y *Branch & Bound* (Ramificación y Poda) [5], que tienen la ventaja de que siempre encuentran la solución óptima, pero, dado que el espacio de soluciones posible puede ser complejo y muy amplio, la búsqueda es difícil de acotar incluso mediante el empleo de heurísticas, y por tanto suelen dar lugar a algoritmos poco eficientes para tamaños de problema medianos o grandes. También se pueden utilizar técnicas de optimización heurísticas no exhaustivas (o no completas) en las que la exploración del espacio de soluciones generalmente se realiza de una forma más "inteligente", pudiendo perderse la solución óptima al no examinarse todas las posibles, pero a cambio se obtienen soluciones próximas al óptimo (o de alta calidad) de forma más eficiente, entendiendo por eficiente el que se ajuste a las restricciones impuestas en el uso de los recursos computacionales dependiendo de la aplicación (por ejemplo, si se trata de una aplicación que debe ejecutarse en un dispositivo móvil como un teléfono, se dispondrá de poco poder computacional y muchas restricciones de memoria, mientras que si se trata de una aplicación usada en una estación de trabajo estas restricciones serán menores y la potencia de cálculo muy superior). Estas técnicas no exhaustivas se conocen con el nombre de *metaheurísticas*. Normalmente, las técnicas metaheurísticas suelen ser esquemas algorítmicos basados en distintas ideas, en muchas ocasiones tomadas del funcionamiento de la naturaleza [6], pero que tienen en común el enfoque del problema resolviéndolo mediante sucesivas mejoras de una solución o conjunto de soluciones, con una exploración del espacio de soluciones más amplia y, en la mayoría de ellos, con cierto factor aleatorio [11].

En este trabajo se estudia la aplicación de algunas de estas técnicas heurísticas a los problemas de diseño en telecomunicaciones, analizando su comportamiento para decidir la conveniencia o no de su aplicación, así como, de entre todas ellas, ver cuáles se comportan mejor, bajo qué condiciones y con qué ajuste de parámetros.

1.1. Técnicas de Optimización Heurísticas

Existen numerosas técnicas de optimización, pero para los problemas que nos ocupan, las vamos a clasificar en dos tipos de técnicas: exhaustivas y no exhaustivas. En ambos casos, se trata de técnicas computacionales no analíticas, y la clasificación se basa en el examen que realizan del espacio de soluciones. También es importante hacer notar que en todas estas técnicas un elemento fundamental que mejora de forma determinante la eficiencia de las mismas es el empleo de heurísticas. Respecto a la heurística, en la *Wikipedia* se nos dice que "*en cualquier problema de búsqueda donde hay* b *opciones en cada nodo y una profundidad* d *al nodo objetivo, un algoritmo de búsqueda ingenuo deberá buscar potencialmente entre* b^d *nodos antes de encontrar la solución. Las heurísticas mejoran la eficiencia de los algoritmos de búsqueda reduciendo el factor de ramificación de* b *a (idealmente) una constante* b*". Básicamente, heurística podríamos definirla en este contexto como "*conocimiento que nos lleva a acercarnos más rápidamente a la solución*".

1.1.1. Técnicas Exhaustivas

Las técnicas de optimización exhaustivas son aquellas que garantizan encontrar siempre el óptimo (máximo o mínimo) recorriendo en el peor de los casos todo el espacio de soluciones (que puede ser enorme). De este tipo de técnica nos vamos a centrar en dos, que son esquemas algorítmicos ampliamente utilizados y bien conocidos: *Backtracking* (vuelta atrás) y *Branch & Bound* (ramificación y poda).

El esquema algorítmico *Backtracking* realiza un recorrido exhaustivo del árbol de soluciones, de una forma ordenada en profundidad. Normalmente se puede mejorar la eficiencia de este esquema algorítmico añadiendo heurísticas en el orden en el que se realiza el recorrido del árbol de soluciones y añadiendo criterios para poder descartar ramas que es imposible que conduzcan a una solución mejor que la actualmente obtenida o que no conducen a una solución que cumpla las restricciones del problema.

Por otro lado, el esquema algorítmico *Branch & Bound* se basa en ir recorriendo el conjunto de nodos activos que representan las soluciones intermedias en base a la bondad que puede tener la solución que de ellos se pueda obtener. Generalmente, de cada uno de los nodos se obtiene una cota superior e inferior mediante alguna heurística, de tal modo que los nodos aún no tratados que tengan una cota superior que sea inferior a la mejor cota inferior pueden ser descartados. De alguna manera, el esquema *Branch & Bound* podría ser visto como una mejora del *Backtracking* cuando existe conocimiento heurístico suficiente del problema como para hallar las citadas cotas, o, al menos, para lograr una mejor ordenación global de la búsqueda y descartar mayor número de ramas del árbol de soluciones.

En este trabajo vamos a aplicar la técnica *Backtracking* a nuestro problema de diseño en telecomunicaciones, aunque por las características del mismo no se adecua especialmente bien, pero así podremos compararlo con las técnicas no exhaustivas y podremos observar lo interesante de estas técnicas para problemas como el planteado.

1.1.2. Técnicas No Exhaustivas

A diferencia de las técnicas exhaustivas, en las técnicas no exhaustivas lo que prima no es encontrar la mejor solución (el óptimo) a cualquier precio, si no que en este caso nos interesa conseguir soluciones suficientemente buenas (de alta calidad) sin que el coste de hallarlas exceda las restricciones de tiempo o memoria que se impongan (hay que tener en cuenta que los problemas de optimización del tipo que estamos tratando tienen un espacio de soluciones sumamente complejo y que recorrerlo por completo puede no ser viable para determinadas aplicaciones y/o entornos de ejecución). En este tipo de técnicas, lo que se hace es ir trabajando con una solución o conjunto de soluciones para obtener nuevas soluciones que se acerquen más al óptimo o que exploren nuevas zonas del espacio de soluciones con el fin de evitar los óptimos locales, y, de forma iterativa, lograr una convergencia a una solución de *alta calidad*. Normalmente no se podrá garantizar que la solución obtenida sea el óptimo (con frecuencia no lo será), pero se podrá garantizar la calidad de la solución, dado que ésta cumplirá con los criterios buscados.

Existen numerosas técnicas de optimización metaheurísticas ya que es un campo donde se está realizando una gran investigación como puede verse en [11]. Algunas de las más conocidas son:

- Algoritmos Genéticos [12]
- Búsqueda Tabú [9]
- Optimización basada en Colonia de Hormigas [7]
- GRASP [20]
- Búsqueda Dispersa (*Scatter Search*) [10]
- Temple Simulado [13]

Estas técnicas, como hemos dicho, están abiertas a la investigación, y es común mezclarlas con otras técnicas o entre ellas para mejorar la calidad de la solución obtenida o para mejorar su eficiencia. En ese caso se conocen como *Metaheurísticas Híbridas* [18].

En este trabajo aplicaremos las técnicas de Búsqueda Dispersa, Algoritmos Genéticos y Temple Simulado al problema de diseño en telecomunicaciones, comparando los resultados obtenidos con cada una de ellas y con las otras técnicas. También se aplicarán mejoras tales como la hibridación con técnicas de búsqueda local para obtener resultados más satisfactorios.

1.2. Trabajos Relacionados

En este apartado vamos a ver qué otros trabajos se han realizado anteriormente sobre el empleo de técnicas de optimización heurísticas en los problemas de diseño en telecomunicaciones. Dado que es un campo de estudio muy amplio, veremos algunos trabajos de este tipo en el ámbito general del diseño en telecomunicaciones, y después nos centraremos en el estudio de los trabajos directamente relacionados con el problema concreto en el que nos hemos centrado.

Entre los problemas de diseño en telecomunicaciones, uno de los más importantes es el diseño de redes de comunicaciones. El diseño de estas redes es un problema complejo que requiere de optimización combinatoria de alto coste. En [21] se revisan diversos trabajos al respecto usando distintas técnicas metaheurísticas.

El problema del diseño de filtros con resonadores ha sido ampliamente analizado durante varios años y se han propuesto diversas estrategias para casos concretos, pero en general no existen reglas para su resolución analítica ni sistemática en todos los casos. M. Uhm, S. Nam y J. Kim han propuesto recientemente el empleo de una técnica metaheurística híbrida basada en un algoritmo genético combinado con un algoritmo voraz de búsqueda local específico para este problema [22].

En este trabajo se exploran diversas técnicas metaheurísticas aplicadas al problema del diseño de filtros con resonadores, pero con un planteamiento más genérico, buscando comparar las diversas técnicas entre sí para la resolución de este problema, pero teniendo en mente la idea de poder aplicar lo aprendido en este estudio a otros problemas de este ámbito o de otros ámbitos en los que el planteamiento sea parecido, y por tanto, evitando siempre la dependencia excesiva del problema concreto y de su representación.

2. Análisis de las Herramientas de Optimización

En este capítulo analizamos las herramientas que vamos a usar para abordar los problemas de optimización planteados. Dado que en el ámbito del diseño en telecomunicaciones es muy habitual trabajar en un entorno de cálculo numérico como MATLAB, nos vamos a centrar en las herramientas (*toolboxes*) disponibles en este entorno relacionadas con la optimización, y, además, veremos otras técnicas heurísticas que hemos implementado en este entorno que no están disponibles en los *toolboxes* y que potencialmente pueden ayudarnos a resolver los problemas planteados, o al menos nos ayudarán a completar el estudio.

2.1. Introducción a la Teoría de Optimización

Las técnicas de optimización, basándonos en [17], se usan para encontrar un conjunto de parámetros de diseño, $x = \{x_1, x_2, ..., x_n\}$, que, de alguna manera, pueden ser definidos como óptimos. Una formulación típica consiste en una función objetivo, $f(x)$, que debe ser minimizada o maximizada, y que podría ser objeto de restricciones, ya sean de igualdad, $G_i(x) = 0\,(i = 1, ..., m_e)$; de desigualdad, $G_i(x) \leq 0\,(i = m_e + 1, ..., m)$; y/o de límites en los parámetros, x_l, x_u.

Una descripción general del problema puede verse como

$$\min_x f(x)$$

sujeto a

$$\begin{cases} G_i(x) = 0 & i = 1, ..., m_e \\ G_i(x) \leq 0 & i = m_e + 1, ..., m \end{cases}$$

donde x es el vector de longitud n de los parámetros de diseño, $f(x)$ es la función objetivo, que devuelve un valor escalar, y la función vectorial $G_i(x)$ devuelve un vector de longitud m que contiene los valores de las restricciones de igualdad y desigualdad evaluados en x.

Un solución eficiente y precisa a este problema depende no sólo del tamaño del problema en términos del número de restricciones y del número de parámetros de diseño sino también de las características de la función objetivo y de las restricciones. Cuando la función objetivo $f(x)$ y las restricciones $G_i(x)$ son funciones lineales de los parámetros de diseño, el problema se dice que es un problema de *Programación Lineal*. La *Programación Cuadrática* trata sobre la minimización o maximización de un función objetivo cuadrática

que esta restringida linealmente. Para ambos tipos de problema, existen procedimientos de resolución efectivos. Más difíciles de resolver son los problemas de *Programación No-lineal* en los que la función objetivo y las restricciones pueden no ser funciones lineales de los parámetros de diseño. La resolución de los problemas de programación no lineal normalmente requiere un procedimiento iterativo de mejora en el que habitualmente en cada iteración se establece la dirección de búsqueda resolviendo un subproblema lineal, cuadrático o sin restricciones.

2.2. *Toolboxes* de optimización de MATLAB

MATLAB, en su versión R2008a, dispone de dos *toolboxes* relacionados con la optimización. Veamos las herramientas que nos ofrece cada uno de ellos que son de interés para nuestro estudio. La información aquí expuesta está extraída de las respectivas guías de usuario que acompañan a dicho software.

2.2.1. *Optimization Toolbox*

El primero de ellos, denominado *Optimization Toolbox* [16], incluye funciones de optimización de muchos tipos, incluyendo:

- Minimización no lineal sin restricciones
- Minimización no lineal con restricciones, incluyendo problemas de minimización semi-infinita
- Programación lineal y cuadrática
- Ajuste de curvas y mínimos cuadrados no lineales
- Mínimos cuadrados lineales con restricciones
- Problemas dispersos y estructurados de larga escala, incluyendo programación lineal y minimización no lineal con restricciones

Nos interesa de este *toolbox*, principalmente, sus funciones de búsqueda (minimización) local (pueden consultarse las funciones más interesantes en el Apéndice A).

2.2.2. *Genetic Algorithm and Direct Search Toolbox*

El segundo *toolbox* de optimización disponible en MATLAB, denominado *Genetic Algorithm and Direct Search Toolbox* [15], se centra en los algoritmos genéticos y de búsqueda directa para resolver problemas de optimización global complejos, como los que hemos presentado. En este *toolbox* nos encontramos con rutinas para resolver problemas de optimización usando:

- Búsqueda Directa
- Algoritmos Genéticos
- Temple Simulado

Además, permite personalizar las reglas o funciones usadas permitiendo al usuario escribir sus propias funciones.

Veamos en que consiste cada una de estas técnicas.

2.2.2.1. Búsqueda Directa

La búsqueda directa es un método para resolver problemas de optimización que no requiere ninguna información acerca del gradiente de la función objetivo. A diferencia de los métodos de optimización más tradicionales que usan la información sobre el gradiente o sucesivas derivadas para buscar un punto óptimo, un algoritmo de búsqueda directa explora un conjunto de puntos alrededor del punto actual, buscando uno donde el valor de la función objetivo sea menor que el valor del punto actual. Esto permite usar la búsqueda directa para resolver problemas con una función objetivo no diferenciable o incluso no continua.

Este *toolbox* incluye dos algoritmos de búsqueda directa llamados algoritmo de búsqueda de patrón generalizada (GPS) y algoritmo de búsqueda de malla adaptativa (MADS). Ambos son patrones de algoritmos de búsqueda de patrones que computan una secuencia de puntos que aproximan al punto óptimo. En cada paso, el algoritmo busca un conjunto de puntos, llamado malla, alrededor del punto actual — el punto calculado en el paso anterior. La malla se forma añadiendo al punto actual un conjunto de vectores (llamados patrón) multiplicados por un escalar. Si el algoritmo de búsqueda de patrones encuentra un punto en la malla que mejora el valor de la función objetivo en el punto actual, el nuevo punto se convierte en el punto actual para el siguiente paso del algoritmo.

2.2.2.2. Algoritmos Genéticos

Los algoritmos genéticos son un método para resolver problemas de optimización tanto con restricciones como sin ellas, basado en la selección natural, el proceso que guía la evolución natural. En este método se modifica repetidamente una población de soluciones individuales. En cada paso, se seleccionan individuos de la población actual para ser padres y se usan para producir hijos que forman la siguiente generación. A través de sucesivas generaciones, la población evoluciona hacia una solución óptima. Se pueden aplicar los algoritmos geneticos para resolver una gran variedad de problemas que no se ajustan bien a los algoritmos de optimización clásicos, incluyendo problemas para los cuales la función objetivo es discontinua, no diferenciable, estocástica o altamente no lineal. Los algoritmos genéticos usan principalmente tres tipos de reglas en cada paso para crear la siguiente generación a partir de la actual:

- *reglas de selección* de los individuos, llamados *padres*, que contribuirán a la población de la siguiente generación
- *reglas de cruzamiento* para combinar los *padres* por parejas y obtener *hijos* para la siguiente generación
- *reglas de mutación* que aplican cambios a *padres* individuales para obtener *hijos*

Algoritmo 2.1 Esquema de los Algoritmos Genéticos

```
1. Crear una población inicial aleatoria.
2. Repetir hasta que se cumpla alguna condición de parada:
 a) Puntuar cada miembro de la población actual con su valor de bondad.
 b) Calcular el valor de expectación en base al valor de bondad.
 c) Seleccionar a los miembros - padres - en base a su valor de expectación.
 d) Crear el grupo de elite con los mejores miembros de la población.
 e) Obtener hijos de los padres, mediante cambios aleatorios a un solo padre -
     mutación - y por combinación de dos padres - cruce.
 f) Actualizar la población tomando la elite y los hijos obtenidos.
```

Los algoritmos genéticos difieren de los algoritmos de optimización clásicos, basados en las derivadas, de dos formas principalmente, que se pueden resumir en la siguiente tabla:

Algoritmo Clásico	**Algoritmo Genético**
Genera un solo punto en cada iteración. La secuencia de puntos se aproxima a la solución óptima.	Genera una población de puntos en cada iteración. El mejor punto de la población se aproxima a la solución óptima.
Selecciona el siguiente punto de la secuencia de una forma determinista.	Selecciona la siguiente población mediante un cálculo que suele basarse, al menos en parte, en el azar.

Un esquema general de esta técnica puede verse en algoritmo 2.1.

2.2.2.3. Temple Simulado

El temple simulado es un método para resolver problemas sin restricciones y con restricciones de límites en los valores de los parámetros. El método modela el proceso físico de calentar un material y después ir enfriándolo lentamente con el fin de decrementar defectos, minimizando de este modo la energía del sistema.

En cada iteración del algoritmo, se genera un nuevo punto al azar. La distancia del nuevo punto al punto actual, o la extensión de la búsqueda, se basa en una distribución de probabilidad con una escala proporcional a la temperatura. El algoritmo acepta todos los nuevos puntos que mejoren el objetivo, pero también, con una cierta probabilidad, puntos que empeoren el objetivo. Al aceptar puntos que empeoren el objetivo, el algoritmo evita quedarse atrapado en un óptimo local, y es capaz de explorar globalmente más posibles soluciones. Se usa un temple periódico para bajar sistemáticamente la temperatura conforme avanza el algoritmo. Como la temperatura baja, el algoritmo reduce la extensión de su búsqueda para converger a un óptimo.

2.3. Otras Técnicas de Optimización

Además de las técnicas de optimización disponibles en los *toolboxes* de MATLAB, hemos implementado otras dos técnicas adicionales que se incluyen en este estudio.

Algoritmo 2.2 Esquema de la técnica metaheurística *Scatter Search*

1. Generar un conjunto inicial de soluciones (P), del cual hay que escoger un conjunto de referencia (R) seleccionando los mejores individuos. La noción de mejor no se refiere en exclusiva a los individuos con mejor valor en la función de bondad, sino que también interesan los individuos que aporten diversidad a R, de tal modo que individuos con peor valor de bondad pero que aporten diversidad también se incluyen en R.
2. Crear Subconjuntos. Crear de forma sistematica los posibles subconjuntos de R a tratar. Por ejemplo todos los conjuntos de pares de elementos de R.
3. Combinar. Consiste en obtener nuevas soluciones a partir de combinaciones de los elementos de los subconjuntos de R creados en el paso 2.
4. Mejorar cada solución obtenida mediante el uso de una heurística.
5. Incorporar las soluciones obtenidas a R cuando mejoren (bondad o diversidad).
6. Repetir los pasos del 2 al 5 hasta que R no cambie o se cumpla alguna de las condiciones de parada. Cuando R no cambie, volver al punto 1 reintroduciendo elementos que mejoren la diversidad de un nuevo P pero conservando los elementos mejores en cuanto a bondad en R.

Las técnicas seleccionadas han sido *Scatter Search* (Búsqueda Dispersa) y *Backtracking* (Vuelta Atrás). *Scatter Search* se ha seleccionado para completar el grupo de técnicas metaheurísticas, ya que es una técnica más moderna que los Algoritmos Genéticos con gran actividad en el campo de la investigación. Por otro lado, *Backtracking* se ha incluido en el estudio por completitud, para poder observar el comportamiento de una técnica exhaustiva en este problema y compararla con el resto, aun sabiendo que por las características del mismo, principalmente por el inmenso espacio de soluciones, no vamos a obtener grandes resultados.

2.3.1. Scatter Search

Scatter Search (Búsqueda Dispersa) es una técnica metaheurística perteneciente a la familia de los algoritmos evolutivos, a la que también pertenecen los algoritmos genéticos, ya que se basa en el trabajo con una población, que mediante sucesivas iteraciones o generaciones, va evolucionando hacia una solución óptima. Un esquema de esta técnica (basado en lo descrito en [10]) se puede ver en la plantilla del algoritmo 2.2.

En el apéndice B puede verse el código fuente de la implementación de esta técnica en MATLAB.

2.3.2. Backtracking

Backtracking (Vuelta Atrás) es una técnica de optimización combinatoria clásica [5] que realiza una exploración exhaustiva de un conjunto de soluciones bien definido (idealmente el espacio de soluciones completo si es posible), como se ha explicado en la introducción.

En el pseudocódigo de algoritmo 2.3 (extraído de [5]) se describe esta técnica.

Las funciones en las que se basa este esquema algorítmico son:

- `generar(nivel,s)` genera una nueva solución para el nivel `nivel` siendo la solución parcial actual `s`.

Algoritmo 2.3 Pseudocódigo de *Backtracking*

```
nivel:=1
repetir
  s[nivel]:=generar(nivel,s)
  si solucion(nivel,s) entonces
    tratar(s)
  finsi
  si criterio(nivel,s) entonces
    nivel:=nivel+1
  sino
    mientras nivel<>0 Y NO mashermanos(nivel,s) hacer
      retroceder(nivel,s)
    finmientras
  finsi
hasta nivel=0
```

- `solución(s)` dice si la solución parcial `s` es una solución viable que cumple todas las condiciones.
- `tratar(s)` realiza el tratamiento oportuno de la solución `s`, en este caso calcular su valor de bondad y compararla con la mejor solución encontrada hasta ahora, guardando la mejor solución.
- `criterio(nivel,s)` dice si se puede seguir construyendo una solución a partir de la solución parcial `s`.
- `mashermanos(nivel,s)` dice si quedan más nodos por explorar en el nivel actual.
- `retroceder(nivel,s)` deshace lo hecho para llegar la solución `s` y retrocede un nivel.

En el apéndice C puede verse el código fuente de la implementación de esta técnica en MATLAB.

3. Aplicación a la Síntesis de Filtros

Vamos a comenzar nuestro estudio centrándonos en el problema de la síntesis de filtros con resonadores acoplados, empezando por un pequeño análisis del problema en sí mismo pero desde un punto de vista computacional, para a continuación aplicar las técnicas vistas en el capítulo 2 que tengan sentido para este problema, realizando un estudio experimental.

3.1. Análisis del Problema de Síntesis de Filtros con Resonadores Acoplados

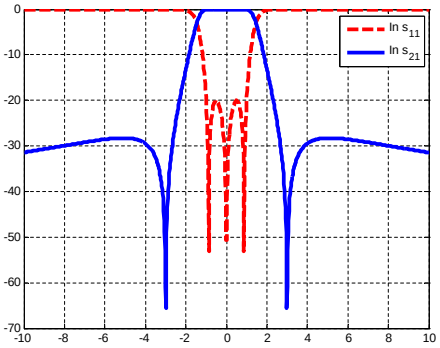
Los resonadores de microondas acoplados son componentes esenciales en los sistemas de comunicaciones modernos. Las estructuras de filtrado con requisitos cada vez más estrictos a menudo sólo pueden conseguirse mediante el uso de resonadores acoplados transversalmente conteniendo un número de ceros de transmisión finito ya que el número de ceros de transmisión está directamente relacionado con las características de los filtros. Una teoría general de este tipo de filtros y otros parecidos [4] permite que, para diseñar un determinado filtro, se parta de una matriz de acoplamientos completa que determinaría el comportamiento del mismo si se dispusiese de una tecnología capaz de implementar una red de resonadores cuya topología fuese que todos los resonadores estuviesen acoplados entre sí. La realidad es que ese tipo de tecnología generalmente no es viable o no se ajusta a las restricciones del dispositivo al que pertenece el filtro que se intenta diseñar, de tal modo que lo que se pretende es hallar la matriz de acoplamientos que, cumpliendo las restricciones impuestas por la tecnología disponible, más se aproxime al comportamiento deseado. Por lo general, las restricciones que debe cumplir la matriz se refieren a la inexistencia de acoplamiento entre dos resonadores dados, hecho que viene reflejado en la matriz mediante un 0 en la posición correspondiente al valor del acoplamiento entre esos dos resonadores [2]. Los valores de los elementos de la matriz también están condicionados por la tecnología subyacente, de tal modo que tendrán que ser valores reales dentro de un rango determinado.

En nuestro estudio vamos a trabajar con 6 diseños concretos, cuatro de los cuales cuentan con 9 parámetros de diseño a optimizar y otros dos con 8. En todos los casos, estos parámetros de diseño pueden tomar valores reales entre -5 y 5 por la tecnología subyacente tal y como se explica en el párrafo anterior. En la figura 3.1 se puede ver de forma visual como es el comportamiento deseado de algunos de los filtros correspondientes a los diseños en estudio (el resto es similar).

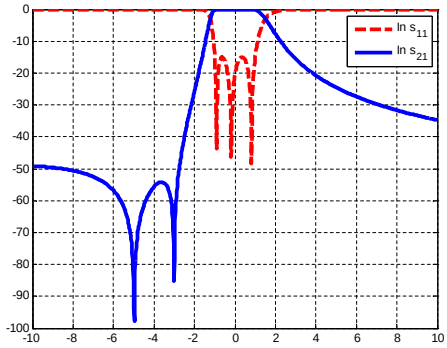
Estos son los nombres con los que hemos designado los 6 diseños:

- `tcometa-3y3`: Topología *Cometa* y ceros en -3 y 3, 8 parámetros de diseño

Figura 3.1.: Representación de la respuesta deseada



(a) Topología *Cometa* con ceros en -3 y 3



(b) Topología *Dos Trisection de Orden Tres* con ceros en -5 y -3

- `tcometa-5y3`: Topología *Cometa* y ceros en -5 y -3, 8 parámetros de diseño
- `ttransversal-3y3`: Topología *Transversal* y ceros en -3 y 3, 9 parámetros de diseño
- `ttransversal-5y3`: Topología *Transversal* y ceros en -5 y -3, 9 parámetros de diseño
- `t2trisection3-3y3`: Topología *Dos Trisection Orden 3* y ceros en -3 y 3, 9 parámetros de diseño
- `t2trisection3-5y3`: Topología *Dos Trisection Orden 3* y ceros en -5 y -3, 9 parámetros de diseño

En el apéndice D se detallan las funciones de bondad asociadas a estos diseños.

3.2. Resultados Experimentales

Dado que el problema que hemos descrito en el apartado anterior trabaja con vectores de números reales dentro del intervalo -5 y 5 y teniendo en cuenta el tipo de gráfica (véase la figura 3.1), este problema puede considerarse no lineal y con restricciones de limites superior e inferior en los valores de los parámetros de diseño.

Esto implica que debemos descartar aquellas herramientas que sólo trabajan sobre una sola variable y las que sólo trabajan sin restricciones. Tampoco nos interesan las ideadas para trabajar con multiobjetivos (las funciones objetivo devuelven un vector en lugar de un escalar) y las que requieren que se le suministren gradientes u otro tipo de información adicional distinta de la función objetivo.

Las herramientas de optimización aplicables las vamos a estudiar de forma experimental en este apartado, realizando en primer lugar un ajuste de los parámetros de cada una

de las herramientas mediante un estudio individual, pasando a continuación a compararlas entre sí teniendo en cuenta tanto la calidad de las soluciones obtenidas como el rendimiento en cuanto a tiempo de ejecución.

3.2.1. Estudio Individual

3.2.1.1. Diseño del Experimento

Para llevar a cabo este estudio, los experimentos se han llevado a cabo teniendo en cuenta los siguientes puntos:

- Se ha repetido múltiples veces la ejecución de cada configuración y se han teniendo en cuenta todos los resultados a través de la media aritmética y la desviación típica.
- Se ha evaluado parámetro por parámetro, manteniendo fijos los que no se están evaluando en cada momento excepto cuando la interacción entre dos o más de ellos es conocida de antemano (parámetros dependientes), en cuyo caso se ha probado todas las combinaciones entre los mismos.
- Para cada herramienta, se han seleccionado para su estudio aquellos parámetros que son significativos para el problema, fijando el resto a sus valores por defecto.
- Los parámetros relativos al número de iteraciones/generaciones y, en su caso, el tamaño de la población, se han tratado de modo distinto para ver su efecto en el resto de los parámetros.
- En el caso de las herramientas que requieren un punto inicial para realizar la optimización, en cada uno de los ciclos de ejecución se ha fijado un punto inicial al azar que se ha mantenido para la evaluación de todas las configuraciones.

3.2.1.2. fmincon

La herramienta `fmincon` pertenece al *Optimization Toolbox* de MATLAB (véase el apéndice A para ver con más detalle el funcionamiento de esta herramienta) y puede catalogarse como una herramienta de búsqueda local con restricciones para funciones multivariable no lineales, ajustándose por tanto a las características de este problema.

De esta herramienta se han estudiado las siguientes características (se describen aquí brevemente por motivos de espacio, para obtener una descripción más completa puede consultarse [16]):

- *LargeScale*: Activa (on) o desactiva (off) las características de algoritmo de gran escala.
- *Algorithm*: Permite especificar el algoritmo que usa `fmincon` interiormente. Existen dos opciones válidas para este problema: interior-point y active-set.

En la figura 3.2 pueden observase algunos los efectos de estos parámetros en el valor de *fitness*. En general, el comportamiento es bastante dispar dependiendo de la función objetivo. El parámetro *LargeScale* parece tener poca influencia en el resultado. En cuanto al parámetro *Algorithm*, con el valor interior-point se obtienen mejores resultados para

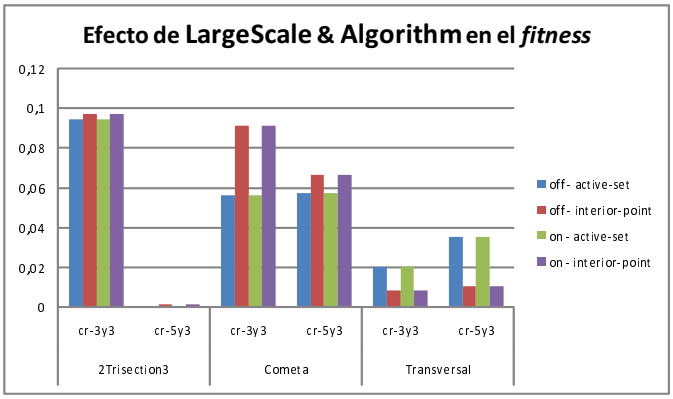


Figura 3.2.: Efecto de los parámetros *LargeScale* y *Algorithm* de `fmincon` en el *fitness*

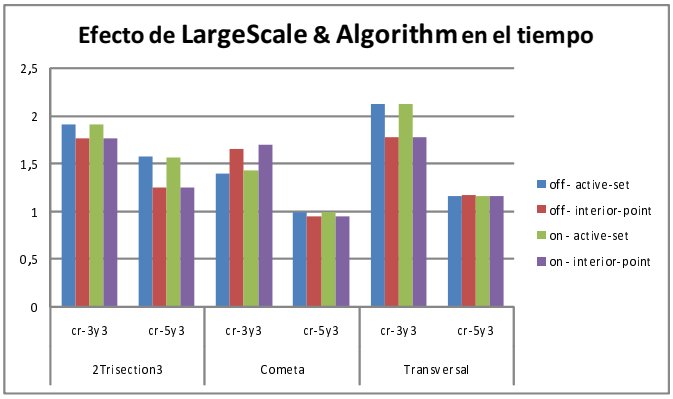


Figura 3.3.: Efecto de los parámetros *LargeScale* y *Algorithm* de `fmincon` en el tiempo

la topología *Transversal*, siendo a la inversa con la topología *Cometa*, mientras que no se aprecian diferencias significativas en la topología *Dos Trisection de Orden Tres*.

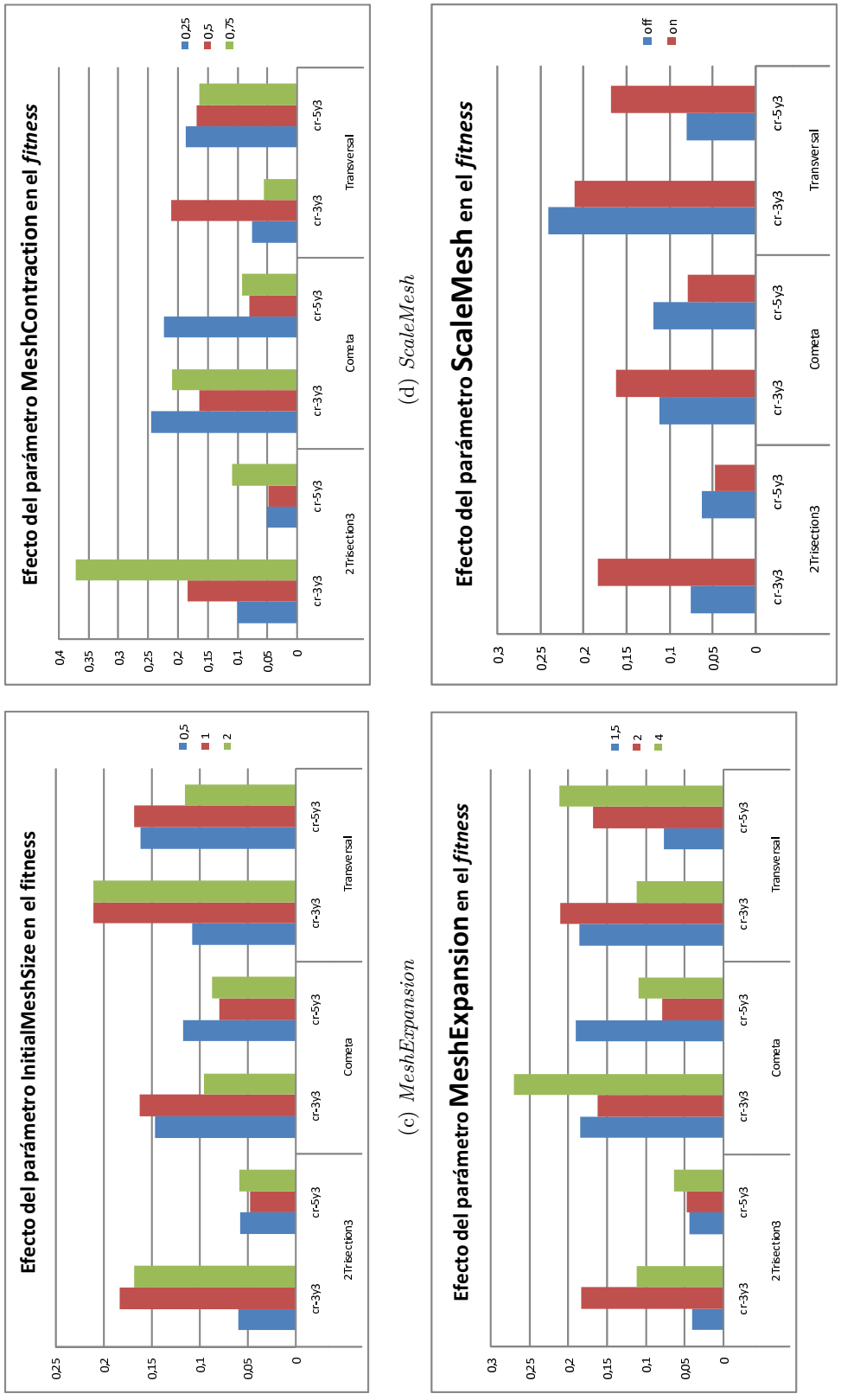
En cuanto a los tiempos de ejecución, en la figura 3.3 no se observan diferencias especialmente destacables entre las configuraciones estudiadas.

3.2.1.3. patternsearch (Búsqueda Directa)

La herramienta `patternsearch` (Búsqueda Directa) pertenece al *Direct Search and Genetic Algorithm Toolbox* de MATLAB [15] (véase el apartado 2.2.2.1 en la página 10), y, al igual que `fmincon`, puede catalogarse como una herramienta de búsqueda local con restricciones para funciones multivariable no lineales.

En este caso, los parámetros que se han evaluado son los siguientes (para un descripción detallada de los mismos se puede consultar [32]):

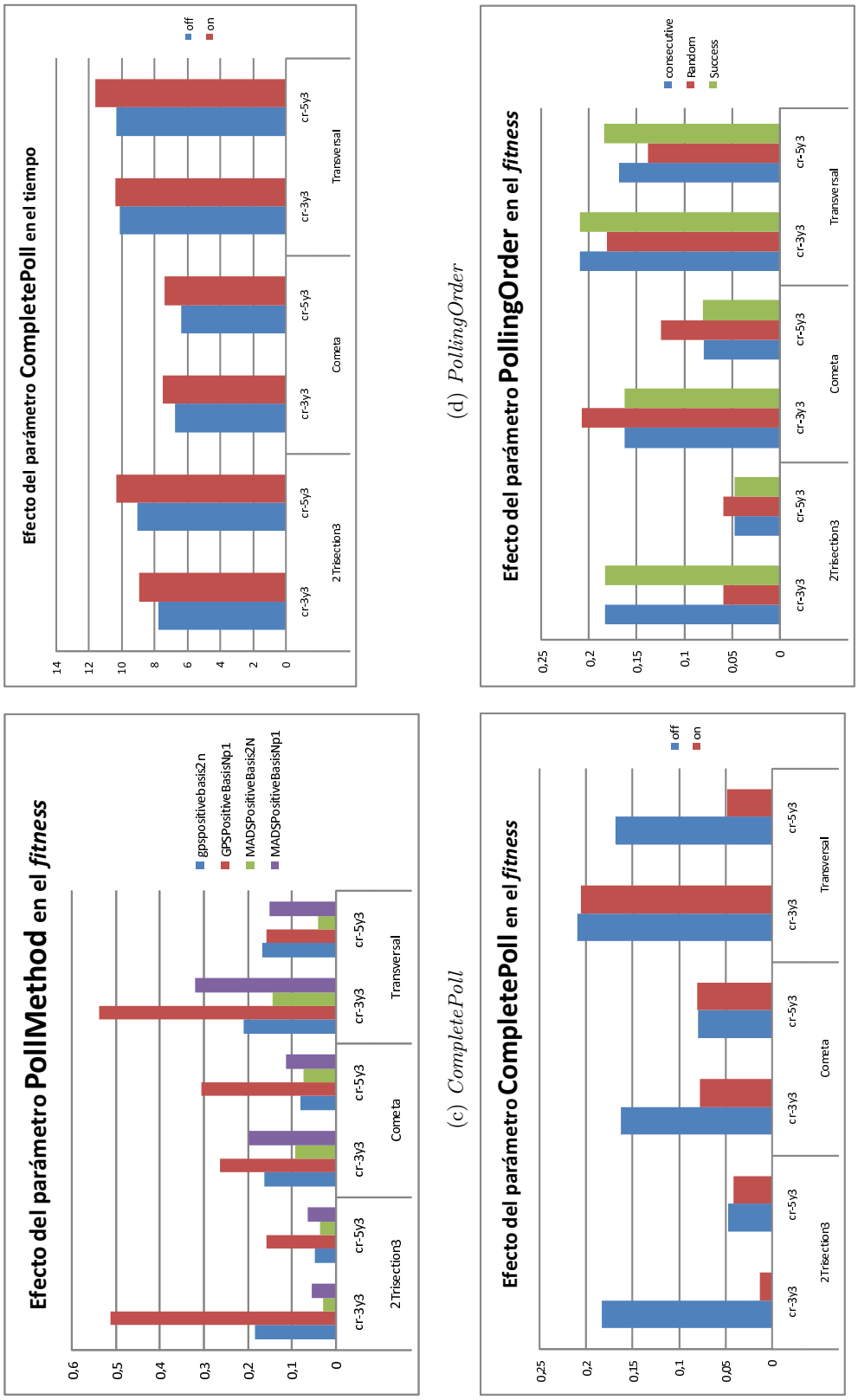
- *InitialMeshSize*: Tamaño inicial de la malla.
- *MeshContraction*: Factor por el que multiplicar la malla para contraerla cuando se aproxima a un mínimo.



(a) *InitialMeshSize* (b) *MeshContraction* (c) *MeshExpansion* (d) *ScaleMesh*

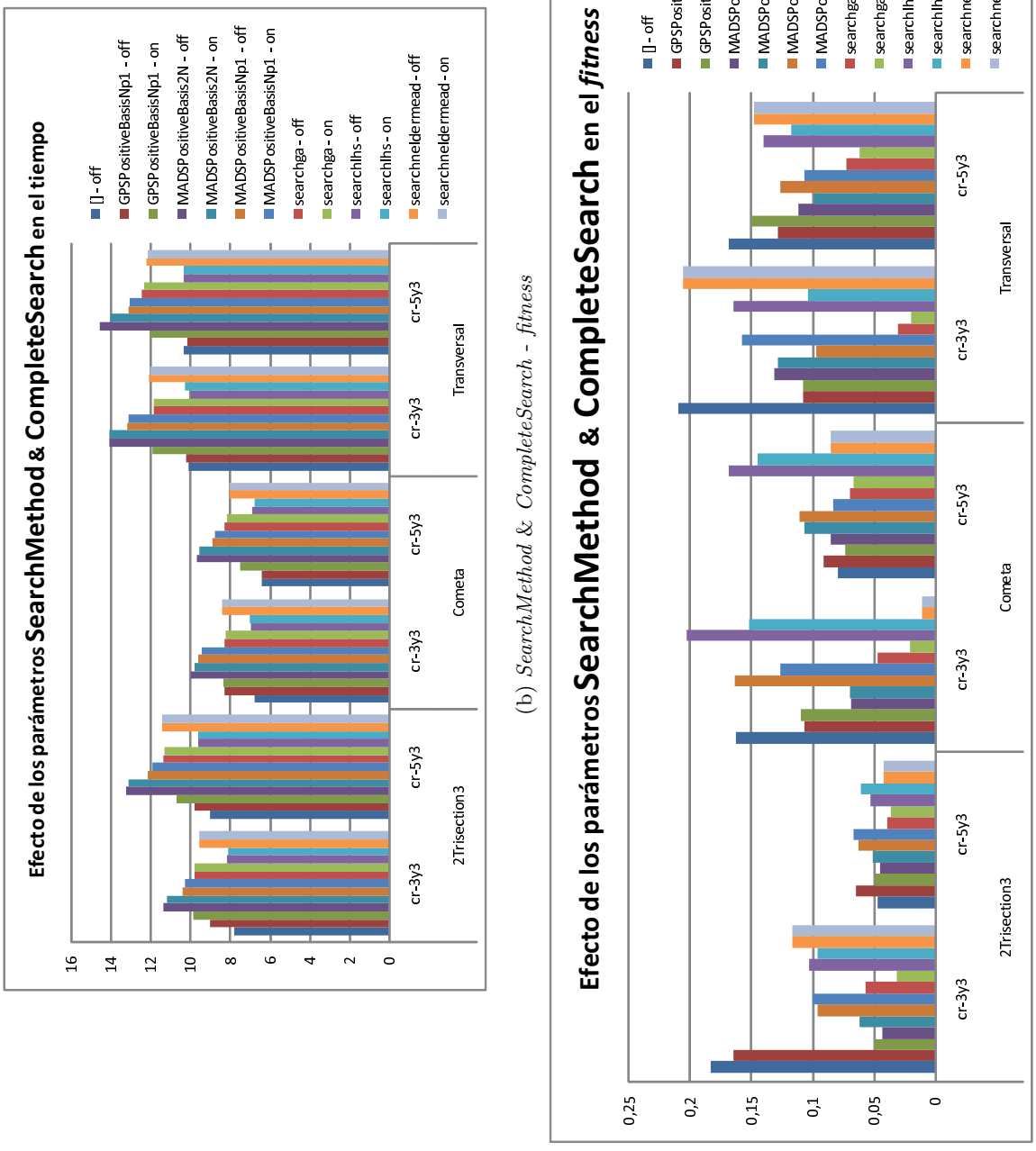
Figura 3.4.: Resultados de la herramienta `patternsearch` (I)

Figura 3.5.: Resultados de la herramienta `patternsearch` (II)



(a) *PollMethod* (b) *CompletePoll* (c) *CompletePoll* (d) *PollingOrder*

Figura 3.6.: Resultados de la herramienta `patternsearch` (III)



(a) *SearchMethod & CompleteSearch* - tiempo

(b) *SearchMethod & CompleteSearch* - *fitness*

- *MeshExpansion*: Factor por el que multiplicar la malla para expandirla cuando se está lejos de un mínimo.
- *ScaleMesh*: Activa (on) o desactiva (off) el escalado de la malla.
- *PollMethod*: Método empleado para realizar el sondeo. Existen las siguientes posibilidades: GPSPositiveBasis2N, GPSPositiveBasisNp1, MADSPositiveBasis2N, MADSPositiveBasisNp1.
- *CompletePoll*: Activa (on) o desactiva (off) el análisis completo de todos los puntos. Si está activo, se examinan todos los puntos de la malla y se selecciona como siguiente punto el mejor. Si no está activo, se elige como siguiente punto el primero que mejore al actual, deteniendo en ese momento el examen.
- *PollingOrder*: Orden para realizar el sondeo. Admite tres opciones: Consecutive, Random y Success.
- *SearchMethod*: Tipo de búsqueda usada en el patrón de búsqueda. Si se indica un método, en cada iteración, antes de realizar el sondeo, se usa el método para buscar un mejor punto, y, si se encuentra, se omite el sondeo. Admite las siguientes opciones: *ninguno*, GPSPositiveBasisNp1, GPSPositiveBasis2N, MADSPositiveBasisNp1, MADSPositiveBasis2N, searchga, searchlhs y searchneldermead.
- *CompleteSearch*: Sólo tiene sentido si se han indicado algunos tipos de búsqueda (*SearchMethod*). Activa (on) o desactiva (off) la búsqueda completa en cada iteración para esos métodos de búsqueda.

En las figuras 3.4, 3.5 y 3.6 se pueden observar los resultados más relevantes de las pruebas realizadas con esta herramienta. Cabe destacar que ciertos parámetros como *InitialMeshSize* (3.4a), *MeshContraction* (3.4b), *MeshExpansion* (3.4c), *ScaleMesh* (3.4d) y *PollingOrder* (3.5d) muestran comportamientos diferentes en cada uno de los problemas, no pudiéndose extraer ninguna conclusión acerca de ellos. En cambio parámetros como *PollMethod* (3.5a), *CompletePoll* (3.5c), *SearchMethod* y *CompleteSearch* (3.6b) sí muestran un comportamiento similar en prácticamente todos los casos, pudiéndose concluir que los parámetros *PollMethod* = MADSPositiveBasis2N, *CompletePoll* = on, *SearchMethod* = searchga y *CompleteSearch* = on obtienen los mejores resultados. A priori se puede presuponer que estos parámetros tendrán un alto coste en cuanto a tiempo se refiere, pero como se puede ver en las figuras 3.5b y 3.6a, aunque es evidente que empeoran algo el rendimiento, no es una penalización muy significativa teniendo en cuenta la mejora que se produce en los resultados.

3.2.1.4. ga (Algoritmo Genético)

La herramienta ga (Algoritmo Genético) pertenece al *Direct Search and Genetic Algorithm Toolbox* de MATLAB (véase el apartado 2.2.2.2 en la página 10) y está pensado especialmente para la optimización global de problemas no lineales con o sin restricciones. Además, aporta una gran flexibilidad, ya que aparte de los múltiples parámetros de los que dispone, es posible personalizar las funciones principales de la herramienta, tales como la función de selección, de cruce, de mutación y de creación de la población inicial, permitiendo proporcionar funciones hechas a medida para el problema de que se

trate. Asimismo permite aplicar una función de búsqueda local, tales como `fmincon` y `patternsearch`, al resultado final obtenido para mejorarlo.

Por este motivo, vamos a dividir el estudio de los resultados en dos partes, una primera examinando los parámetros estándar disponibles y una segunda introduciendo funciones personalizadas.

Los parámetros que vamos a estudiar de esta herramienta son (para un descripción detallada de los mismos se puede consultar [15]):

- *PopulationSize*: Tamaño de la población.
- *Generations*: Número máximo de generaciones.
- *EliteCount*: Número de individuos pertenecientes a la élite que sobreviven de una generación.
- *CrossoverFraction*: Fracción de nuevos individuos que se obtienen por cruce en contraposición a los que se obtienen por mutación.
- *FitnessScalingFcn*: Función de escalado del valor de *fitness* para obtener una valor de expectativa. Los posibles valores en estudio son fitscalingrank (asigna un valor de expectativa igual a su valor de orden de mejor a peor valor de *fitness*) y fitscalingprop (asigna un valor de expectativa proporcional al valor de *fitness*).
- *CreationFcn*: Función de creación de la población inicial. Debido a las características del problema, sólo hay una función de entre las que vienen de serie que se ajusta a nuestro problema: gacreationlinearfeasible (crea un población inicial al azar normalmente distribuida dentro de los límites establecidos). Se ha implementado una función personalizada para este parámetro llamada generapinicial que se basa en la anterior pero que además aplica una búsqueda local con `fmincon` a cada elemento generado y cuyo código fuente puede verse en el apéndice E.1 en la página 65.
- *SelectionFcn*: Función de selección de padres en base al valor de expectativa. Los valores posibles son: selectionremainder (selecciona a los padres de forma proporcional a la parte entera de su valor de expectación, y usa un método de ruleta basándose en la parte decimal), selectionstochunif (usando un muestreo universal estocástico), selectionroulette (usando una ruleta en la que cada padre tiene un área proporcional a su valor de expectación) y selectiontournament (selecciona varios padres al azar y se queda con el que tiene mejor valor de expectación).
- *CrossoverFcn*: Función de cruce de parejas de padres. Las funciones disponibles de serie son: crossoverscattered (cada uno de los genes se coge o de un padre o de otro de forma aleatoria), crossoverheuristic (se obtiene un nuevo elemento situado en la línea que une los cromosomas de ambos padres interpretados como puntos en el espacio, más cerca del padre con más expectativa), crossoverintermediate (se obtiene un nuevo elemento situado entre ambos padres considerando sus cromosomas como vectores), crossoversinglepoint (se cruzan los cromosomas de los padres por un punto), crossovertwopoint (se cruzan los cromosomas de los padres por dos puntos) y crossoverarithmetic (se obtiene la media aritmética de ambos padres considerando sus cromosomas como vectores). En este caso también se ha implementado una función personalizada basada en una combinación lineal como

la descrita en [10] con ciertas variaciones, llamada sscombineadapt, y que puede verse en el apéndice E.4 en la página 67.

- *MutationFcn*: Función de mutación. De entre las funciones disponibles de serie, solamente mutationadaptfeasible (mutación al azar teniendo en cuenta los límites) es adecuada para este problema. Además se ha implementado una versión personalizada basada en la anterior pero que además aplica una búsqueda local con `fmincon` a determinados elementos en determinadas iteraciones del algoritmo. Se llama mimutacion y puede verse su código fuente en el apéndice E.5 en la página 67.
- *HybridFcn*: Función de hibridación. Es posible aplicar una función de búsqueda local al mejor elemento obtenido por el algoritmo a su termino, con el objetivo de buscar entre sus vecinos una solución de más calidad más eficientemente. Las opciones son: ninguna, fmincon y patternsearch.

Funciones Estándar En las figuras 3.7, 3.8 y 3.9 pueden verse los resultados más interesantes obtenidos en este experimento, limitándonos siempre a las funciones estándar que vienen en MATLAB y sin emplear la función de hibridación.

El comportamiento de los parámetros de tamaño de población (*PopulationSize*) y número máximo de generaciones (*Generations*) tienen el efecto previsible tanto en mejora del resultado (3.7a, 3.7c) como en empeoramiento del tiempo de ejecución (3.7b, 3.7d), siendo ambos efectos inversamente proporcionales entre sí.

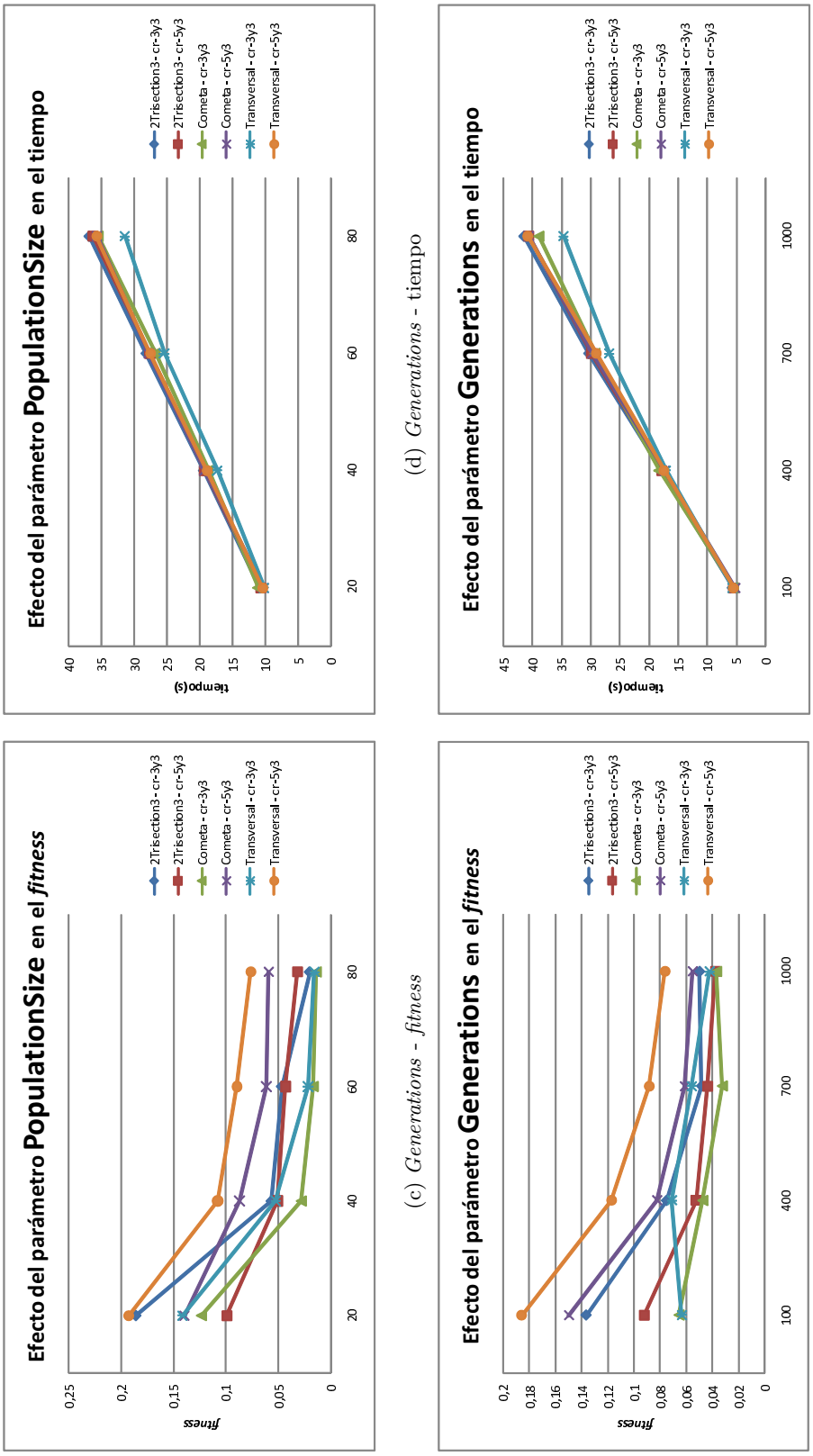
En cuanto a los parámetros *EliteCount* (3.8a) y *CrossoverFraction* (3.8b) destacar que no tienen el mismo efecto en todas los casos, aunque en el caso de *EliteCount*, parece que el valor 2 suele ser el mejor en la mayoría de los casos, mientras que para el otro parámetro, situarlo alrededor de los valores 0.4 ó 0.6 (dependiendo del caso) parece ser la mejor decisión, lo que implica que la mutación aparenta hacer un buen trabajo en este problema.

Observando la figura 3.8c se puede ver que la mejor función de selección es, en promedio, selectionstochunif. En cuanto a los tiempos, vemos en la figura 3.8d que no hay diferencias significativas, a excepción de selectiontournament, que siempre es un poco peor que las demás en este aspecto.

En cuanto al parámetro *FitnessScalingFcn*, podemos ver en la figura 3.9a que la función fitscalingrank es mejor en casi todos los casos, pero teniendo en cuenta que tiene un coste en tiempo de ejecución tal y como podemos ver en la figura 3.9b.

Por último, el efecto del parámetro *CrossoverFcn* podemos verlo en las figuras 3.9c y 3.9d. Se observa que con la función crossoverheuristic se obtienen, en promedio, los resultados más satisfactorios, sin que se vean diferencias significativas en el tiempo de ejecución entre las distintas posibilidades.

Funciones Personalizadas y Uso de la Hibridación En la figura 3.10 pueden verse los resultados obtenidos en este experimento usando las versiones personalizadas de las funciones comentadas anteriormente y activando la opción de hibridación (aplicación de una función de búsqueda local al finalizar la ejecución normal del algoritmo).



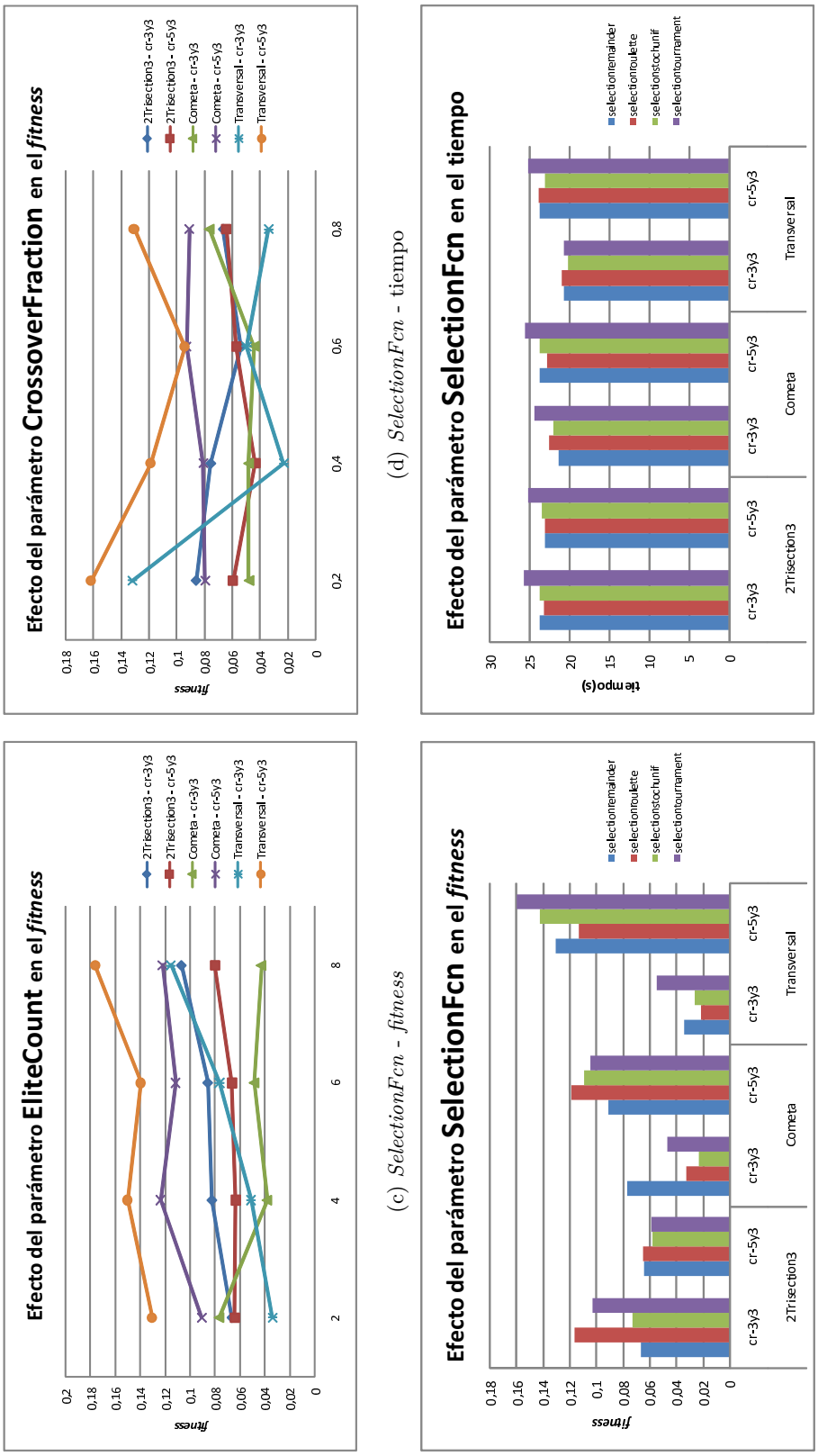
(a) *PopulationSize* - *fitness*

(b) *PopulationSize* - tiempo

(c) *Generations* - *fitness*

(d) *Generations* - tiempo

Figura 3.7.: Resultados de la herramienta ga usando parámetros estándar (I)



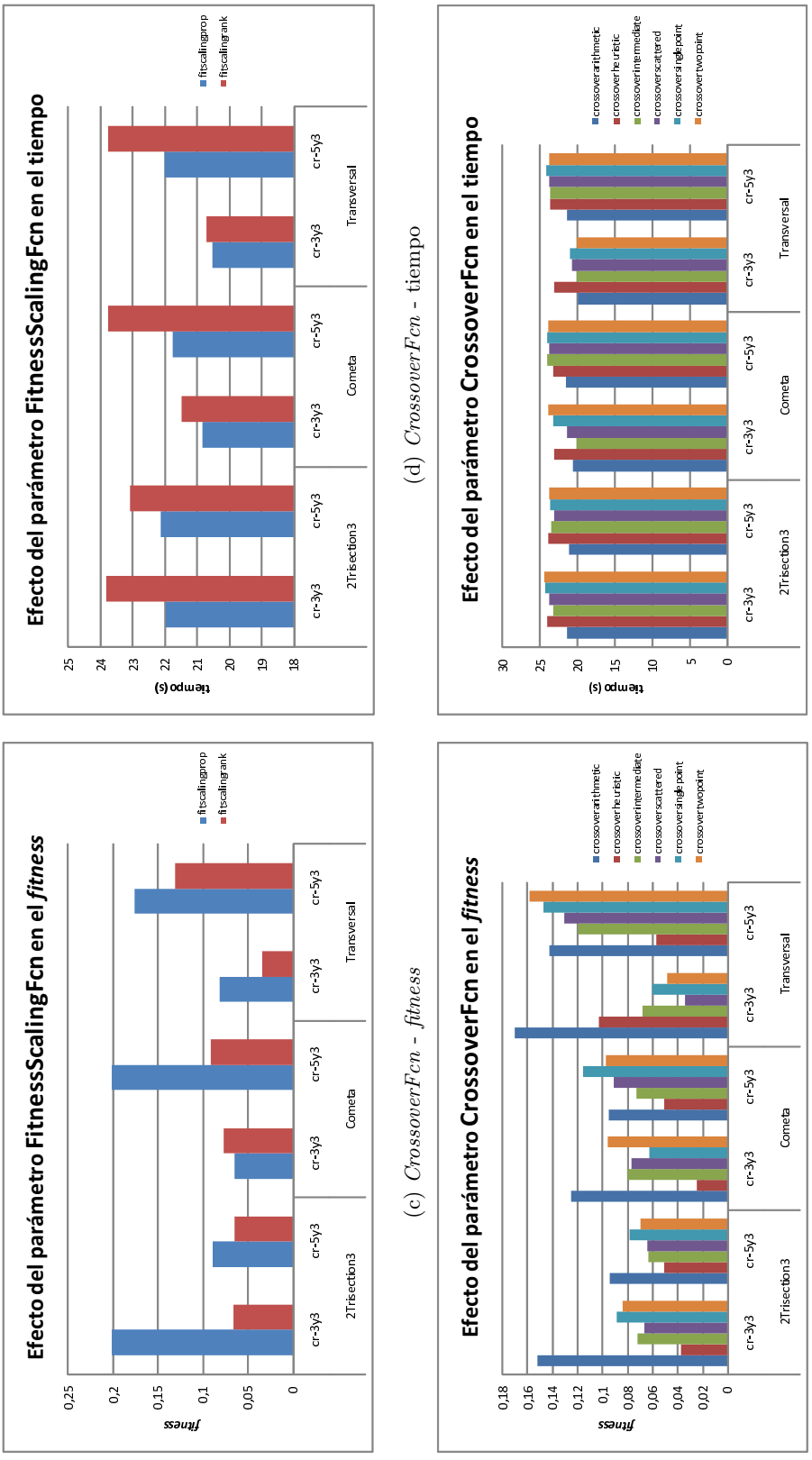
(a) *EliteCount* - *fitness*

(b) *CrossoverFraction* - *fitness*

(c) *SelectionFcn* - *fitness*

(d) *SelectionFcn* - tiempo

Figura 3.8.: Resultados de la herramienta ga usando parámetros estándar (II)



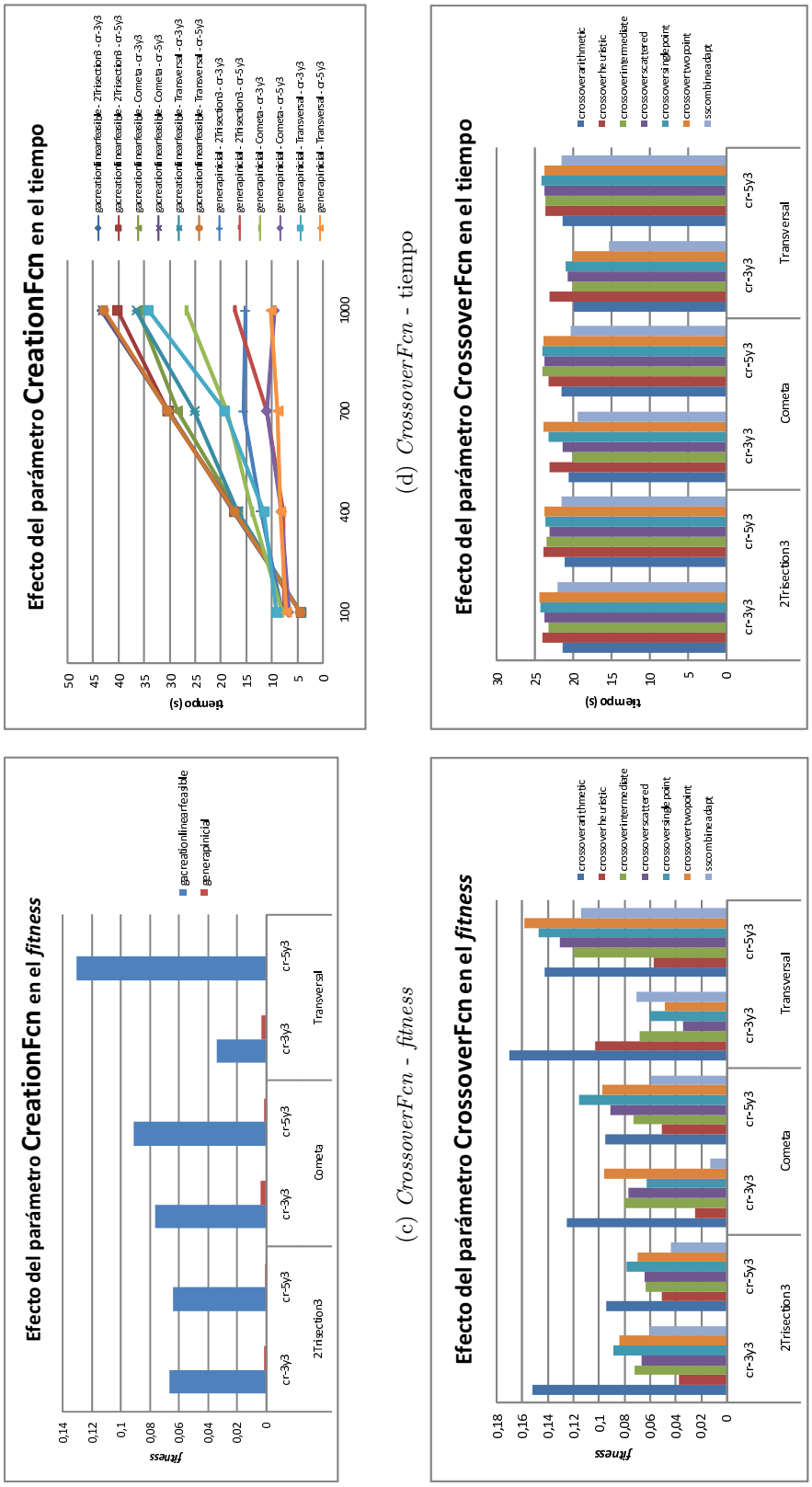
(a) *FitnessScaling* - *fitness*

(b) *FitnessScaling* - tiempo

(c) *CrossoverFcn* - *fitness*

(d) *CrossoverFcn* - tiempo

Figura 3.9.: Resultados de la herramienta `ga` usando parámetros estándar (III)



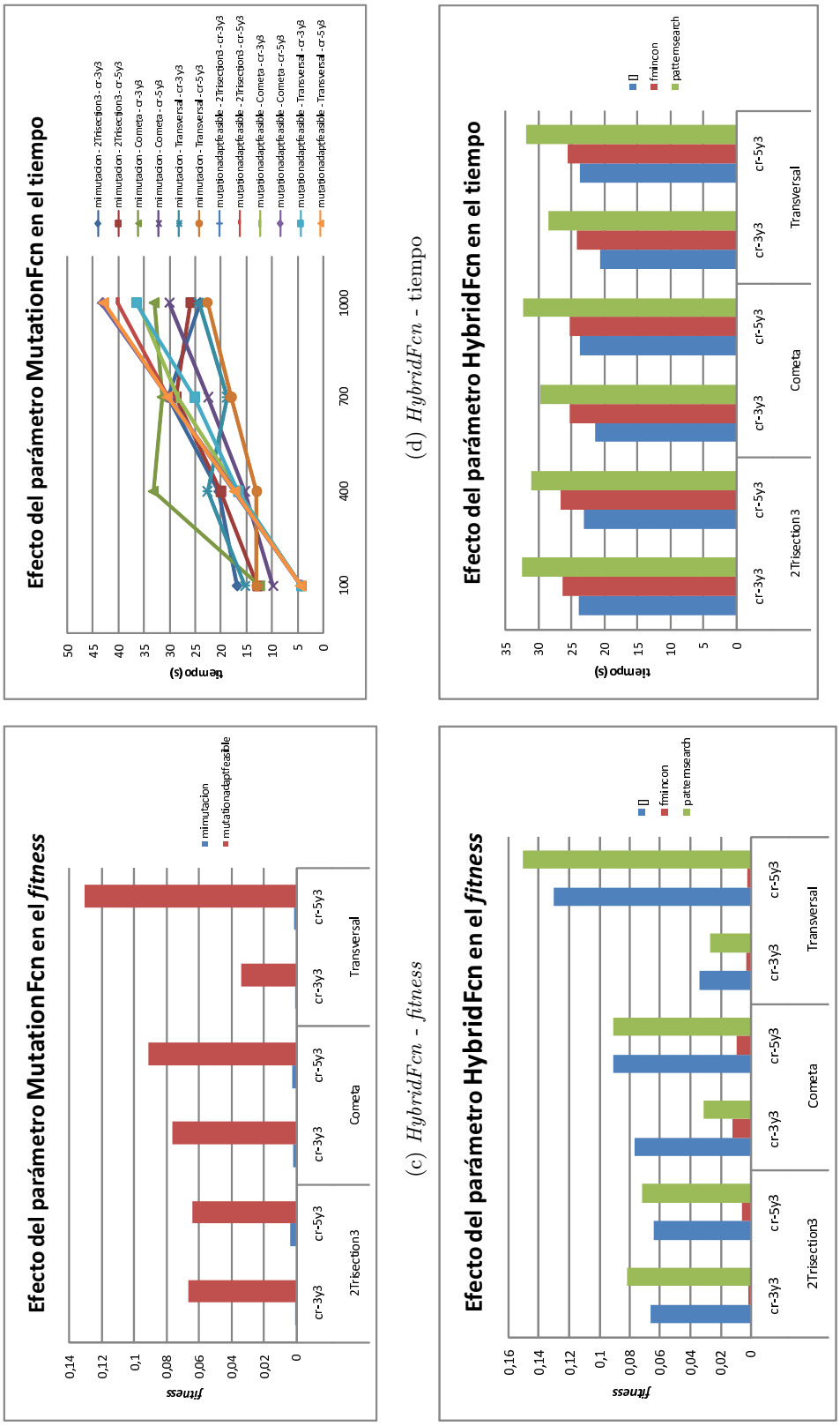
(a) *CreationFcn* - *fitness*

(b) *CreationFcn* - tiempo

(c) *CrossoverFcn* - *fitness*

(d) *CrossoverFcn* - tiempo

Figura 3.10.: Resultados de la herramienta ga usando funciones personalizadas



(a) *MutationFcn* - *fitness*

(b) *MutationFcn* - tiempo

(c) *HybridFcn* - *fitness*

(d) *HybridFcn* - tiempo

Figura 3.11.: Resultados de la herramienta `ga` usando funciones personalizadas (II)

El usar una creación de población inicial (*CreationFcn*) personalizada (generapinicial) tiene un efecto muy importante en el resultado como puede verse en la figura 3.10a, siendo sólo penalizada en cuanto a tiempo (figura 3.10b) en el caso de limitarse la ejecución a pocas generaciones, pero mejorando también en este aspecto cuando se aumentan.

En cuanto al efecto sobre la función de cruce (*CrossoverFcn*) de la personalización (sscombine), podemos ver que si bien su resultado en cuanto al valor de la función de *fitness* (figura 3.10c) no siempre es el mejor (aunque suele ser de los mejores), en tiempo de ejecución (figura 3.10d) siempre es el mejor o empata con el mejor.

En el caso de la mutación (*MutationFcn*), el resultado de la personalización (mimutacion) es igual de espectacular que en el caso de la creación (figura 3.11a), ocurriendo también lo mismo en cuanto al tiempo (figura 3.11b), es decir, a mayor número de generaciones, mejor se comporta.

Por último, y no por ello menos importante, tenemos la función de hibridación (*HybridFcn*), que se aplica al termino del algoritmo genético al mejor individuo obtenido por este. Claramente, la función fmincon da los mejores resultados (figura 3.11c) en todos los aspectos con sólo una ligera penalización en cuanto al tiempo de ejecución (figura 3.11d), siendo, por tanto, muy interesante su uso.

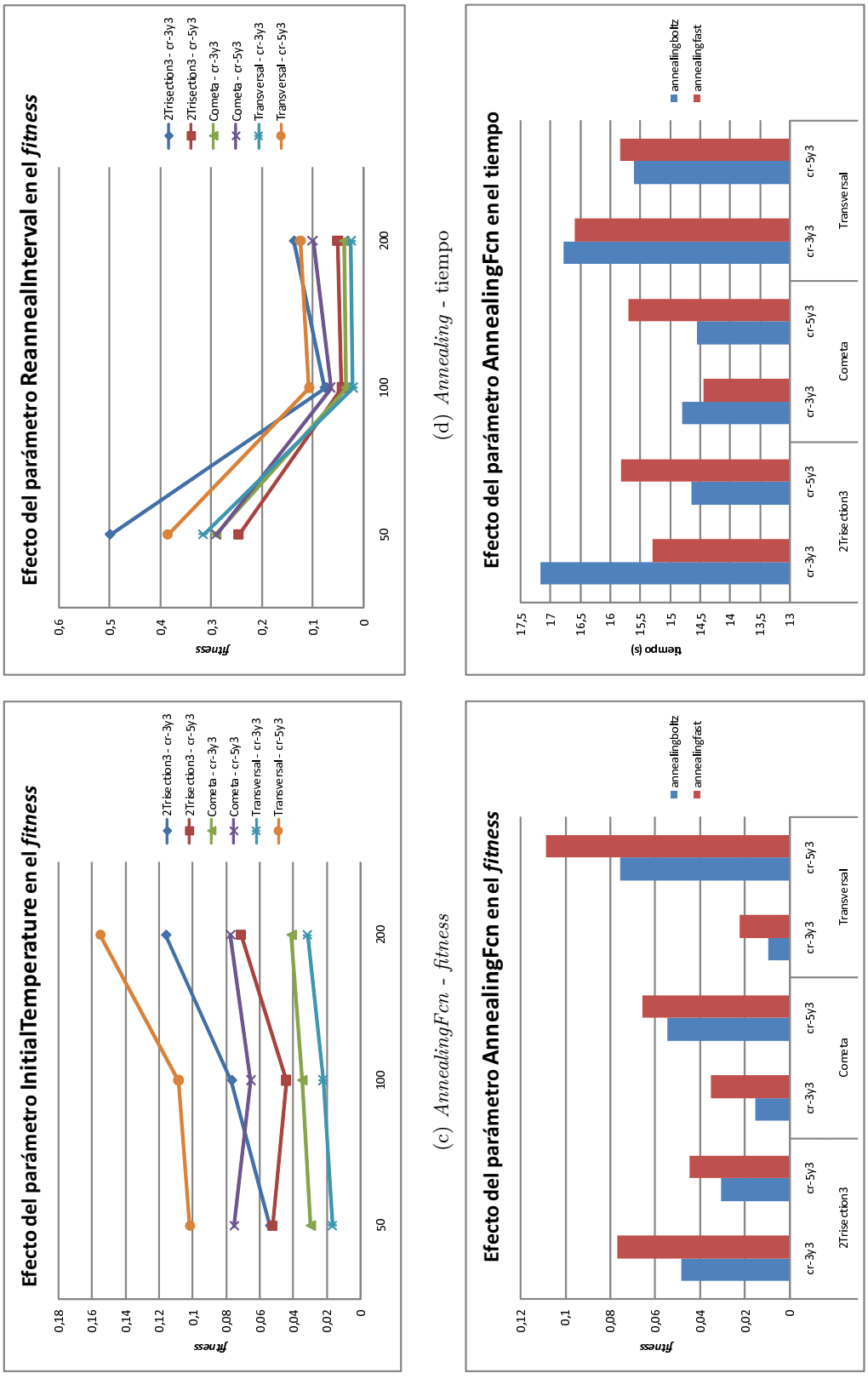
3.2.1.5. simulannealbnd (Temple Simulado)

La herramienta `simulannealbnd` (Temple Simulado) pertenece al *Direct Search and Genetic Algorithm Toolbox* de MATLAB (véase el apartado 2.2.2.3 en la página 11), y al igual que `fmincon` y `patternsearch`, puede catalogarse como herramienta de búsqueda local con restricciones puesto que necesita un punto inicial en el que comenzar la búsqueda.

En este caso, los parámetros que se han evaluado son los siguientes (para una explicación más detallada puede consultarse [15]):

- *AnnealingFcn*: Función de templanza. Admite dos posibles funciones de serie: annealingfast y annealingboltz.
- *InitialTemperature*: Temperatura inicial (un valor positivo).
- *ReannealInterval*: Intervalo de "retemplanza".
- *TemperatureFcn*: Función de cambio de la temperatura. Las opciones son: temperatureexp, temperatureboltz y temperaturefast.
- *HybridFcn*: Función híbrida. Las opciones que vamos a evaluar son: ninguna, fmincon y patternsearch.
- *HybridInterval*: Indica cuando hay que aplicar la función híbrida. Las opciones son: never, end o un valor positivo.

En las figuras 3.12 y 3.13 se pueden observar los resultados más significativos del experimento. En cuanto al parámetro *InitialTemperature*, podemos ver que el valor 50 ó 100 (dependiendo del caso) da los mejores resultados (figura 3.12a). Con el parámetro *ReannealInterval* tenemos un caso parecido, aunque en esta ocasión sí queda claro que el valor 100 es el mejor (figura 3.12b). Por otro lado, el parámetro *AnnealingFcn* funciona mejor con el valor annealingboltz (figuras 3.12c y 3.12d) al igual que sucede con



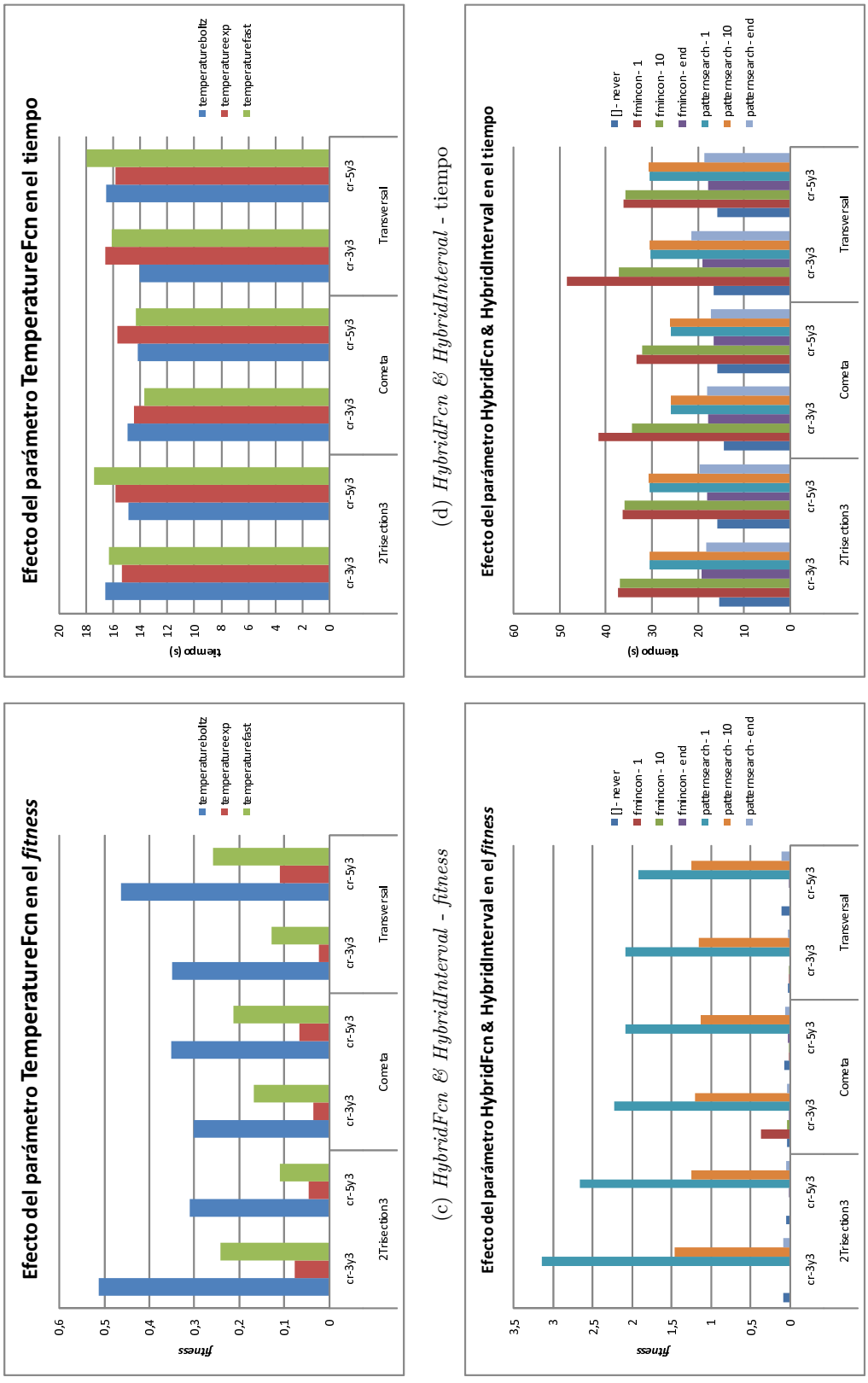
(a) *InitialTemperature* - *fitness*

(b) *ReannealInterval* - *fitness*

(c) *AnnealingFcn* - *fitness*

(d) *Annealing* - tiempo

Figura 3.12.: Resultados de la herramienta `simulannealbnd` (I)



(a) *TemperatureFcn* - *fitness*

(b) *TemperatureFcn* - tiempo

(c) *HybridFcn & HybridInterval* - *fitness*

(d) *HybridFcn & HybridInterval* - tiempo

Figura 3.13.: Resultados de la herramienta `simulannealbnd` (II)

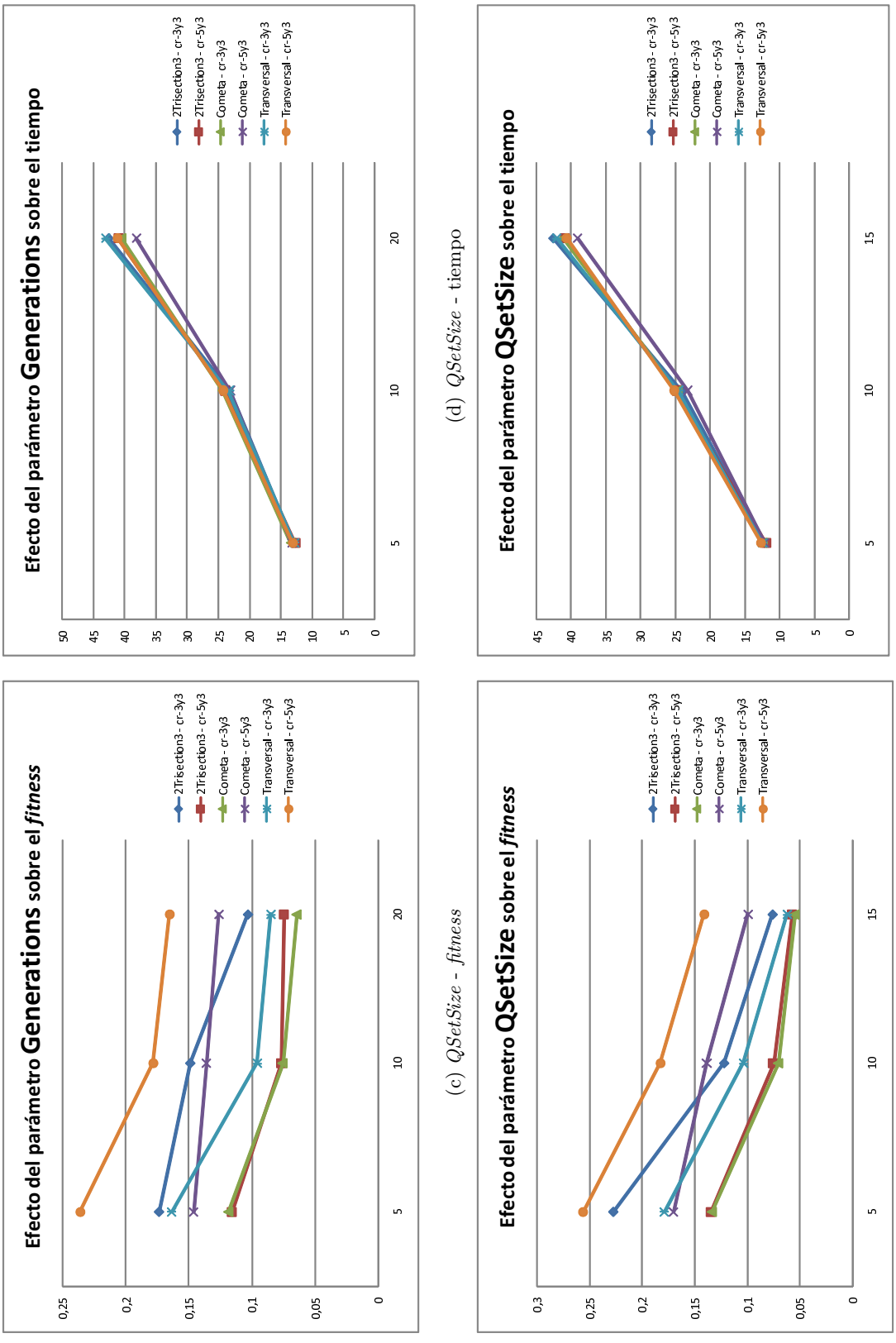
el valor temperatureexp para el parámetro *TemperatureFcn* (figuras 3.13a y 3.13b). En cambio, vemos un efecto sorprendente en el parámetro *HybridFcn* y su parámetro dependiente *HybridInterval* en la figura 3.13c, ya que al usar patternsearch se produce un empeoramiento muy significativo, obteniéndose los mejores resultados (varios ordenes de magnitud mejores) con la función fmincon. En cuanto al tiempo, este depende directamente del intervalo de aplicación como puede verse en la figura 3.13d, donde vemos que el aplicarlo con intervalo 1 tiene un coste en tiempo muy importante no siendo los resultados significativamente mejores que en el caso 10 o end, siendo el coste en tiempo de este último caso muy pequeño, y por tanto, con mejor relación coste/resultado.

3.2.1.6. ScatterSearch (Búsqueda Dispersa)

La herramienta `ScatterSearch` (Búsqueda Dispersa) es una técnica metaheurística descrita en el apartado 2.3.1 en la página 12, que al igual que la herramienta `ga` (Algoritmo Genético) realiza optimización global con o sin restricciones. Al igual que esta, admite personalización de las funciones principales, como son la creación de una población inicial, la de combinación de soluciones, la de mejora y la de hibridación final. Como se ha mencionado anteriormente, hemos realizado la implementación de esta herramienta para este trabajo y se pueden consultar los detalles de la implementación en el apéndice B en la página 50.

Los parámetros que vamos a evaluar son:

- *QSetSize*: Tamaño del subconjunto de referencia de calidad.
- *DSetSize*: Tamaño del subconjunto de referencia de diversidad.
- *Generations*: Número de generaciones o iteraciones.
- *CreationFnc*: Función de creación de la población inicial o conjunto P. Tenemos las mismas posiblidades que en ga: gacreationlinearfeasible y generapinicial.
- *OptimFnc*: Función de mejora. Se aplica a algunos elementos antes de intentar añadirlos al conjunto de referencia, y suele ser una función de búsqueda local. Tenemos como opciones: ninguna, fmincon y patternsearch.
- *OptimFactor*: Factor de aplicación de la función de mejora. Admite valores reales entre 0, que significa no aplicar, y 1, que significa aplicar siempre. Este parámetro, de alguna manera, permite ajustar el nivel de esfuerzo a aplicar en la mejora con respecto a la combinación, de tal modo que a menor factor de mejora, menor esfuerzo se dedica a la misma y viceversa.
- *CombineFnc*: Función de combinación. Como posibilidades tenemos la mismas que en ga: sscombine, crossoverscattered adaptado, crossoverheuristic adaptado, crossoverintermediate adaptado, crossoversinglepoint adaptado, crossovertwopoint adaptado y crossoverarithmetic adaptado. La adaptación se realiza mediante la función GACROSSOVERADAPT, cuyo código fuente se puede ver en el apéndice E.3 en la página 66, y que realiza la conversión de parámetros y valores de retorno para poder usar las funciones de cruce disponibles para `ga` en nuestro `ScatterSearch`.



(a) *Generations* - *fitness*

(b) *Generations* - tiempo

(c) *QSetSize* - *fitness*

(d) *QSetSize* - tiempo

Figura 3.14.: Resultados de la herramienta `ScatterSearch` relativos a tamaños (I)

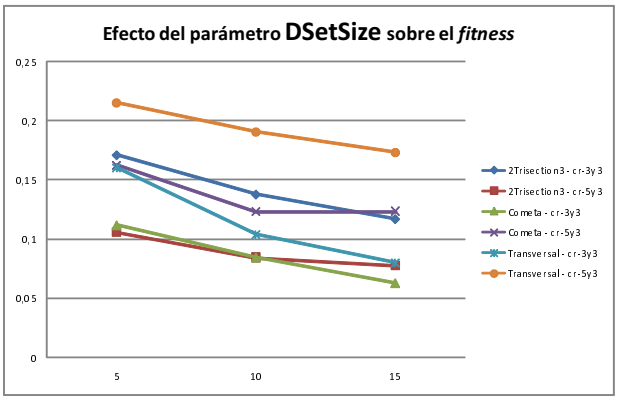


Figura 3.15.: Resultados de la herramienta `ScatterSearch` para *DSetSize* en el *fitness*

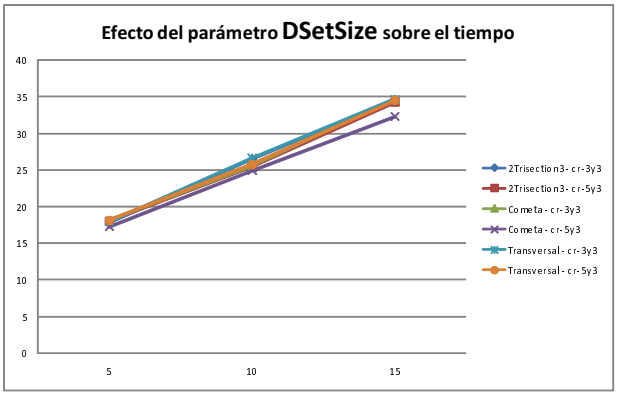
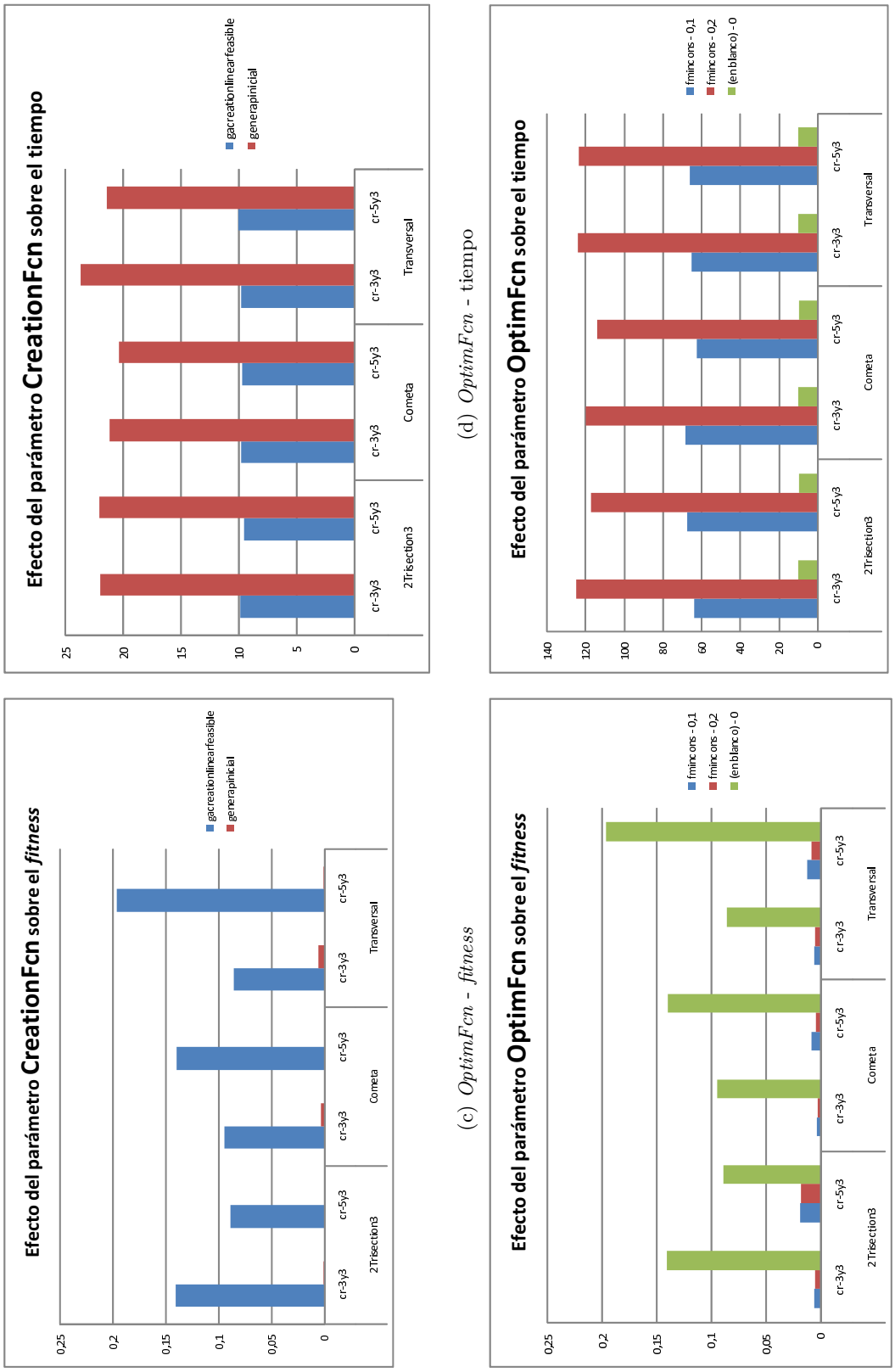


Figura 3.16.: Resultados de la herramienta `ScatterSearch` para *DSetSize* en el tiempo

- *HybridFnc*: Función de hibridación. Se aplica al finalizar el algoritmo al mejor individuo obtenido. Al igual que ocurre en el caso de la combinación, tenemos las mismas posibilidades que en ga, a saber: ninguna, fmincon y patternsearch.

En la figuras 3.14, 3.15 y 3.16 se muestran los resultados más relevantes de los experimentos realizados con esta herramienta con respecto a los parámetros *Generations*, *QSetSize* y *DSetSize*. El parámetro *Generations* tiene el efecto esperado, cuantas más generaciones, mejor calidad tiene la solución (figura 3.14a), pero más tiempo se necesita (figura 3.14b), aunque cabe destacar que el empeoramiento del rendimiento es mayor que la mejora en el *fitness*. De igual modo se comportan tanto el parámetro *QSetSize* (figuras 3.14c y 3.14d) como *DSetSize* (figuras 3.15 y 3.16) relativos al tamaño del conjunto de referencia, aunque la mejora asociada al aumento de *QSetSize* es mayor y también su coste en cuanto a tiempo.

En las figuras 3.17 y 3.18 se muestran los resultados de los experimentos relativos a los parámetros que admiten funciones. Viendo las figuras 3.17a y 3.17b de la función de creación (*CreationFcn*), podemos ver que el efecto de la función generapinicial es ciertamente impresionante, con varios ordenes de magnitud de mejora respecto a la otra función en cuanto a calidad de la solución, con un coste relativamente bajo en cuanto al aumento del tiempo (sólo el doble).



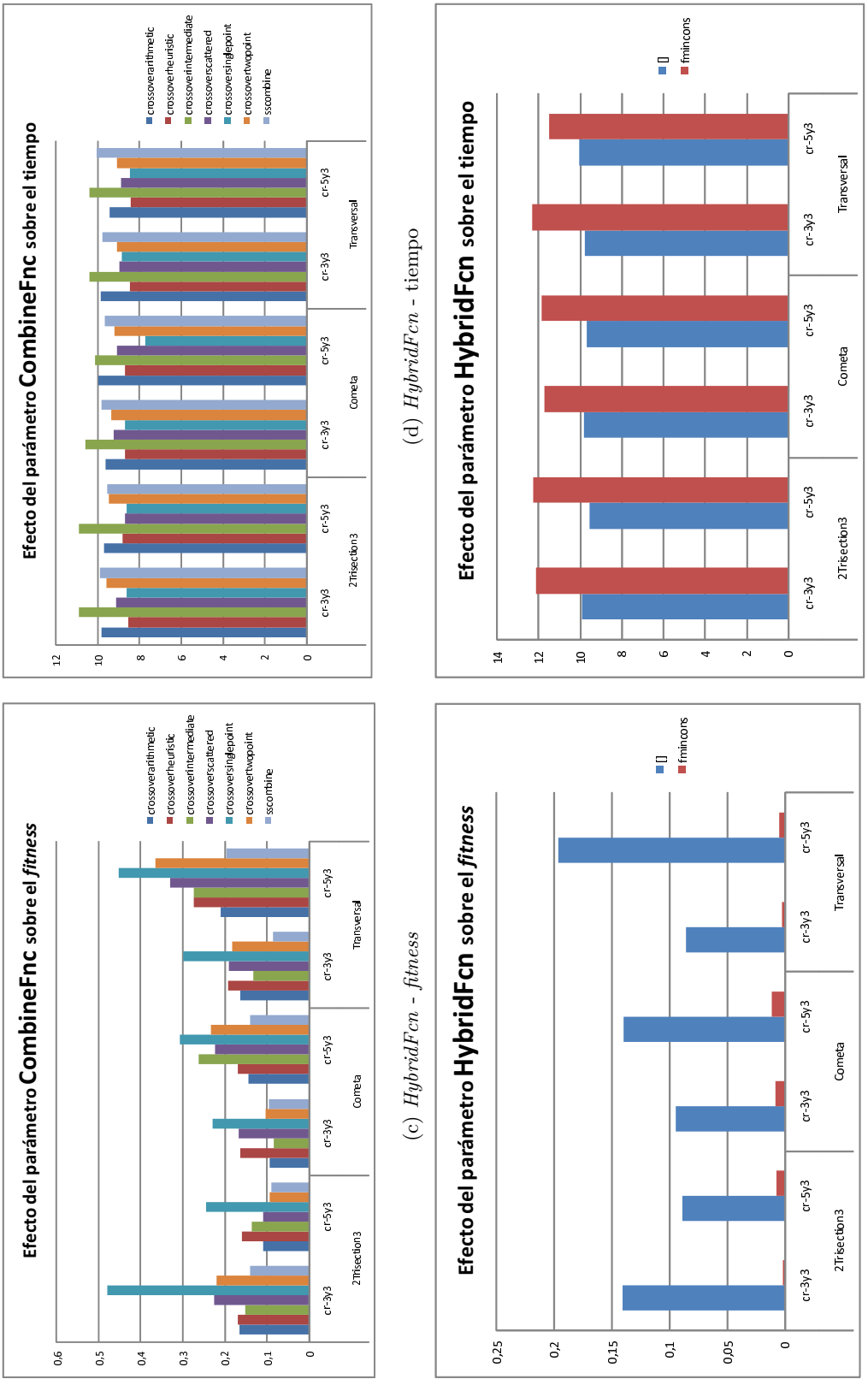
(a) *CreationFcn* - *fitness*

(b) *CreationFcn* - tiempo

(c) *OptimFcn* - *fitness*

(d) *OptimFcn* - tiempo

Figura 3.17.: Resultados de la herramienta `ScatterSearch` relativos a funciones (I)



(a) *CombineFnc - fitness*

(b) *CombineFnc* - tiempo

(c) *HybridFcn - fitness*

(d) *HybridFcn* - tiempo

Figura 3.18.: Resultados de la herramienta `ScatterSearch` relativos a funciones (II)

En cuanto a la función de mejora (*OptimFcn*), en las figuras 3.17c y 3.17d, también comprobamos que la aplicación de fmincon produce una mejora espectacular con simplemente aplicarlo en una fracción pequeña (0.1 = 10 %) y con un número muy reducido de iteraciones máximas en la búsqueda local para no ralentizar en exceso el proceso, aunque el coste en tiempo también es importante pero proporcionalmente menor que la mejora obtenida, no resultando tan significativo el aumento de la fracción de aplicación a 0.2 (20 %) en cuanto a mejora pero sí en cuanto a coste en tiempo (el doble).

Los resultados relativos a la función de combinación (*CombineFnc*) muestran, en las figuras 3.18a y 3.18b, que en cuanto al coste en tiempo son todas muy parecidas, y aunque sscombine es la mejor en cuanto al valor de la función de *fitness*, las diferencias no son tan significativas como en el caso de la creación y la mejora.

Por último, en cuanto a la función de hibridación (*HybridFcn*), en las figuras 3.18c y 3.18d, también podemos ver que el efecto de aplicar fmincon es muy significativo, siendo su coste en tiempo muy poco importante, siendo por tanto un parámetro muy a tener en cuenta.

3.2.1.7. backtracking (Vuelta Atrás)

La herramienta `backtracking` (Vuelta Atrás) es una técnica basada en la exploración sistemática tal y como se describe en el apartado 2.3.2 en la página 12 y cuyos detalles de implementación pueden consultarse en el apéndice C en la página 58. Dada su naturaleza sistemática y la amplitud del espacio de soluciones (8 ó 9 dimensiones, valores reales), no parece la técnica más apropiada. No obstante, hemos aplicado una versión reducida de la misma para realizar una exploración de grano grueso del espacio de soluciones e intentar detectar en que zonas puede encontrarse el óptimo. Los resultados obtenidos nos sirven para poder comparar esta técnica exhaustiva con las técnicas metaheurísticas anteriormente estudiadas aplicadas a este tipo de problemas.

El único parámetro en este caso es el número de puntos por dimensión que se van a evaluar (es decir, el número de valores que tomará cada uno de los parámetros de diseño dentro de los limites establecidos). Estos puntos se distribuirán a igual distancia uno de otro, incluyendo los limites superior e inferior. Dada la complejidad del problema, en la tabla 3.1 podemos ver como crece de forma desorbitada el número de puntos a evaluar conforme se aumenta este parámetro, denominado *amplitud*, teniendo en cuenta el número de variables o dimensión del problema.

Tabla 3.1.: Crecimiento del número de puntos a evaluar

amplitud	*8-D*	*9-D*
3	512	729
5	32768	59049
7	2097152	4782969
9	134217728	387420489

En la figura 3.19 podemos ver los resultados obtenidos variando el parámetro *amplitud*. Como era de esperar, al aumentar la amplitud el resultado mejora más o menos linealmente, pero, en la figura 3.20, vemos como el tiempo de ejecución empeora de forma exponencial, especialmente en el caso de topologías con 9 variables de diseño como es

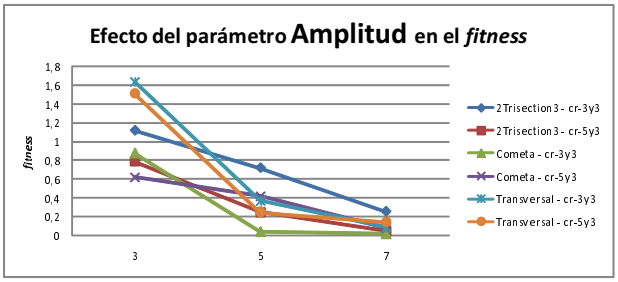


Figura 3.19.: Efecto del parámetro *Amplitud* de `backtracking` en el *fitness*

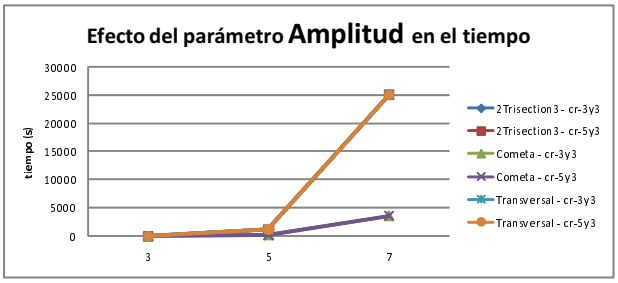


Figura 3.20.: Efecto del parámetro *Amplitud* de `backtracking` en el tiempo

el caso de la *Transversal*. Esto nos hace intuir que la principal utilidad que debemos darle a esta herramienta es la de generar una población inicial formada por los mejores resultados de la exploración realizada y que sirva como entrada a otras herramientas. Hemos llevado a cabo unos experimentos preliminares en relación con esto, pero no hemos incluido los resultados en este trabajo por no ser suficientemente satisfactorios todavía.

3.2.2. Estudio Comparativo

Después de haber estudiado de forma individual cada una de las herramientas que tenemos disponibles, vamos ahora a comparar los resultados de todas ellas con sus mejores opciones.

3.2.2.1. Diseño del experimento

Para que la prueba sea justa, vamos a limitar cada una de las pruebas con un límite de tiempo, pudiendo así obtener resultados comparables a igualdad de coste máximo en tiempo. Para que los resultados sean más fiables, repetiremos cada prueba varias veces obteniendo el promedio.

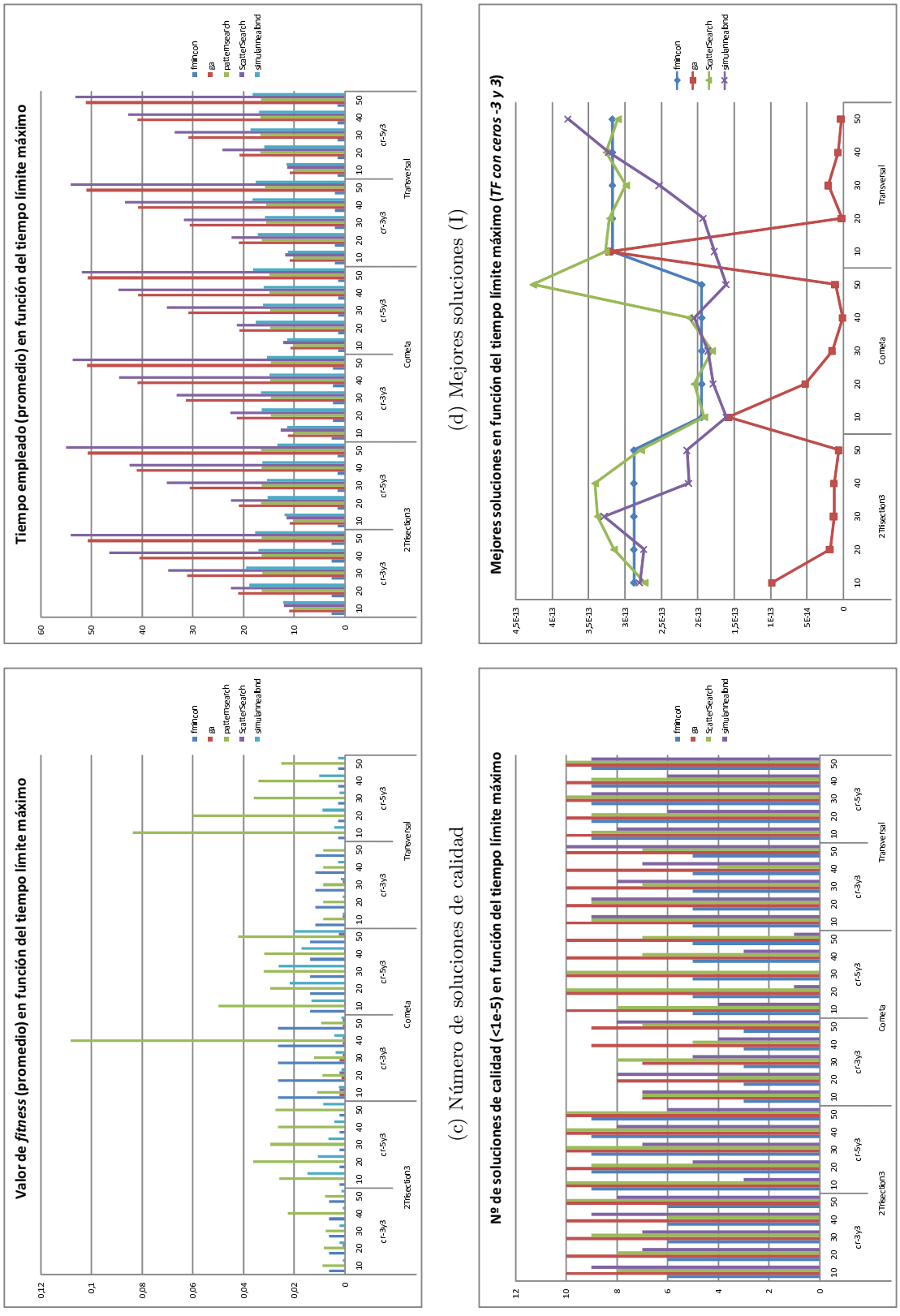
Para cada una de las herramientas de este experimento hemos usado la siguiente configuración obtenida seleccionando los valores de los parámetros que han obtenido resultados satisfactorios en las pruebas individuales:

- `fmincon`: *Algorithm* = interior-point, *LargeScale* = off

- `patternsearch`: *InitialMeshSize* = 0.5, *MeshContraction* = 0.25, *MeshExpansion* = 1.5, *SearchMethod* = searchga, *CompleteSearch* = on, *ScaleMesh* = on, *PollMethod* = MADSPositiveBasis2N, *CompletePoll* = on, *PollingOrder* = Random.
- ga: *PopulationSize* = 40, *EliteCount* = 2, *CrossoverFraction* = 0.6, *FitnessScalingFcn* = fitscalingrank, *SelectionFcn* = selectionstochunif, *CreationFcn* = generapinicial, *CrossoverFcn* = crossoverheuristic, *MutationFcn* = mimutacion, *HybridFcn* = fmincon.
- `simulannealbnd`: *AnnealingFcn* = annealingboltz, *InitialTemperature* = 100, *ReannealInterval* = 100, *TemperatureFcn* = temperatureexp, *HybridFcn* = fmincon, *HybridInterval* = end.
- `ScatterSearch`: *QSetSize* = 10, *DSetSize* = 5, *CreationFnc* = generapinicial, *OptimFnc* = fmincon, *OptimFactor* = 0.1, *CombineFnc* = sscombine, *HybridFnc* = fmincon.

3.2.2.2. Resultados

En las figuras 3.21 y 3.22 pueden observarse los resultados más interesantes de la prueba. Debe hacerse notar que la función `fmincon` no cuenta con ningún parámetro para limitar el tiempo de ejecución, aunque este no es en ningún caso superior a ninguno de los limites impuestos, sino más bien al contrario, ya que en general consume sólo unos pocos segundos (figura 3.21b). Por otro lado, tanto `fmincon`, como `patternsearch` y `simulannealdbnd` son herramientas de optimización local (necesitan un punto a partir del cual iniciar la búsquda), por lo que se han generado puntos al azar dentro del espacio de soluciones y se han usado los mismos para cada una de ellas, permitiendo que el resultado entre ellas sea comparable. Evidentemente, tanto `ga` como `ScatterSearch` (usando `fmincon` como función auxiliar) obtienen los mejores resultados en promedio en cuanto a calidad de la solución obtenida (figura 3.22d), como era previsible, debido sobre todo a que son optimizadores globales y exploran más regiones del espacio de soluciones. De ellos dos, cabe destacar la regularidad de `ga`, que obtiene siempre las mejores soluciones (figuras 3.21a, 3.21d, 3.22a) y las menos malas (figuras 3.22b y 3.22c). También es necesario comentar los resultados de `patternsearch`, que en general son mucho peores que los de los demás, especialmente cuando comparamos las mejores soluciones obtenidas por cada método (figuras 3.21d y 3.22a), de donde han tenido que suprimirse por estar fuera de la escala. En la figura 3.21c podemos ver el número de soluciones aceptables (por debajo de un umbral mínimo) que nos da cada método según el límite de tiempo impuesto en 10 ejecuciones, pudiendo destacar que en ella se resume de forma bastante clara lo comentado anteriormente sobre la calidad de las soluciones ofrecidas por cada método.



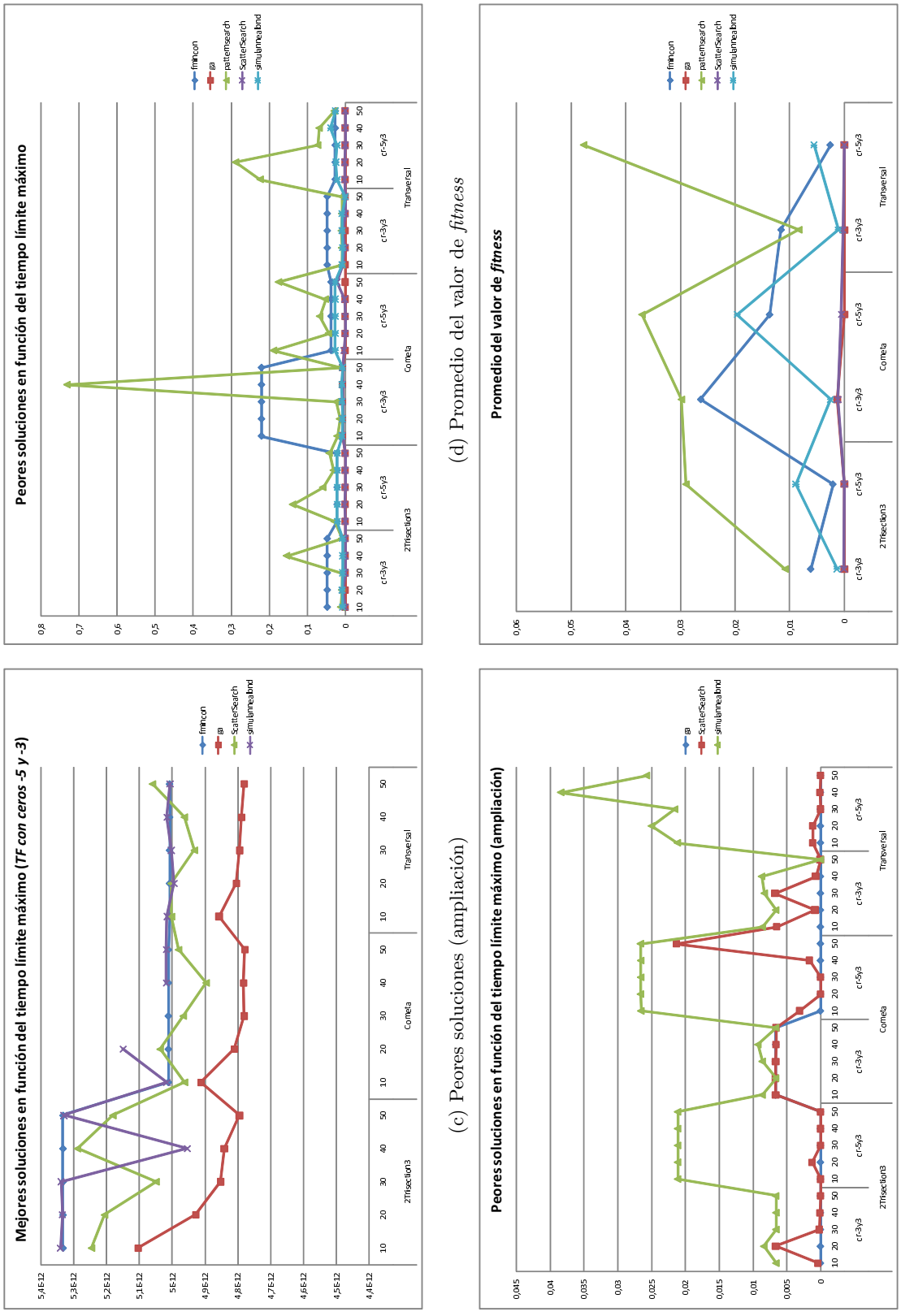
(a) *Fitness* (promedio)

(b) Tiempo (promedio)

(c) Número de soluciones de calidad

(d) Mejores soluciones (I)

Figura 3.21.: Resultados del estudio comparativo



(a) Mejores soluciones (II)

(b) Peores soluciones

(c) Peores soluciones (ampliación)

(d) Promedio del valor de *fitness*

Figura 3.22.: Resultados del estudio comparativo (II)

4. Conclusiones y Trabajos Futuros

Tras todo los visto en el capítulo 3 llega el momento de extraer conclusiones. Hemos visto como para el diseño de filtros con resonadores acoplados debemos usar una herramienta de optimización no analítica ya que nos enfrentamos a un problema no lineal con restricciones y múltiples variables. También hemos visto que es inviable afrontar el problema mediante el empleo de una técnica exhaustiva debido a que el espacio de soluciones es enorme (8 ó 9 dimensiones y números reales entre -5 y 5 en cada una de ellas). Por ello hemos optado por emplear diversas herramientas de optimización. De entre ellas, las que mejores resultados obtienen son `ga` (*Algoritmo Genético*), del *toolbox* de MATLAB pero personalizado con varias funciones a medida, y `ScatterSearch` (*Búsqueda Dispersa*), implementado para MATLAB por nosotros en este estudio, ambas de la familia de los algoritmos evolutivos. También hemos evaluado varias herramientas que podemos catalogar de búsqueda local en tanto en cuanto precisan de un punto inicial a partir del que iniciar la búsqueda. Hemos visto que de estas herramientas es especialmente interesante `fmincon`, debido a su rendimiento y efectividad, pudiéndose emplear como base para otros algoritmos (como el caso de los dos mencionados anteriormente).

Con todos estos resultados se está preparando un trabajo para el MAEB 2009 que se celebrará en febrero de 2009 (http://maeb09.lcc.uma.es).

En base a todo lo visto, se plantean los siguientes trabajos futuros:

- Aplicación de otras técnicas, como pueden ser GRASP, Colonias de Hormigas, Búsqueda Tabú, etc., a este problema y sus variantes en colaboración con el grupo de *Electromagnetismo aplicado a las Telecomunicaciones* de la Universidad Politécnica de Cartagena, planificada dentro de un proyecto Séneca.
- Aplicación de estas técnicas, incluyendo *Backtracking* y *Branch & Bound*, a otros problemas de telecomunicaciones, como al problema de la decodificación de la comunicación sobre un canal MIMO [19], en colaboración con el grupo de *Programación Paralela de la Universidad Politécnica* de Valencia, dentro de un proyecto nacional.
- Investigación de la aplicación del paralelismo, especialmente en los problemas donde el cálculo conlleve un coste importante en tiempo, utilizando para ello técnicas metaheurísticas [3, 14].
- Desarrollo de un *toolbox* con interfaz de usuario atractiva para MATLAB de *Scatter Search* (Búsqueda Dispersa).

El desarrollo de esta investigación se enmarcaría dentro de la realización de la tesis doctoral sobre la aplicación de técnicas metaheurísticas y paralelas a problemas de telecomunicaciones.

Bibliografía

[1] F. Almeida, M. J. Blesa Aguilera, C. Blum, J. M. Moreno-Vega, M. Pérez Pérez, A. Roli, and M. Sampels, editors. *Hybrid Metaheuristics, Third International Workshop, HM 2006, Gran Canaria, Spain, October 13-15, 2006, Proceedings*, volume 4030 of *Lecture Notes in Computer Science*. Springer, 2006.

[2] S. Amari. Synthesis of cross-coupled resonator filters using an analytical gradient-based optimization technique. *Microwave Theory and Techniques, IEEE Transactions on*, 48(9):1559–1564, September 2000.

[3] W. Bozejko and M. Wodecki. Parallel scatter search algorithm for the flow shop sequencing problem. In Wyrzykowski et al. [23], pages 180–188.

[4] R. J. Cameron. General coupling matrix synthesis methods for Chebyshev filtering functions. *Microwave Theory and Techniques, IEEE Transactions on*, 47:433–442, 1999.

[5] D. Giménez Cánovas, J. Cervera López, G. García Mateos, and N. Marín Pérez. *Algoritmos y Estructuras de Datos*, volume 2. Diego Marin Librero-Editor, 2003.

[6] A. Colorni, M. Dorigo, F. Maffioli, V. Maniezzo, G. Righini, and M. Trubian. Heuristics from nature for hard combinatorial optimization problems. *International Transactions in Operational Research*, 3(1):1–21, 1996.

[7] M. Dorigo and G. Di Caro. The ant colony optimization meta-heuristic. In David Corne, Marco Dorigo, and Fred Glover, editors, *New Ideas in Optimization*, pages 11–32. McGraw-Hill, London, 1999.

[8] C. Feremans, M. Labbé, and G. Laporte. Generalized network design problems. *European Journal of Operational Research*, 148(1):1–13, July 2003.

[9] F. Glover. Tabu search. *Part I. ORSA Journal on Computing*, (1):190–206, 1989.

[10] F. Glover, M. Laguna, and R. Martí. Fundamentals of scatter search and path relinking. *Control and Cybernetics*, 29(3):653–684, 2000.

[11] Fred W. Glover and Gary A. Kochenberger. *Handbook of Metaheuristics (International Series in Operations Research & Management Science)*. Springer, January 2003.

[12] **D. E. Goldberg. Genetic algorithms. In *Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.**

[13] **S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.**

[14] **S. Lukasik, Z. Kokosinski, and G. Swieton. Parallel simulated annealing algorithm for graph coloring problem. In Wyrzykowski et al. [23], pages 229–238.**

[15] **The MathWorks, http://www.mathworks.com. *Genetic Algorithm and Direct Search Toolbox User's Guide*. Version 2.3.**

[16] **The MathWorks, http://www.mathworks.com. *Optimization Toolbox User's Guide*. Version 4.0.**

[17] **Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, New York, second edition, 2006.**

[18] **G. R. Raidl. A unified view on hybrid metaheuristics. In Almeida et al. [1], pages 1–12.**

[19] **T. S. Rappaport, A. Annamalai, R. M. Buehrer, and W. H. Tranter. Wireless communications: Part events and a future perspective. *IEEE Communications Magazine*, 40(5):148–161, May 2002.**

[20] **M. G. C. Resende and C. C. Ribeiro. Greedy randomized adaptive search procedures. Technical report, AT&T Labs Research, August 2002. version 2.**

[21] **C. C. Ribeiro, S. L. Martins, and I. Rosseti. Metaheuristics for optimization problems in computer communications. *Computer Communications*, 30:656–669, 2007.**

[22] **M. Uhm, S. Nam, and J. Kim. Synthesis of resonator filters with arbitrary topology using hybrid method. *Microwave Theory and Techniques, IEEE Transactions on*, 55(10):2157–2167, October 2007.**

[23] **R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors. *Parallel Processing and Applied Mathematics, 7th International Conference, PPAM 2007, Gdansk, Poland, September 9-12, 2007, Revised Selected Papers*, volume 4967 of *Lecture Notes in Computer Science*. Springer, 2008.**

A. Funciones de Minimización del *Optimization Toolbox*

En el *Optimization Toolbox* de MATLAB, tenemos disponibles varias funciones de optimización (minimización) para funciones escalares de variables múltiples, de las cuales vamos a detallar un poco aquellas que se han usado en algún momento (no se describen aquí el resto de funciones disponibles en este *toolbox*). La información de este apéndice esta extraída de la documentación del *toolbox* de la guía de usuario que se incluye en el software y que también está disponible *on-line* en la página web del fabricante [33]. Es recomendable la lectura de esta documentación para una mejor comprensión de las herramientas de este *toolbox*.

A.1. fmincon

Encuentra un mínimo de una función multivariable con restricciones no lineales. *fmincon* es un método basado en el gradiente que esta diseñado para trabajar con problemas donde la función objetivo y las funciones de restricciones son continuas y tienen primera derivada.

A.1.1. Descripción

Encuentra un mínimo del problema especificado por

$$\min_x f(x) \text{ tal que } \begin{cases} c(x) \leq 0 \\ ceq(x) = 0 \\ A \cdot x \leq b \\ Aeq \cdot x = beq \\ lb \leq x \leq ub \end{cases}$$

donde *x*, *b*, *beq*, *lb*, y *ub* son vectores, *A* y *Aeq* son matrices, *c(x)* y *ceq(x)* son funciones que devuelven vectores, y *f(x)* es una función que devuelve un escalar. *f(x)*, *c(x)*, y *ceq(x)* pueden ser funciones no lineales.

A.1.2. Algoritmo

fmincon admite múltiples opciones, entre ellas la selección del tipo de algoritmo que se desea emplear en función de la naturaleza del problema. De este modo tenemos los siguientes algoritmos:

Optimización *Trust-Region-Reflective* El algoritmo *trust-region-reflective* es un método de región de confianza de subespacio y está basado en el método de Newton reflexivo-interior descrito en [27, 28]. Cada iteración consiste en la aproximación de la solución de un sistema lineal grande usando el método de gradientes conjugados precondicionados (PCG).

Optimización Active-Set *fmincon* usa un método de programación cuadrática secuencial (SQP). En este método, la función resuelve un subproblema cuadrático (QP) en cada iteración. *fmincon* actualiza una estimación del Hessiano del Lagrangiano en cada iteración usando la formula BFGS (descrito en [35, 34]).

Optimización Interior-Point Este algoritmo está descrito en [26, 28, 37].

A.2. fminimax

fminimax minimiza el valor del peor caso (el más grande) de un conjunto de funciones multivariable, empezando en un punto inicial estimado. A este tipo de problema se le suele llamar problema *minimax*.

A.2.1. Descripción

Encuentra un mínimo del problema especificado por

$$\min_x \max_i F_i(x) \text{ tal que } \begin{cases} c(x) \leq 0 \\ ceq(x) = 0 \\ A \cdot x \leq b \\ Aeq \cdot x = beq \\ lb \leq x \leq ub \end{cases}$$

donde *x*, *b*, *beq*, *lb*, y *ub* son vectores, *A* y *Aeq* son matrices, *c(x)*, *ceq(x)* y *F(x)* son funciones que devuelven vectores. *F(x)*, *c(x)*, y *ceq(x)* pueden ser funciones no lineales.

A.2.2. Algoritmo

fminimax internamente reformula el problema *minimax* como un problema de programación no lineal equivalente añadiendo restricciones adicionales de la forma $F_i(x) = y$ a las restricciones dadas en la formulación original, y entonces minimiza *y* sobre *x*. *fminimax* usa un método de programación cuadrática secuencial (SQP) [24] para resolver este problema.

La función a minimizar debe ser continua. *fminimax* podría dar soluciones locales óptimas solamente.

A.3. fminsearch

Encuentra el mínimo de una función multivariable sin restricciones usando un método que no necesita derivadas.

A.3.1. Descripción

Encuentra el mínimo de un problema especificado por

$$\min_x f(x)$$

donde x es un vector y *f(x)* es una función que devuelve un escalar.

A.3.2. Algoritmo

fminsearch usa el método de búsqueda *simplex* descrito en [31]. Se trata de un método de búsqueda directa que no usa gradientes ni numéricos ni analíticos como hace *fminunc*.

Si n es la longitud de x (número de componentes del vector), un espacio *simplex* n-dimensional se caracteriza por $n+1$ vectores distintos que son sus vértices. En el espacio bidimensional, un *simplex* es un triángulo; en el tridimensional es una pirámide. En cada paso de la búsqueda, se genera un punto dentro o cerca del actual *simplex*. El valor de la función en el nuevo punto se compara con el valor de la función en los vértices del *simplex*, y, normalmente, uno de los vértices es reemplazado por el nuevo punto, obteniendo un nuevo *simplex*. Este paso se repite hasta que el diámetro del *simplex* es menor que un valor de tolerancia dado.

fminsearch es generalmente menos eficiente que *fminunc* para problemas de orden mayor que dos. Sin embargo, para problemas con alta discontinuidad, *fminsearch* puede ser más robusto.

A.4. fminunc

Encuentra el mínimo de una función multivariable sin restricciones.

A.4.1. Descripción

Encuentra el mínimo de un problema especificado por

$$\min_x f(x)$$

donde x es un vector y *f(x)* es una función que devuelve un escalar.

A.4.2. Algoritmos

fminunc intenta encontrar un mínimo de una función escalar de varias variables, empezando por un punto inicial estimado. Normalmente se conoce a esto como *optimización no lineal sin restricciones*. *fminunc* utiliza dos algoritmos distintos en función del problema:

Optimización de Gran Escala Por defecto *fminunc* elige el algoritmo de gran escala si se le proporciona el gradiente en la función. Este algoritmo es un método de región de confianza de subespacio y se basa en el método de Newton de reflexión interior descrito en [27, 28]. Cada iteración conlleva aproximarse a la solución de un sistema lineal grande usando el método de los gradientes conjugados precondicionados (PCG).

Optimización de Media Escala *fminunc*, cuando así se le indica, usa el usa el método Quasi-Newton BFGS con un procedimiento de búsqueda de línea cúbica. Este método quasi-Newton usa la formula BFGS ([25, 29, 30, 36]) para actualizar la aproximación de la matriz hessiana.

Bibliografía

[24] R. K. Brayton, S. W. Director, G. D. Hachtel, and L. Vidigal. New algorithm for statistical circuit design based on quasi-newton methods and function splitting. *IEEE Trans. Circuits and Systems*, CAS-26:784–794, September 1979.

[25] C. G. Broyden. The convergence of a class of double-rank minimization algorithms. *Journal Inst. Math. Applic.*, 6:76–90, 1970.

[26] R. H. Byrd, J. C. Gilbert, and J. Nocedal. A trust region method based on interior point techniques for nonlinear programming. *Mathematical Programming*, 89:149–185, 2000.

[27] T. F. Coleman and Y. Li. On the convergence of reflective newton methods for large-scale nonlinear minimization subject to bounds. *Mathematical Programming*, 67(2):189–224, 1994.

[28] T. F. Coleman and Y. Li. An interior, trust region approach for nonlinear minimization subject to bounds. *SIAM Journal on Optimization*, 6:418–445, 1996.

[29] R. Fletcher. A new approach to variable metric algorithms. *Computer Journal*, 13:317–322, 1970.

[30] D. Goldfarb. A family of variable metric updates derived by variational means. *Mathematics of Computing*, 24:23–26, 1970.

[31] J. C. Lagarias, J. A. Reeds, M. H. Wright, and P. E. Wright. Convergence properties of the nelder-mead simplex method in low dimensions. *SIAM Journal of Optimization*, 9(1):112–147, 1998.

[32] The MathWorks, http://www.mathworks.com/access/helpdesk/help/toolbox/gads. *Genetic Algorithm and Direct Search Toolbox User's Guide*. Version 2.3.

[33] The MathWorks, http://www.mathworks.com/access/helpdesk/help/toolbox/optim. *Optimization Toolbox User's Guide*. Version 4.0.

[34] M. J. D. Powell. *The Convergence of Variable Metric Methods For Nonlinearly Constrained Optimization Calculations*. Academic Press, Amsterdam, The Netherlands, 1978.

[35] M. J. D. Powell. A fast algorithm for nonlinearly constrained optimization calculations. *Lecture Notes in Mathematics*, 630:144–157, 1978.

[36] D. F. Shanno. Conditioning of quasi-newton methods for function minimization. *Mathematics of Computing*, 24:647–656, 1970.

[37] R. A. Waltz, J. L. Morales, J. Nocedal, and D. Orban. An interior algorithm for nonlinear optimization that combines line search and trust region steps. *Mathematical Programming*, 107:391–408, 2006.

B. Implementación de *Scatter Search*

Esta implementación se basa en la implementación en C de *Scatter Search* de *R. Martí* (http://www.uv.es/%7Ermarti), que a su vez esta basada en lo expuesto en [10]. Esta implementación esta adaptada al entorno de MATLAB, y, además, se ha adaptado la nomenclatura para que sea similar a la de los algoritmos genéticos del *toolbox* de MATLAB, y también se ha seguido su filosofía en cuanto a la posibilidad de cambiar las funciones principales del algoritmo (creación de la población inicial, combinación, mejora e hibridación final) y en cuanto a los parámetros de ajuste y las condiciones de parada, aunque se ha optado por usar una programación orientada a objetos en lugar de la programación estructurada/funcional utilizada en el *toolbox* de MATLAB.

```
1  classdef ScatterSearch < handle
2  % ScatterSearch Es el resolutor/optimizador de problemas basado en Scatter Search
3  %   ScatterSearch es un clase que implementa un resolutor de problemas basado en
4  %   la tecnica metaheuristica Scatter Search.
5
6  % Ejemplo de uso:
7  %  [ceros, raices, cte] = cerosraices5();
8  %  pos = t2trisection3();
9  %  fitnessfnc=@(x)funcoste(ceros,raices,cte,pos,x);
10 %  miss = ScatterSearch(fitnessfnc, length(pos), 10, 10, 200, -5, 5);
11 % Para hacerlo mas informativo:
12 %  miss.Verbose=1;
13 % Y para empezar la optimizacion:
14 %  [x,fval]=miss.optimize();
15
16     properties
17         FitnessFnc = []; % Funcion de fitness
18
19         DiverFnc = @minqdist; % Funcion de medida de la diversidad
20
21         CreationFnc = {@gacreationlinearfeasible,ScatterSearch.opCreateFnc(), ...
22             'PopulationSize', 'PopInitRange'}; % Funcion de creacion
23         % La funcion de creacion se pasa en un cell con la estructura:
24         %   CreationFnc{1} => Manejador de la funcion
25         %   CreationFnc{2} => Opciones
26         %   CreationFnc{3} => Campo donde se indica el tamano de la Poblacion
27         %   CreationFnc{4} => Campo donde se indica la matriz de limites: fila
28         %    * la primera fila sera el vector con los limites inferiores
29         %    * la segunda fila sera el vector con los limites superiores
30
31         OptimFncOpt = optimset('MaxIter', 20, 'Display', 'off',...
32             'UseParallel', 'always'); % Opciones de funcion de optimizacion
33
34         OptimFnc = ScatterSearch.OptimFncDefault(); % Funcion de Mejora
35         % La funcion de mejora debe tener la siguiente signatura:
36         % [x, fval] = optimfnc(fitnessfnc, x, lb, ub, options)
37         OptimFactor = 0.2; % Tanto por 1 de aplicacion de la mejora
38
39         CombineFnc = @sscombine; % Funcion de combinacion de elementos
40         % La funcion de CombineFnc debe tener la siguiente signatura
41         % [offsprings] = CombineFnc(elems, n)
42         % donde elems son los elementos a combinar (cada fila un elemento),
43         % y n el numero de nuevos elementos que deben resultar de la
44         % combiancion en la matriz de salida offsprings.
45
46         NVar = uint8(0); % Numero de variables del problema
47         UpperBound = []; % Limite superior
48         LowerBound = []; % Limite inferior
49         Generations = uint32(50); % Numero de Maximo de Iteraciones
50         QSetSize = uint32(10); % Tamano del conjunto de calidad
51         DSetSize = uint32(10); % Tamano del conjunto de diversidad
52         PSize = uint32(200); % Tamano de la poblacion inicial
53         FitnessLimit = -Inf; % Parar al alcanzar este valor de fitness
54         HybridFnc = {ScatterSearch.HybridFncDefault(), ...
55             optimset('Algorithm', 'interior-point', 'MaxIter', 1000,...
56             'Display', 'off', 'UseParallel', 'always')}; % Funcion hibrida
57         OutputFcns = [];
```

```
58          StallGenLimit =  uint32(2); % Iteraciones de estancamiento.
59          StallTimeLimit = Inf; % Tiempo maximo sin mejorar
60          TimeLimit = Inf; % Tiempo maximo para llevar a cabo la optimizacion
61          Verbose = uint32(0); % Cada cuantas iteraciones muestra informacion
62      end
63
64      properties (SetAccess = 'protected') % , GetAccess = 'protected'
65          Generation = uint32(0); % Numero de Iteraciones actuales
66
67          QSet = []; % Soluciones de calidad, Size = QSetSize X (NVar+2)
68          % En QSet no solo guardamos las soluciones, sino tambien:
69          %   QSet(1) => Valor de calidad (funcion FitnessFnc)
70          %   QSet(2) => N. de iteraciones
71          %   QSet(3,end) => Valores de la solucion
72
73          DSet = []; % Soluciones de diversidad, Size = DSetSize X (NVar+2)
74          % En DSet no solo guardamos las soluciones, sino tambien:
75          %   DSet(1) => Valor de diversidad (funcion DiverFnc)
76          %   DSet(2) => N. de iteraciones
77          %   DSet(3,end) => Valores de la solucion
78          LastCombine = uint32(0);   % Iteración de ultima combinacion de elementos
79          NewElements = false;   % true se se han anadido nuevos elementos
80          ApplyOptimFnc = false; % Variable que indica si hay que hacer mejora o no
81          NApplyOptimFnc = uint32(0); % Numero de veces que se ha optimizado
82          % Variables de control para la parada
83          LastAdvanceTime = []; % Momento de la ultima de mejora
84          LastAdvance = uint32(0); % Iteracion de la ultima mejora
85          StopCause = []; % Causa de la parada
86          TolFun = 0; % Tolerancia en la funcion de fitness
87          XTol = 0; % Tolerancia empleada en las comparaciones de elem.
88          DTol = 0; % Tolerancia en la funcion de diversidad
89      end
90
91      properties (Dependent = true)
92          x;
93          fval;
94      end
95
96      methods (Static = true, Access = 'private')
97          function opciones = opCreateFnc()
98              opciones = gaoptimset(gaoptimset(@ga), 'Display', 'off',...
99                  'UseParallel', 'always');
100             opciones.LinearConstr = struct('type', 'boundconstraints');
101         end
102
103         function handle = OptimFncDefault()
104             handle = @(fun, x, lb, ub, opt) fminsearch(fun,x,opt);
105         end
106
107         function handle = HybridFncDefault()
108             handle = @(fun, x, lb, ub, opt) fmincon(fun, x, [], [], [],...
109                 [], lb, ub, [], opt);
110         end
111     end
112     methods
113         function ssObj = ScatterSearch(fitnessFnc, nvar, sizeQSet,...
114                 sizeDSet, populationSize, lowerBound, upperBound)
115             % Construccion del objeto
116             ssObj.FitnessFnc = fitnessFnc;
117             ssObj.NVar = nvar;
118             ssObj.PSize = populationSize;
119             ssObj.QSetSize = sizeQSet;
120             ssObj.DSetSize = sizeDSet;
121             ssObj.LowerBound = lowerBound;
122             ssObj.UpperBound = upperBound;
123         end
124
125         function [x,fval] = optimize(ss)
126             % Tiempo de inicio
127             tini = clock;
128
129             % Reserva de memoria
130             ss.QSet = zeros(ss.QSetSize, ss.NVar + 2);
131             ss.DSet = zeros(ss.DSetSize, ss.NVar + 2);
132
133             % Preparamos los argumentos para las funciones Creacion
134             ss.CreationFnc{2}.(ss.CreationFnc{3}) = ss.PSize;
135             ss.CreationFnc{2}.(ss.CreationFnc{4}) =...
136                 [ss.LowerBound ; ss.UpperBound];
137
138             % Mejora
139             ss.ApplyOptimFnc = isa(ss.OptimFnc, 'function_handle');
140             ss.NApplyOptimFnc = uint32(0);
141
142             % Calculo de la tolerancia inicial
143             ss.XTol = (ss.UpperBound-ss.LowerBound) ./ double(ss.PSize);
144
145             % Tratamos el caso de Verbose > 0
146             if (ss.Verbose > 0)
```

```
147                 regen = 0;
148                 disp('Iniciando. Generando conjunto inicial...');
149             end
150
151             % Inicializamos el conjunto de Referencia
152             ss.Initiate_RSet();
153
154             % Realizamos la optimizacion
155             while (ss.Generation < ss.Generations)
156                 if (ss.Verbose>0 && mod(ss.Generation, ss.Verbose)==0)
157                     fprintf('Gen. %d, fval: best=%.7g, mean=%.7g, x=' ,...
158                         ss.Generation, ss.fval, mean(ss.QSet(:,1)));
159                     disp(ss.x);
160                 end
161                 if (ss.NewElements)
162                     ss.Combine_RefSet();
163                 else
164                     ss.Update_DSet();
165
166                     if (ss.Verbose > 0)
167                         regen = regen + 1;
168                         fprintf('Generacion %d, **** REGENERACION n. %d\n' ,...
169                             ss.Generation, regen);
170                     end
171
172                     ss.Combine_RefSet();
173                 end
174
175                 % Examinamos las condiciones de parada
176                 if (isfinite(ss.FitnessLimit) && ...
177                         (ss.FitnessLimit - ss.fval > ss.TolFun))
178                     if (ss.Verbose > 0)
179                         disp('**** Parando al alcanzar FitnessLimit ****');
180                     end
181                     ss.StopCause = 'FitnessLimit';
182                     break;
183                 end
184                 if (isfinite(ss.StallTimeLimit) && ...
185                         etime(clock, ss.LastAdvanceTime)>ss.StallTimeLimit)
186                     if (ss.Verbose > 0)
187                         fprintf('**** No hay mejora en  %.0f segundos****\n' ,...
188                             etime(clock, ss.LastAdvanceTime)>ss.StallTimeLimit);
189                     end
190                     ss.StopCause = 'StallTimeLimit';
191                     break;
192                 end
193                 if (isfinite(ss.StallGenLimit) && ...
194                         ss.Generation-ss.LastAdvance >(ss.StallGenLimit+1))
195                     if (ss.Verbose > 0)
196                         fprintf('**** No mejora en  %d generaciones****\n' ,...
197                             ss.Generation - ss.LastAdvance - 1);
198                     end
199                     ss.StopCause = 'StallGenLimit';
200                     break;
201                 end
202                 if (isfinite(ss.TimeLimit) &&...
203                         etime(clock, tini)>ss.TimeLimit)
204                     if (ss.Verbose > 0)
205                         fprintf('Agotado Tiempo Limite, %.0f s.\n' ,...
206                             etime(clock, ss.LastAdvanceTime)>ss.StallTimeLimit);
207                     end
208                     ss.StopCause = 'TimeLimit';
209                     break;
210                 end
211             end
212             if (ss.Generation >= ss.Generations)
213                 ss.StopCause = 'Generations';
214             end
215
216             % Vemos si hay que aplicar la funcion hibrida
217             if (~isempty(ss.HybridFnc))
218                 if (ss.Verbose > 0)
219                     disp('>>>>>>> Conmutando a funcion hibrida <<<<<<<<');
220                 end
221                 if (isa(ss.HybridFnc, 'function_handle')) % Handle directo
222                     [x, fval] = ss.HybridFnc(ss.FitnessFnc, ss.x);
223                 else
224                     [x, fval] = ss.HybridFnc{1}(ss.FitnessFnc, ss.x, ...
225                         ss.LowerBound, ss.UpperBound, ss.HybridFnc{2});
226                 end
227                 if (ss.fval - fval > ss.TolFun)
228                     la = ss.LastAdvance;
229                     lat = ss.LastAdvanceTime;
230                     ss.try_add_QSet(ss.corrigeLimites(x));
231                     ss.LastAdvance = la;
232                     ss.LastAdvanceTime = lat;
233                 end
234
235             end
```

```
236
237             x = ss.x;
238             fval = ss.fval;
239             if (ss.Verbose > 0)
240                 fprintf('FINAL: Generation %d, fval=%.15g, x=',...
241                     ss.Generation, ss.fval);
242                 disp(ss.x);
243             end
244         end
245
246         function x = get.x(ss)
247             if (~isempty(ss.QSet))
248                 x = ss.QSet(1, 3:end);
249             else
250                 x = NaN;
251             end
252         end
253
254         function fval = get.fval(ss)
255             if (~isempty(ss.QSet))
256                 fval = ss.QSet(1,1);
257             else
258                 fval = NaN;
259             end
260         end
261
262         function set.QSetSize(ss, sizeQSet)
263             if (sizeQSet>ss.PSize/2)
264                 ss.QSetSize = uint32(ss.PSize/2);
265             else
266                 ss.QSetSize = uint32(sizeQSet);
267             end
268         end
269
270         function set.DSetSize(ss, sizeDSet)
271             if (sizeDSet>ss.PSize/2)
272                 ss.DSetSize = uint32(ss.PSize/2);
273             else
274                 ss.DSetSize = uint32(sizeDSet);
275             end
276         end
277
278         function set.PSize(ss, sizeP)
279             if (sizeP>2)
280                 ss.PSize = uint32(sizeP);
281                 ss.DSetSize = ss.DSetSize;
282                 ss.QSetSize = ss.QSetSize;
283             end
284         end
285
286         function set.NVar(ss, nvar)
287             if (nvar>0)
288                 ss.NVar = uint8(nvar);
289                 ss.LowerBound = ss.LowerBound;
290                 ss.UpperBound = ss.UpperBound;
291             end
292         end
293
294         function set.LowerBound(ss, lb)
295             if (~isempty(lb))
296                 if (length(lb)<ss.NVar)
297                     ss.LowerBound = ones(1, ss.NVar) * lb(1);
298                 else
299                     ss.LowerBound = lb(1:ss.NVar);
300                 end
301             else
302                 ss.LowerBound = lb;
303             end
304         end
305
306         function set.UpperBound(ss, ub)
307             if (~isempty(ub))
308                 if (length(ub)<ss.NVar)
309                     ss.UpperBound = ones(1, ss.NVar) * ub(1);
310                 else
311                     ss.UpperBound = ub(1:ss.NVar);
312                 end
313             else
314                 ss.UpperBound = ub;
315             end
316         end
317
318         function set.StallGenLimit(ss, sgl)
319             if (ss.StallGenLimit ~= sgl)
320                 if (isfinite(sgl))
321                     ss.StallGenLimit = uint32(sgl);
322                 else
323                     ss.StallGenLimit = sgl;
324                 end
```

```
325             end
326         end
327     end
328
329     methods (Access = 'protected')
330         function Initiate_RSet(ss)
331             % Consideramos que esta es la primera generacion
332             ss.Generation = uint32(1);
333
334             % Y por supuesto vamos a tener nuevos elementos
335             ss.NewElements = true;
336
337             % Debemos reservar la memoria para las soluciones
338             solutions = zeros(ss.PSize, ss.NVar + 2);
339             % En soluciones guardamos, en la posicion (1) el fitness y
340             % en la posicion 2 la diversidad y a continuacion los valores
341             % de la solucion (3:end).
342
343             % Ahora generamos los valores para la poblacion de trabajo
344             % inicial, para ello usaremos una funcion de creacion
345             solutions(:,3:end) = ss.CreationFnc{1}(ss.NVar,...
346                 ss.FitnessFnc, ss.CreationFnc{2});
347
348             % Calculamos el fitness de cada solucion
349             for l=1:size(solutions,1)
350                 solutions(l,1)=ss.FitnessFnc(solutions(l, 3:end));
351             end
352             % Ahora ordenamos los valores en funcion del fitness (de menor
353             % a mayor)
354             solutions = sortrows(solutions, 1);
355             % Y le aplicamos una mejora a los 10 mejores antes de pasarlos
356             % al conjunto de referencia si exite
357             if (ss.ApplyOptimFnc)
358                 for ind=1:ss.QSetSize
359                     % ss.OptimFnc{2}.(ss.OptimFnc{4}) = ...
360                     % solutions(ind,2:end);
361                     [ss.QSet(ind,3:end),ss.QSet(ind,1)] =...
362                         ss.OptimFncSimple(solutions(ind,3:end),...
363                         solutions(ind,1));
364                 end
365                 ss.QSet = sortrows(ss.QSet, 1);
366             else
367                 ss.QSet(:, 3:end) = solutions(1:ss.QSetSize, 3:end);
368                 ss.QSet(:, 1) = solutions(1:ss.QSetSize, 1); % Fitness
369             end
370             ss.QSet(:, 2) = ss.Generation; % Generation
371             ss.Update_TolFun(); % Actualizamos la tolerancia
372
373             % Creamos el conjuto de diversidad
374             ss.Update_DSet(solutions(ss.QSetSize+1:end,:));
375
376             ss.LastCombine = uint32(0);
377         end
378
379         function Update_DSet(ss, solutions)
380             % Funcion interna. Actualiza el conjunto de referencia
381             % dedicado a la diversidad. Si no se le pasa un conjunto
382             % de soluciones pregenerado, lo genera el mismo.
383             if (nargin==1 || isempty(solutions))
384                 % Aumentamos el valor de la generacion
385                 ss.Generation = ss.Generation + 1;
386
387                 % Aumentamos un poco la tolerancia
388                 ss.XTol = ss.XTol / 2;
389
390                 % Reseteamos la generacion del conjunto QSet para permitir
391                 % nuevas combinaciones
392                 ss.QSet(:,2) = ss.Generation;
393
394                 % Debemos reservar la memoria para las soluciones
395                 solutions = zeros(ss.PSize, ss.NVar + 2);
396                 % En soluciones guardamos, en la posicion (1) el fitness,
397                 %  la distancia minima al resto en la posicion (2),
398                 %  y a continuacion los valores de la solucion (3:end).
399
400                 % Ahora generamos los valores para la poblacion de trabajo
401                 % inicial, para ello usaremos una funcion de creacion
402                 ss.CreationFnc{2}.(ss.CreationFnc{3}) = ss.PSize;
403                 ss.CreationFnc{2}.(ss.CreationFnc{4}) =...
404                     [ss.LowerBound ; ss.UpperBound];
405                 solutions(:,3:end) = ss.CreationFnc{1}(ss.NVar,...
406                     ss.FitnessFnc, ss.CreationFnc{2});
407
408                 for l=1:size(solutions,1)
409                     solutions(l,1)=ss.FitnessFnc(solutions(l, 3:end));
410                 end
411             end
412
413             % Calculamos la diversidad a la solucion Q
```

```
414             for l=1:size(solutions,1)
415                 solutions(l,2)=ss.DiverFnc(solutions(l, 3:end),...
416                     ss.QSet(:,3:end));
417             end
418
419             % Primero ordenamos por fitness
420             solutions = sortrows(solutions, 1);
421             % Nos quedamos con la mitad que tenga mejor fitness y
422             % del resto las DSetSize/2 con mejor diversidad
423             solutionsQ = solutions(1:round(size(solutions,1)/2),:);
424
425             % Ahora ordenamos por diver
426             solutions = sortrows(solutions, -2);
427             solutionsD = solutions(1:ss.DSetSize/2,:);
428
429             % Y unimos ambas
430             solutions = [solutionsQ ; solutionsD];
431
432             % Vamos seleccionando de uno en uno los mas diversos,
433             % actualizando la diversidad del resto respecto a este.
434             for l=1:ss.DSetSize
435                 % Ordenamos los valores en funcion de la diversidad
436                 % (de mayor a menor)
437                 solutions = sortrows(solutions, -2);
438                 ss.DSet(l, 3:end) = solutions(1, 3:end); % Solucion
439                 ss.DSet(l, 2) = ss.Generation; % Iteracion
440                 ss.DSet(l, 1) = solutions(1, 2); % Diversidad
441
442                 if (l<ss.DSetSize)
443                     % Actualizamos los valores de diversidad teniendo en cuenta
444                     % el valor que hemos anexado.
445                     for ll=1:size(solutions,1)
446                         solutions(ll,2)=min([solutions(ll, 2),...
447                             ss.DiverFnc(solutions(ll,3:end),...
448                             ss.DSet(l,3:end))]);
449                     end
450                 end
451             end
452
453             % Actualizamos la diversidad por completo del conjuto de
454             % referencia de diversidad (DSet), sin incluir el QSet ya
455             % que lo hemos tratado previamente.
456             ss.Update_DSet_Diver(false);
457         end
458
459         function Update_DSet_Diver(ss, completo)
460             %Update_DSet_Diver permite actualizar el conjuto DSet en
461             %cuanto a diversidad.
462             % Update_DSet_Diver(ss, completo) => completo=true indica
463             % recalcular completamente la diversidad volviendo a medirla
464             % respecto al conjunto QSet.
465             if (nargin>=2 && completo)
466                 % Primero calculamos la diversidad de todas las soluciones
467                 % del conjunto RefSet respecto al conjunto QSet, ya que
468                 % se ha seleccionado un recalculo completo.
469                 for l=1:ss.DSetSize
470                     ss.DSet(l,1)=ss.DiverFnc(ss.DSet(l, 3:end),...
471                         ss.QSet(:,3:end));
472                 end
473             end
474
475             % Actualizamos la diversidad del conjunto DSet completo
476             ss.DSet(1, 1) = min([ss.DSet(1, 1),...
477                 ss.DiverFnc(ss.DSet(1, 3:end),...
478                 ss.DSet(2:end, 3:end))]);
479             for l=2:ss.DSetSize-1
480                 ss.DSet(l, 1) = min([ss.DSet(l, 1), ss.DiverFnc(...
481                     ss.DSet(l,3:end), ss.DSet(1:l-1,3:end)),...
482                     ss.DiverFnc(ss.DSet(l,3:end),...
483                     ss.DSet(l+1:end,3:end))]);
484             end
485             ss.DSet(end, 1) = min([ss.DSet(end, 1),...
486                 ss.DiverFnc(ss.DSet(end, 3:end),...
487                 ss.DSet(1:ss.DSetSize-1, 3:end))]);
488
489             % Por ultimo ordenamos el conjunto de diversidad de mas a menos
490             ss.DSet = sortrows(ss.DSet, -1);
491
492             % Y actualizamos la tolerancia
493             ss.DTol = (ss.DSet(1,1)-ss.DSet(end,1))/double(ss.DSetSize/2);
494         end
495
496         function Combine_RefSet(ss)
497             ss.NewElements = false;
498
499             % Extraemos cuantos elementos nuevos hay en cada conjunto
500             nq =  sum(ss.QSet(:,2)>ss.LastCombine);
501             nd =  sum(ss.DSet(:,2)>ss.LastCombine);
502
```

```
503             if (nq>=2)
504                 qxqSize = 4*(factorial(nq)/...
505                     (factorial(nq-2)*2)+nq*(ss.QSetSize-nq));
506             else
507                 qxqSize = 4*nq*(ss.QSetSize-nq);
508             end
509
510             if (nd>=2)
511                 dxdSize = 2*(factorial(nd)/...
512                     (factorial(nd-2)*2)+ nd*(ss.DSetSize-nd));
513             else
514                 dxdSize = 2*(nd*(ss.DSetSize-nd));
515             end
516
517             total_size=...
518                 qxqSize+...
519                 3*(nq*ss.DSetSize + nd*ss.QSetSize - nq*nd)+...
520                 dxdSize;
521
522             np = zeros(total_size,ss.NVar);
523             np_size = 1;
524
525             % Combinamos primero entre los elementos del QSet
526             % que en teoria debe contener mucha informacion sobre
527             % la solucion optima.
528             for ii=1:ss.QSetSize
529                 for jj=ii+1:ss.QSetSize
530                     if (ss.QSet(ii, 2) > ss.LastCombine || ...
531                             ss.QSet(jj, 2) > ss.LastCombine)
532                         np(np_size:np_size+3,:) = ss.CombineFnc([
533                             ss.QSet(ii,3:end); ss.QSet(jj,3:end)], 4);
534                         np_size = np_size + 4;
535                     end
536                 end
537             end
538
539             % Combinamos a continuacion los elementos del QSet con los
540             % del DSet, para garantizar explorar guiados por los mejores
541             for ii=1:ss.QSetSize
542                 for jj=1:ss.DSetSize
543                     if (ss.QSet(ii, 2) > ss.LastCombine || ...
544                             ss.DSet(jj, 2) > ss.LastCombine)
545                         np(np_size:np_size+2,:) = ss.CombineFnc([
546                             ss.QSet(ii,3:end); ss.DSet(jj,3:end)], 3);
547                         np_size = np_size + 3;
548                     end
549                 end
550             end
551
552             % Por ultimo combinamos entre si los elementos del DSet
553             % buscando sobre todo mejorar la diversidad
554             for ii=1:ss.DSetSize
555                 for jj=ii+1:ss.DSetSize
556                     if (ss.DSet(ii, 2) > ss.LastCombine || ...
557                             ss.DSet(jj, 2) > ss.LastCombine)
558                         np(np_size:np_size+1,:) = ss.CombineFnc([
559                             ss.DSet(ii,3:end); ss.DSet(jj,3:end)], 2);
560                         np_size = np_size + 2;
561                     end
562                 end
563             end
564
565             if (mod(ss.Generation, ss.Verbose)==0)
566                 fprintf('* %d cand. ',np_size-1);
567                 if (total_size ~= (np_size-1))
568                     fprintf('Fallo al calcular total_size (=%d)\n',...
569                         total_size);
570                 end
571             end
572
573             % Actualizamos las variables de control
574             ss.LastCombine = ss.Generation;
575             ss.Generation = ss.Generation + 1;
576
577             % Corregimos las soluciones por si acaso
578             np=ss.corrigeLimites(np(1:np_size-1, :));
579
580             % Vemos si alguna de las soluciones generadas nos sirve
581             for ii=1:np_size-1
582                 ss.try_add_QSet(np(ii,:));
583                 ss.try_add_DSet(np(ii,:));
584             end
585         end
586
587         function try_add_QSet(ss, sol)
588             %ry_add_QSet. Intenta anadir una solucion al conjunto de
589             % referencia QSet, aplicando la funcion de mejora.
590
591             if (ss.ApplyOptimFnc && rand()<=ss.OptimFactor)
```

```
592                 [sol, fitness] = ss.OptimFncSimple(sol);
593             else
594                 fitness = ss.FitnessFnc(sol);
595             end
596
597             if ((ss.QSet(end,1)-fitness>ss.TolFun)&&(ss.is_new(sol)))
598                 % Es mejor que el ultimo y no existe previamente, lo
599                 % incorporamos en el conjunto de referencia QSet
600                 ss.NewElements = true; % Indicamos que tenemos nuevo elem.
601                 ss.LastAdvance = ss.Generation; % Generacion de mejora
602                 ss.LastAdvanceTime = clock; % Momento de esta mejora
603                 ss.QSet(end, 1) = fitness; % Valor de fitnes
604                 ss.QSet(end, 2) = ss.Generation; % Generacion
605                 ss.QSet(end, 3:end) = sol; % Solucion
606                 ss.QSet = sortrows(ss.QSet, 1); % Reordenamos QSet
607                 ss.Update_TolFun();
608                 ss.Update_DSet_Diver(true); % Recalculamos Diver
609             end
610         end
611
612         function try_add_DSet(ss, sol)
613             %try_add_DSet. Intenta incorporar una elem. al conjunto de
614             % referencia DSet, sin aplicar funcion de mejora (interesa
615             % la diversidad y no la bondad).
616             diver = min([ss.DiverFnc(sol, ss.QSet(:,3:end)),...
617                 ss.DiverFnc(sol, ss.DSet(:,3:end))]);
618
619             if (diver - ss.DSet(end, 1) > ss.DTol)
620                 % Mejoramos la diversidad peor del conjunto DSet
621                 ss.NewElements = true; % Indicamos que tenemos nuevo elem.
622                 ss.DSet(end, 3:end) = sol; % Guardamos la sol
623                 ss.DSet(end, 2) = ss.Generation;
624                 ss.DSet(end, 1) = diver;
625                 ss.Update_DSet_Diver(true); % Recalculamos Diver
626             end
627         end
628
629         function [solfeasible] = corrigeLimites(ss, sol)
630             solfeasible = sol;
631             if (~isempty(ss.UpperBound))
632                 mihigh = repmat(ss.UpperBound, size(solfeasible, 1), 1);
633                 indmal = solfeasible > mihigh;
634                 if (any(any(indmal)))
635                     solfeasible(indmal) = mihigh(indmal);
636                 end
637             end
638             if (~isempty(ss.LowerBound))
639                 milow = repmat(ss.LowerBound, size(solfeasible, 1), 1);
640                 indmal = solfeasible < milow;
641                 if (any(any(indmal)))
642                     solfeasible(indmal) = milow(indmal);
643                 end
644             end
645         end
646
647         function [isnew] = is_new(ss, sol)
648             % Comprueba si la solucion existe en el conjunto QSet
649             isnew = all(sum(abs(ss.QSet(:, 3:end)...
650                 - repmat(sol, ss.QSetSize, 1)) > ss.XTol, 2));
651         end
652
653         function [nx, nfval] = OptimFncSimple(ss, x, fval)
654             ss.NApplyOptimFnc = ss.NApplyOptimFnc + 1;
655             [nx, nfval] = ss.OptimFnc(ss.FitnessFnc, x,...
656                 ss.LowerBound, ss.UpperBound, ss.OptimFncOpt);
657             nx = ss.corrigeLimites(nx);
658             if (nargin>2 && isfinite(fval) && fval-nfval<ss.TolFun)
659                 % No mejoramos, posiblemente al corregir los limites
660                 nx = x;
661                 nfval = fval;
662             end
663         end
664
665         function Update_TolFun(ss)
666             ss.TolFun=(ss.QSet(end,1)-ss.QSet(1,1))/double(ss.QSetSize/2);
667         end
668     end
669 end
```

C. Implementación de *Backtracking*

Dado que precisamos una implementación de *Backtracking* que busque en un espacio de soluciones potencialmente infinito y no discreto como se realiza de forma clásica, en esta versión se requiere que se defina el número de valores de cada variable que se van a estudiar dentro de los límites definidos, concretándose de ese modo el conjunto de soluciones que se va a explorar. La función devuelve, además de la mejor solución, una matriz con las mejores soluciones ordenadas por calidad de tal modo que esta función puede emplearse para generar una entrada de cierta calidad para otra que realice una búsqueda en las cercanías de un punto como las funciones de búsqueda local.

```
1  function [ x, fval, xsel ] = backtracking( fitnessfnc, nvars, lb, ub,...
2      meshsize, numsol, verbose )
3  %BACKTRACKING Resuelve el problema mediante el uso de backtracking
4  % Encuentras los NUMSOL mejores soluciones en cuanto a FITNESSFNC (min.)
5  % realizando un recorrido exhaustivo entre los valores de LB y UB (limites
6  % inferior y superior respectivamente), en MESHSIZE pasos (debe ser impar),
7  % probando todas las combinaciones para las NVARS variables.
8  % Devuelve en X la mejor solucion, en FVAL el valor de fitness de esa
9  % solución y en XSEL una matriz donde la primera columna es el valor de
10 % fitness de la solución contenida en esa misma fila.
11
12 xsel = zeros(numsol, nvars + 1);
13 xsel(:, 1) = Inf; %En la primera columna guardamos el valor de fitnes
14 xsel(:, 2:end) = NaN; %En el resto la solución correspondiente
15
16 if (length(lb)<nvars)
17     lb = ones(1, nvars) * lb(1);
18 end
19
20 if (length(ub)<nvars)
21     ub = ones(1, nvars) * ub(1);
22 end
23
24 sol = zeros(1, nvars);
25 ind = zeros(1, nvars);
26
27 nivel = 1;
28 cont = 0;
29
30 if (nargin<7)
31     verbose=false;
32 end
33
34 total = (meshsize^nvars);
35
36 %Esto es verbose
37 if (verbose)
38     fprintf('Realizando un backtracking completo de %d iteraciones:\n',...
39         total);
40 end
41
42 while (nivel>0)
43     %Generamos siguiente solución a evaluar
44     sol(nivel) = (lb(nivel)+ub(nivel))/2 + ((-1)^ind(nivel))...
```

```
45          * ceil(ind(nivel)/2)*(ub(nivel)-lb(nivel))/(meshsize-1);
46
47      ind(nivel)=ind(nivel) + 1; % Preparamos para el siguiente hermano
48
49      if (nivel == nvars) % Comprobamos si es solución
50          nfval = fitnessfnc(sol); % Calculamos su fitness
51
52          % Esto es para hacer test
53          %disp(sol);
54
55          % Esto es verbose
56          if (verbose)
57              cont = cont + 1;
58              if (mod(cont, 1000)==0)
59                  fprintf('.');
60                  if (mod(cont, 70000)==0)
61                      fprintf(' %g%%\n', (100*cont/total));
62                  end
63              end
64          end
65
66          if (nfval < xsel(end, 1)) % Mejoramos
67              xsel(end, 1) = nfval; % Guardamos la mejor solución (fval)
68              xsel(end, 2:end) = sol; % Guardamos la mejor solución (x)
69              xsel = sortrows(xsel, 1);
70              if (verbose)
71                  fprintf('********\n*MEJORA* -> fval=%g, x:', nfval);
72                  disp(sol);
73              end
74          end
75      else
76          nivel = nivel + 1; % Pasamos al siguiente nivel
77      end
78
79      while (nivel>0 && ind(nivel)>=meshsize) % Hemos examinado todos los hermanos?
80          ind(nivel) = 0; % Debemos retroceder de nivel
81          nivel = nivel - 1;
82      end
83  end
84  fval = xsel(1, 1);
85  x = xsel(1, 2:end);
86  end
```

D. Funciones de Bondad para el Diseño de Filtros

La función de bondad empleada para ver cuan buenos son los parámetros de diseño dados para cada problema planteado en el capítulo 3 recibe como entrada para definir el problema la topología del problema (posiciones no nulas de la matriz de acoplamientos), los ceros y polos de la función de transferencia, y una constante relacionada con la respuesta del filtro. El último parámetro de la función es el vector con los parámetros de diseño que se están evaluando. La versión original de estas funciones nos las ha proporcionado el grupo de investigación de *Electromagnetismo aplicado a las Telecomunicaciones* del *Departamento de Tecnologías de la Información y las Comunicaciones* de la Universidad Politécnica de Cartagena. Sobre esa base, se ha rehecho el código, extrayendo las partes comunes en nuevas funciones auxiliares y realizando una serie de optimizaciones necesarias para que el rendimiento del código fuese satisfactorio para las pruebas a realizar (mejorando los tiempos en 3 órdenes de magnitud).

D.1. Función de Bondad General

La función de bondad está implementada con el nombre `funcoste`, que emplea a su vez la función auxiliar `parametros`. A continuación se muestra el código fuente de ambas funciones.

D.1.1. funcoste.m

```
1  function [coste]=funcoste(ceros, raices, cte, posiciones, acoplos)
2  %FUNCOSTE es una función generica de coste de un determinado ajuste de
3  % acoplos con respecto a lo buscado (ceros, raices y cte) usando la estructura
4  % indicada en posiciones (que lugares ocupan los
5  % valores de 'acoplos' en la matriz base llamada Mini)
6  % Ejemplo:
7  %   [ceros, raices] = cerosraices([3*j, -3*j],[0, 0.8744, -0.8744]);
8  %   cte = 1/10
9  %   Mini = zeros(5);
10 %% %M11 M22 M33 Ms1 Ms2 M1l M23 M2l
11 %   posiciones={[7,10],[13],[19],[2,6],[3,11],[10,22],[14,18],[15,23]};
12 %% %Deben tener valores entre -5 y +5.
13 %% %El vector inicial no puede tener valores nulos
14 %   acoplos = [-4, -3, -2, -1, 1, 2, 3, 4];
15 %   coste = funcoste(ceros, raices, cte, Mini, posiciones, acoplos);
16 %  => coste = 2.9884
17 %% %Tenemos una solución exacta conocida para este ejemplo:
18 %   acoplos=[0   0   0 0.8025 0.7054 -0.8025 -1.4305 0.7054];
19 %   => coste = 7.1543e-010
20
```

```
21  %% Rellenamos la matriz Mini con los datos
22  Mini = zeros(5);
23  for a=1:length(posiciones)
24      Mini(posiciones{a}) = acoplos(a);
25  end
26
27  %% CALCULO DEL COSTE
28  [S11_wz]=parametros(raices,1,1,Mini);
29  [S11_wp,S21_wp]=parametros(ceros,1,1,Mini);
30  [S11_menos1]=parametros(-1,1,1,Mini);
31  [S11_mas1]=parametros(1,1,1,Mini);
32
33  ccoste1=sum(abs(S11_wz).^2);
34  ccoste2=sum(abs(S21_wp).^2);
35  ccoste3=(abs(S11_menos1)-cte).^2;
36  ccoste4=(abs(S11_mas1)-cte).^2;
37
38  coste=ccoste1+ccoste2+ccoste3+ccoste4;
```

D.1.2. parametros.m

```
1   function [s11,s21]=parametros(w,R1,Rn,M)
2
3   N=length(M);
4   R=zeros(N,N);
5   R(1,1)=R1;
6   R(N,N)=Rn;
7
8   V=zeros(N,1);
9   V(1)=1;
10  I=eye(N);
11  I(1,1)=0;
12  I(N,N)=0;
13
14  Mr=R+j.*M;
15  lw=length(w);
16  s11=zeros(lw,1);
17  s21=zeros(lw,1);
18
19  for i=1:lw
20      s=j*(w(i));
21      Z=(s.*I)+Mr;
22      solucion=Z\V;
23
24      Ainv11=j*solucion(1);
25      s11(i)=1+(2*j*R1*Ainv11);
26
27      AinvN1=j*solucion(end);
28      s21(i)=-2*sqrt(R1*Rn)*j*AinvN1;
29  end
```

D.2. Topologías

Las topologías en estudio son tres: *Cometa*, *Transversal* y *Dos Trisection de Orden Tres*. Una topología está descrita por las posiciones no nulas de la matriz base que define los acoplos entre la red de resonadores del filtro. A continuación se muestran las posiciones de cada una de ellas como funciones de MATLAB así como la matriz resultante.

D.2.1. Cometa

D.2.1.1. Función

```
1  function [posiciones] = tcometa()
2  %TCOMETA Devuelve la estructura de la topología cometa para ser usada en la
3  %función funcoste.
4
5  persistent posicionesp
6
7  if (isempty(posicionesp))
8      posicionesp = {7, 13, 19, [2,6], [4,16], [8,12], [10,22], [20,24]};
9  end
10
11 posiciones = posicionesp;
12
13 end
```

D.2.1.2. Matriz

Esta es la matriz correspondiente a las posiciones indicadas por la función `tcometa`:

$$M = \begin{pmatrix} 0 & a_4 & 0 & a_5 & 0 \\ a_4 & a_1 & a_6 & 0 & a_7 \\ 0 & a_6 & a_2 & 0 & 0 \\ a_5 & 0 & 0 & a_3 & a_8 \\ 0 & a_7 & 0 & a_8 & 0 \end{pmatrix}$$

D.2.2. Transversal

D.2.2.1. Función

```
1  function [posiciones] = ttransversal()
2  %TTRANSVERSAL Devuelve la estructura de la topología transversal para ser
3  %usada en la función funcoste.
4
5  persistent posicionesp
6
7  if (isempty(posicionesp))
8      posicionesp = {7,13,19,[2,6],[3,11],[4,16],[10,22],[15,23],[20,24]};
9  end
10
11 posiciones = posicionesp;
12
13 end
```

D.2.2.2. Matriz

Esta es la matriz correspondiente a las posiciones indicadas por la función `ttransversal`:

$$M = \begin{pmatrix} 0 & a_4 & a_5 & a_6 & 0 \\ a_4 & a_1 & 0 & 0 & a_7 \\ a_5 & 0 & a_2 & 0 & a_8 \\ a_6 & 0 & 0 & a_3 & a_9 \\ 0 & a_7 & a_8 & a_9 & 0 \end{pmatrix}$$

D.2.3. Dos Trisection de Orden Tres

D.2.3.1. Función

```
function [posiciones] = t2trisection3()
%T2TRISECTION3 Devuelve la estructura de la topología t2trisection3 para
%ser usada en la función funcoste.

persistent posicionesp

if (isempty(posicionesp))
    posicionesp = {7,13,19,[2,6],[3,11],[8,12],[14,18],[15,23],[20,24]};
end

posiciones = posicionesp;

end
```

D.2.3.2. Matriz

Esta es la matriz correspondiente a las posiciones indicadas por la función `t2trisection3`:

$$M = \begin{pmatrix} 0 & a_4 & a_5 & 0 & 0 \\ a_4 & a_1 & a_6 & 0 & 0 \\ a_5 & a_6 & a_2 & a_7 & a_8 \\ 0 & 0 & a_7 & a_3 & a_9 \\ 0 & a_7 & a_8 & a_9 & 0 \end{pmatrix}$$

D.3. Funciones de Transferencia

Para definir la función de bondad asociada a un problema de diseño de filtros acoplados, además de la topología se usan unos parámetros adicionales que corresponden a los ceros y polos de la función de transferencia del sistema, así como una constante relacionada con la respuesta esperada del mismo. En los problemas descritos en este documento se emplean dos configuraciones para estos parámetros, `cerosraices5` y `cerosraices3`, que a continuación se detallan.

D.3.1. cerosraices5

```
function [ ceros, raices, cte ] = cerosraices5()
%CEROSRAICES5 Devuelve los ceros y raizes, asi como la constante de la tf

persistent cerosp raicesp ctep

if (isempty(cerosp))
    %Calculamos los ceros y raices
    ceros_s = [-5*j, -3*j];
    polos = [0.8153, -0.9067, -0.1812];
    [cerosp, raicesp] = cerosraices(ceros_s, polos);

    %Y calculamos la constante
    RL_DB = 15;
    ctep = 10^(-RL_DB/20);
```

```
15 end
16
17 ceros = cerosp;
18 raices = raicesp;
19 cte = ctep;
20
21 end
```

D.3.2. cerosraices3

```
1  function [ ceros, raices, cte ] = cerosraices3()
2  %CEROSRAICES3 Devuelve los ceros y raizes de tf con ceros en -3 y 3
3
4  persistent cerosp raicesp ctep
5
6  if (isempty(cerosp) || isempty(raicesp))
7      ceros_s=[3*j, -3*j];
8      polos=[0, 0.8744, -0.8744];
9      [cerosp, raicesp] = cerosraices(ceros_s, polos);
10
11     %Y calculamos la constante
12     RL_DB = 20;
13     ctep = 10^(-RL_DB/20);
14 end
15
16 ceros = cerosp;
17 raices = raicesp;
18 cte = ctep;
19
20 end
```

E. Funciones Personalizadas para ga y ScatterSearch

Tanto `ga` (Algoritmo Genético) como `ScatterSearch` (Búsqueda Dispersa) admiten entre sus parámetros el fijar las funciones que se usan para realizar ciertas tareas, como son la selección, la combinación, el cruce, la mutación, la mejora y la creación de una población inicial.

Para algunas de ellas se han implementado funciones personalizadas cuyo código fuente se detalla en este apéndice.

E.1. generapinicial (creación de población inicial)

```
1  function pinicial = generapinicial(variables, funcoste, options)
2  %GENERAPINICIAL Permite generar una población inicial del tamaño 'tamano'
3  %para la función 'funcoste' dada, que admite 'variables' variables
4  %Util para los algoritmos de busqueda. Ej.
5  %  poblacion = generapinicial(@funcostetc, 8, 100)
6  %Para generar la población se usa un algoritmo de busqueda directa
7  %partiendo de un punto al azar.
8
9  %Llamamos a la función por defecto de generación
10 %y trabajaremos sobre los valores que devuelva
11 pinicial = gacreationlinearfeasible(variables, funcoste, options);
12
13 range = options.PopInitRange;
14 minimo = range(1,:);
15 maximo = range(2,:);
16
17 %Creamos las opciones que vamos a usar en la busqueda
18 persistent opciones
19 if isempty(opciones)
20     opciones = optimset('Algorithm', 'interior-point', 'MaxIter', 100,...
21         'Display', 'off');
22 end
23
24 %Definimos la estructura a usar para el problema de optimización
25 problema=struct('objective', funcoste, 'x0', [], 'Aineq', [],...
26     'bineq', [], 'Aeq', [], 'beq', [], 'lb', minimo, 'ub', maximo, ...
27     'nonlcon', [], 'solver', 'fmincon', 'options', opciones);
28
29 %Iteramos hasta completar el tamaño (solo optimizamos EliteCount*2 elementos)
30 tamano=size(pinicial,1);
31 cada=ceil(tamano/(options.EliteCount*2));
32 for l=1:cada:tamano
33     problema.x0 = pinicial(l,:);
34     t=fmincon(problema);
35     if (isfinite(t))
36         pinicial(l,:)=t;
37     end
38 end
```

```
39
40 end
```

E.2. sscombine (combinación)

```
1  function [ offsprings ] = sscombine( elementos, n )
2  %SSCOMBINE Genera las n combinaciones siguiendo el metodo por defecto de SS
3
4  if (size(elementos,1)<2)
5      offsprings = elementos;
6  else
7      %De momento solo vamos a trabajar con 2 elementos...
8      x = elementos(1, :);
9      y = elementos(2, :);
10
11     %Reservamos la memoria para el numero de hijos esperado
12     offsprings = zeros(n, length(x));
13
14     %Calculamos un vector director base para la combinacion
15     d = (y - x) ./ 2;
16
17     %Obtenemos un valor aleatorio que nos servira para tomar
18     %decisiones.
19     a = floor(rand() * 3);
20
21     for ind=1:n
22         r = rand();
23         switch (mod(a, 3))
24             case 0 %Generamos C2
25                 offsprings(ind, :) = x + (r .* d);
26             case 1 %Generamos C1
27                 offsprings(ind, :) = x - (r .* d);
28             otherwise %Generamos C3
29                 offsprings(ind, :) = y + (r .* d);
30         end
31         a = a + 1;
32     end
33 end
34 end
```

E.3. gacrossoveradapt (combinación)

```
1  function [ offsprings ] = gacrossoveradapt(cof, opciones, ff, elementos,...
2      n, cofargs)
3  %GACROSSOVERADAPT Adapta a ScatterSearch una función de cruce de los alg.
4  %geneticos del toolbox de MATLAB.
5  %   Los parametros necesarios son:
6  %     * cof: Funcion de cruce de geneticos
7  %     * opciones: opciones a usar en la función de cruce de geneticos
8  %     * ff: funcion de fitness
9  %     * cofargs: argumentos adicionales para la función de geneticos
10 %     * elementos: elementos a cruzar de Scatter Search
11 %     * n: numero de hijos a generar
12 %
13 %Ejemplo (suponiendo que miss es un objeto de la clase ScatterSearch):
14 %     opciones = miss.CreationFnc{2};
15 %     opciones.LinearConstr.lb = miss.LowerBound'; %Transpuesta de lb
16 %     opciones.LinearConstr.ub = miss.UpperBound'; %Transpuesta de ub
17 %     miss.CombineFnc = @(e,n) gacrossoveradapt(@crossoverheuristic,...
18 %         opciones, [], e, n)
19
20 %Calculamos el numero de elementos a pasar a la funcion
```

```
21  necesarios = ceil(n*2 / (size(elementos, 1)));
22  if (necesarios>1)
23      indices = repmat(1:size(elementos,1), 1, necesarios);
24  else
25      indices = 1:n*2;
26  end
27
28  if (nargin < 6)
29      cofargs = {};
30  end
31
32  offsprings=cof(indices,opciones,size(elementos,2), ff,...
33      ones(necesarios,1),elementos,cofargs{:});
34  end
```

E.4. sscombineadapt (cruce)

```
1   function xoverKids = sscombineadapt(parents, options, nvars, FitnessFcn, thisScore
        , thisPopulation)
2   %SSCOMBINEADAPT es una funcion que adapta a los algoritmos geneticos la
3   %funcion de combinacion sscombine creada originalmente para Scatter Search
4   %Los parametros que debe admitir esta funcion son:
5   % * parents: Vector con los padres elegidos por la funcion de seleccion
6   % * options: Estructura con las opciones - no se usa
7   % * nvars: Numero de variables
8   % * FitnessFcn: Funcion de Fitness - no se usa
9   % * thisScore: Matriz de puntuacion - no se usa
10  % * thisPopulation: La matriz con la poblacion actual
11
12  %Cuantos hijos hay que generar?
13  nKids = length(parents)/2;
14
15  %Reserva el espacio para los hijos del cruze
16  xoverKids = zeros(nKids,nvars);
17
18  for ind=1:nKids
19      %Hacemos un cruce y pedimos solo un hijo
20      xoverKids(ind,:)=sscombine(thisPopulation(parents(ind*2-1:ind*2),:),1);
21  end
22
23  %Extract information about linear constraints, if any
24  linCon = options.LinearConstr;
25  if ~isequal(linCon.type,'unconstrained')
26      xoverKids = corrigelim(xoverKids, linCon.lb', linCon.ub');
27  end
```

E.5. mimutacion (mutación)

```
1   function mutationChildren = mimutacion(parents,options,GenomeLength, ...
2       FitnessFcn,state,thisScore,thisPopulation,varargin)
3   %MIMUTACION Operador de mutación que aplica una busqueda local para mejorar.
4   %  MUTATIONCHILDREN = MIMUTACION(PARENTS,OPTIONS,GENOMELENGTH,...
5   %  FITNESSFCN,STATE,THISSCORE,THISPOPULATION, VARARGIN) Crea los hijos mutados
6   %  usando una mutacion adaptativa y realizando una busqueda local
7   %
8   %   Ejemplo:
9   %      options = gaoptimset('MutationFcn',{@mimutacion});
10  %
11  %   Esto especifica que la funcion de mutacion a usar sera MIMUTACION
12  %
13  %   Copyright 2008 Jose Ceferino Ortega
14  %   $Revision: 1.0.0.0 $  $Date: 2008/07/23 13:42:42 $
```

```
15
16  % Guardamos las opciones para mejorar el rendimiento
17  persistent opciones
18  if isempty(opciones)
19      opciones = optimset('Algorithm', 'interior-point', 'MaxIter', 25,...
20          'Display', 'off');
21  end
22
23  %% Llamamos a la funcion que genera la mutación original
24  mutationChildren = mutationadaptfeasible(parents,options,GenomeLength, ...
25      FitnessFcn,state,thisScore,thisPopulation,varargin);
26
27  %% Tratamos el caso
28  if (mod(state.Generation, options.EliteCount*2) == 0)
29      range = options.PopInitRange;
30      minimo = range(1,:);
31      maximo = range(2,:);
32
33      % Definimos la estructura a usar para el problema de optimización
34      problema=struct('objective', FitnessFcn, 'x0', [], 'Aineq', [],...
35          'bineq', [], 'Aeq', [], 'beq', [], 'lb', minimo, 'ub', maximo, ...
36          'nonlcon', [], 'solver', 'fmincon', 'options', opciones);
37
38      % Iteramos hasta completar el n. de elementos
39      % Pero solo mejoramos EliteCount de ellos (el resto no)
40      tamano=size(mutationChildren,1);
41      cada=round(tamano/options.EliteCount);
42      for l=1:cada:tamano
43          problema.x0 = mutationChildren(l,:);
44          t=fmincon(problema);
45          if (isfinite(t))
46              mutationChildren(l,:)=t;
47          end
48      end
49  end
50  end
```