UNGenético: Una librería en C++ de Algoritmos Genéticos con Codificación Híbrida

Prof. Óscar Germán Duarte Velasco M.Sc., Ph.D.

21 de julio de 2001

Prólogo

El área trabajo de la *computación flexible* al interior de la Facultad de Ingeniería de la Universidad Nacional de Colombia ha venido consolidándose en los últimos años. Un grupo cada vez más amplio de profesores ha abordado la exploración de estrategias como las Redes Neuronales, los Algoritmos Genéticos y la Lógica Difusa, tanto a nivel de investigación como de aplicación y de desarrollo de herramientas.

Precisamente en esta última línea de trabajo, el desarrollo de herramientas, la facultad ha aportado a la comunidad académica nacional e internacional importantes productos, entre los que vale la pena destacar *UNNeuro*, *UNFUZZY*, y *SISC* entre otros.

UNNeuro fue desarrollado como trabajo de grado de los hoy profesores Jonathan Gómez y Fabio González, bajo la dirección del profesor Gustavo Pérez en 1993, y se constituyó en la principal herramienta de trabajo en las universidades colombianas para el desarrollo de aplicaciones basadas en Redes Neuronales. Una nueva versión de este programa está próxima a aparecer públicamente.

La primera versión de *UNFuzzy* nació como mi proyecto de tesis en la maestría de Automatización Industrial en 1997, también con la dirección de Gustavo Pérez. La acogida que ha tenido esta herramienta a nivel nacional e internacional ha sido bastante satisfactoria: se conocen multiples aplicaciones de lógica difusa basadas en *UNFUZZY* desarrolladas no sólo en Colombia, sino también en México, España, Cuba, Italia, Argentina, Brasil, Ecuador, Bolivia, Perú, Estados Unidos, Canadá, Bélgica y otros paises.

Precisamente esta gran acogida dio origen a la idea de desarrollar una herramienta más que completara el trio de estrategias de computación flexible: *UNGenético* es una librería de clases en lenguaje C++ para la implementación de los Algoritmos Genéticos. Este texto constituye la documentación básica para poder entender y emplear la librería.

*UNGenético*contempla una opción novedosa respecto a las versiones tradicionales de los Algoritmos Genéticos, consistente en la posibilidad de tener una codificación híbrida de la información; ésta permite manejar simultaneamente, y en un mismo *genoma*, genes de distintos tipos: Reales, Booleanos, Enteros o cualquier otro tipo que defina el usuario (ver 1.3).

La tendencia en investigación a nivel internacional indica que los esfuerzos deben ahora encaminarse hacia el desarrollo de sistemas que combinen dos o más de las técnicas básicas de computación flexible. En ese sentido ya existe un primer precedente, ya que *SISC* (Sistema Integrado de Supervisión y Control, herramienta desarrollada por un gruop amplio de colaboradores bajo la dirección de Gustavo Pérez) permite definir sistemas complejos que incluyan Redes Neuronales y Lógica Difusa. Continuando con esa línea, en este documento se incluyen dos ejemplos sobre cómo pueden emlpearse los Algoritmos

Genéticos para optimizar Sistemas de Lógica Difusa.

Debe mencionarse también que los trabajos en computación flexible desarrollados en la Facultad no se restringe a los nombres que han sido citados en estos párrafos. También deben citarse los trabajos de los profesores Alberto Delgado, José J. Martinez, Luis Fernando Niño y Germán Hernández, entre otros.

Este texto está organizado de la siguiente manera: el capítulo 1 presenta una breve introducción a los Algoritmos Genéticos, cuyo propósito es el de fijar los términos que se emplearán en el resto del documtno. El capítulo 2 muestra en detalle la estructura de clases incluida en *UNGenético*, mientras que el capítulo 3 explica sus fundamentos de utilización mediante unos ejemplos elementales. El capítulo 4 se ha reservado para unas aplicaciones más sofisticadas que combinan Lógica Difusa con Algoritmos Genéticos. El capítulo 5 plantea una serie de trabajos futuros que pueden desarrollarse a partir de los resultados que se obtengan con esta primera versión de *UNGenético*.

En los apéndices se ha organizado el código fuente relacionado con este trabajo: El apéndice A contiene el código fuente completo de *UNGenético*; los apéndices B y C contienen el código fuente de los ejemplos de los capítulos 3 y 4, respectivamente, mientras que el apéndice D muestra el código fuente de la clase genérica *Arreglo*, empleada intensamente en la librería.

El Autor.

Índice General

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción a los Algoritmos Genéticos

Al enfrentar la tarea de hallar la solución a un determinado problema, uno de los pasos fundamentales consiste en la elección de la herramienta adecuada para abordar dicha tarea; esta elección está obviamente condicionada por el tipo de problema a solucionar y por su complejidad.

En múltiples ocasiones podemos emplear un procedimiento explícito para hallar la solución *exacta* del problema, mediante una expresión matemática directa, la interpretación de alguna gráfica, la ejecución de un algoritmo iterativo o cualquier otra estrategia.

No obstante, existen problemas en los que no es viable obtener una solución exacta, bien porque no se conoce algún método que provea esa solución, o bien porque conociéndolo éste resulta ser muy complejo de implementar o muy lento en su ejecución.

Las *heurísticas* (véase [4]) son estrategias de búsqueda de soluciones particularmente útiles en estos casos. La idea fundamental de una heurística es la de *buscar* "buenas" soluciones al problema, aunque no se asegure que se encuentre su solución "exacta" [1]. Los Algoritmos Genéticos (AG) son un caso particular de estas técnicas heurísticas.

La razón por la cual los Algoritmos Genéticos tienen ese nombre, es porque se trata de estrategias que fueron concebidas como una imitación simplificada de la mecánica que ha gobernado los procesos genéticos de selección natural que se dan en la naturaleza. Por tanto, ha tomado algunos términos propios de las ciencias biológicas para describir los componentes del algoritmo (se habla de *genes*, *mutación*, etc.). No hay que perder de vista, sin embargo, que los AG no pretenden modelar el proceso biológico, tan sólo se trata de una estrategía heurística de búsqueda de soluciones inspirada en dicho proceso.

Este capítulo resume las ideas básicas de los AG en la sección 1.1; una presentación más extensa puede encontrarse en [10], citelagunaIntroGeneticos y [17]. En la sección 1.2 se destaca la importancia crucial que tiene para los AG el problema de codificación de la información, mientras que en la sección 1.3 se propone una nueva estrategia de

[1] Curiosamente, aunque estas técnicas se encargan de *buscar*, el término *heurístico* proviene del griego *heuriskien*, que significa *encontrar*

codificación que es implementada en la librería *UNGenético* (ver capítulo 2).

1.1 El algoritmo básico

El objetivo de un Algoritmo Genético es la optimización de una *Función de Evaluación* (también denominada *"fitness"*), sujeta a ciertas restricciones. Para ello, las variables de la función se codifican en *genes*; una colección de genes, que corresponde a un punto en el espacio de entrada de la función de evaluación, es un *individuo*. En un AG se produce una colección de individuos (una *población*) que a través de un cierto ciclo de iteraciones se va mejorando. La población de un determinado ciclo es la *generación* correspondiente a ese ciclo.

La estragetia básica de los AG se resume en el siguiente algoritmo ($P(t)$ es la población en la generación t):

1. $t = 0$
2. Crear $P(t)$
3. Hacer hasta que se cumpla la condición de parada
   - **Evaluar** la $P(t)$
   - **Seleccionar** $P(t+1)$ a partir de $P(t)$
   - **Recombinar** $P(t+1)$
   - $t = t + 1$
4. Fin del ciclo

La creación de $P(0)$ generalmente se hace de forma aleatoria. La **Evaluación** de $P(t)$ consiste en calcular el valor de la Función de Evaluación para cada individuo de la población.

El proceso de **Selección** consta de dos etapas:

1. Asignar una probabilidad de supervivencia a cada individuo basada en la función de evaluación, para ello se emplea un **Operador de Probabilidad**
2. Ejecutar algún algoritmo de muestreo (de selección) para decidir cuántas copias de cada individuo se crean en la siguiente generación; se emplea un **Operador de Selección**

El proceso de **Recombinación** consta de dos etapas:

1. El proceso de **Cruce**: Es la generación de nuevos individuos (hijos) a partir de individuos de la actual generación (padres). Para efectuar este proceso se necesita:
   - Asignar parejas con un **Operador de Asignación de Parejas**.
   - Decidir si cada pareja se cruza y cuántos individuos genera con un **Operador de Reproducción**.

Tabla 1.1: Operadores de los AG

Tipo de Operador	**observaciones**
Operador de probabilidad	no depende de la codificación
Operador de seleccion	no depende de la codificación
Operador de asignación de parejas	no depende de la codificación
Operador de reproduccion	no depende de la codificación
Operador de cruce	si depende de la codificación
Operador de mutación	si depende de la codificación
Operador de adaptación	es opcional y depende de los otros operadores seleccionados

- Cruzar los individuos con un **Operador de Cruce**

2. El proceso de Mutación: Es la alteración aleatoria de alguna de las características (genes) de un individuo mediante un **Operador de Mutación**.

Adicionalmente al ciclo básico, existe la posibilidad de adaptar el algoritmo genético modificando alguna de sus características conforme avanza el algoritmo. Para esto se requiere algún **Operador de adaptación**. Es usual seguir la estrategia de **Elitismo**, según la cual el mejor individuo de una generación siempre pasa a la siguiente.

De acuerdo a lo anterior, un algoritmo genético requiere los operadores que se resumen en la tabla 1.1. Para cada tipo de operador existe una amplia gama de posibilidades en la literatura, lo cual de origen a mútliples versiones del algoritmo básico. Algunos operadores dependen del tipo de codificación genética empleada (ver sección 1.2); a continuación se muestran algunos de los operadores que no dependen del tipo de codificación.

El desempeño de un AG suele analizarse siguiendo la función de evaluación del mejor de los individuos de cada generación; sin embargo, también pueden seguirse la del peor individuo, el promedio de los funciones de evaluación de todos los individus y/o su desviación estándar. Existen también otras medidas de desempeñocomo las siguientes:

Medida Online: Promedio de las funciones de evaluación de todos los individuos que han aparecido hasta ese momento en todas las generaciones.

Medida Offline: Promedio de las funciones de evaluación de los mejores individuos de cada generación hasta ese momento.

1.1.1 Operadores de Probabilidad

El operador de probabilidad asigna una *probabilidad de supervivencia* $p(i$ a los N individuos de una cierta generación. Un individuo con una buena función de evaluación $f(i)$ deberá tener una mayor probabilidad de supervivencia que otro con una peor función de evaluación. Estos son algunos de los operadores más usuales:

Ranking Lineal: Se ordena la población de mejor individuo a peor. La probabilidad de supervivencia depende de su posición i según la siguiente ecuación (n_{min} es un

parámetro seleccionable por el usuario):

$$n_{max} = 2 - n_{min}$$

$$p(i) = (n_{max} - (n_{max} - n_{min}) * ((i)/(N-1)))/N$$

Proporcional: La probabilidad de supervivencia depende de la función de evaluación (mejor función significa mayor probabilidad).

Para maximizar: $p(i) = \frac{f(i)}{\sum_{j=1}^{N} f(j)}$

Para minimizar: $p(i) = 1.0 - \frac{f(i)}{\sum_{j=1}^{N} f(j)}$

Homogénea: A todos los individuos se les asigna la misma probabilidad de supervivencia $\frac{1.0}{N}$

1.1.2 Operadores de Selección

El Operador de Selección establece cuántas copias se crearán de cada individuo en la siguiente generación. Suele establecerse que el número de individuos en todas las generaciones se mantenga constante. Estos son algunos de los operadores más usuales:

Estocástica con Remplazo: Se crea un "ruleta"con tantas casillas como individuos tenga la población; el ángulo de cada casilla es proporcional a la probabilidad de supervivencia. Se juega a la ruleta tantas veces como individuos tenga la población y se crea una copia de cada individuo ganador.

Estocástica Universal: se crea una ruleta como en el caso anterior, pero sólo se juega una vez; las casillas ganadoras están igualmente espaciadas, y hay tantas como individuos tenga la población.

1.1.3 Operadores de Asignación de Parejas

El Operador de Asignación de Parejas decide qué individuos se cruzan entre sí pra generar nuevos individuos. Estos son algunos de los operadores más usuales:

Aleatoria: se buscan aleatoriamente las parejas.

Siguiente: La pareja de cada individuo es la siguiente en el arreglo de individuos que forma la población.

Extremos: Al primer individuo le corresponde el último, al segundo el penúltimo, etc.

1.1.4 Operadores de Reproducción

El Operador de Reproducción establece una estrategia general de reproducción entre parejas que se cruzan. Estos son algunos de los operadores más usuales:

Dos padres dos hijos: por cada pareja se crean dos hijos que remplazan a sus padres.

Mejor padre-mejor hijo: por cada pareja se crean dos hijos, pero sólo el mejor hijo remplaza al peor padre

Mejores entre padres e hijos: por cada pareja se crean dos hijos; de los cuatro se mantienen los dos mejores.

1.2 Codificación del problema

Tal como se comentó en la seccion 1.1, para optimizar una función mediante AG es necesario codificar las variable de dicha función en *genes*. No existe una única forma de codificar dichas variables; en esta sección se muestran dos de las estrategias más comunes para ese proceso, la *codificación binaria* y la *codificación real*, si bien en la literatura se encuentrabn aplicaciones con codificación entera, en listas y arreglos, etc.

La elección de uno u otro tipo de codificación puede afectar el desempeño de un AG, principalmente por que ([16]): "1) los AG manipulan una representación codificada del problema y 2) la representación puede limitar en gran medida la visión que tiene un sistema sobre su entorno". Por otra parte, ciertas codificaciones pueden acomodarse más fácilmente para representar ciertos tipos de variables, y por tanto pueden resultar en una implementación más rápida y efectiva del AG. Es necesario resltar que los operadores de cruce y mutación dependen del tipo de codificación empleada.

1.2.1 Codificación Binaria

Históricamente, esta fue la primera estrategia de representación empleada. Se busca representar el espacio de búsqueda $E = E_1 \times E_2 \times \ldots \times E_n$ mediante un alfabeto binario. Para ello, se codifica cada variable del espacio de búsqueda mediante una función $cod_i : E_i \rightarrow \{0,1\}^{L_i}$ con $L_i \in N$ y $i = 1, 2, \ldots, n$; es decir, cada variable del espacio de búsqueda, E_i se representa mediante una cadena de unos y ceros de longitud L_i.

La representación de un elemento cualquiera $x = (x_1, x_2, \ldots, x_n) \in E$, tal que $x_i \in E_i$ se lleva a cabo mediante la concatenación de las n cadenas $cod(x) = cod_1(x_1)cod_2(x_2)\ldots cod_n(x_n)$. Esta nueva cadena de L casillas se denomina el *genoma*[2], y cada una de las casillas contiene un *gen*, que será un uno o un cero.

Cruce Binario

El operador de cruce es el operador que permite que dos individuos compartan su información genética, para producir nuevos individuos. Estos son algunos de los operadores de cruce con codificación binaria más usuales:

[2] algunos autores emplean el término *cromosoma*

Cruce simple: Dados dos individuos (*padres*) con genomas $P_1 = (c_1^1, \ldots, c_L^1)$ y $P_2 = (c_1^2, \ldots, c_L^2)$, respectivamente, se generan dos nuevos individuos (*hijos*) con genomas $H_1 = (c_1^1, \ldots, c_i^1, c_{i+1}^2, \ldots, c_L^2)$ y $H_2 = (c_1^2, \ldots, c_i^2, c_{i+1}^1, \ldots, c_L^1)$, donde i (denominado el *punto de cruce*)es un número entero aleatorio del intervalo $[1, L-1]$.

Cruce doble: Similar al anterior, pero con dos puntos de cruce aleatorios i, j. Los genomas de los hijos serán $H_1 = (c_1^1, \ldots, c_i^1, c_{i+1}^2, \ldots, c_j^2, c_{j+1}^1, \ldots, c_L^1)$ y $H_2 = (c_1^2, \ldots, c_i^2, c_{i+1}^1, \ldots, c_j^1, c_{j+1}^2, \ldots, c_L^2)$.

Mutación Binaria

El operador de mutación efectúa alteraciones aleatorias en la información genética. Con esta función se busca explorar nuevas zonas del espacio de búsqueda, y asi intentar escapar de posibles óptimos locales en los que pueda quedar atrapado el algoritmo. Estos son algunos de los operadores de mutación con codificación binaria más usuales:

Uniforme: Para cada se genera un número aleatorio $m \in [0, 1]$. Si $m \leq p_m$ se altera el valor del gen (si es un 1 cambia a 0, y viceversa). p_m es un número del intervalo $[0, 1$ denominado *probabilidad de mutación*, seleccionado por el usuario y constante en toda la ejecución del algoritmo.

Np Uniforme: Similar al anterior, pero la probabilidad de mutación disminuye conforme avanza la ejecución del algoritmo, usualmente siguiendo una curva exponencial $p_m = p_{max} e^{-b\frac{t}{T}}$, donde t es el número de la generación, T el número máximo de generaciones, y p_{max}, b dos parámetros seleccionados por el usuario ($p_{max} \in [0, 1]$ y $b > 0$).

1.2.2 Codificación Real

En la codificación real se busca representar el espacio de búsqueda $E = E_1 \times E_2 \times \ldots \times E_n$ mediante un arreglo de números reales, mediante una función $cod : E \to R^n$. Es usual emplear n funciones de codificación, una para cada una de las variables del espacio de búsqueda: $cod_i : E_i \to R$; en estos casos la representación de un elemento cualquiera $x = (x_1, x_2, \ldots, x_n) \in E$, tal que $x_i in E_i$ se lleva a cabo mediante la concatenación de los n reales $cod(x) = [cod_1(x_1) cod_2(x_2) \ldots cod_n(x_n)]$. Este arreglo de n elementos es el *genoma*, y cada uno de los elementos del arreglo es un *gen*, que será un número real.

Cruce Real

La variedad de operadores de cruce para codificación real es mucho mayor que para la codificación binaria. Algunos de los operadores más empleados se han listado a continuación. En todos los casos se ha supuesto que los dos individuos $C_1 = \{c_1^1, c_2^1, \ldots, c_n^1\}$ y $C_2 = \{c_1^2, c_2^2, \ldots, c_n^2\}$ se desean cruzar para generar m hijos $H_k = \{h_1^k, h_2^k, \ldots, h_n^k\}$ con $k = 1, 2, ...m$. Cada gen h_i^k debe pertenecer al intervalo $[a_i, b_i]$:

Plano: Se genera un hijo ($m = 1$). h_i^k se genera aleatoriamente en el intervalo $[c_i^1, c_i^2]$.

Cruce simple: Una posición j se escoge aleatoriamente ($j < n-1$). Se generan dos hijos así:

$$H_1 = \{c_1^1, c_2^1, \ldots, c_j^1, c_{j+1}^2, \ldots, c_n^2\}$$
$$H_1 = \{c_1^2, c_2^1, \ldots, c_j^2, c_{j+1}^1, \ldots, c_n^1\}$$

Cruce Aritmético: Se generan dos hijos ($\lambda \in [0,1]$ es un parámetro seleccionable por el usuario); cada gen se obtiene así:

$$h_i^1 = \lambda c_i^1 + (1-\lambda)c_i^2$$
$$h_i^2 = \lambda c_i^2 + (1-\lambda)c_i^1$$

Cruce $BLX-\alpha$: Se generan dos hijos ($\alpha \in [0,1]$ es un parámetro seleccionable por el usuario); cada gen se obtiene aleatoriamente del intervalo $[c_{min} - I\alpha, c_{max} + I\alpha]$ en donde $c_{min} = \min(c_i^1, c_i^2)$, $c_{max} = \max(c_i^1, c_i^2)$, $I = c_{max} - c_{min}$. El cruce correspondiente a $\alpha = 0$ es igual al cruce plano.

Cruce Lineal: Se generan tres hijos, de los cuales se seleccionan sólo los dos con mejor función de evaluación. Cada gen de los tres hijos se genera así:

$$h_i^1 = \frac{1}{2}c_i^1 + \frac{1}{2}c_i^2$$
$$h_i^1 = \frac{3}{2}c_i^1 - \frac{1}{2}c_i^2$$
$$h_i^1 = -\frac{1}{2}c_i^1 + \frac{3}{2}c_i^2$$

Cruce Discreto: Cada gen h_i se escoge aleatoriamente entre c_i^1 y c_i^2.

Cruce Línea extendida: $h_i = c_i^1 + \alpha(c_i^2 - c_i^1)$. α se escoge aleatoriamente en el intervalo $[-0.25, 1.25]$.

Cruce Extendido intermedio: $h_i = c_i^1 + \alpha_i(c_i^2 - c_i^1)$. α_i se escoge aleatoriamente en el intervalo $[-0.25, 1.25]$. Equivale al cruce $BLX - 0.25$.

Cruce Heurístico de Wrigth: Supóngase que C_1 es el padre con la mejor función de evaluación. En esas condiciones $h_i = r(c_i^1 - c_i^2) + c_i^1$ con r un número aleatorio perteneciente al intervalo $[0,1]$.

Cruce BGA Lineal: En las mismas condiciones del cruce de Wrigth, se obtiene cada gen así:

$$h_i = c_i^1 + rango_i \gamma \Delta$$
$$\delta = \frac{c_i^2 - c_i^1}{||C_1 - C_2||}$$
$$\gamma = \sum_{k=0}^{15} \alpha_k 2^{-k}$$

α_k puede ser 0 ó 1, con probabilidad de ser 1: $p(\alpha_i = 1) = \frac{1}{16}$. Usualmente, $rango_i = 0.5(b_i - a_i)$

Mutación Real

Al igual que con los operadores de cruce, existe una mayor variedad de operadores de mutación con codificación real que con codificación binaria. Algunos de los más usados son los siguientes:

Uniforme: Cada gen tiene una probabilidad de mutar. La probabilidad es seleccionable por el usuario. Si un individuo está formado por los genes $c_1, c_2, ...c_n$, y el gen c_i debe pertenecer al intervalo $[a_i, b_i]$, entonces el nuevo gen $c_i\prime$ es un número aleatorio perteneciente al intervalo $[a_i, b_i]$

No Uniforme: al aplicar este operador en la generación t, con el número máximo de generaciones es g_{max},el nuevo gen $c_i\prime$ es

$$c_i\prime = \begin{cases} c_i + \Delta(t, b_i - c_i) & \text{si } \tau > 0 \\ c_i - \Delta(t, b_i - c_i) & \text{si } \tau \leq 0 \end{cases}$$

$$\Delta(t, y) = y\left(1 - r^{(1-\frac{t}{g_{max}})^b}\right)$$

b es un parámetro seleccionable por el usuario; r es un número aleatorio en el intervalo $[0, 1]$; τ es un número aleatorio que puede valer 0 ó 1.

Mutación de Muhlenbein:

$$c_i\prime = c_i \pm \gamma rango_i$$
$$\gamma = \sum_{k=0}^{15} \alpha_k 2^{-k}$$

$rango_i$ define el rango de la mutación, y usualmente se emplea como $0.1(b_i - a_i)$. El sigmo + ó − se escoge aleatoriamente con igual probabilidad, y α_k puede ser 0 ó 1, con probabilidad de ser 1: $p(\alpha_k = 1) = \frac{1}{16}$.

1.3 Codificación Híbrida

Al analizar los distintos operadores de las codificaciones real y binaria, puede verse que quizás la primera de ellas tiene algunas propiedades en las que supera a la segunda; por ejemplo la posibilidad de controlar más facilmente la relación *Exploración-Explotación* del algoritmo, gracias a la existencia de operadores parametrizables como el operador de cruce $BLX - \alpha$.

No obstante, las comparaciones que se hacen sobre las distintas estrategias de codificación demuestran que no puede afirmarse que una de ellas se superior a las otras en todos los casos. Por el contrario, lo único que puede concluirse es que la elección de la estrategia de codificación más adecuada depende del tipo de problema a optimizar. Tómese por ejemplo el caso en el que una de las variables del espacio de búsqueda sea una variable booleana; no parece adecuado representar esta variable mediante un número

real. También es sabido que la codificación binaria suele presentar inconvenientes cuando los espacios de búsqueda son contínuos.

El problema se complica un poco más si consideramos que en un gran número de aplicaciones reales el espacio de búsqueda está compuesto por variables heterogéneas: algunas serán reales, otras binarias, otras enteras, etc. A título de ejemplo, supóngase que se desea optimizar la función $f(x_1, x_2, x_3)$

$$f(x_1, x_2, x_3) : R \times \{0, 1\} \times Z \rightarrow R$$

es decir, una función en la que la primera variable es real, la segunda es booleana y la tercera es entera. ¿cuál puede ser la estrategia de codificación más adecuada en este caso?

Justamente para enfrentar ese tipo de problemas, se propone aqui emplear una *Codificación Híbrida*, en la que en un mismo genoma coexistan genes de distinto tipo. Retomando el ejemplo anterior, se propone emplear un genoma de tres genes: el primero con codificación real, el segundo con codificación binaria y el tercero con codificación entera.

Es claro que el problema anterior puede intentar solucionarse empleando un único tipo de codificación (por ejemplo codificación binaria) para todas las variables. Si esto es así, ¿cuál es la ganancia de este enfoque? La respuesta está en la posibilidad de tener una *representación más natural* del problema, y asi obteber una implementación más sencilla y eficiente del algoritmo.

Debido a que los operadores de cruce y mutación dependen del tipo de codificación empleada, al tener una codificación híbrida será necesario definir distintos operadores de cruce y mutación para cada uno de los distintos tipos de genes presentes en el genoma.

Esta propuesta ha sido implementada en una librería de clases escrita en lenguage C++, que se detalla en el capítulo 2, y que se reproduce completa en el apéndice A. La librería ha sido denominada *UNGenético*.

Capítulo 2

Librería de clases C++

En este capítulo se presenta en forma detallada la implementación de la Librería de clases escrita en lenguaje C++ con enfoque orientado a objetos, que ha sido diseñada para emplear algoritmos genéticos con codificación híbrida. La tabla 2.1 presenta el nombre de los archivos que componen la librería y su contenido sintetizado. El Apéndice A contiene el código completo de estos archivos.

En la sección 2.1 se da una primera visión de las distintas clases, que permite comprender la forma en que éstas se relacionan entre sí. La sección 2.2 presenta las clases Principales, es decir, aquellas con las que se ha implementado el Algoritmo Genético con Codificación Híbrida. La mayoría de estas clases "principales" son virtuales; las clases heredadas de ellas se presentan en las secciones 2.3 y 2.4.

2.1 Jerarquía de clases

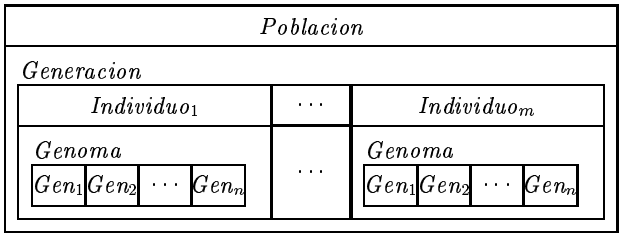
En la librería *UNGenético* se han definido varias clases que encapsulan el comportamiento de los diversos componentes de un Algoritmo Genético; de ellas, las principales son:

- *Gen*

Tabla 2.1: Archivos de la librería

Archivos	**Contenido**
genetico.hpp, genetico.cpp	Definiciones básicas del algoritmo genético
genbool.hpp, genbool.cpp	Definiciones de los genes codificados como Booleanos (*bool*) y sus operadores
genint.hpp, genint.cpp	Definiciones de los genes codificados como Enteros (*long*) y sus operadores
genreal.hpp, genreal.cpp	Definiciones de los genes codificados como Reales (*double*) y sus operadores

Figura 2.1: La Clase *Poblacion*



- *Individuo*
- *Poblacion*
- *AlgoritmoGenetico*
- *Operador Mutacion*
- *OperadorCruce*
- *OperadorSeleccion*
- *OperadorProbabilidad*
- *OperadorParejas*
- *OperadorReproduccion.*

La figura 2.1 muestra la forma en que se relacionan las clases *Gen*, *Individuo*, *Poblacion*:

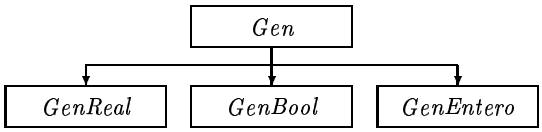
Un objeto de la clase *Poblacion* contiene un arreglo de objetos de la clase *Individuo* denominado *Generacion* (Una población es un conjunto de individuos); Cada objeto de la clase *Individuo* contiene un arreglo de objetos de la clase *Gen* denominado *Genoma.*

La clase *Gen* es una clase abstracta [1]. Cada *Gen* en el *Genoma* puede ser un objeto de cualquiera de las clases herederas de *Gen* que se observan en la figura 2.2: *GenBool*, *GenEntero*, *GenReal.* A cada una de estas clases herederas de *Gen* le corresponden dos grupos de clases herederas de las clases *OperadorCruce* y *OperadorMutacion.*

Cada elemento del *Genoma*, es decir cada objeto de la clase *Gen*, contiene una parte de la información que define al *Individuo* codificada en una forma específica. Esta estructura es la base de la Codificación Híbrida. Debido a que cada gen puede ser de un tipo diferente, es indispensable que cada uno de ellos tenga asociado un operador de cruce y un operador de mutación; estos operadores forman parte de la clase *AlgoritmoGenetico.* En un algoritmo genético sin Codificación Híbrida, es decir en uno en el que todos los

[1] Una clase abstracta tiene una o más funciones (procedimientos) declarados como virtuales puros; estas funciones deben necesariamente ser redefinidas por las clases herederas para que éstas no sean abstractas. No pueden crearse instancias de clases abstractas

Figura 2.2: Jerarquía de la Clase *Gen*



genes son del mismo tipo, bastaría con un único operador de mutación y un único operador de cruce para todos los genes.

La clase *Individuo* también es una clase abstracta. Para una aplicación específica del algoritmo genético es necesario crear una clase heredera de *Individuo* (ver capítulo 3). Estas clases tomarán toda la funcionalidad de la clase padre *Individuo*, y se les adicionará la funcionalidad del problema específico a optimizar [2].

Por otra parte, la figura 2.4 muestra la forma en que se relacionan las clases *Poblacion*, *AlgoritmoGenetico*, *OperadorProbabilidad*, *OperadorSeleccion*, *OperadorParejas*, *OperadorReproduccion*, *OperadorMutacion* y *OperadorCruce* según se explica a continuación:

Un objeto de la clase *AlgoritmoGenetico* contiene un objeto de la clase *Poblacion* denominado *GeneracionActual*, y cuatro objetos *OpProbabilidad*, *OpSeleccion*, *OpParejas* y *OpReproduccion*, de cada una de las clases *Operador Probabilidad*, *OperadorSeleccion*, *OperadorParejas* y *Operador Reproduccion*, respectivamente . Estas últimas cuatro clases son virtuales, siendo sus herederas las clases que se muestran en la figura 2.3.

La clase *AlgoritmoGenetico* es la encargada de efectuar las operaciones del algoritmo sobre la *GeneracionActual* (mutación, selección, cruce etc.), empleando para ello cuatro operadores:

OpProbabilidad es un objeto de la clase *OperadorProbabilidad* que se encarga de asignar la probabilidad de supervivencia de cada *Individuo* de la *GeneracionActual*.

OpSeleccion es un objeto de la clase *OperadorSeleccion* que se encarga de efectuar el proceso de Selección sobre la *GeneracionActual*.

OpParejas es un objeto de la clase *OperadorParejas* que se encarga de asignar las parejas que serán cruzadas para dar origen a una nueva *GeneracionActual*.

OpReproduccion es un objeto de la clase *OperadorReproduccion* que se encarga de definir la estrategia general de reproducción.

Adicionalmente, la clase *AlgoritmoGenetico* contiene un arreglo de operadores de mutación denominado *ListaOperadorMutacion* y uno de operadores de cruce denominado *ListaOperadorCruce*. El tamaño de estos arreglos debe coincidir con el número de genes en el genoma; cada operador debe ser del tipo adecuado para el gen correspondiente: si en la posición i del el genoma hay un gen real, entonces en la posición i de los arreglos de operadores de mutación y cruce deben haber unos que operen sobre genes reales. La clase *Individuo* tiene una función para llenar estos arreglos.

[2]En los ejemplos del capítulo 4 se busca optimizar unos Sistemas de Lógica Difusa, y por tanto la clase Heredada de *Individuo* adiciona la funcionalidad propia de estos sistemas

Figura 2.3: Jerarquía de las Clases de Operadores del Algoritmo Genético

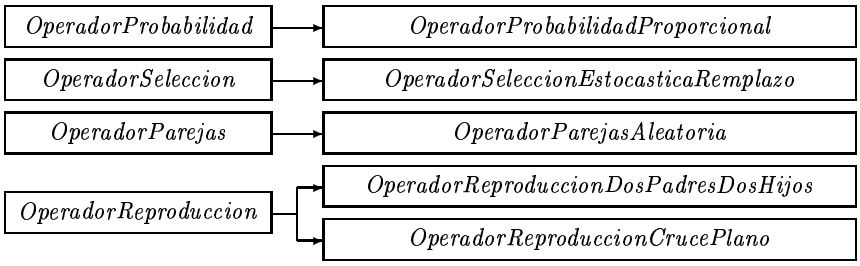
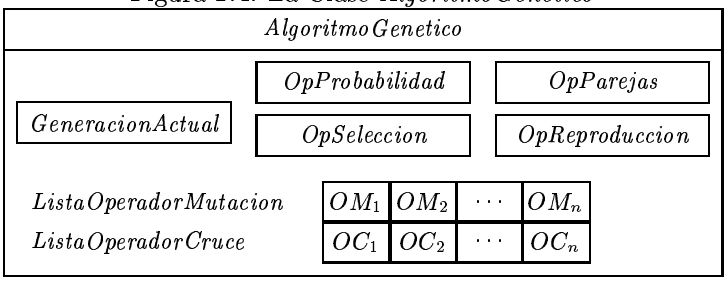


Figura 2.4: La Clase *AlgoritmoGenetico*



2.2 Clases principales

2.2.1 Clase *Gen*

Gen es una clase abstracta cuyo propósito principal es el de almacenar la información genética. Debido a que ésta puede estar codificada en diversas formas, son las clases herederas de *Gen* (ver sección 2.3) las que definen la propiedad en la que se almacena la información genética; por ejemplo, la clase *GenReal*, que es heredera de *Gen* define la variable *double Valor* en la que se consigna la información genética con codificación real, mientras que la clase *GenEntero*, tambien heredera de *Gen* define la variable *long Valor* en la que se consigna la información genética con codificación entera.

Por cada clase que se defina como heredera de *Gen* deben definirse al menos dos clases más, una de ellas heredera de la clase *OperadorMutacion* y otra heredera de la clase *OperadorCruce*. Por ejemplo, al definir la clase *GenBool* se han definido también las clases *OperadorMutacionBoolUniforme* (heredera de *OperadorMutacion*) y *OperadorCruceBoolPlano* (heredera de *OperadorCruce*)

La tabla 2.2 contiene las propiedades (variables) de la clase *Gen*. Los procedimientos se explican a continuación:

- *void ag(AlgoritmoGenetico *Ag)*:

  Modificador de AG. Modifica también la propiedad AG de OpMutacion y OpCruce.

Tabla 2.2: Propiedades de la clase *Gen*

Propiedad	**Descripción**
*AlgoritmoGenetico *AG*	Apuntador al Algoritmo Genético

Tabla 2.3: Propiedades de la clase *OperadorMutacion*

Propiedad	**Descripción**
*AlgoritmoGenetico *AG*	Apuntador al Algoritmo Genético
double ProbabilidadMutacion	Especifica la probabilidad de mutar

- *virtual void crearAleatorio()=0*:

  Función virtual para generar un gen de valor aleatorio.

- *virtual OperadorMutacion operadorMutacionDefecto()=0*:

  Función virtual para asignar un operador de mutación por defecto.

- *virtual OperadorCruce operadorCruceDefecto()=0*:

  Función virtual para asignar un operador de cruce por defecto.

2.2.2 Clase *OperadorMutacion*

OperadorMutacion es una clase abstracta cuyo propósito principal es el de efectuar una mutación sobre un objeto de la clase *Gen*. En general, cada clase heredera de *OperadorMutacion* corresponde a una clase heredera de *Gen*.

La tabla 2.3 contiene las propiedades (variables) de la clase *OperadorMutacion*. Los procedimientos se explican a continuación:

- *void mutar(Gen *g)*:

  Produce un valor aleatorio en el intervalo $[0, 1]$. Si este valor es inferior a *ProbabilidadMutacion*, entonces se ejecuta *mutarGen(g)*.

- *virtual void mutarGen(Gen *g)=0*:

  Función virtual para definir el procedimiento de mutación.

2.2.3 Clase *OperadorCruce*

OperadorCruce es una clase abstracta cuyo propósito principal es el de efectuar un cruce entre dos objetos de una clase heredera de la clase *Gen*, para producir un número determinado de hijos de la misma clase. En general, cada clase heredera de *OperadorCruce* corresponde a una clase heredera de *Gen*.

Tabla 2.4: Propiedades de la clase *OperadorCruce*

Propiedad	**Descripción**
*AlgoritmoGenetico *AG*	Apuntador al Algoritmo Genético

La tabla 2.4 contiene las propiedades (variables) de la clase *OperadorCruce*. Los procedimientos se explican a continuación:

- *void cruzar(Gen *madre,Gen *padre, Arreglo<Gen> *hijos, int numHijos)*:

  Ejecuta la función *cruzarGenes(madre,padre,hijos,numhijos)*, que produce *numhijos* objetos de la misma clase que *madre* y *padre* y los almacena en *hijos*. Asigna a cada uno de los objetos almacenados en *hijos* el valor de la propiedad *AG*.

- *virtual void cruzarGenes(Gen *madre, Gen *padre, Arreglo<Gen> *hijos, int num Hijos)=0*:

  Función virtual para definir el procedimiento de cruce.

2.2.4 Clase *Individuo*

Individuo es una clase abstracta cuyo propósito principal es el de encapsular el comportamiento de los miembros de una Población del Algor¡tmo Genético. La propiedad más importante de un objeto de la clase *Individuo* es el arreglo de genes *Arreglo<Gen> Genoma*[3], en donde se almacena la información genética codificada del objeto.

El objetivo del algoritmo genético es el de encontrar uno (o más) individuos que optimicen una cierta función. Por ello, para emplear la clase *AlgoritmoGenetico* (ver sección 2.2.6) debe primero definirse una clase heredera de *Individuo*, y crear un objeto de dicha clase que será empleado como modelo para construir todos los objetos de la clase *Individuo* que se necesitan en la ejecución del algoritmo genético.

Las clases herederas de *Individuo* deben definir todas las propiedades y métodos propios del problema a optimizar (ver capítulos 3 y 4). También deben proveer métodos para codificar y decodificar la información geética, así como para calcular la función objetivo.

La tabla 2.5 contiene las propiedades (variables) de la clase *Individuo*. Los procedimientos se explican a continuación:

- *void calcularObjetivo()*:

  Ejecuta los métodos *decodificar* y *objetivo*. El valor retornado por éste último se almacena en *Objetivo*

- *void copiar(Individuo *other)*:

  Copia las propiedades *Objetivo* y *Probabilidad* de *other*, y ejecuta el método *copiarDetalles(other)*

[3]la clase *Arreglo* es la implementación de un contenedor genérico que se explica en el apéndice D

- *virtual Individuo* crearCopia()=0*:

  Función virtual cuyo propósito es el de crear un objeto de la misma clase que el objeto actual (una clase heredera de *Individuo*), retornando el apuntador a *Individuo*

- *virtual void copiarDetalles(Individuo *other)=0*:

  Función virtual cuyo propósito es el de copiar todas las propiedades específicas de la clase heredera

- *virtual double objetivo()=0*:

  Función virtual cuyo propósito es el de calcular la función objetivo para el individuo

- *virtual void codificar()=0*:

  Función virtual cuyo propósito es el de codificar las propiedades específicas del individuo en los genes almacenados en *Genoma*

- *virtual void decodificar()=0*:

  Función virtual cuyo propósito es el de decodificar la información contenida en los genes almacenados en *Genoma*, para conocer las propiedades específicas del individuo.

- *virtual void crearOperadores(Arreglo<OperadorMutacion> *LM, Arreglo<Operador Cruce> *LC)*:

  Esta función llena los arreglos LM y LC con operadores de Mutación y Cruce, respectivamente, coclocando tantos operadores como genes existan en el genoma. Cad operador es del tipo adecuado para manejar el gen respectivo. Esta función asigna los operadores por defecto que proveen las funciones *operadorMutacionDefecto()* y *operadorCruceDefecto* de la clase *Gen*. Esta funciám puede ser redefinida en las clases herederas de *Individuo*, para asignar otros operadores diferentes.

- *void crearAleatorio()*:

  Ejecuta el método *crearAleatorio()* de cada uno de los genes en *Genoma*

- *void mutar()*:

  Ejecuta la mutación de cada uno de los genes en *Genoma*

- *void ag(AlgoritmoGenetico *Ag)*:

  Modificador de *AG*. También ejecuta el método *ag(Ag)* de cada uno de los genes en *Genoma*

2.2.5 Clase *Poblacion*

la clase *Poblacion* consiste escencialmente de un conjunto de individuos con la funcionalidad necesaria para ejecutar varias operaciones sobre ellos. La tabla 2.6 contiene las propiedades (variables) de la clase *Poblacion*. Los procedimientos se explican a continuación:

Tabla 2.5: Propiedades de la clase *Individuo*

Propiedad	**Descripción**
*AlgoritmoGenetico *AG*	Apuntador al Algoritmo Genético
Arreglo<Gen> Genoma	Arreglo de genes que contiene la totalidad de la información genética del objeto
double Objetivo	Almacena el valor de la función objetivo calculado en *objetivo()*
double Probabilidad	Almacena el valor de probabilidad de supervivencia calculado por el operador de probabilidad de *AG*
*Individuo *Pareja*	Apuntador a la pareja con la que se efectuará el cruce. La selección de la pareja la efectúa el operador de parejas de *AG*

- *void crearGeneracion(int tam)*:

  Crea *tam* individuos y los almacena en *Generacion*. Para crear cada individuo emplea el método *crearCopia()* de un objeto de la clase *Individuo* cuyo nombre es *Modelo*

- *void crearGeneracionAleatoria()*:

  Ejecuta el método *crearAleatorio()* para cada uno de los miembros de *Generación*

- *void calcularObjetivo()*:

  Ejecuta el método *calcularObjetivo()* para cada uno de los miembros de *Generación*

- *void ordenar(int descendente=0)*:

  Ordena los miembros de *Generación* según su propiedad *Objetivo*, en orden descendente (si *descendente* es distinto de cero) o ascendente(si *descendente* es igual a cero).

- *void sumaObjetivo()*:

  Efectua la suma de las propiedades *Objetivo* de todos los miembros de *Generación*

- *void mutar()*:

  Ejecuta el método *mutar()* para cada uno de los miembros de *Generación*

- *void buscaElite(int Maximizar)*:

  Determina cuál es el mejor individuo en *Generacion*, y produce una copia de él en *Elite*, empleando lel método *copiar* de *Elite*. Si *Maximizar* es igual a cero, el mejor individuo será aquél couya propiedad *Objetivo* sea mínima; en caso contraria será aquél en que sea máxima.

Tabla 2.6: Propiedades de la clase *Poblacion*

Propiedad	**Descripción**
*AlgoritmoGenetico *AG*	Apuntador al Algoritmo Genético
*Individuo *Modelo*	Apuntador a un objeto de una clase heredera de *Individuo* que se emplea como modelo para producir copias de él.
*Individuo *Elite*	Copia del mejor individuo de la población, es decir, del objeto en *Generación* cuya propiedad *Objetivo* es la mejor
Arreglo<Individuo> Generacion	Arreglo de objetos de una clase heredera de *Individuo*, que constituyen una generación del algoritmo genético
double SumaObjetivo	variable que almacena el resultado de la función *sumaObjetivo()*

- *void elitismo()*:

  Copia *Elite* en el último lugar de *Generacion*

- *void ag(AlgoritmoGenetico *Ag)*:

  Modificador de *AG*. También ejecuta el método *ag(Ag)* de cada uno de los individuos en *Generacion*

2.2.6 Clase *AlgoritmoGenetico*

La clase *AlgoritmoGenetico* encapsula el comportamiento global del algoritmo genético. Para que un objeto de la clase *AlgoritmoGenetico* pueda efectuar el proceso de optimización del algoritmo genético, primero debe definirse una clase heredera de *Individuo*, crear un objeto de esa clase, y asignarlo como *Modelo* del algoritmo (ver capítulos 3 y 4).

Existen dos posibilidades para ejecutar el proceso de optimización (ver tabla 3.1):

- Empleando el método *optimizar()*, que ejecuta todo el proceso.
- Empleando los métodos *iniciarOptimizacion()*, *iterarOptimizacion()*, *parada()* y *finalizarOptimizacion()*, para tener los resultados intermedios en cada una de las iteraciones del proceso.

Con cualquiera de los dos posibilidades anteriores, puede crearse un archivo de texto en el que se almacenen algunos valores representativos de las iteraciones. Puede especificarse el intervalo de iteraciones cuyos valores desean almacenarse. Los valores en cuestión son los siguientes:

- Número de Generación. (cuántas iteraciones han transcurrido)

- Función objetivo del mejor individuo en toda la historia
- Generación en que apareció el mejor individuo en toda la historia
- Función objetivo del mejor individuo en la generación actual
- Función objetivo del peor individuo en la generación actual
- Promedio de las funciones objetivo de los individuos de la generación actual
- Desviación estándar de las funciones objetivo de los individuos de la generación actual
- Valor de la medida OnLine para la generación actual
- Valor de la medida OffLine para la generación actual

Las tablas 2.7 a 2.9 contienen las propiedades (variables) de la clase *AlgoritmoGenetico*. Los procedimientos se explican a continuación:

- *void asignarProbabilidad(int descendente)*:

  Ordena a *OpProbabilidad* que asigne la probabilidad de supervivencia a los individuos de la generación actual

- *void seleccionar()*:

  Ordena a *OpSeleccion* que seleccione los individuos que sobrevivirán en la generación actual

- *void asignarParejas()*:

  Ordena a *OpParejas* que asigne las parejas para efectuar los cruces con los individuos de la generación actual

- *void reproducir()*:

  Ordena a *OpReproduccion* que ejecute la estrategia general de reproducción.

- *void mutar()*:

  Ordena a todos los individuos de la generación actual que ejecuten el método *mutar()*

- *void calcularObjetivo()*:

  Ordena a todos los individuos de la generación actual que ejecuten el método *calcularObjetivo()*

- *void buscaElite()*:

  Ejecuta el método *buscaElite()* de la propiedad *GeneracionActual*

- *void elitismo()*:

  Si *Elitismo* es distinto de cero, Ejecuta el método *elitismo()* de la propiedad *GeneracionActual*

- *void optimizar()*:

  Ejecuta *iniciarOptimizacion()*; ejecuta iterativamente *iterarOptimizacion()* hasta que la función *parada()* retorna un valor distinto de cero; ejecuta *finalizarOptimizacion()*

- *void iniciarOptimizacion()*:

  Prepara el algoritmo para su ejecución.

- *void iterarOptimizacion()*:

  Efectua una iteración más en el algoritmo

- *void finalizarOptimizacion()*:

  Finaliza el algoritmo

- *int parada()*:

  Devuelve un valor distinto de cero si se ha cumplido el criterio de parada.

- *void actualizarMedidas()*:

  Calcula los valores intermedios en cada iteración.

- *virtual void adaptacion()*:

  Actualmente no se usa. Servirá en futuras versiones para efectuar adaptación de los parámetros del algoritmo

- *void modelo(Individuo *Ind)*:

  Con esta función se especifica que *Ind* servirá de modelo para crear las distintas instancias de *Individuo* que requiere el algoritmo.

- *void mostrarMedidas()*:

  Presenta por la salida estándar los valores intermedios de la generación actual

- *void salvar()*:

  Almacena en *arch* los valores intermedios de la generación actual

2.2.7 Clase *OperadorProbabilidad*

OperadorProbabilidad es una clase abstracta cuyo propósito principal es el de asignar una probabilidad de supervivencia a cada uno de los individuos presentes en un objeto de la clase *Poblacion*. La tabla 2.10 contiene las propiedades (variables) de la clase *OperadorProbabilidad*. Los procedimientos se explican a continuación:

- *virtual void asignarProbabilidad(int descendente)=0*:

  Función virtual para efectuar el proceso de asignación de probabilidad de supervivencia

Tabla 2.7: Propiedades de la clase *AlgoritmoGenetico*. Parte A

Propiedad	**Descripción**
Poblacion GeneracionActual	Contiene los individuos de cada generación
*OperadorProbabilidad *OpProbabilidad*	Operador para asignar la probabilidad de supervivencia de cada individuo
*OperadorSeleccion *OpSeleccion*	Operador para efectuar el proceso de Selección en cada generación
*OperadorParejas *OpParejas*	Operador para asignar parejas de cruce dentro de los individuos de cada generación
*OperadorReproduccion *OpReproduccion*	Operador para determinar la estrategia general de reproducción
*Arreglo¡OperadorMutacion¿ *ListaOperadorMutacion*	Arreglo de operadores de mutación
*Arreglo¡OperadorCruce¿ *ListaOperadorCruce*	Arreglo de operadores de cruce
int Elitismo	Si es distinto de cero se emplea la estrategia de elitismo
int Maximizar	Si es distinto de cero se maximiza la función objetivo, en caso contrario se minimiza
int TamanoPoblacion	Determina el número de individuos en cada generación
*Individuo *MejorEnEstaGeneracion*	Copia del mejor individuo de la generación actual
*Individuo *PeorEnEstaGeneracion*	Copia del peor individuo de la generación actual
*Individuo *MejorEnLaHistoria*	Copia del mejor individuo de toda la historia
int Conteo	Variable auxiliar para determinar si deben guardarse los valores de la generación actual en *arch*
int Generacion	Número de la generación actual
int GeneracionMaxima	Máximo número de generaciones
int GeneracionDelMejorEnLaHistoria	Número de la generación en que apareció el mejor individuo en la historia
double EvaluacionMedia	Promedio de funciones objetivo de la generación actual
double MedidaOnLine	Medida OnLine de la generación actual
double MedidaOffLine	Medida OffLine de la generación actual

Tabla 2.8: Propiedades de la clase *AlgoritmoGenetico*. Parte B

Propiedad	**Descripción**
double AcumuladoOnLine	Variable auxiliar para el cálculo de la medida OnLine
double AcumuladoOffLine	Variable auxiliar para el cálculo de la medida OffLine
double Desviacion	Desviación Estándar de funciones objetivo de la generación actual
int IndicadorArchivo	Si es distinto de cero se usará *arch* para salvar los valores intermedios de las iteraciones
int IntervaloSalvar	Establece cada cuántas iteraciones deben salvarse los valores intermedios en *arch*
*FILE *arch*	apuntador al archivo en que se salvan los valores intermedios de las iteraciones
char NombreArchivo[400]	Nombre del archivo al que apunta *arch*
int IndicadorMostrar	Si es distinto de cero se mostrarán por la salida estándar los valores intermedios de las iteraciones
int IndicadorMostrarGeneracion	Si es distinto de cero se salva y/o muestra el número de generación actual
int IndicadorMostrarMejorEnHistoria	Si es distinto de cero se salva y/o muestra la función objetivo del mejor individuo en la historia
int IndicadorMostrarGeneracionMejorHistorico	Si es distinto de cero se salva y/o muestra la generación en que apareció el mejor individuo en la historia
int IndicadorMostrarMejorEnGeneracion	Si es distinto de cero se salva y/o muestra la función objetivo del mejor individuo de la generación actual

Tabla 2.9: Propiedades de la clase *AlgoritmoGenetico*. Parte C

Propiedad	**Descripción**
int IndicadorMostrarPeorEnGeneracion	Si es distinto de cero se salva y/o muestra la función objetivo del peor individuo de la generación actual
int IndicadorMostrarMedia	Si es distinto de cero se salva y/o muestra la media de funciones objetivo de la generación actual
int IndicadorMostrarDesviacion	Si es distinto de cero se salva y/o muestra la desviación estándar de las funciones objetivo de la generación actual
int IndicadorMostrarOnLine	Si es distinto de cero se salva y/o muestra la medida OnLine para la genereación actual
int IndicadorMostrarOffLine	Si es distinto de cero se salva y/o muestra la medida OffLine para la genereación actual
int CriterioParada	Determina qué criterio de parada emplear

Tabla 2.10: Propiedades de la clase *OperadorProbabilidad*

Propiedad	**Descripción**
*AlgoritmoGenetico *AG*	Apuntador al Algoritmo Genético
*Poblacion *Pob*	Apuntador a la población sobre la cual opera

Tabla 2.11: Propiedades de la clase *OperadorSeleccion*

Propiedad	**Descripción**
*AlgoritmoGenetico *AG*	Apuntador al Algoritmo Genético
*Poblacion *Pob*	Apuntador a la población sobre la cual opera

Tabla 2.12: Propiedades de la clase *OperadorParejas*

Propiedad	**Descripción**
*AlgoritmoGenetico *AG*	Apuntador al Algoritmo Genético
*Poblacion *Pob*	Apuntador a la población sobre la cual opera

2.2.8 Clase *OperadorSeleccion*

OperadorSeleccion es una clase abstracta cuyo propósito principal es el de efectuar el proceso de selección en un objeto de la clase *Poblacion*. La tabla 2.11 contiene las propiedades (variables) de la clase *OperadorSeleccion*. Los procedimientos se explican a continuación:

- *virtual void seleccionar()=0*:

  Función virtual para efectuar el proceso de selección

2.2.9 Clase *OperadorParejas*

OperadorParejas es una clase abstracta cuyo propósito principal es el de asignar las parejas para el proceso de reproducción a cada uno de los individuos presentes en un objeto de la clase *Poblacion*. La tabla 2.12 contiene las propiedades (variables) de la clase *OperadorParejas*. Los procedimientos se explican a continuación:

- *virtual void asignarParejas()=0*:

  Función virtual para efectuar el proceso de asignación de probabilidad de supervivencia

2.2.10 Clase *OperadorReproduccion*

OperadorReproducción es una clase abstracta cuyo propósito principal es el de ejecutar una estrategia general de reproducción en un objeto de la clase *Poblacion*. La tabla 2.13

Tabla 2.13: Propiedades de la clase *OperadorReproduccion*

Propiedad	**Descripción**
*AlgoritmoGenetico *AG*	Apuntador al Algoritmo Genético
*Poblacion *Pob*	Apuntador a la población sobre la cual opera

contiene las propiedades (variables) de la clase *OperadorReproduccion*. Los procedimientos se explican a continuación:

- *virtual void reproducir()=0*:

  Función virtual para ejecutar la estrategia general de reproducción

2.3 Clases Herederas de *Gen*

La clase *Gen*, cuya definición se explica en la sección 2.2.1, es una clase abstracta, y por lo tanto para ser empleada es necesario definir al menos una clase heredera no abstracta. Para definir una clase heredera de *Gen* es necesario tener en cuenta que:

- En la clase *Gen* no se ha definido ninguna propiedad que permita almacenar la información genética, por lo tanto, en las clases herederas deberán definirse estas propiedades, y deberán ser del tipo adecuado a la codificación que se desee realizar. Así, si se desea efectuar codificación real, será necesario definir una propiedad del tipo *float* o *double*, mientras que si se desea efectuar codificación entera, será necesario definir una Propiedad del tipo *int* o *long*.

- En la clase *Gen* se define la función *void crearAleatorio()* como puramente virtual, y por lo tanto es necesario redefinirla en las clases herederas. El propósito de esta función es el de asignar a la propiedad en la que se almacena la información genética (ver ítem inmediatamente anterior) un valor aleatorio válido.

- Por cada clase heredada de *Gen* debe definirse al menos una clase heredada de *OperadorMutacion* y una clase heredada de *OperadorCruce*, encargadas de efectuar la mutación y el cruce, respectivamente, de los genes de la nueva clase (ver sección 2.4).

- Las funciones *operadorMutacionDefecto()* y *operadorCruceDefecto()* deben retornar unos apuntadores a objetos de las clases a que hace referencia el ítem inmediatamente anterior.

En la librería *UNGenético* se han incluido tres clases herederas *GenReal*, *GenEntero* y *GenBool*, empleadas para efectuar codificación Real, Entera y Binaria, respectivamente. Estas clases se detallan en las tablas 2.14, 2.15 y 2.16, .

Tabla 2.14: clase *GenReal*

Nuevas Propiedades	• *double Valor*: Almacena la Información genética en codificación Real • *double Minimo*: Mínimo valor válido de la información Genética; por defecto es 0.0 • *double Maximo*: Máximo valor válido de la información Genética; por defecto es 1.0
Nuevos Métodos	• *void crearAleatorio()*: Almacena en *Valor* un valor aleatorio (distribución uniforme) entre *Minimo* y *Maximo* • *void operadorMutacionDefecto()*: crea un objeto de la clase *OperadorMutacionRealUnforme* y retorna su apuntador. • *void operadorCruceDefecto()*: crea un objeto de la clase *OperadorCruceRealPlano* y retorna su apuntador.
Operador de Mutación asignado por defecto	un objeto de la clase *OperadorMutacionRealUniforme*
Operador de Cruce asignado por defecto	un objeto de la clase *OperadorCruceRealPlano*

Tabla 2.15: clase *GenEntero*

Nuevas Propiedades	• *long Valor*: Almacena la Información genética en codificación Real • *long Minimo*: Mínimo valor válido de la información Genética; por defecto es -10 • *long Maximo*: Máximo valor válido de la información Genética; por defecto es 10
Nuevos Métodos	• *void crearAleatorio()*: Almacena en *Valor* un valor aleatorio (distribución uniforme) entre *Minimo* y *Maximo* • *void operadorMutacionDefecto()*: crea un objeto de la clase *OperadorMutacionEnteroUnforme* y retorna su apuntador. • *void operadorCruceDefecto()*: crea un objeto de la clase *OperadorCruceEnteroPlano* y retorna su apuntador.
Operador de Mutación asignado por defecto	un objeto de la clase *OperadorMutacionEnteroUniforme*
Operador de Cruce asignado por defecto	un objeto de la clase *OperadorCruceEnteroPlano*

Tabla 2.16: clase *GenBool*

Nuevas Propiedades	• *int Valor*: Almacena la Información genética en codificación Real
Nuevos Métodos	• *void crearAleatorio()*: Almacena en *Valor* un valor aleatorio (distribución uniforme) entre *Minimo* y *Maximo* • *void operadorMutacionDefecto()*: crea un objeto de la clase *OperadorMutacionBoolUnforme* y retorna su apuntador. • *void operadorCruceDefecto()*: crea un objeto de la clase *OperadorCruceBoolPlano* y retorna su apuntador.
Operador de Mutación asignado por defecto	un objeto de la clase *OperadorMutacionBoolUniforme*
Operador de Cruce asignado por defecto	un objeto de la clase *OperadorCruceBoolPlano*

2.4 Clases Herederas de los Operadores

En la sección 2.2 se mencionan varias clases abstractas que encapsulan el comportamiento de diversos tipos de operadores. Para emplear el algoritmo genético es necesario definir al menos una clase heredera de ellas. En las secciones siguientes se presentan los detales de implementación de esas clases herederas. Las clases herederas a las que hacemos referencia son

- *OperadorMutacion* (sección 2.2.2)
- *OperadorCruce* (sección 2.2.3)
- *OperadorProbabilidad* (sección 2.2.7)
- *OperadorSeleccion* (sección 2.2.8)
- *OperadorParejas* (sección 2.2.9)
- *OperadorReproduccion* (sección 2.2.10)

2.4.1 Clases Herederas de *OperadorMutacion*

La clase *OperadorMutacion*, cuya definición se explica en la sección 2.2.2, es una clase abstracta, y por lo tanto para ser empleada es necesario definir al menos una clase heredera no abstracta.

Para definir una clase heredera de *OperadorMutacion* es necesario tener en cuenta que:

- Cada clase heredera de *OperadorMutacion* corresponde a una clase heredera de *Gen* (ver sección 2.3). El propósito de una clase heredera de *OperadorMutacion* es el de efectuar una mutación sobre un gen de la clase heredera de *Gen* asociada.
- La clase *OperadorMutacion* define la propiedad *double ProbabilidadMutacion*, cuyo valor por defecto es 0.1. La mutación de una clase heredera se realiza con probabilidad igual a *ProbabilidadMutacion*.
- La clase *OperedorMutacion* define la función virtual pura *mutarGen(Gen *g)*. Toda clase heredera debe redefinir esta función, y es en ella en donde debe realizarse la mutación del gen *g*. Típicamente, el código de esta función será de la siguiente forma (se ha tomado como ejemplo el código de la clase *OperadorMutacionRealUniforme*, que opera sobre objetos de la clase *GenReal*):

```
void OperadorMutacionRealUniforme::mutarGen(Gen *g)
{
// Apuntador a una clase heredera de Gen
  GenReal *gr;

// Se está forzando a que el apuntador de entrada
```

Tabla 2.17: Clases Herederas de *OperadorMutacion*

Clase	**Clase heredera de *Gen* asociada**	**Mutación**
OperadorMutacion RealUniforme	*GenReal*	Mutación Uniforme
OperadorMutacion RealNoUniforme	*GenReal*	Mutación No Uniforme, controlada con la propiedad *double B*, que por defecto vale 0.5
OperadorMutacion RealBGA	*GenReal*	Mutación BGA controlada con la propiedad *double Factor* que por defecto vale 0.1
OperadorMutacion EnteroUniforme	*GenEntero*	Mutación Uniforme
OperadorMutacion EnteroNoUniforme	*GenEntero*	Mutación No Uniforme, controlada con la propiedad *double B*, que por defecto vale 0.5
OperadorMutacion EnteroBGA	*GenEntero*	Mutación BGA controlada con la propiedad *double Factor* que por defecto vale 0.1
OperadorMutacion BoolUniforme	*GenBool*	Mutación Uniforme

```
// apunte a la clase heredera
  gr=(GenReal*)g;

// A partir de este punto se efectúa la mutación
  double azar;
  azar=(double)((double)rand()/(double)RAND_MAX);
  gr->Valor=gr->Minimo+azar*(gr->Maximo-gr->Minimo);
}
```

En la librería *UNGenético* se han incluido las clases herederas de *OperadorMutacion* que se detallan en la tabla 2.17.

2.4.2 Clases Herederas de *OperadorCruce*

La clase *OperadorCruce*, cuya definición se explica en la sección 2.2.3, es una clase abstracta, y por lo tanto para ser empleada es necesario definir al menos una clase heredera no abstracta.

Para definir una clase heredera de *OperadorCruce* es necesario tener en cuenta que:

- Cada clase heredera de *OperadorCruce* corresponde a una clase heredera de *Gen* (ver sección 2.3). El propósito de una clase heredera de *OperadorMutacion* es el de efectuar un cruce entre dos genes (padre y madre) de la clase heredera de *Gen* asociada, para generar un conjunto de genes de la misma clase (hijos).
- La clase *OperedorCruce* define la función virtual pura *void cruzarGenes(Gen *madre,Gen *padre, Arreglo<Gen> *h, int numHijos)*. Toda clase heredera debe redefinir esta función, y es en ella en donde debe realizarse el cruce de los genes *madre* y *padre* para generar *numHijos* genes que se almacenan en el arreglo *h*. Típicamente, el código de esta función será de la siguiente forma (se ha tomado como ejemplo el código de la clase *OperadorCruceRealPlano*, que opera sobre objetos de la clase *GenReal*):

```
void OperadorCruceRealPlano::cruzarGenes(Gen *m,Gen *p,
                        Arreglo$<$Gen$>$ *h, int numHijos)
{
// apuntadores a la clase heredera de Gen asociada
  GenReal *madre,*padre;

// se esta forzando a que los genes madre y padre
// apunten a la clase asociada
  madre=(GenReal*)m;
  padre=(GenReal*)p;

// hijos es un arreglo temporal
  Arreglo<GenReal> hijos;

// se crea un ciclo para generar tantos hijos como
// indique numHijos
  int i,tam;
  for(i=0;i<numHijos;i++)
  {

// se define un gen hijo de la clase asociada
    GenReal *gr;
    gr=new GenReal;

// en este punto se le dan las características propias
// del método de cruce.
    double azar;
    azar=(double)((double)rand()/(double)RAND_MAX);
    gr->Valor=madre->Valor+azar*(padre->Valor-madre->Valor);
    gr->Minimo=madre->Minimo;
    gr->Maximo=madre->Maximo;
```

```
    if(gr->Valor<gr->Minimo){gr->Valor=gr->Minimo;}
    if(gr->Valor>gr->Maximo){gr->Valor=gr->Maximo;}

// se adiciona en el arreglo temporal
    hijos.Add(gr);
  }

// se adicionan los hijos al arreglo h
  tam=hijos.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    h->Add(hijos.dato(i));
  }

// se limpia el arreglo temporal
  hijos.FlushDetach();
}
```

En la librería *UNGenético* se han incluido las clases herederas de *OperadorCruce* que se detallan en la tabla 2.18.

2.4.3 Clases Herederas de *OperadorProbabilidad*

La clase *OperadorProbabilidad*, cuya definición se explica en la sección 2.2.7, es una clase abstracta, y por lo tanto para ser empleada es necesario definir al menos una clase heredera no abstracta.

Para definir una clase heredera de *OperadorProbabilidad* es necesario tener en cuenta que la clase *OperadorProbabilidad* define la función virtual pura *void asignarProbabilidad(int descendente)*. Toda clase heredera debe redefinir esta función; Es en ella en donde debe realizarse la asignación de probabilidad de Supervivencia a cada individuo en la población; el argumento indica si la mayor probabilidad debe asignarse al individuo con mayor función objetivo (*descendente* igual a cero) o con menor función objetivo (*descendente* distinto de cero).

En la librería *UNGenético* se ha incluido la clase heredera *OperadorProbabilidadProporcional* que asigna una probabilidad de supervivencia proporcional a la función objetivo.

2.4.4 Clases Herederas de *OperadorSeleccion*

La clase *OperadorSeleccion*, cuya definición se explica en la sección 2.2.8, es una clase abstracta, y por lo tanto para ser empleada es necesario definir al menos una clase heredera no abstracta.

Para definir una clase heredera de *OperadorSeleccion* es necesario tener en cuenta que la clase *OperadorSeleccion* define la función virtual pura *void seleccionar()*. Toda clase heredera debe redefinir esta función; Es en ella en donde debe realizarse la selección de los individuos que sobreviven en cada iteración del algoritmo.

Tabla 2.18: Clases Herederas de *OperadorCruce*

Clase	**Clase heredera de *Gen* asociada**	**Cruce**
OperadorCruce RealPlano	*GenReal*	Cruce Plano
OperadorCruce RealAritmetico	*GenReal*	Cruce Aritmético controlado por *double lambda*
OperadorCruce RealBLX	*GenReal*	Cruce BLX-α controlado por *Alfa*
OperadorCruce RealLineal	*GenReal*	Cruce Lineal
OperadorCruce RealDiscreto	*GenReal*	Cruce Discreto
OperadorCruce RealIntermedio Extendido	*GenReal*	Cruce Intermedio Extendido controlado por *double Alfa*
OperadorCruce RealHeuristico	*GenReal*	Cruce Heurístico
OperadorCruce EnteroPlano	*GenEntero*	Cruce Entero Plano
OperadorCruce EnteroAritmetico	*GenEntero*	Cruce Aritmético controlado por *double lambda*
OperadorCruce EnteroBLX	*GenEntero*	Cruce BLX-α controlado por *Alfa*
OperadorCruce EnteroLineal	*GenEntero*	Cruce Lineal
OperadorCruce EnteroDiscreto	*GenEntero*	Cruce Discreto
OperadorCruce EnteroIntermedio Extendido	*GenEntero*	Cruce Intermedio Extendido controlado por *double Alfa*
OperadorCruce Heuristico	*GenEntero*	Cruce Heurístico
OperadorCruce BoolPlano	*GenBool*	Cruce Plano

En la librería *UNGenético* se ha incluido la clase heredera *OperadorSeleccionEstocasticaRemplazo* que selecciona de acuerdo al método tradicional de la ruleta.

2.4.5 Clases Herederas de *OperadorParejas*

La clase *OperadorParejas*, cuya definición se explica en la sección 2.2.9, es una clase abstracta, y por lo tanto para ser empleada es necesario definir al menos una clase heredera no abstracta.

Para definir una clase heredera de *OperadorParejas* es necesario tener en cuenta que la clase *OperadorParejas* define la función virtual pura *void asignarParejas()*. Toda clase heredera debe redefinir esta función; Es en ella en donde debe realizarse la asignación de parejas que se cruzarán para dar origen a los individuos de la nueva generación.

En la librería *UNGenético* se ha incluido la clase heredera *OperadorParejasAleatoria* que asigna parejas en forma aleatoria.

2.4.6 Clases Herederas de *OperadorReproduccion*

La clase *OperadorReproduccion*, cuya definición se explica en la sección 2.2.10, es una clase abstracta, y por lo tanto para ser empleada es necesario definir al menos una clase heredera no abstracta.

Para definir una clase heredera de *OperadorReproduccion* es necesario tener en cuenta que la clase *OperadorReproduccion* define la función virtual pura *void reproducir()*. Toda clase heredera debe redefinir esta función; Es en ella en donde debe ejecutarse la estrategia general de reproducción.

En la librería *UNGenético* se han incluido las clases herederas de *OperadorReproduccionDosPadresDosHijos* y *OperadorReproduccionCrucePlano*. La primera de ellas genera dos hijos por cada dos padres, cruzando uno a uno sus genes con los operadores de cruce respectivos. La segunda genera también dos hijos por cada dos padres, pero intercambiando los genes respectivos (ignora los operadores de cruce).

2.5 Clases Herederas de *Individuo*

La clase *Individuo*, cuya definición se explica en la sección 2.2.4, es una clase abstracta, y por lo tanto para ser empleada es necesario definir al menos una clase heredera no abstracta. Quizás ésta sea la clase más importantes desde la perspectiva del usuario de la librería, ya que a través de ella se define el problema a optimizar, tal como se explica en los capítulos 3 y 4.

Para definir una clase heredera de *Individuo* es necesario tener en cuenta que:

- Cada individuo posee un arreglo de genes denominado *Genoma*. Allí pueden contenerse genes de diferentes tipos, es decir, objetos de distintas clases herederas de *Gen*.

- Las características que definen a cada Individuo son propiedades diferentes a *Genoma*. mediante las función *codificar()* esas propiedades se almacenan en el *Genoma*

siguiendo alguna estrategia de codificación, mientras que con la función *decodificar()* se lee la información genética de *Genoma*, y se actualizan las propiedades del Individuo.

- La clase *Individuo* define la función virtual pura *codificar()*. Toda clase heredera debe redefinir esta función; se encarga de codificar las propiedades del Individuo en el *Genoma*
- La clase *Individuo* define la función virtual pura *decodificar()*. Toda clase heredera debe redefinir esta función; se encarga de decofificar las propiedades del *Genoma* y actualizar los valores de las propiedades del Individuo.
- La clase *Individuo* define la función virtual pura *Individuo* crearCopia()*. Toda clase heredera debe redefinir esta función; en ella debe crearse un objeto de la misma clase, y retornar el apuntador a la clase padre *Individuo*
- La clase *Individuo* define la función virtual pura *copiarDetalles(Individuo *other)*. Toda clase heredera debe redefinir esta función; en esta función se copian las propiedades de *other* y se codifican
- La clase *Individuo* define la función virtual pura *double objetivo()*. Toda clase heredera debe redefinir esta función; aqui se calcula el valor de la función objetivo del Individuo.
- La clase *Individuo* define la función virtual *void crearOperadores(Arreglo¡OperadorMutacion¿ *LM,Arreglo¡OperadorCruce¿ *LC)*. Si se desea trabajar con operadotred de cruce y mutación distintos a los definidos por defecto, la clase heredera debe redefinir esta función.

Capítulo 3

Utilización de la librería

El propósito de este capítulo es el de presentar la forma en que debe emplearse la librería detallada en el capítulo 2, para efectuar una optimización mediante algoritmos genéticos. En la sección 3.1 se explican los fundamentos de esta utilización, y en las secciones 3.2 y 3.3 se muestran algunos ejemplos sencillos. El capítulo 4 se reserva para unos ejemplos más sofisticados.

3.1 Fundamentos

Para emplear la librería *UNGenético* es necesario efectuar dos pasos:

1. Definir una clase heredera de *Individuo* en la que se debe reppresentar el problema a optimizar.

2. Ejecutar el algoritmo siguiento mediante una corta secuencia de instrucciones.

El primer paso debe ejecutarse acorde con los lineamientos de la sección 2.5. La ejecución del algoritmo puede hacerse de dos formas: Ejecución Contínua ó Ejecución Paso a Paso. La primera alternativa ejecuta al algoritmo en una sóla instrucción, mientras que la segunda permite definir un ciclo para cada iteración (para cada generación), teniendo asi acceso a los resultados intermedios del algoritmo.

Para mostrar la utilización de estas dos alternativas, supóngase que en el paso 1 se ha definido la clase *IndividuoHeredero*, que es heredera de la clase *Individuo*. La tabla 3.1 muestra cómo podrían ser las líneas de código necesarias para emplear el algoritmo.

Tal como se especifica en la sección 2.2.6, la funcion *modelo(Individuo *Ind)* de la clase *AlgoritmoGenetico* se emplea para informar al algoritmo qué tipo de Individuo debe optimizar.

Para obtener información acerca de los resultados (Parciales y/o definitivos) del proceso de optimización, existen tres posibilidades:

- *Mediante apuntadores a individuos de la población:* En cada generación el algoritmo crea copias del mejor y del peor individuo de la generación actual y del mejor individuo en todas las generaciones hasta la actual; los apuntadores (del tipo

Tabla 3.1: Ejecución del Algoritmo Genético

Ejecución Contínua	*Ejecución Paso a Paso*

`AlgoritmoGenetico MiAg;`
`IndividuoHeredado Ind;`
`MiAg.modelo(&Ind);`
`MiAg.optimizar();`
	`AlgoritmoGenetico MiAg;`
	`IndividuoHeredado Ind;`
	`MiAg.modelo(&Ind);`
	`MiAg.iniciarOptimizacion();`
	`do`
	`{`
	`  MiAg.iterarOptimizacion();`
	`}while(!MiAg.parada());`
	`MiAg.finalizarOptimizacion();`

Individuo) a estas copias son *MejorEnEstaGeneracion*, *PeorEnEstaGeneracion* y *MejorEnLaHistoria*, respectivamente. También se actualizan en cada generación algunas propiedades que sirven de indicadores sobre el desarrollo de la optimización. La tabla 3.2 consigna el nombre y significado de esas propiedades.

- *Mediante un archivo de texto:* El algoritmo puede salvar en un archivo de texto algunos valores que sirven como indicadores sobre el desarrollo del algoritmo. Puede especificarse el nombre del archivo (que por defecto es "prueba.txt") empleando la propiedad *char NombreArchivo[400].* También puede especificarse cada cuantas generaciones se desea que el algoritmo archive esos valores, mediante la propiedad *int IntervaloSalvar* (que por defecto es 10). Existe un grupo de propiedades que sirven como banderas, para seleccionar cuáles de esos valores se desean imprimir; el valor por defecto de todas ellas es 1, lo que indica que por defecto se imprimen todos los valores. La tabla 3.3 resume los valores que pueden salvarse y la bandera que los controla; adicionalmente existe la bandera *IndicadorMostrar* que debe estar activada (1) para poder salvar cualquier de los valores. Los valores de una misma generación se salvan en una misma línea, en el mismo orden en que aparecen en la tabla 3.3

- *Mediante impresión por la salida estándar:* Los indicadores mencionados en el ítem anterior también pueden desplegarse por la salida estándar mediante la función *void mostrarMedidas().* Las banderas reseñadas en la tabla 3.3 también se emplean para controlar qué indicadores se despliegan con esta función.

3.2 Un ejemplo con codificación Real

En este primer ejemplo se busca minimizar la ecuación

Tabla 3.2: Indicadores del desarrollo del algoritmo

Propiedad	**Descripción**
int Generacion	Número de generación actual
int GeneracionDelMejorEnLaHistoria	Generación en la que ha aparecido el mejor individuo hasta la generación actual
double EvaluacionMedia	Promedio de las funciones objetivo de todos los individuos de la población, en la generación actual
double Desviacion	desviación estándar de las funciones objetivo de todos los individuos de la población, en la generación actual
double MedidaOnLine	Medida "OnLine"en la generación actual
double MedidaOffLine	Medida "OffLine"en la generación actual

Tabla 3.3: Indicadores que pueden salvarse

Indicador	**Bandera**
Número de Generación actual	*IndicadorMostrar Generacion*
Mejor función objetivo hasta la generación actual	*IndicadorMostrar MejorEnHistoria*
Generación en que apareció el mejor individuo hasta la generación actual	*IndicadorMostrar GeneracionMejorHistorico*
Mejor función objetivo en la generación actual	*IndicadorMostrar MejorEnGeneracion*
Peor función objetivo en la generación actual	*IndicadorMostrar PeorEnGeneracion*
Promedio de las funciones objetivo de la generación actual	*IndicadorMostrar Media*
Desviación estándar de las funciones objetivo de la generación actual	*IndicadorMostrar Desviacion*
Medida "OnLine"de la generación actual	*IndicadorMostrar OnLine*
Medida "OffLine"de la generación actual	*IndicadorMostrar OffLine*

Figura 3.1: Definición de la clase *VectorReal*

```
class VectorReal:public Individuo
{
public:
  VectorReal(int dim);
  ~VectorReal();
//// Funciones virtuales de Individuo:
  Individuo* crearCopia();
  void copiarDetalles(Individuo *other);
  double objetivo();
  void codificar();
  void decodificar();
  void crearOperadores(Arreglo<OperadorMutacion> *LM,Arreglo<OperadorCruce> *LC);
//// Funciones adicionales:
  void mostrar(); // muestra el vector y su F.O.
//// Propiedades adicionales:
  int Dimension;  // Tamaño del vector
  double *x;        // vector de enteros
};
```

$$y = \sqrt{\sum_{0}^{n-1}(x_i - i)^2} \qquad n \in N; x_i \in R \qquad -2n \leq x_i \leq 2np \tag{3.1}$$

Obviamente la solución de este problema de optimización se da para la n-upla $x_{min} = \{0, 1, \ldots, n-1\}$, cuyo valor de la función objetivo es cero.

De acuerdo a lo establecido en la sección 3.1, el primer paso para resolver este problema de optimización empleando la librería *UNGenético* consiste en crear una clase heredera de *Individuo*. Se ha dado a esa clase el nombre de
VectorReal, y su definición se muestra en la figura 3.1.

En esta clase sólo se han definido dos nuevas propiedades, *long *x*, que es el vector que contiene los reales x_i de la ecuación 3.1, y *int Dimension*, que especifica el tamaño del vector, es decir, que equivale a n en dicha ecuación. Nótese que estas propiedades *no son el Genoma*, que hasta este punto no se ha creado aún.

El genoma de la clase *VectorReal* puede contener genes de cualquier tipo. Debido a la naturaleza de este problema, resulta natural sugerir el uso de genes reales, es decir, de genes de la clase *GenReal*. La estrategia de codificación que se ha empleado es la más sencilla posible: a cada valor del vector de Reales en la ecuación 3.1 le corresponde un gen del tipo *GenReal* en el genoma. Los valores del gen y del real correspondiente en el vector coinciden.

El genoma debe crearse en el constructor de la clase heredera, tal como se muestra en la figura 3.2; dicho constructor recibe como argumento la dimensión del problema (n en la ecuación 3.1). El destructor de la clase se muestra en la figura 3.3

Figura 3.2: Constructor de la clase *VectorReal*

```
VectorReal::VectorReal(int dim)
{
  Dimension=dim;
  x=new double[Dimension];      // Creación del vector de enteros
  for(int i=0;i<Dimension;i++) // Creación de los genes enteros
  {
    GenReal *gen;
    gen= new GenReal; // creación de un gen
    gen->Minimo=-2.0*Dimension; // valor mínimo del gen
    gen->Maximo=2.0*Dimension+1; // valor máximo del gen
    Genoma.Add(gen);    // adición al Genoma
  }
}
```

Figura 3.3: Destructor de la clase *VectorReal*

```
VectorReal::~VectorReal()
{
  Genoma.FlushDestroy(); // Destruir los genes del Genoma
  delete[] x;            // Destruir el vector de enteros
}
```

Figura 3.4: Función *Individuo* crearCopia()* de la clase *VectorReal*

```
Individuo* VectorReal::crearCopia()
{
  Individuo *Ind;
  Ind=new VectorReal(Dimension); // Individuo de la misma clase
  return Ind;
}
```

Figura 3.5: Función *void copiarDetalles(Individuo *other)* de la clase *VectorReal*

```
void VectorReal::copiarDetalles(Individuo *other)
{
// Supone que los dos objetos son de la misma Dimensión
  VectorReal *VR;
  VR=(VectorReal*)other;   // Forzado como VectorReal
  for(int i=0;i<Dimension;i++)
  {
    x[i]=VR->x[i];    // se copian los valores del vector
  }
  codificar();
}
```

En la definición de la clase (figura 3.1) se observa que se han declarado cinco funciones que redefinen las correspondientes cinco funciones virtuales puras de la clase *Individuo*, y que se muestran en las figuras 3.4 a 3.8; también se ha creado la función virtual *crearOperadores()*para mostrar cómo pueden utilizarse operadores de cruce y mutación distintos a los que se definen por defecto, asi como una función *void mostrar()* (ver figura 3.9), para desplegar por la salida estándar el vector y su Función Objetivo (ver figura 3.10). Como puede verse, el código de cada una de estas funciones es extremadamente simple y fácil de entender.

Una vez definida la clase *VectorReal*, heredera de *Individuo*, el siguiente paso, de acuerdo con la sección 3.1 es la ejecución del algoritmo; el programa *PruebaReal* se encarga de ello. La figura 3.11 muestra una versión simple de la función principal del programa, diseñada según las especificaciones de la tabla 3.1 para una ejecución contínua del algoritmo; en esta versión, el valor de n en la ecuación 3.1 es de 5.

En la figura 3.12 se muestra una versión modificada de la función principal del programa. En ella se han incluido unas facilidades adicionales:

- El usuario puede especificar la dimensión del problema (n en la ecuación 3.1).
- El usuario puede establecer el Tamaño de la población y el número de generaciones del algoritmo, mediante las propiedades *TamanoPoblacion* y *GeneracionMaxima* de la clase *AlgoritmoGenetico* (ver sección 2.2.6).

Figura 3.6: Función *double objetivo()* de la clase *VectorReal*

```
double VectorReal::objetivo()
{
  double res=0.0;
  for(int i=0;i<Dimension;i++)
  {
    res+=((x[i]-(i+1))*(x[i]-(i+1)));  // Función objetivo
  }
  return sqrt(res);
}
```

Figura 3.7: Función *void codificar()* de la clase *VectorReal*

```
void VectorReal::codificar()
{
  for(int i=0;i<Dimension;i++)
  {
    GenReal *g;
    g=(GenReal*)Genoma.dato(i); // gen forzado como  GenReal
    g->Valor=x[i];  // Valor del gen = Valor del entero en el vector
  }
}
```

Figura 3.8: Función *void decodificar()* de la clase *VectorReal*

```
void VectorReal::decodificar()
{
  for(int i=0;i<Dimension;i++)
  {
    GenReal *g;
    g=(GenReal*)Genoma.dato(i); // gen forzado como GenReal
    x[i]=g->Valor;  // Valor del entero en el vector = Valor del gen
  }
}
```

Figura 3.9: Función *void operadoresEntero()* de la clase *VectorReal*

```
void VectorReal::crearOperadores(Arreglo<OperadorMutacion> *LM,Arreglo<OperadorCruce> *LC)
{
  OperadorMutacion *OM;
  OperadorCruce *OC;
  OM=new OperadorMutacionRealUniforme;
  OC=new OperadorCruceRealPlano;
  for(int i=0;i<Dimension;i++)
  {
    LM->Add(OM);
    LC->Add(OC);
  }
}
```

Figura 3.10: Función *void mostrar()* de la clase *VectorReal*

```
void VectorReal::mostrar()
{
  calcularObjetivo();
  for(int i=0;i<Dimension;i++)
  {
    cout << x[i] << " ";
  }
  cout << ":" << Objetivo << "\n";
}
```

Figura 3.11: Función principal del programa *PruebaReal* - Versión 1

```
void main()
{
  VectorReal *Mod;
  Mod=new VectorReal(5);

  AlgoritmoGenetico MiAg;
  MiAg.modelo(Mod);
  MiAg.optimizar();
  MiAg.mostrarMedidas();
}
```

- Se utiliza una ejecución Paso a Paso; en cada iteración se muestra información sobre el mejor Individuo.
- Despues de la optimización, el programa despliega información sobre los 5 mejores Individuos de la última generación.

La figura 3.13 muestra los resultados finales de una ejecución del programa *pruebaReal* - Versión 2. Los resultados de esa ejecución se han salvado en un archivo de texto *PruebaReal.txt*, y con él se han obtenido varias gráficas que permiten visualizar el comportamiento del algoritmo genético [1]. Estas gráficas corresponden a algunos de los indicadores consignados en la tabla 3.3, y se muestran en las figuras 3.14 a 3.19

3.3 Un ejemplo con codificación Híbrida

En este segundo ejemplo se ha modificado la ecuación 3.1 para que incluya variables enteras y booleanas, además de las variables reales x_i. La nueva ecuación es la siguiente:

$$y = \sqrt{\sum_0^{n-1}(x_i - i)^2 + K^2 + B} \quad \begin{array}{l} n \in N; x_i \in R \quad -2n \leq x_i \leq 2n \\ K \in Z; -10 \leq K \leq 10; B \in \{0,1\} \end{array} \tag{3.2}$$

Obviamente la solución de este problema de optimización se da para la n-upla $x_{min} = \{0, 1, \ldots, n-1\}$, $K = 0$, $B = 0$ cuyo valor de la función objetivo es cero.

A diferencia del ejemplo de la sección 3.2, en el que todas las variables a optimizar eran del mismo tipo (Real), en este ejemplo hay n variables reales, una variable entera y una variable booleana a optimizar. Se propone entonces emplear una estrategia de codificación híbrida, con n genes de tipo real, un gen de tipo entero y un gen de tipo booleano.

Se ha impementado un programa, alque se la ha dado el nombre *pruebaHíbrida*. La clase heredera de *Individuo* se ha denominado *VectorHibrido*. Las diferencias respecto al ejemplo de la sección 3.2 son mínimas, y se resumen a continuación:

- En la definición de la clase se han incluido dos propiedades más: *long* K y *int* B, que representan las variables K yB de la ecuación 3.2.
- El constructor de la clase incluye unas líneas adicionales a las mostradas en la figura 3.2, para adicionar al genoma los nuevos genes, uno entero y uno booleano, tal como se muestra en la figura 3.20
- En la función *crearCopia()* se crea un individuo del tipo *VectorHibrido* en lugar de uno del tipo
  vectorReal, tal como se muestra en la figura 3.21.
- En la funcion
  copiarDetalles(Individuo *Ind) se adicionan dos líneas para copiar los valores de las nuevas propiedades K y B (figura 3.22).

[1] Las gráficas se han obtenido con el programa *gnuplot* Copyright(C) 1986 - 1993 Thomas Williams, Colin Kelley, con licencia pública GNU

Figura 3.12: Función principal del programa *pruebaReal* - Versión 2

```
void main()
{
// Unas variables auxiliares:
  int Dim,Tamano,Generacion;
  cout << "Ejemplo de Aplicación de los Algoritmos Genéticos:\n\n";
  cout << "Dimensión del problema: ";cin >> Dim;
  cout << "Tamaño de la Población: ";cin >> Tamano;
  cout << "Número máximo de generaciones: ";cin >> Generacion;
// Definicion del algoritmo, y asignacion del modelo
  VectorReal *Mod;
  Mod=new VectorReal(Dim);
  AlgoritmoGenetico MiAg;
  MiAg.modelo(Mod);
  MiAg.TamanoPoblacion=Tamano;
  MiAg.GeneracionMaxima=Generacion;
  MiAg.IntervaloSalvar=1;
// Utilizacion Paso a Paso del Algoritmo
  int c=0;
  MiAg.iniciarOptimizacion();
  do
  {
    MiAg.iterarOptimizacion();
    // aqui pueden emplearse los resultados de la iteracion
    // para cualquier cosa, por ejemplo:
    c++;gotoxy(1,7);cout << c << ":";
    VectorReal *VR;
    VR=(VectorReal*)MiAg.MejorEnLaHistoria;
    VR->mostrar();
  }while(!MiAg.parada());
  MiAg.finalizarOptimizacion();

  MiAg.GeneracionActual.ordenar(0);
  int casos;
  if (Tamano<5){casos=Tamano;}else{casos=5;}
  cout << "Estos son los mejores " << casos << " individuos:\n";
  for(int i=0;i<casos;i++)
  {
    VectorReal *VR;
    VR=(VectorReal*)MiAg.GeneracionActual.Generacion.dato(i);
    VR->mostrar();
  }

  MiAg.mostrarMedidas();
}
```

Figura 3.13: Resultados de una ejecución del programa *pruebaReal* - Versión 2

```
Ejemplo de Aplicación de los Algoritmos Genéticos:

Dimensión del problema: 5
Tamaño de la Población: 61
Número máximo de generaciones: 100

100:-0.018845 1.01937 1.9597 2.97291 4.00995 :0.056459415

Estos son los mejores 5 individuos:
-0.018845 1.01937 1.9597 2.97291 4.00995 :0.0564594
-0.0188232 1.0306 1.95968 3.00188 4.01982 :0.0575534
-0.0188232 1.03065 1.95968 3.00572 4.01987 :0.0578502
-0.0188383 1.0275 1.95969 2.97956 4.0162 :0.0584488
-0.018837 1.03211 1.95969 2.98315 4.01564 :0.059496
Generacion:100
MejorHistorico:0.0564594
GeneracionMejorHistorico:89
MejorActual:0.0564594
PeorActual:17.2273
Media:3.3481
Desviación:4.48585
OnLine:3.01687
OffLine:0.276938
```

Figura 3.14: Ejemplo Real: Mejor en cada generación

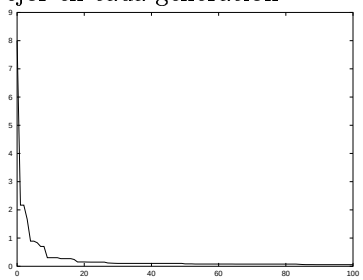


Figura 3.15: Ejemplo Real: Peor en cada generación

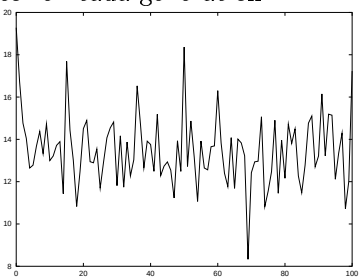


Figura 3.16: Ejemplo Real: Promedio en cada generación

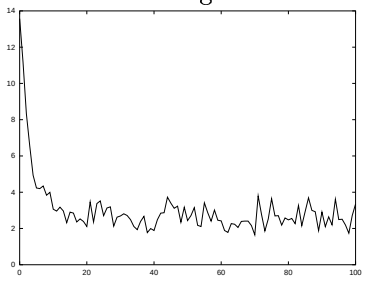


Figura 3.17: Ejemplo Real: Desviación Estándar en cada generación

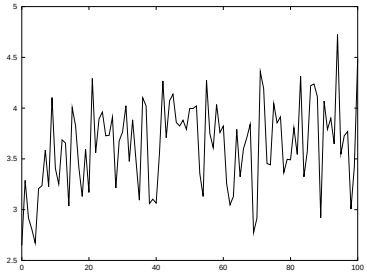


Figura 3.18: Ejemplo Real: Medida OnLine en cada generación

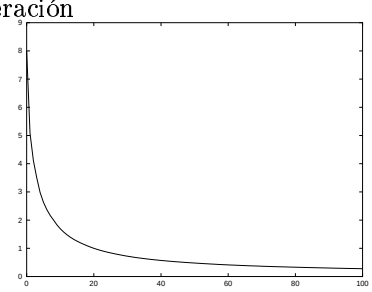


Figura 3.19: Ejemplo Real: Medida OffLine en cada generación

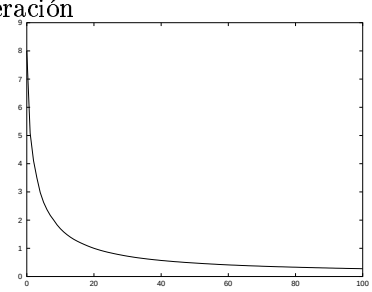


Figura 3.20: Constructor de la clase *VectorHibrido*

```
VectorHibrido::VectorHibrido(int dim)
{
  Dimension=dim;
  x=new double[Dimension];      // Creación del vector de enteros
  for(int i=0;i<Dimension;i++) // Creación de los genes enteros
  {
    GenReal *gen;
    gen= new GenReal; // creación de un gen
    gen->Minimo=-2.0*Dimension; // valor mínimo del gen
    gen->Maximo=2.0*Dimension+1; // valor máximo del gen
    Genoma.Add(gen);    // adición al Genoma
  }
  GenEntero *genE;
  genE= new GenEntero;
  genE->Minimo=-10;
  genE->Maximo=10;
  Genoma.Add(genE);

  GenBool *genB;
  genB= new GenBool;
  Genoma.Add(genB);
}
```

- La función *codificar()* incluye unas líneas para codificar los valores de K y B en los genes respectivos (figura 3.23).
- La función *decodificar()* incluye unas líneas para decodificar los valores de K y B de los genes respectivos (figura 3.24).
- La función *objetivo()* se ha adecuado para representar a la ecuación 3.2 (figura 3.25).
- La función *crearOperadores()* se ha adecuado para definir operadores a los genes reales, booleano y entero.
- La funcion *mostrar()* despliega también información sobre las nuevas propiedades K y B(figura 3.27).
- En la función principal del programa se define un objeto de la clase *VectorHibrido* como modelo para el algoritmo genético, en lugar de un onjeto de la clase *VectorReal* (figura 3.28).

La figura 3.29 muestra los resultados finales de una ejecución del programa *pruebaHíbrida*. Los resultados de esa ejecución se han salvado en un archivo de texto *Prue-*

Figura 3.21: función *Individuo *crearCopia()* de la clase *VectorHibrido*

```
Individuo* VectorHibrido::crearCopia()
{
  Individuo *Ind;
  Ind=new VectorHibrido(Dimension);
  return Ind;
}
```

Figura 3.22: función *void copiarDetalles(Individuo *other)* de la clase *VectorHibrido*

```
void VectorHibrido::copiarDetalles(Individuo *other)
{
  VectorHibrido *VR;
  VR=(VectorHibrido*)other;
  for(int i=0;i<Dimension;i++)
  {
    x[i]=VR->x[i];
  }
  K=VR->K;
  B=VR->B;
  codificar();
}
```

Figura 3.23: función *void codificar()* de la clase *VectorHibrido*

```
void VectorHibrido::codificar()
{
  for(int i=0;i<Dimension;i++)
  {
    GenReal *g;
    g=(GenReal*)Genoma.dato(i);
    g->Valor=x[i];
  }
  GenEntero *gE;
  gE=(GenEntero*)Genoma.dato(i);
  gE->Valor=K;
  i++;
  GenBool *gB;
  gB=(GenBool*)Genoma.dato(i);
  gB->Valor=B;
}
```

Figura 3.24: función *void decodificar()* de la clase *VectorHibrido*

```
void VectorHibrido::decodificar()
{
  for(int i=0;i<Dimension;i++)
  {
    GenReal *g;
    g=(GenReal*)Genoma.dato(i);
    x[i]=g->Valor;
  }
  GenEntero *gE;
  gE=(GenEntero*)Genoma.dato(i);
  K=gE->Valor;
  i++;
  GenBool *gB;
  gB=(GenBool*)Genoma.dato(i);
  B=gB->Valor;
}
```

Figura 3.25: función *double objetivo()* de la clase *VectorHibrido*

```
double VectorHibrido::objetivo()
{
  double res=0.0;
  for(int i=0;i<Dimension;i++)
  {
    res+=((x[i]-(i))*(x[i]-(i)));
  }
  res+=(double)(K*K)+(double)B;
  return sqrt(res);
}
```

Figura 3.26: función *void crearOperadores()* de la clase *VectorHibrido*

```
void VectorHibrido::crearOperadores(Arreglo<OperadorMutacion> *LM,Arreglo<OperadorCruce> *LC)
{
  OperadorMutacion *OM;
  OperadorCruce *OC;
  OM=new OperadorMutacionRealUniforme;
  OC=new OperadorCruceRealPlano;
  for(int i=0;i<Dimension;i++)
  {
    LM->Add(OM);
    LC->Add(OC);
  }
  OM= new OperadorMutacionEnteroUniforme;
  LM->Add(OM);
  OC= new OperadorCruceEnteroPlano;
  LC->Add(OC);
  OM= new OperadorMutacionBoolUniforme;
  LM->Add(OM);
  OC= new OperadorCruceBoolPlano;
  LC->Add(OC);
}
```

Figura 3.27: función *void mostrar()* de la clase *VectorHibrido*

```
void VectorHibrido::mostrar()
{
  calcularObjetivo();
  for(int i=0;i<Dimension;i++)
  {
  cout << x[i] << " ";
  }
  cout << K << " " << B << " ";
  cout << ":" << Objetivo << "\n";
}
```

Figura 3.28: Función principal del programa *pruebaHíbrida*

```
void main()
{
// Unas variables auxiliares:
  int Dim,Tamano,Generacion;
  cout << "Ejemplo de Aplicación de los Algoritmos Genéticos:\n\n";
  cout << "Dimensión del problema: ";cin >> Dim;
  cout << "Tamaño de la Población: ";cin >> Tamano;
  cout << "Número máximo de generaciones: ";cin >> Generacion;
// Definicion del algoritmo, y asignacion del modelo
  VectorHibrido *Mod;
  Mod=new VectorHibrido(Dim);
  AlgoritmoGenetico MiAg;
  MiAg.modelo(Mod);
  MiAg.TamanoPoblacion=Tamano;
  MiAg.GeneracionMaxima=Generacion;
  MiAg.IntervaloSalvar=1;
// Utilizacion Paso a Paso del Algoritmo
  int c=0;
  MiAg.iniciarOptimizacion();
  do
  {
    MiAg.iterarOptimizacion();
    // aqui pueden emplearse los resultados de la iteracion
    // para cualquier cosa, por ejemplo:
    c++;gotoxy(1,7);cout << c << ":";
    VectorHibrido *VR;
    VR=(VectorHibrido*)MiAg.MejorEnLaHistoria;
    VR->mostrar();
  }while(!MiAg.parada());
  MiAg.finalizarOptimizacion();

  MiAg.GeneracionActual.ordenar(0);
  int casos;
  if (Tamano<5){casos=Tamano;}else{casos=5;}
  cout << "Estos son los mejores " << casos << " individuos:\n";
  for(int i=0;i<casos;i++)
  {
    VectorHibrido *VR;
    VR=(VectorHibrido*)MiAg.GeneracionActual.Generacion.dato(i);
    VR->mostrar();
  }

  MiAg.mostrarMedidas();
}
```

Figura 3.29: Resultados de una ejecución del programa *pruebaHibrida*

```
Ejemplo de Aplicación de los Algoritmos Genéticos:

Dimensión del problema: 5
Tamaño de la Población: 61
Número máximo de generaciones: 100

100:0.0518956 1.00816 1.98617 3.0015 4.003 0 0 :0.0544258362

Estos son los mejores 5 individuos:
0.0518956 1.00816 1.98617 3.0015 4.003 0 0 :0.0544258
0.0522189 1.00857 1.98593 3.0015 4.00309 0 0 :0.0548641
0.0522659 1.00857 1.98606 3.0015 4.0031 0 0 :0.0548766
0.0523016 1.00976 1.9855 3.0015 4.00328 0 0 :0.0552623
0.0523212 1.00989 1.9852 3.0015 4.00362 0 0 :0.0554061
Generacion:100
MejorHistorico:0.0544258
GeneracionMejorHistorico:86
MejorActual:0.0544258
PeorActual:10.6682
Media:3.02275
Desviación:3.60671
OnLine:3.44936
OffLine:0.330601
```

baHibrida.txt, y con él se han obtenido varias gráficas que permiten visualizar el comportamiento del algoritmo genético [2]. Estas gráficas corresponden a algunos de los indicadores consignados en la tabla 3.3, y se muestran en las figuras 3.30 a 3.35

[2] Las gráficas se han obtenido con el programa *gnuplot* Copyright(C) 1986 - 1993 Thomas Williams, Colin Kelley, con licencia pública GNU

Figura 3.30: Ejemplo Híbrido: Mejor en cada generación

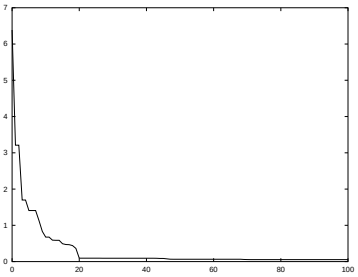


Figura 3.31: Ejemplo Híbrido: Peor en cada generación

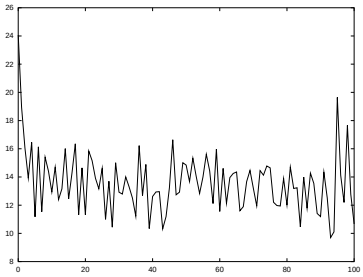


Figura 3.32: Ejemplo Híbrido: Promedio en cada generación

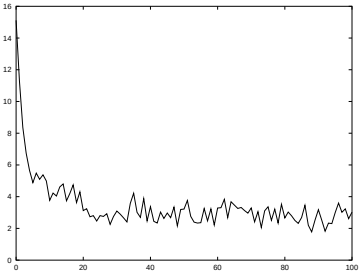


Figura 3.33: Ejemplo Híbrido: Desviación Estándar en cada generación

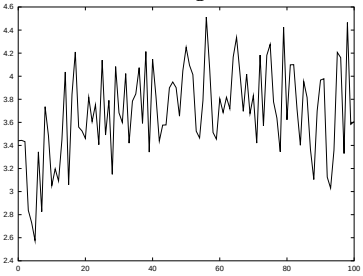


Figura 3.34: Ejemplo Híbrido: Medida OnLine en cada generación

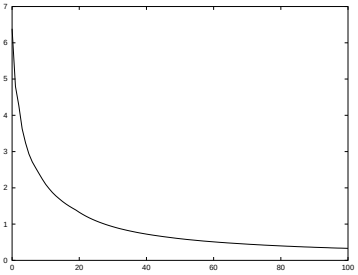
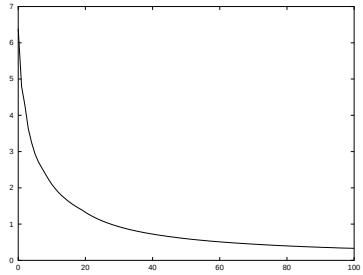


Figura 3.35: Ejemplo Híbrido: Medida OffLine en cada generación



Capítulo 4

Ejemplos de Aplicación

Los ejemplos presentados en el capítulo 3 son ejemplos triviales, cuya optimización no requiere de técnicas heurísticas tales como los algoritmos genéticos. En este capítulo se muestra la aplicación de *UNGenético* en un problema no trivial de optimización: La identificación de un sistema no lineal mediante un Sistema de Lógica Difusa.

En la sección 4.1 se hace una presentación resumida de los Sistemas de Lógica Difusa, que es una adaptación del primer capítulo de [6]; una presentación más detallada puede obtenerse en [5], [13], [14], [18], y [19]. En las sección 4.3 se detallan dos estrategias distintas para emplear tales sistemas en la identificación de sistemas no lineales mediante *UNGenético*; el sistema escogido como ejemplo se muestra en la sección 4.2.

4.1 Sistemas de lógica difusa

Existen varios tipos de Sistemas de Lógica Difusa. En este capítulo se trabajará con "Sistemas de Lógica Difusa con Difusor y Concresor", también conocidos como "Sistemas Tipo Mamdani", cuya estructura se muestra en la figura 4.1 y se detalla en la sección 4.1.2. Se trata de un sistema no lineal de múltiples entradas y múltiples salidas cuyos principios de funcionamiento se apoyan en las nociones de Conjunto Difuso y Lógica Difusa, presentadas inicialmente por Zadeh en [26], [27], [28], [29]; estas nociones se resumen en la sección 4.1.1.

4.1.1 Conjuntos Difusos, Lógica Difusa y Variables Lingüísticas

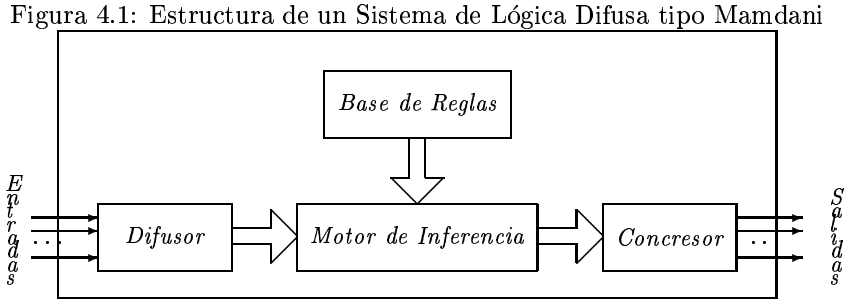
Dado un conjunto U denominado *universo de discurso* o *universo de referencia*, se define un conjunto difuso C como el conjunto de parejas

$$C = \{(x, \mu_C(x)) | x \in U\} \tag{4.1}$$

en donde $\mu_C(x)$ es una función al intervalo unitario denominada *función de pertenencia* o *función característica* del conjunto C:

$$\mu_C(x) : U \rightarrow [0, 1] \tag{4.2}$$

Figura 4.1: Estructura de un Sistema de Lógica Difusa tipo Mamdani



La función de pertenencia $\mu_C(x)$ mide qué tanto pertenece x al conjunto difuso C; si $\mu_C(x) = 0$ significa que el elemento x no pertenece en absoluto a C; si $\mu_C(x) = 1$ significa que pertenece completamente a C; si $0 < \mu_C(x) < 1$ significa que pertenece parcialmente a C. De esta forma se está definiendo C como un conjunto cuyas fronteras no son exactas, sino que son *difusas.*

Las operaciones usuales entre conjuntos tradicionales (usualmente denominados *conjuntos crisp* o *conjuntos concretos*), tales como la Unión, la Intersección y el Complemento pueden extenderse a los conjuntos difusos de diversas maneras [1]. Las más usuales son las siguientes:

- Intersección: La Intersección de dos conjuntos difusos C y D definidos sobre el mismo universo de discurso U y con funciones de pertenencia $\mu_C(x)$ y $\mu_D(x)$ respectivamente, es un nuevo conjunto difuso $C \cap D$ sobre U cuya función de pertenencia es

$$\mu_{C\cap D}(x) = \min(\mu_C(x), \mu_C(x)) \tag{4.3}$$

- Unión: La Unión de dos conjuntos difusos C y D definidos sobre el mismo universo de discurso U y con funciones de pertenencia $\mu_C(x)$ y $\mu_D(x)$ respectivamente, es un nuevo conjunto difuso $C \cup D$ sobre U cuya función de pertenencia es

$$\mu_{C\cup D}(x) = \max(\mu_C(x), \mu_C(x)) \tag{4.4}$$

- Complemento: El complemento de un conjunto difuso C definido sobre el universo de discurso U y con función de pertenencia $\mu_C(x)$, es un nuevo conjunto difuso C' sobre U cuya función de pertenencia es

$$\mu_{C'}(x) = 1 - \mu_C(x) \tag{4.5}$$

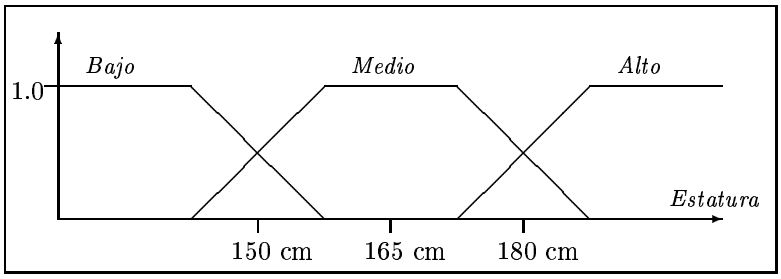
Existe un isomorfismo entre la teoría de conjuntos tradicionales y la lógica booleana, que se resume en la table 4.1. Este isomorfismo puede emplearse para definir la *lógica difusa* a partir de la teoría de conjuntos difusos, tal como se muestra en el siguiente ejemplo:

[1] existen familias de funciones que permiten extender las operaciones clásicas, conocidas como las T-Normas y las S-Normas, para mayor información véase [11]

Tabla 4.1: Isomorfismo entre la teoría de conjuntos y la lógica

Operador de Conjuntos	**Operador Lógico**
Intersección	AND
Unión	OR
Complemento	NOT

Figura 4.2: Ejemplo de Variable Lingüística: *Estatura*



Las proposiciones simples p, q se representas por un conjunto difuso cada una, sobre los universos de discurso U y V, con funciones de pertenencia $\mu_p(x)$ y $\mu_q(y)$. La proposición compuesta p *AND* q se representa por un nuevo conjunto difuso $p_{AND}q$ sobre el universo de discurso UxV con función de pertenencia

$$\mu_{p_{AND}q}(x,y) = \min(\mu_p(x), \mu_q(y)) \tag{4.6}$$

A partir de las operaciones básicas *AND*, *OR* y *NOT*, pueden desarrollarse otras tales como la implicación (*IF - THEN*) y la doble implicación (*IFF*).

En la lógica booleana el grado de verdad de una proposición sólo puede ser *Falso* o *Verdadero*, usualmente representado por los valores 0 y 1 respectivamente. Por el contrario, las proposiciones en la lógica difusa pueden tomar cualquier grado de verdad entre 0 y 1, y por ello se dice que ésta es una forma de *lógica multivaluada.*

Una *Variable Lingüística* es una variable cuyos posibles valores son ciertas *Etiquetas.* Por ejemplo, puede definires la variable lingüística *Edad*, con las etiquetas *Bebe, Niño, Adolescente, Adulto, etc.*. En el contexto de la lógica difusa, a cada etiqueta de una variable lingüística se le asigna un conjunto difuso. La figura 4.2 muestra una posible definición de la Variable Lingüística *Estatura* empleando tres etiquetas: *Bajo, Medio, Alto*[2].

[2]La selección de las etiquetas, y de los conjuntos difusos asociados a éstas, depende de la aplicación de la variable, y en ocasiones de criterios subjetivos

4.1.2 Sistema de Lógica Difusa tipo Mamdani

Tal como se mencionó antes, la estructura de un Sistema de Lógica Difusa tipo Mamdani es la que se muestra en la figura 4.1. Se trata de un sistema de múltiples entradas y múltiples salidas, en cuyo interior la información se representa y procesa empleando la lógica difusa. Cada variable de Entrada y cada variable de salida se representa al interior del sistema por medio de una variable Lingüística como la de la figura 4.2.

La función del *Difusor* es la de tomar las entradas concretas y convertirlas en conjuntos difusos; estos conjuntos serán procesados por el *Motor de Inferencia* empleando la información contenida en la *Base de Reglas* para producir un grupo de conjuntos difusos que serán tomados por el *Concresor* para producir unas salidas concretas[3].

Difusor

El bloque difusor recibe las p entradas concretas que llegan al Sistema de Lógica Difusa, y produce un Conjunto Difuso por cada una de ellas. Cada conjunto difuso producido por este bloque está definido sobre el Universo de Discurso de la variable lingüística respectiva, está centrado en el valor concreto de entrada, y tiene una función de pertenencia cuya forma puede ser distinta para cada variable de entrada. La estrategia más usual consiste en crea un conjunto tipo *singleton*, cuya función de pertenencia $\mu(x)$vale 1 para el valor concreto de entrada, y cero para cualquier otro valor

Concresor

El bloque de concreción recibe los $m \times q$ conjuntos difusos generados por el motor de Inferencia, y produce q valores concretos correspondientes a cada una de las variables de salida del Sistema de Lógica Difusa

En general, para producir cada uno de los q valores concretos, el Concresor toma los m Conjuntos Difusos correspondientes a cada Variable de Salida, y mediante algún algoritmo produce un valor concreto. Ciertos algoritmos de concreción efectúan la *Agregación* de los m conjuntos difusos, usualmente mediante la Unión o la Intersección de estos conjuntos. Posteriormente se calcula algún valor representativo del conjunto resultante, tal como su centro de gravedad o el promedio de los valores con máxima función de pertenencia, y dicho valor representativo es la salida del bloque concresor.

Base de Reglas

La base de reglas es un conjunto de m reglas, cada una de la forma:

IF *(entrada 1 es etiqueta* e_{i1} ***AND*** $\ldots$ ***AND*** *entrada* p *es etiqueta* e_{ip}*)*
THEN *(salida 1 es etiqueta* f_{i1} ***AND*** $\ldots$ ***AND*** *salida* q *es etiqueta* f_{iq}*)*

En donde i es el número de la regla, *etiqueta* $e_i k$ es una de las etiquetas definidas en la variable lingüística de la entrada k, y *etiqueta* $f_i k$ es una de las etiquetas definidas en la variable lingüística de la salida k.

[3]En la siguiente explicación se ha supuesto un sistema de p entradas y q salidas

Opcionalmente, cada etiqueta de las variables de entrada puede estar afectada por un *modificador lingüístico*. Cada modificador (mod_i) es un número real positivo que se emplea como exponente de la función de pertenencia de la etiqueta respectiva. Si el modificador es menor que 1 se interpreta como el adverbio *poco*, y se es mayor que 1 se interpreta como el adverbio *muy*.

En cada regla pueden distinguirse dos partes: el *Antecedente* y el *Consecuente*, de tal forma que puede escribirse en forma abreviada como

IF *Antecedente* ***THEN*** *Consecuente*

Motor de Inferencia

El Motor de Inferencia recibe los p conjuntos difusos producidos por el Difusor, y los aplica a cada una de las m reglas de la Base de Reglas, para producir $m \times q$ Conjuntos Difusos (un conjunto difuso por cada variable de salida en cada una de las reglas) definidos sobre los Universos de Discurso de la Variables Lingüísticas de salida.

La forma en que se define la función de pertenencia de cada uno de los $m \times q$ Conjuntos Difusos producidos es la siguiente:

Supóngase que el Difusor produce p conjuntos difusos $Dif_1, Dif_2, \ldots Dif_p$, con funciones de pertenencia

$$\mu_{Dif_1}(x_1), \mu_{Dif_2}(x_2), ...\mu_{Dif_p}(x_p) \tag{4.7}$$

Supóngase ademas que la regla número i es de la forma

IF *(entrada 1 es etiqueta* e_{i1} ***AND*** $\ldots$ ***AND*** *entrada p es etiqueta* e_{ip}*)*
THEN *(salida 1 es etiqueta* f_{i1} ***AND*** $\ldots$ ***AND*** *salida q es etiqueta* f_{iq}*)*

En donde las etiquetas e_ik y f_ik tienen asociados los conjuntos difusos c_{ik} y d_{ij} respectivamente, cuyas funciones de pertenencia son

$$\mu_{c_{i1}}(x_1), \mu_{c_{i2}}(x_2), ...\mu_{c_{ip}}(x_p) \tag{4.8}$$

$$\mu_{d_{i1}}(y_1), \mu_{d_{i2}}(y_2), ...\mu_{d_{ip}}(y_q) \tag{4.9}$$

Supóngase que el conjunto B_{ij} es uno de los $m \times q$ conjuntos difusos generados por el Motor de Inferencia, correspondiente a la regla i y a la variable de salida j. Dicho conjunto B_{ij} tendrá por funcion de pertenencia:

$$\mu_{B_{ij}}(y_j) = composicion(\mu_{Dif}(X), \mu_{Imp}(X, y_j)) \tag{4.10}$$

$$\mu_{B_{ij}}(y_j) = \max_X(\mu_{Dif}(X) \otimes \mu_{Imp}(X, y_j)) \tag{4.11}$$

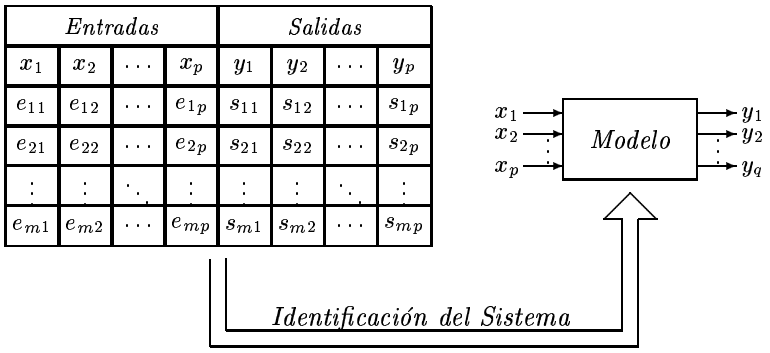
En donde X corresponde a un vector de las p variables de entrada $[x_1\ x_2\ ...\ x_p]$; $\otimes$ corresponde a un operador T-Norma, y $\mu_{Dif}(X)$, $\mu_{Imp}(X, y_j)$ se definen a continuación:

$$\mu_{Dif}(X) = \mu_{Dif_1}(x_1)\ AND\ \ldots\ AND\ \mu_{Dif_p}(x_p) \tag{4.12}$$

$$\mu_{Imp}(X, y_j) = (\mu_{Antecedente}(X) \Rightarrow \mu_{Consecuente}(y_j)) \tag{4.13}$$

$$\mu_{Antecedente}(X) = \mu_{c_{i1}}(x_1)\ AND\ \ldots\ AND\ \mu_{c_{ip}}(x_p) \tag{4.14}$$

Figura 4.3: Identificación de Sistemas



$$\mu_{Consecuente}(y_j) = \mu_{d_{ij}}(x_j) \tag{4.15}$$

En donde el operador AND corresponde a un operador T-Norma, y el operador $\Rightarrow$ corresponde a una Implicación.

En caso de que la regla i incluya los modificadores $mod_{i1} \ldots mod_{ip}$ para cada una de las etiquetas del antecedente el único cambio que debe hacerse para la determinación de las funciones de pertenencia es el siguiente:

$$\mu_{Antecedente}(X) = (\mu_{c_{i1}}(x1))^{mod_{i1}}\ AND\ \ldots\ AND(\mu_{c_{ip}}(x_p))^{mod_{ip}} \tag{4.16}$$

4.1.3 Identificación de sistemas mediante Sistemas de Lógica Difusa

El problema de Identificación de Sistemas puede entenderse como el problema de obtener un modelo de un sistema real, a partir de mediciones efectuadas sobre las entradas y salidas de ese sistema[4], tal como se visualiza en la figura 4.3. Aunque existen múltiples metodologías para el proceso de identificación, una gran parte de ellas sigue los siguientes pasos:

1. Seleccionar un tipo de *modelo paramétrico.* El proceso de identificación consistirá en encontrar el "mejor" modelo.

2. Separar los datos de la tabla (por renglones) en dos grupos: Uno de Entrenamiento y otro de Prueba. La separación usualmente es aleatoria, y en una proporción cercana al 70% para entrenamiento y 30% para prueba.

3. Seleccionar algún *criterio* para medir qué tan bueno es un modelo del sistema a identificar.

4. Emplear los datos de entrenamiento para encontrar los mejores parámetros de modelo, mediante algún *algoritmo de entrenamiento.*

[4]en [15] se hace una magnifica exposición de este tema

5. Evaluar el modelo obtenido con los datos de prueba

Es posible proponer que el modelo a que se refiere el item 1 sea un Sistema de Lógica Difusa como el de la figura 4.1. Los parámetros de este modelo pueden ser (según el algoritmo del item 4), la forma de los conjuntos difusos de las variables lingüísticas, las etiquetas de la base de reglas, los modificadores de los antecedentes de las reglas, etc.

Uno de los algoritmos de identificación de sistemas difusos más eficientes es el conocido como *Algoritmo de Universos Fijos* o *Algoritmo de Wang y Mendel* (véase [23]).

1. Para la k-ésima línea de entranamiento en la tabla, determinar los grados de pertenencia de $e_{k1}, e_{k2}, \ldots, e_{kp}, s_{k1}, s_{k2}, \ldots, s_{kq}$ a cada uno de los Valores Lingüísticos de las respectivas Variables Lingüísticas.

2. Seleccionar los Valores Lingüísticos $Lx_{k1}, Lx_{k2}, \ldots, Lx_{kp}, Ly_{k1}, Ly_{k2}, \ldots, Ly_{kq}$, para los cuales los grados de pertenencia respectivos son máximos.

3. Crear una regla de la forma

$$\textbf{\textit{IF}}\ x_1\ \textit{es}\ Lx_{k1}\ \textbf{\textit{AND}}\ \ldots\ \textbf{\textit{AND}}\ x_p\ \textit{es}\ Lx_{kp}$$
$$\textbf{\textit{THEN}}\ y_1\ \textit{es}\ Ly_{k1}\ \textbf{\textit{AND}}\ \ldots\ \textbf{\textit{AND}}\ y_q\ \textit{es}\ Ly_{qk}$$

4. Asignar a la regla anterior un factor de *certeza*, calculado como el producto de los grados de pertenencia a cada Valor Lingüístico.

5. Verificar si en la Base de Reglas existe ya una regla con el mismo antecedente (y quizás distinto consecuente); de ser así, dejar en la Base aquella que tenga un mayor factor de *certeza*. Si aún no hay en la Base de Reglas una regla con el mismo antecedente, adicionar la nueva regla a la Base.

6. repetir el procedimiento para todas las líneas de entrenamiento de la tabla

Existen otras múltiples propuestas en la literatura, algunas de las cuales están expuestas en [8]. Específicamente, la utilización de algoritmos genéticos como el algoritmo de entrenamiento del item 4 se explora en [9] y en [1]. En las secciones 4.3.1 y 4.3.2 se muestran dos alternativas diferentes, ambas basadas en *UNGenético*, para identificar el sistema que se muestra en la sección 4.2

4.2 Descripción del Sistema a Identificar

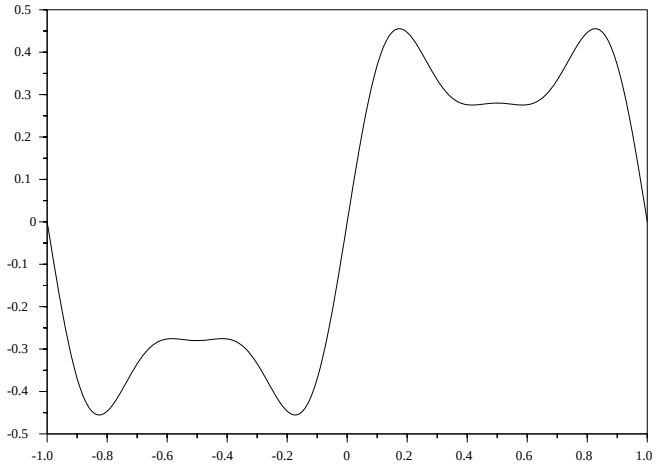
Como ejemplo de aplicación de a propuesto el problema de identificar el sistema no lineal de una entrada y una salida ($p = q = 1$ en la figura 4.3):

$$y = 0.7(0.6sin(x) + 0.3sin(3x) + 0.1sin(5x)) \tag{4.17}$$

que es alimentado por una entrada de la forma

$$x = sin(0.008\pi k) \qquad k = 0, 1, \ldots, 250 \tag{4.18}$$

Figura 4.4: Relación Entrada Salida del sistema



La figura 4.4 muestra la gráfica $yvsx$ correspondiente a la ecuación 4.17. Cuando la entrada tiene el comportamiento de la ecuación 4.18, la salida del sistema es la que se observa en la figura 4.5; los puntos con los que se ha construido esta gráfica se han salvado en el archivo *datos.txt*. Este archivo se empleará como la tabla de datos que aparece en la figura 4.3, con una separación aleatoria de 70% de datos de entrenamiento y 30% de prueba.

El criterio para evaluar la bondad de los modelos será la suma de los errores cuadráticos producidos por el modelo en cada una de las parejas entrada-salida:

$$E = \frac{\sum_i |s_i - \Phi(e_i)|}{\sum_i |s_i|} \tag{4.19}$$

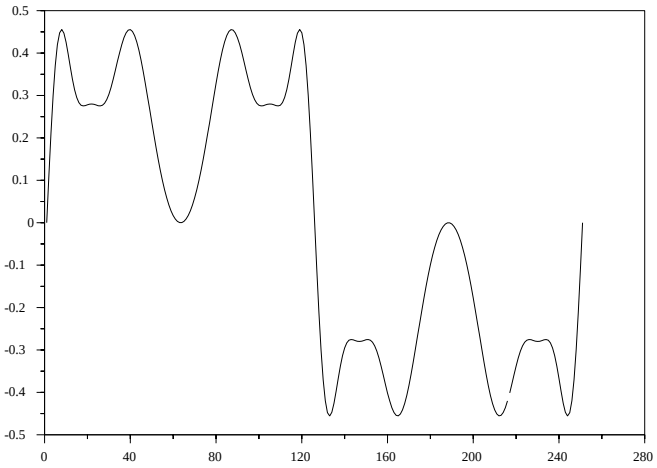
en donde e_i y s_i son la entrada y la salida número i de la tabla, respectivamente, y $\Phi(\cdot)$ es la función que calcula la salida del modelo. La sumatoria se efectuará sobre los datos de entrenamiento o de prueba, según sea el caso.

4.3 Identificación difusa de sistemas

Se proponen dos estrategias distintas de identificación, mediante Sistemas de Lógica Difusa tipo Mamdani optimizados por algoritmos genéticos; ambas propuestas tienen las siguientes características comunes:

- El sistema tiene una entrada y una salida.

Figura 4.5: Salida del sistema cuando la entrada es $x = sin(0.008\pi k) \quad k = 0, 1, \ldots, 250$



- La variable de entrada puede tomar valores en el intervalo $[-1, 1]$.
- La variable de salida puede tomar valores en el intervalo $[-0.5, 0.5]$.
- El bloque difusor es del tipo *singleton*.
- El bloque concresor es del tipo *altura*.
- La T-norma $\otimes$ de la ecuación 4.11, el operador AND de las ecuaciones 4.12, 4.14 y el operador $\Rightarrow$ de la ecuación 4.13 son en todo los casos el mínimo (ecuación 4.6).
- Para cada estrategia de identificación se han probado modelos con distintos números de etiquetas lingüísticas en las variables de entrada y de salida (siempre el mismo número de etiquetas en entrada y salida).
- Los modificadores que aparecen en la ecuación 4.16 se obtienen mediante optimización por algoritmos genéticos.
- El código fuente en C++ para la implementación de los Sistemas de Lógica Difusa ha estado basado en el código generado por la herramienta UNFUZZY (véase [6])

Los puntos en donde difieren las dos estrategias son los siguientes:

- Forma de los conjuntos difusos En la primera estrategia se define estáticamente, mientras que en la segunda se obtiene mediante optimización por algoritmos genéticos.

Figura 4.6: herencia múltiple en la estrategia 1

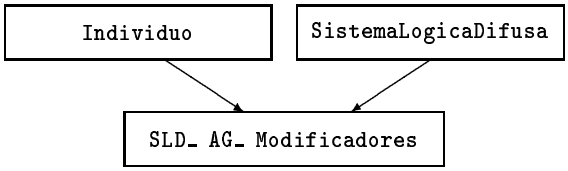
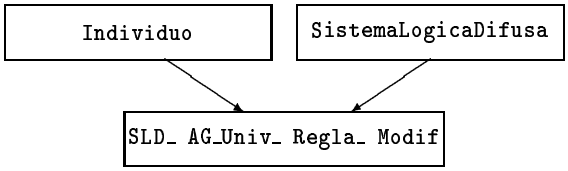


Figura 4.7: herencia múltiple en la estrategia2



- Base de Reglas En la primera estrategia se crea mediante el Algoritmo de Wang y Mendel, mientras que en la segunda se crea mediante optimización por algoritmos genéticos.

En el capítulo 3 se explica que para emplear la librería *UNGenético*, es necesario crear una clase heredera de la clase *Individuo*. Esta clase debe encapsular el comportamiento del objeto a optimizar, que en este caso es un Sistema de Lógica Difusa. La implementación seleccionada aprovecha el código fuente generado por la herramienta UNFUZZY, que provee una clase denominada *SistemaLogicaDifusa*, incluida en los archivos *fuzzy.hpp* y *fuzzy.cpp*.

En cada una de las dos estrategias se ha creado una nueva clase que toma en herencia múltiple las propiedades de las clases *Individuo* de *UNGenético* y *SistemaLogicaDifusa* de UNFUZZY, tal como se visualiza en las figuras 4.6 y 4.7. En las secciones 4.3.1 y 4.3.2 se explican los puntos más importantes de la implementación de las dos estrategias, asi como los resultados de las pruebas; El código fuente completo se ha incluido en el apéndice C.

Las clases herederas que se muestran en las figuras 4.6 y 4.7 son muy parecidas entre si. Algunas de las características comunes de estas clases son las siguientes:

- Contienen apuntadores a tres arreglos en los que está contenida la tabla de datos de la figura 4.3: *Patrones* apunta a la tabla completa, *PatronesEntrenamiento* apunta a una tabla con los datos de entrenamiento, y *PatronesPrueba* apunta a una tabla con los datos de prueba.
- Poseen funciones *objetivo()* y *objetivoTest()* que calculan el error (Ecuación 4.19) en los datos de entrenamiento y de prueba respectivamente.
- Poseen la función *EntrenarUFijo()* que ejecuta el algoritmo de entrenamiento de Wang y Mendel con los datos de entrenamiento.

- Poseen la función *grabaModificadores(char *cad)* que adiciona en el archivo de nombre *cad* los valores que se están optimizando.

4.3.1 Estrategia 1

En esta estrategia la base de datos se crea con el algoritmo de Wang y Mendel, y se busca optimizar los modificadores de la ecuación 4.16. Si el sistema tiene m reglas y p entradas, existirán un total de $m \times p$ modificadores, y todos ellos serán números reales. De esta forma, el genoma estará constituido por $m \times p$ genes reales. Se ha adoptado como rango de valores válidos para cada modificador el intervalo $[0.1, 4.0]$.

La clase *SistemaLogicaDifusa* provee, entre otras muchas, las siguientes funciones:

- *int motor->numeroReglas()*; entrega el número de reglas en la Base de Reglas.
- *int numeroEntradas()*: entrega el número de entradas del sistema.
- *float motor->regla(int i)->modificador(int j)*: entrega el valor del modificador de la regla i, para la entrada j
- *void motor->regla(int i)->modificador(int j, double m)*: hace que el el modificador de la regla i, para la entrada j tome el valor m.

Con las anteriores funciones pueden entonces definirse las funciones *codificar()* y *decodificar()* que son obligatorias para emplear *UNGenético*. Estas funciones se muestran en las figuras 4.8 y 4.9

Esta estrategia se ha probado para 3, 5, 7, 9 y 25 etiquetas en las variables lingüísticas de entrada y de salida. En las figuras[5] 4.3.1 a 4.3.1 se muestran las relaciones entrada-salida de los distintos modelos antes y después de la optimización. Por otra parte, en las figuras 4.3.1 a 4.3.1 se muestra la salida de los modelos con una entrada como la de la ecuación 4.18, antes y despues de la optimización; en estas figuras se ha dibujado también el comportamiento real del sistema (ecuaciones 4.17 y 4.18, figuras 4.4 y 4.5), para efectos de comparación.

Es posible observar cómo en todos los casos el algoritmo genético acerca el comportamiento de los modelos real del sistema. También es posible observar que el poder de representación numérica del sistema de lógica difusa se incrementa al tomar más conjuntos difusos [6].

Vale la pena recordar que en esta estrategia sólo se están optimizando los modificadores lingüísticos, y que éstos sólo afectan las zonas de transición entre una regla y otra. Puede notarse cómo los modelos antes de la optimización son modelos a trazos rectos, que se suavizan al optimizar los modificadores.

Cuando un conjunto difuso tiene un intervalo zona cuya función de pertenencia es 1, como en los conjuntos *Bajo* y *Alto* de la figura 4.2, los modificadores lingüísticos no alteran para nada su incidencia en el sistema ($1.0^{mod} = 1.0$). Este hecho puede verse en los trozos rectos horizontales que están al comienzo y al final de las curvas en las figuras

[5] Las gráficas se han obtenido con el programa *gnuplot* Copyright(C) 1986 - 1993 Thomas Williams, Colin Kelley, con licencia pública GNU

[6] esto sucede en detrimento de su poder de representación lingüístico, ya que no es fácil asignar más de siete etiquetas lingüísticas interpretables.

Figura 4.8: función *codificar()* en la Estrategia 1

```
void SLD_AG_Modificadores::codificar()
{
  int i,j,tam,tam2,cont=0;
  tam=motor->numeroReglas();
  for(i=0;i<tam;i++)
  {
    tam2=numeroEntradas();
    for(j=0;j<tam2;j++)
    {
      GenReal *g;
      g=(GenReal*)Genoma.dato(cont);
      g->Valor=motor->regla(i)->modificador(j);
      cont++;
    }
  }
}
```

4.3.1 a 4.3.1, que no se ven afectados por el proceso de optimización. Estas zonas son las que dificultan que la salida de los modelos difiera de la salida del sistema real en los dos picos que se observan en las figuras 4.3.1 a 4.3.1; tan sólo el modelo de 25 conjuntos logra hacer una adecuada simulación.

4.3.2 Estrategia 2

En esta estrategia se busca optimizar simultáneamente la forma de los conjuntos difusos, la Base de Reglas, y los modificadores lingüísticos. La definición de los conjuntos difusos es particularmente crítica, ya que es necesario impedir que ocurran algunos inconvenientes. Para explicar este hecho, supóngase que se desea optimizar la definición de los conjuntos de la variable lingüística *Estatura* que aparece en la figura 4.2; Al modificar la forma de estos conjuntos pueden darse los siguientes casos no deseados:

- zonas indefinidas (figura 4.20): Si una región del espacio de entrada no tiene ninguna etiqueta que la califique, no estará cubierta por ninguna regla y por lo tanto el modelo no podrá simular el comportamiento del sistema en esta región.

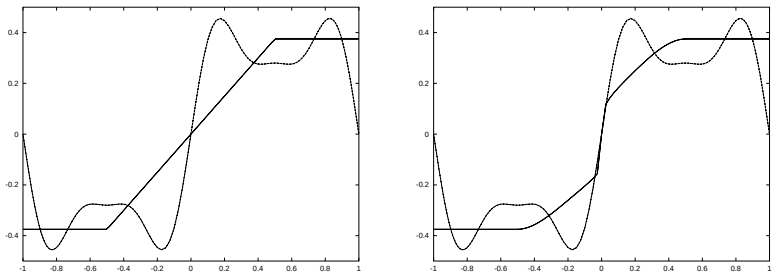
- conjuntos subsumidos(figura 4.21): Si un conjunto queda contenido dentro de otro, está región quedará doblemente etiquetada, y existirán reglas igualmente válidas (y posiblemente contradictorias) para la región.

- etiquetas ilógicas(figura 4.22): Es posible que la Base de Reglas pierda todo su significado si las etiquetas lingüísticas cambian de orden (*Medio* mayor que *Alto*).

Figura 4.9: función *decodificar()* en la Estrategia 1

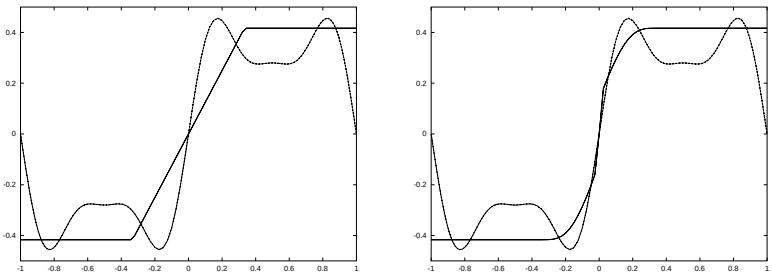
```
void SLD_AG_Modificadores::decodificar()
{
  int i,j,tam,tam2,cont=0;
  tam=motor->numeroReglas();
  for(i=0;i<tam;i++)
  {
    tam2=numeroEntradas();
    for(j=0;j<tam2;j++)
    {
      GenReal *g;
      g=(GenReal*)Genoma.dato(cont);
      motor->regla(i)->modificador(j,g->Valor);
      cont++;
    }
  }
}
```

Figura 4.10: Relación Entrada-Salida del modelo con 3 etiquetas. Estrategia 1



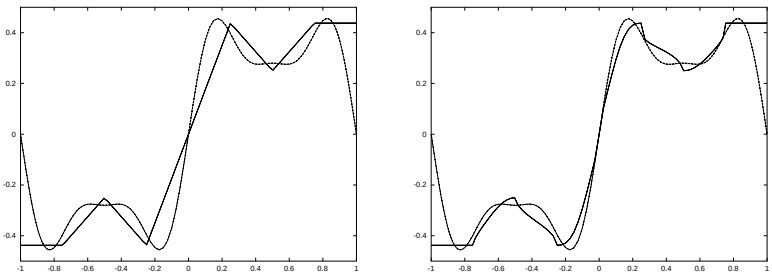
(a) Antes de optimización (b) Despues de optimización

Figura 4.11: Relación Entrada-Salida del modelo con 5 etiquetas. Estrategia 1



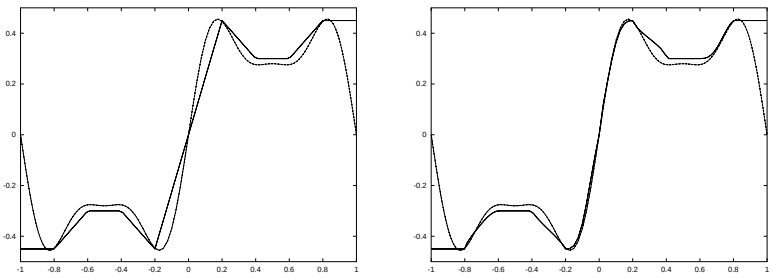
(a) Antes de optimización (b) Despues de optimización

Figura 4.12: Relación Entrada-Salida del modelo con 7 etiquetas. Estrategia 1



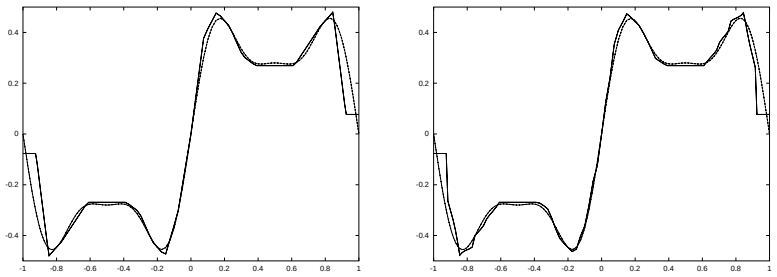
(a) Antes de optimización (b) Despues de optimización

Figura 4.13: Relación Entrada-Salida del modelo con 9 etiquetas. Estrategia 1



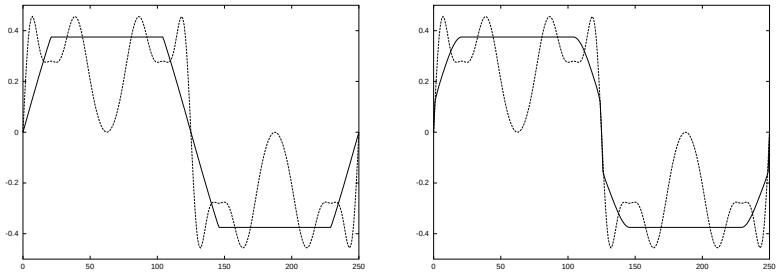
(a) Antes de optimización (b) Despues de optimización

Figura 4.14: Relación Entrada-Salida del modelo con 25 etiquetas. Estrategia 1



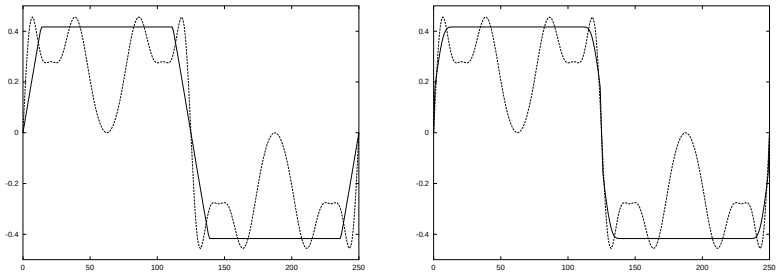
(a) Antes de optimización (b) Despues de optimización

Figura 4.15: Salida del modelo con 3 etiquetas. Estrategia 1



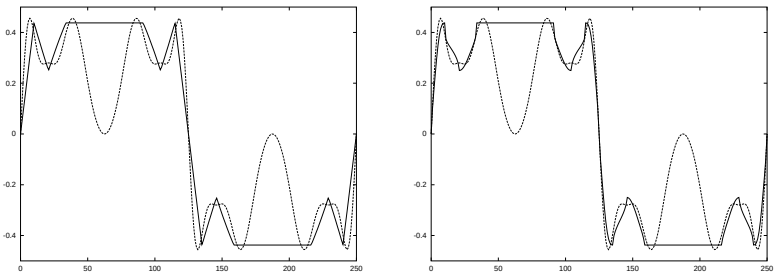
(a) Antes de optimización (b) Despues de optimización

Figura 4.16: Salida del modelo con 5 etiquetas. Estrategia 1



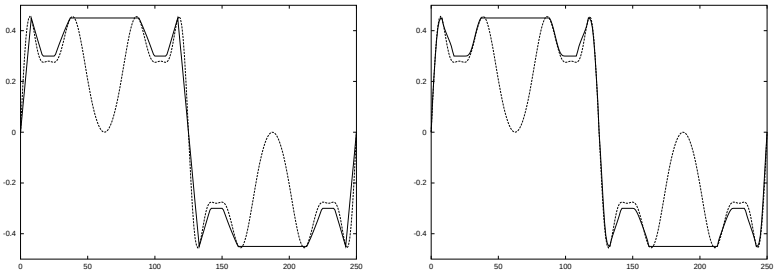
(a) Antes de optimización (b) Despues de optimización

Figura 4.17: Salida del modelo con 7 etiquetas. Estrategia 1



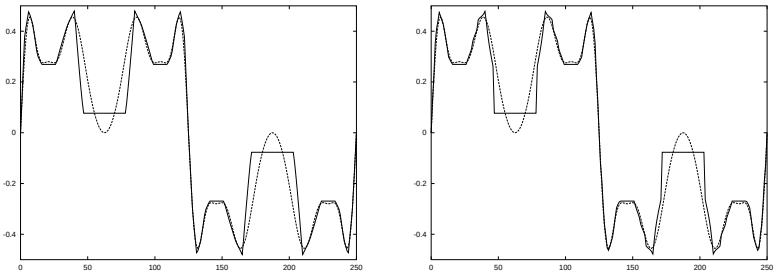
(a) Antes de optimización (b) Despues de optimización

Figura 4.18: Salida del modelo con 9 etiquetas. Estrategia 1



(a) Antes de optimización (b) Despues de optimización

Figura 4.19: Salida del modelo con 25 etiquetas. Estrategia 1



(a) Antes de optimización (b) Despues de optimización

Por esta razón es necesario restringir la forma que los conjuntos difusos puedan tomar, para asegurar que el espacio de entrada tenga una *partición*[7]. Una posibilidad para lograrlo consiste en definir la forma de los conjuntos con ciertos parámetros, según se explica en el siguiente procedimiento, referido a la figura 4.23:

1. Si el número de etiquetas a definir es E, dividir el universo de discurso en $N = 2E-1$ intervalos de igual tamaño. Cada intervalo tendrá un ancho Δ, y estarán limitado por los puntos $p_1, p_2, \ldots, p_{N-2}$.

2. Restringir la forma de los conjuntos difusos a trapecios, en forma tal que los vértices de las rampas de un conjunto coincidan en ubicación con los del conjunto siguiente.

3. Denominar a los puntos en donde aparecen los vértices de los trapecios $v_1, v_2, \ldots, v_{N-2}$.

4. Restringir los valores de $v_1, v_2, \ldots, v_{N-2}$ de forma tal que $\delta_i = |v_i - p_i| \leq \frac{\Delta}{2}$.

Esta estrategia no sólo asegura la existencia de una partición correcta, sino que además parametriza la forma de los conjuntos en función de un número pequeño de parámetros $\delta_0, \delta_1, \ldots, \delta_{N-2}$, que pueden ser optimizados.

En cuanto a la Base de Reglas, la estrategia de codificación empleada ha consistido en definir una base completa de reglas (una regla por cada etiqueta de las variables de entrada) y permitr que el algoritmo busque cuál es el consecuente adecuado para cada regla. La etiqueta del consecuente se determina por un número entero *et* que indica cuál es la etiqueta correspondiente de la variable lingüística de salida. Los modificadores se han codificado de la misma forma que en la estrategia 1.

De acuerdo a lo anterior, el genoma de cada individuo deberá contener:

- El conjunto de parámetros que definen la forma de los conjuntos $\delta_0, \delta_1, \ldots, \delta_{N-2}$, mediante genes en codificación real.

- El conjunto de etiquetas de los consecuentes de la base de reglas, mediante genes en codificación entera.

- El conjunto de modificadores lingüísticos, mediante genes en codificación real.

Esta estrategia se ha probado para 3, 5, 7 y 9 etiquetas en las variables lingüísticas de entrada y de salida. En las figuras[8] 4.3.2 a 4.3.2 se muestran las relaciones entrada-salida de los distintos modelos antes y después de la optimización. Por otra parte, en las figuras 4.3.2 a 4.3.2 se muestra la salida de los modelos con una entrada como la de la ecuación 4.18, antes y despues de la optimización; en estas figuras se ha dibujado también el comportamiento real del sistema (ecuaciones 4.17 y 4.18, figuras 4.4 y 4.5), para efectos de comparación.

Al comparar en estas figuras el comportamient antes de la optimización respecto al mismo tipo de comportamiento en la estrategia 1, se hace evidente que en este caso se está partiendo de sistemas de lógica difusa con una pobre definición (No existe una

[7]Una partición es una definición de conjuntos difusos tales que la sumatoria de las funciones de pertenencia de todo elemento del universo de discurso a la totalidad de los conjuntos es 1.0

[8]Las gráficas se han obtenido con el programa *gnuplot* Copyright(C) 1986 - 1993 Thomas Williams, Colin Kelley, con licencia pública GNU

Figura 4.20: Zonas indefinidas

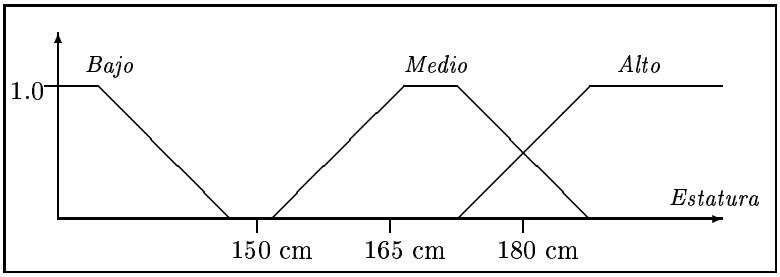
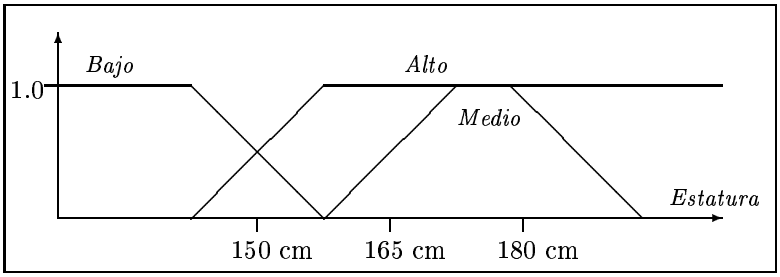


Figura 4.21: Conjuntos subsumidos



base de reglas!). No obstante, el algoritmo logra reproducir bastante bien la salida del sistema con menos etiquetas que en la estrategia 1 (véase la figura 4.3.2); esto es debido principalmente a que el algoritmo puede modificar la forma de los conjuntos, que en la anterior estrategia están predefinidos, y asi definir más adecuadamente la región en que actúa cada una de las reglas.

Figura 4.22: Etiquetas ilógicas

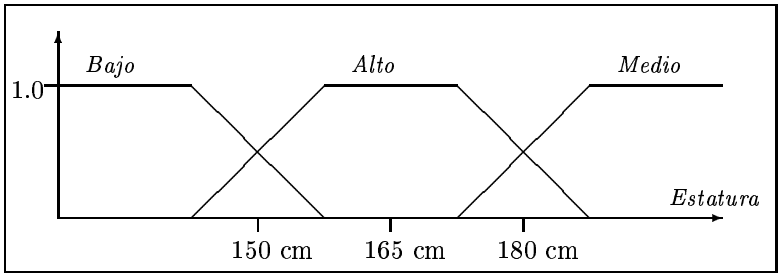


Figura 4.23: Partición parametrizada

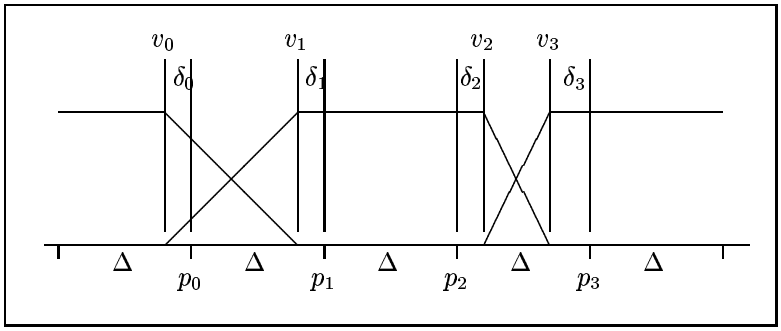
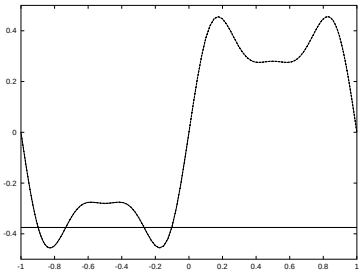
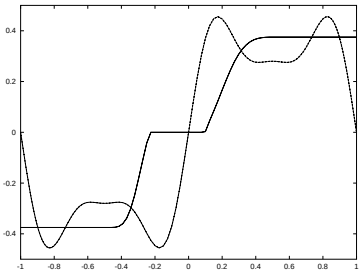


Figura 4.24: Relación Entrada-Salida del modelo con 3 etiquetas. Estrategia 2

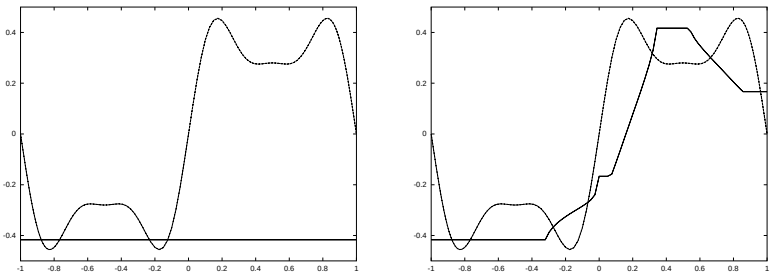


(a) Antes de optimización



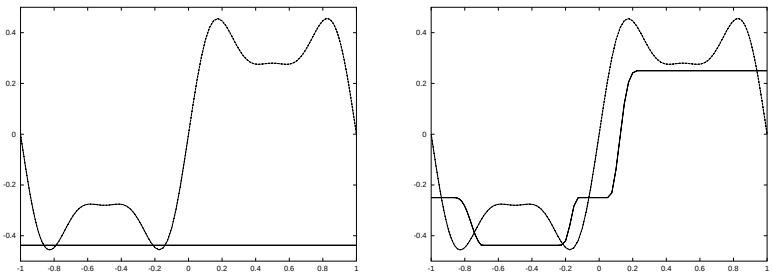
(b) Despues de optimización

Figura 4.25: Relación Entrada-Salida del modelo con 5 etiquetas. Estrategia 2



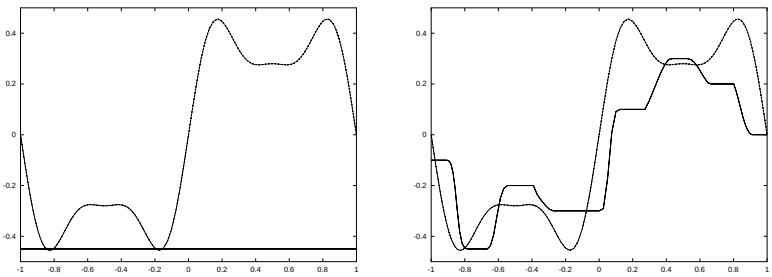
(a) Antes de optimización (b) Despues de optimización

Figura 4.26: Relación Entrada-Salida del modelo con 7 etiquetas. Estrategia 2



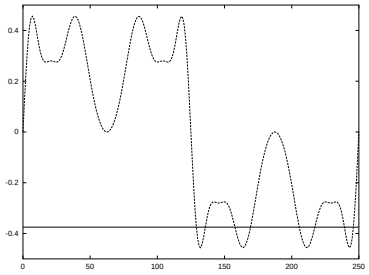
(a) Antes de optimización (b) Despues de optimización

Figura 4.27: Relación Entrada-Salida del modelo con 9 etiquetas. Estrategia 2

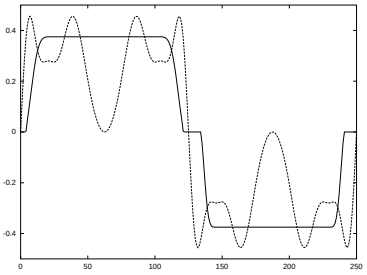


(a) Antes de optimización (b) Despues de optimización

Figura 4.28: Salida del modelo con 3 etiquetas. Estrategia 2

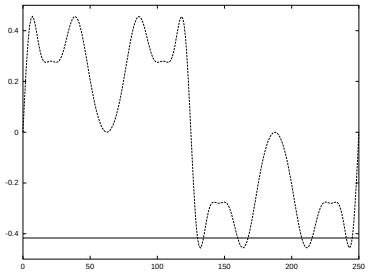


(a) Antes de optimización

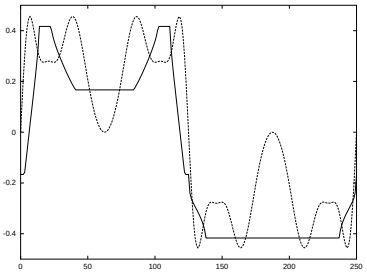


(b) Despues de optimización

Figura 4.29: Salida del modelo con 5 etiquetas. Estrategia 2

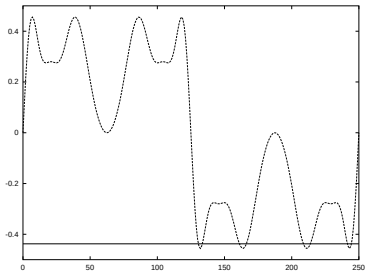


(a) Antes de optimización

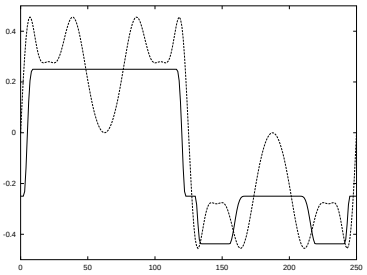


(b) Despues de optimización

Figura 4.30: Salida del modelo con 7 etiquetas. Estrategia 2

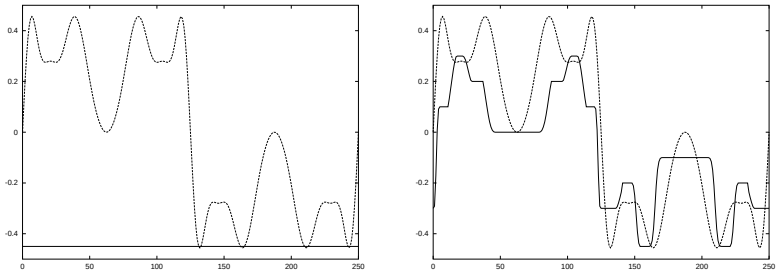


(a) Antes de optimización



(b) Despues de optimización

Figura 4.31: Salida del modelo con 9 etiquetas. Estrategia 2



(a) Antes de optimización

(b) Despues de optimización

Capítulo 5

Trabajos futuros

Los ejemplos de los capítulos 3 y 4 muestran que *UNGenético* es una librería completa para la implementación de Algoritmos Genéticos con la estrategia de cofificación híbrida propuesta en la sección 1.3. No obstante, esta primera versión de la librería puede ser ampliada en diversos sentidos. A continuación se presenta una lista de los trabajos que deberán abordarse para crear en el futuro una nueva versión:

Nuevos tipos de genes: Si bien es cierto que los tipos de genes actualmente disponibles (Real, Booleano y Entero) son los tipos básicos que permiten codificar cualquier tipo de variable, tambien es cierto que contar con otros tipos de genes puede facilitar la aplicación de la librería al facilitar la definición de la clase heredera de *Individuo*. Algunos de los tipos de genes candidatos a ser implementados en posteriores versiones son los siguientes: Arreglos de longitud fija, Arreglos de longitud variable, matrices n-dimensionales de tamaño fijo y variable, e instrucciones de programación (éstas últimas para abordar problemas de programación genética).

Nuevos Operadores: El espectro de operadores es bastante amplio, pero en la literatura existe una gran cantidad de operadores de todo tipo (probabilidad, selección, etc.), cada uno de ellos con propiedades que pueden ser útiles para una aplicación dada. Además, cada nuevo tipo de gen que se defina implica la definición de los operadores de cruce y mutación correspondientes.

Estrategias de adaptación de AG: La clase *AlgoritmoGenetico* provee la función *adaptacion()*, que actualmente no esta en uso, pero que permitirá en futuras versiones implementar operadores de adaptación del AG, es decir, de operadores que modifiquen algunas de las características del algoritmo (por ejemplo la probabilidad de mutación) en tiempo de ejecución, y en función de alguna medida de desmpeño como las medidas OnLine y OffLine

Computación paralela: Cuando el problema a optimizar es muy complejo, puede ser conveniente emplear varios computadores simultáneamente para aumentar la velocidad de ejecución del algoritmo. Esta estrategia de computación paralela no está incluida en la primera versión de *UNGenético*, pero mediante los esquemas de redes disponibles en la actualidad podrá estarlo en futuras versiones.

Algoritmos distribuidos: Una de las tendencias actuales de investigación en los AG consiste en subdividir la población total en distintos nichos, y permitir que cada uno de ellos evolucione independientemente (en forma distribuida); eventualmente, algunos de los individuos de un nicho se mezclan con los de otro. Esta estrategia puede ser implementada en futuras versiones.

Librería de ejemplos: Conviene disponer de una librería de ejemplos de aplicación adicionales a los de los capítulos 3 y 4. El propósito de esta librería sería el de facilitar la comprensión de cómo emplear *UNGenético* a aquellas personas que la estén conociendo.

Difusión de la librería: *UNGenético* es un producto de *software libre*, y por tanto deberá estar disponible en Internet. La difusión de esta librería dentro de la comunidad internacional, empleando para ello estrategias como anuncios en las listas de discusión, presentación en revistas y congresos, etc. Los comentarios y sugerencias que hagan los usuarios de la librería serán los verdaderos indicadores de su efectividad, y las guías para la elaboración de versiones futuras.

Apéndice A

Código fuente

A.1 *Genetico.hpp*

```
#ifndef _GENETICO_HPP
#define _GENETICO_HPP
#endif

#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
#include<iostream.h>
#include<math.h>

#ifndef __ARREGLOS_HPP
#include"arreglos.hpp"
#endif

class Gen;
class OperadorMutacion;
class OperadorCruce;
class OperadorProbabilidad;
class OperadorParejas;
class OperadorReproduccion;
class OperadorSeleccion;
class Individuo;
class Poblacion;
class AlgoritmoGenetico;

class Gen
{
public:
```

```
  Gen(){AG=NULL;}
  ~Gen(){}
  AlgoritmoGenetico *AG;

  void ag(AlgoritmoGenetico *Ag);

  virtual void crearAleatorio()=0;
  virtual OperadorMutacion *operadorMutacionDefecto()=0;
  virtual OperadorCruce *operadorCruceDefecto()=0;

  int operator==(const Gen &other)
  {
    return (this==&other);
  }
};

class Individuo
{
public:
  Individuo();
  ~Individuo();
  AlgoritmoGenetico *AG;

  Arreglo<Gen> Genoma;
  double Objetivo;
  double Probabilidad;
  Individuo *Pareja;

  void calcularObjetivo()
  {
    decodificar();
    Objetivo=objetivo();
  }
  void copiar(Individuo *other)
  {
    Objetivo=other->Objetivo;
    Probabilidad=other->Probabilidad;
    copiarDetalles(other);
  }

  virtual Individuo* crearCopia()=0;
  virtual void copiarDetalles(Individuo *other)=0;
  virtual double objetivo()=0;
  virtual void codificar()=0;
  virtual void decodificar()=0;
  virtual void crearOperadores(Arreglo<OperadorMutacion> *LM,
```

```
          Arreglo<OperadorCruce> *LC);
  void crearAleatorio();
  void mutar();
  void ag(AlgoritmoGenetico *Ag);

  int operator==(const Individuo &other)
  {
    return (this==&other);
  }

};

class Poblacion
{
public:
  Poblacion();
  ~Poblacion();
  AlgoritmoGenetico *AG;

  Individuo *Modelo;
  Individuo *Elite;
  Arreglo<Individuo> Generacion;
  double SumaObjetivo;

  void crearGeneracion(int tam);
  void crearGeneracionAleatoria();
  void calcularObjetivo();
  void ordenar(int descendente=0);
  void sumaObjetivo();
  void mutar();
  void buscaElite(int Maximizar);
  void elitismo();

  void ag(AlgoritmoGenetico *Ag);

};

class AlgoritmoGenetico
{
public:
  AlgoritmoGenetico();
  ~AlgoritmoGenetico();

  Poblacion GeneracionActual;
  OperadorProbabilidad *OpProbabilidad;
  OperadorSeleccion *OpSeleccion;
```

```
OperadorParejas *OpParejas;
OperadorReproduccion *OpReproduccion;
Arreglo<OperadorMutacion> *ListaOperadorMutacion;
Arreglo<OperadorCruce> *ListaOperadorCruce;
int Elitismo;
int Maximizar;
int TamanoPoblacion;

Individuo *MejorEnEstaGeneracion;
Individuo *PeorEnEstaGeneracion;
Individuo *MejorEnLaHistoria;
int Conteo;
int Generacion;
int GeneracionMaxima;
int GeneracionDelMejorEnLaHistoria;
double EvaluacionMedia;
double MedidaOnLine;
double MedidaOffLine;
double AcumuladoOnLine;
double AcumuladoOffLine;
double Desviacion;

int IndicadorArchivo;
int IntervaloSalvar;
FILE *arch;
char NombreArchivo[400];
int IndicadorMostrar;
int IndicadorMostrarGeneracion;
int IndicadorMostrarMejorEnHistoria;
int IndicadorMostrarGeneracionMejorHistorico;
int IndicadorMostrarMejorEnGeneracion;
int IndicadorMostrarPeorEnGeneracion;
int IndicadorMostrarMedia;
int IndicadorMostrarDesviacion;
int IndicadorMostrarOnLine;
int IndicadorMostrarOffLine;
int CriterioParada; //0=GeneracionMaxima

void asignarProbabilidad(int descendente);
void seleccionar();
void asignarParejas();
void reproducir();
void mutar();
void calcularObjetivo();
void actualizarMedidas();
void buscaElite();
```

```
  void elitismo();
  void optimizar();
  void iniciarOptimizacion();
  void iterarOptimizacion();
  void finalizarOptimizacion();
  int parada();

  virtual void adaptacion(){}

  void modelo(Individuo *Ind);

  void mostrarMedidas();
  void salvar();
};

class OperadorMutacion
{
public:
  OperadorMutacion()
  {
    AG=NULL;
      ProbabilidadMutacion=0.1;
  }
  ~OperadorMutacion(){}
  AlgoritmoGenetico *AG;

  double ProbabilidadMutacion;

  void mutar(Gen *g);
  virtual void mutarGen(Gen *g)=0;

  int operator==(const OperadorMutacion &other)
  {
    return (this==&other);
  }
};

class OperadorCruce
{
public:
  OperadorCruce(){AG=NULL;}
  ~OperadorCruce(){}
  AlgoritmoGenetico *AG;

  void cruzar(Gen *madre,Gen *padre, Arreglo<Gen> *hijos,
          int numHijos);
```

```
  virtual void cruzarGenes(Gen *madre,Gen *padre, Arreglo<Gen>
          *hijos,int numHijos)=0;

  int operator==(const OperadorCruce &other)
  {
    return (this==&other);
  }
};

class OperadorProbabilidad
{
public:
  OperadorProbabilidad(){AG=NULL;}
  ~OperadorProbabilidad(){}
  AlgoritmoGenetico *AG;

  virtual void asignarProbabilidad(int descendente)=0;

  Poblacion *Pob;
};

class OperadorSeleccion
{
public:
  OperadorSeleccion(){AG=NULL;}
  ~OperadorSeleccion(){}
  AlgoritmoGenetico *AG;

  virtual void seleccionar()=0;

  Poblacion *Pob;
};

class OperadorParejas
{
public:
  OperadorParejas(){AG=NULL;}
  ~OperadorParejas(){}
  AlgoritmoGenetico *AG;

  virtual void asignarParejas()=0;

  Poblacion *Pob;
};

class OperadorReproduccion
```

```
{
public:
  OperadorReproduccion(){AG=NULL;}
  ~OperadorReproduccion(){}
  AlgoritmoGenetico *AG;

  virtual void reproducir()=0;

  Poblacion *Pob;
};

class OperadorReproduccionDosPadresDosHijos:public OperadorReproduccion
{
public:
  OperadorReproduccionDosPadresDosHijos(){}
  ~OperadorReproduccionDosPadresDosHijos(){}

  void reproducir();
};

class OperadorProbabilidadProporcional:public OperadorProbabilidad
{
public:
  OperadorProbabilidadProporcional(){}
  ~OperadorProbabilidadProporcional(){}

  void asignarProbabilidad(int descendente);
};

class OperadorSeleccionEstocasticaRemplazo:public OperadorSeleccion
{
public:
  OperadorSeleccionEstocasticaRemplazo(){}
  ~OperadorSeleccionEstocasticaRemplazo(){}

  void seleccionar();
};

class OperadorParejasAleatoria:public OperadorParejas
{
public:
  OperadorParejasAleatoria(){}
  ~OperadorParejasAleatoria(){}

  void asignarParejas();
};
```

```
class OperadorReproduccionCrucePlano:public OperadorReproduccion
{
public:
  OperadorReproduccionCrucePlano(){}
  ~OperadorReproduccionCrucePlano(){}

  void reproducir();
};
```

A.2 *Genetico.cpp*

```
#ifndef _GENETICO_CPP
#define _GENETICO_CPP
#endif

#ifndef _GENETICO_HPP
#include"genetico.hpp"
#endif

void Gen::ag(AlgoritmoGenetico *Ag)
{
  AG=Ag;
}

Individuo::Individuo()
{
  Probabilidad=0.0;
  Pareja=NULL;
  AG=NULL;
}

Individuo::~Individuo()
{
  Pareja=NULL;
}

void Individuo::mutar()
{
  int i,tam;
  tam=Genoma.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    AG->ListaOperadorMutacion->dato(i)->mutar(Genoma.dato(i));
  }
```

```
}

void Individuo::crearOperadores(Arreglo<OperadorMutacion> *LM,
          Arreglo<OperadorCruce> *LC)
{
  OperadorMutacion *OM;
  OperadorCruce *OC;
  int i,tam;
  tam=Genoma.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    OM=Genoma.dato(i)->operadorMutacionDefecto();
    OC=Genoma.dato(i)->operadorCruceDefecto();
    LM->Add(OM);
    LC->Add(OC);
  }
}

void Individuo::crearAleatorio()
{
  int i,tam;
  tam=Genoma.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    Genoma.dato(i)->crearAleatorio();
  }
}

void Individuo::ag(AlgoritmoGenetico *Ag)
{
  AG=Ag;
  int i,tam;
  tam=Genoma.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    Genoma.dato(i)->ag(AG);
  }
}

Poblacion::Poblacion()
{
  Modelo=NULL;
  Elite=NULL;
  AG=NULL;
}
```

```
Poblacion::~Poblacion()
{
  Modelo=NULL;
  Elite=NULL;
  AG=NULL;
  Generacion.FlushDestroy();
}

void Poblacion::crearGeneracion(int tam)
{
  int i;
  Generacion.FlushDestroy();
  for(i=0;i<tam;i++)
  {
    Individuo *Ind;
    Ind=Modelo->crearCopia();
    Ind->ag(AG);
    Generacion.Add(Ind);
  }
}

void Poblacion::crearGeneracionAleatoria()
{
  int i,tam;
  tam=Generacion.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    Generacion.dato(i)->crearAleatorio();
  }
}

void Poblacion::mutar()
{
  int i,tam;
  tam=Generacion.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    Generacion.dato(i)->mutar();
  }
}

void Poblacion::calcularObjetivo()
{
  int i,tam;
  tam=Generacion.GetItemsInContainer();
  for(i=0;i<tam;i++)
```

```
  {
    Generacion.dato(i)->calcularObjetivo();
  }
  Arreglo<Individuo> Temp;

}

void Poblacion::ordenar(int descendente)
{
  int i,j,tam,tam2;
  calcularObjetivo();

  Arreglo<Individuo> Temp;
  tam=Generacion.GetItemsInContainer();
  if(tam<=0){return;}
  for(j=0;j<tam;j++)
  {
    int ItemExtremo;
    double Extremo;
    tam2=Generacion.GetItemsInContainer();
    if(tam2<=0){return;}
    Extremo=Generacion.dato(0)->Objetivo;
    ItemExtremo=0;
    for(i=0;i<tam2;i++)
    {
      if(descendente)
      {
        if(Generacion.dato(i)->Objetivo>Extremo)
        {
          Extremo=Generacion.dato(i)->Objetivo;
          ItemExtremo=i;
        }
      }else
      {
        if(Generacion.dato(i)->Objetivo<Extremo)
        {
          Extremo=Generacion.dato(i)->Objetivo;
          ItemExtremo=i;
        }
      }
    }
    Temp.Add(Generacion.dato(ItemExtremo));
    Generacion.Detach(ItemExtremo);
  }
  tam=Temp.GetItemsInContainer();
  for(i=0;i<tam;i++)
```

```
  {
    Generacion.Add(Temp.dato(i));
  }
  Temp.FlushDetach();

}

void Poblacion::sumaObjetivo()
{
  SumaObjetivo=0;
  int i,tam;
  tam=Generacion.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    SumaObjetivo+=Generacion.dato(i)->Objetivo;
  }
}

void Poblacion::buscaElite(int Maximizar)
{
  int i,tam;
  tam=Generacion.GetItemsInContainer();
  int ItemExtremo;
  double Extremo;
  if(tam<=0){return;}
  Extremo=Generacion.dato(0)->Objetivo;
  ItemExtremo=0;

  for(i=0;i<tam;i++)
  {
    if(Maximizar)
    {
      if(Generacion.dato(i)->Objetivo>Extremo)
      {
        Extremo=Generacion.dato(i)->Objetivo;
        ItemExtremo=i;
      }
    }else
    {
      if(Generacion.dato(i)->Objetivo<Extremo)
      {
        Extremo=Generacion.dato(i)->Objetivo;
        ItemExtremo=i;
      }
    }
  }
```

```
  Elite->copiar(Generacion.dato(ItemExtremo));
}

void Poblacion::elitismo()
{
  Generacion.dato(Generacion.GetItemsInContainer()-1)->copiar(Elite);
}

void Poblacion::ag(AlgoritmoGenetico *Ag)
{
  AG=Ag;
  Modelo->ag(Ag);
   Elite->ag(Ag);
  int i,tam;
  tam=Generacion.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    Generacion.dato(i)->ag(AG);
  }
}

AlgoritmoGenetico::AlgoritmoGenetico()
{
  GeneracionActual.AG=this;

  MejorEnEstaGeneracion=NULL;
  PeorEnEstaGeneracion=NULL;
  MejorEnLaHistoria=NULL;
  ListaOperadorMutacion=NULL;
  ListaOperadorCruce=NULL;

  OpProbabilidad=new OperadorProbabilidadProporcional;
  OpSeleccion=new OperadorSeleccionEstocasticaRemplazo;
  OpParejas=new OperadorParejasAleatoria;
  OpReproduccion=new OperadorReproduccionDosPadresDosHijos;

  OpProbabilidad->Pob=&GeneracionActual;
  OpSeleccion->Pob=&GeneracionActual;
  OpParejas->Pob=&GeneracionActual;
  OpReproduccion->Pob=&GeneracionActual;

  OpProbabilidad->AG=this;
  OpSeleccion->AG=this;
  OpParejas->AG=this;
  OpReproduccion->AG=this;
```

```
  Elitismo=1;
  Maximizar=0;

  Generacion=0;
  GeneracionMaxima=100;
  TamanoPoblacion=10;
  GeneracionDelMejorEnLaHistoria=0;
  EvaluacionMedia=0.0;
  MedidaOnLine=0.0;
  MedidaOffLine=0.0;
  AcumuladoOnLine=0.0;
  AcumuladoOffLine=0.0;
  Desviacion=0.0;

  IndicadorArchivo=1;
  IntervaloSalvar=10;
  sprintf(NombreArchivo,"prueba.txt");
  IndicadorMostrar=0;
  IndicadorMostrarGeneracion=1;
  IndicadorMostrarMejorEnHistoria=1;
  IndicadorMostrarGeneracionMejorHistorico=1;
  IndicadorMostrarMejorEnGeneracion=1;
  IndicadorMostrarPeorEnGeneracion=1;
  IndicadorMostrarMedia=1;
  IndicadorMostrarDesviacion=1;
  IndicadorMostrarOnLine=1;
  IndicadorMostrarOffLine=1;
  CriterioParada=0;
}

AlgoritmoGenetico::~AlgoritmoGenetico()
{
  MejorEnEstaGeneracion=NULL;
  PeorEnEstaGeneracion=NULL;
  MejorEnLaHistoria=NULL;

  delete OpProbabilidad;
  delete OpSeleccion;
  delete OpParejas;
  delete OpReproduccion;
}

void AlgoritmoGenetico::modelo(Individuo *Ind)
{
  if(Ind==NULL){return;}
  if(ListaOperadorMutacion==NULL)
```

```
{
  ListaOperadorMutacion=new Arreglo<OperadorMutacion>;
}else
{
  ListaOperadorMutacion->FlushDestroy();
}
if(ListaOperadorCruce==NULL)
{
  ListaOperadorCruce=new Arreglo<OperadorCruce>;
}else
{
  ListaOperadorCruce->FlushDestroy();
}
Ind->crearOperadores(ListaOperadorMutacion,ListaOperadorCruce);

int i,tam;
tam=ListaOperadorMutacion->GetItemsInContainer();
for(i=0;i<tam;i++)
{
  ListaOperadorMutacion->dato(i)->AG=this;
}
tam=ListaOperadorCruce->GetItemsInContainer();
for(i=0;i<tam;i++)
{
  ListaOperadorCruce->dato(i)->AG=this;
}

if(GeneracionActual.Modelo!=NULL)
{
  delete GeneracionActual.Modelo;
}
if(GeneracionActual.Elite!=NULL)
{
  delete GeneracionActual.Elite;
}
if(MejorEnEstaGeneracion!=NULL)
{
  delete MejorEnEstaGeneracion;
}
if(PeorEnEstaGeneracion!=NULL)
{
  delete PeorEnEstaGeneracion;
}
if(MejorEnLaHistoria!=NULL)
{
  delete MejorEnLaHistoria;
```

```
  }

  GeneracionActual.Modelo=Ind;
  GeneracionActual.Elite=GeneracionActual.Modelo->crearCopia();
  MejorEnEstaGeneracion=GeneracionActual.Modelo->crearCopia();
  PeorEnEstaGeneracion=GeneracionActual.Modelo->crearCopia();
  MejorEnLaHistoria=GeneracionActual.Modelo->crearCopia();

  GeneracionActual.Modelo->AG=this;
  GeneracionActual.Elite->AG=this;
  MejorEnEstaGeneracion->AG=this;
  PeorEnEstaGeneracion->AG=this;
  MejorEnLaHistoria->AG=this;
}

void AlgoritmoGenetico::asignarProbabilidad(int descendente)
{
  if(OpProbabilidad!=NULL)
  {
    OpProbabilidad->asignarProbabilidad(descendente);
  }
}

void AlgoritmoGenetico::seleccionar()
{
  if(OpSeleccion!=NULL)
  {
    OpSeleccion->seleccionar();
  }
}

void AlgoritmoGenetico::asignarParejas()
{
  if(OpParejas!=NULL)
  {
    OpParejas->asignarParejas();
  }
}

void AlgoritmoGenetico::reproducir()
{
  if(OpReproduccion!=NULL)
  {
    OpReproduccion->reproducir();
  }
}
```

```
void AlgoritmoGenetico::mutar()
{
  GeneracionActual.mutar();
}

void AlgoritmoGenetico::buscaElite()
{
  if(Elitismo)
  {
    GeneracionActual.buscaElite(Maximizar);
  }
}

void AlgoritmoGenetico::elitismo()
{
  if(Elitismo)
  {
    GeneracionActual.elitismo();
  }
}

void AlgoritmoGenetico::calcularObjetivo()
{
  GeneracionActual.calcularObjetivo();
}

void AlgoritmoGenetico::actualizarMedidas()
{
  int i;
  double AcumuladoMedia=0;
  double mejor,peor;
  int LugarMejor,LugarPeor;
  mejor=peor=GeneracionActual.Generacion.dato(0)->Objetivo;
  LugarMejor=LugarPeor=0;

  for(i=0;i<GeneracionActual.Generacion.GetItemsInContainer();i++)
  {
    double valor;
    valor=GeneracionActual.Generacion.dato(i)->Objetivo;

    AcumuladoMedia+=valor;
    if(((valor<mejor)&&(Maximizar==0))||((valor>mejor)&&(Maximizar==1)))
    {
      mejor=valor;
      LugarMejor=i;
```

```
    }
    if(((valor>peor)&&(Maximizar==0))||((valor<peor)&&(Maximizar==1)))
    {
      peor=valor;
      LugarPeor=i;
    }
  }
  EvaluacionMedia=AcumuladoMedia/GeneracionActual.Generacion.
          GetItemsInContainer();
  MejorEnEstaGeneracion->copiar(GeneracionActual.Generacion.
          dato(LugarMejor));
  PeorEnEstaGeneracion->copiar(GeneracionActual.Generacion.
          dato(LugarPeor));

  double valor1,valor2;
  valor1=MejorEnEstaGeneracion->Objetivo;
  valor2=MejorEnLaHistoria->Objetivo;

  if((Generacion==0)||((valor1<valor2)&&(Maximizar==0))||((valor1>valor2)
        &&(Maximizar==1)))
  {
    MejorEnLaHistoria->copiar(MejorEnEstaGeneracion);
    GeneracionDelMejorEnLaHistoria=Generacion;
  }

  valor2=MejorEnLaHistoria->Objetivo;
  AcumuladoOffLine+=valor2;
  MedidaOffLine=AcumuladoOffLine/(Generacion+1.0);
  AcumuladoOnLine+=EvaluacionMedia;
  MedidaOnLine=AcumuladoOnLine/(Generacion+1.0);
  double acumDesv=0;
  for(i=0;i<GeneracionActual.Generacion.GetItemsInContainer();i++)
  {
    double valor;
    valor=GeneracionActual.Generacion.dato(i)->Objetivo;
    acumDesv=acumDesv+(valor-EvaluacionMedia)*(valor-EvaluacionMedia);
  }
  Desviacion=sqrt(acumDesv/GeneracionActual.Generacion.
        GetItemsInContainer());
}

void AlgoritmoGenetico::mostrarMedidas()
{
  if(IndicadorMostrarGeneracion)
  {
    cout << "Generacion:" << Generacion << "\n";
```

```
  }
  if(IndicadorMostrarMejorEnHistoria)
  {
    cout << "MejorHistorico:" << MejorEnLaHistoria->Objetivo << "\n";
  }
  if(IndicadorMostrarGeneracionMejorHistorico)
  {
    cout << "GeneracionMejorHistorico:" << GeneracionDelMejorEnLaHistoria
         << "\n";
  }
  if(IndicadorMostrarMejorEnGeneracion)
  {
    cout << "MejorActual:" << MejorEnEstaGeneracion->Objetivo << "\n";
  }
  if(IndicadorMostrarPeorEnGeneracion)
  {
    cout << "PeorActual:" << PeorEnEstaGeneracion->Objetivo << "\n";
  }
  if(IndicadorMostrarMedia)
  {
    cout << "Media:" << EvaluacionMedia << "\n";
  }
  if(IndicadorMostrarDesviacion)
  {
    cout << "Desviación:" << Desviacion << "\n";
  }
  if(IndicadorMostrarOnLine)
  {
    cout << "OnLine:" << MedidaOnLine << "\n";
  }
  if(IndicadorMostrarOffLine)
  {
    cout << "OffLine:" << MedidaOffLine << "\n";
  }
}

void AlgoritmoGenetico::salvar()
{
  if(IndicadorMostrarGeneracion)
  {
    fprintf(arch,"%d ",Generacion);
  }
  if(IndicadorMostrarMejorEnHistoria)
  {
    fprintf(arch,"%le ",MejorEnLaHistoria->Objetivo);
  }
```

```
  if(IndicadorMostrarGeneracionMejorHistorico)
  {
    fprintf(arch,"%d ",GeneracionDelMejorEnLaHistoria);
  }
  if(IndicadorMostrarMejorEnGeneracion)
  {
    fprintf(arch,"%le ",MejorEnEstaGeneracion->Objetivo);
  }
  if(IndicadorMostrarPeorEnGeneracion)
  {
    fprintf(arch,"%le ",PeorEnEstaGeneracion->Objetivo);
  }
  if(IndicadorMostrarMedia)
  {
    fprintf(arch,"%le ",EvaluacionMedia);
  }
  if(IndicadorMostrarDesviacion)
  {
    fprintf(arch,"%le ",Desviacion);
  }
  if(IndicadorMostrarOnLine)
  {
    fprintf(arch,"%le ",MedidaOnLine);
  }
  if(IndicadorMostrarOffLine)
  {
    fprintf(arch,"%le ",MedidaOffLine);
  }
  fprintf(arch,"\n");
}

void AlgoritmoGenetico::iniciarOptimizacion()
{
  if(IndicadorArchivo)
  {
    arch=fopen(NombreArchivo,"wt");
    fclose(arch);
  }

  Conteo=0;
  Generacion=0;

  GeneracionDelMejorEnLaHistoria=0;
  EvaluacionMedia=0;
  MedidaOnLine=0;
  MedidaOffLine=0;
```

```
  AcumuladoOnLine=0;
  AcumuladoOffLine=0;

  randomize();

  GeneracionActual.crearGeneracion(TamanoPoblacion);
  GeneracionActual.crearGeneracionAleatoria();
  calcularObjetivo();
  actualizarMedidas();

  if(IndicadorArchivo==1)
  {
    arch=fopen(NombreArchivo,"at");
    salvar();
    fclose(arch);
    if(IndicadorMostrar)
    {
      mostrarMedidas();
    }
  }
  buscaElite();
  asignarProbabilidad(Maximizar);
}

void AlgoritmoGenetico::iterarOptimizacion()
{
  adaptacion();
  seleccionar();
  asignarParejas();
  reproducir();
  mutar();
  elitismo();
  calcularObjetivo();
  asignarProbabilidad(Maximizar);
  buscaElite();
  Generacion++;
  actualizarMedidas();
  Conteo++;
  if((IndicadorArchivo==1)&&(Conteo==IntervaloSalvar))
  {
    Conteo=0;
    arch=fopen(NombreArchivo,"at");
    salvar();
    fclose(arch);
    if(IndicadorMostrar)
    {
```

```
      mostrarMedidas();
    }
  }
}

void AlgoritmoGenetico::finalizarOptimizacion()
{
}

void AlgoritmoGenetico::optimizar()
{
  iniciarOptimizacion();

  do
  {
    iterarOptimizacion();
  }while(!parada());

  finalizarOptimizacion();
}

int AlgoritmoGenetico::parada()
{
  int res;
  switch(CriterioParada)
  {
    case 0: res=(Generacion>=GeneracionMaxima);break;
    default: res=(Generacion>=GeneracionMaxima);break;
  }
  return res;
}

void OperadorMutacion::mutar(Gen *g)
{
  double azar;
  azar=(double)((double)rand()/(double)RAND_MAX);
  if(azar<ProbabilidadMutacion)
  {
    mutarGen(g);
  }
}

void OperadorCruce::cruzar(Gen *madre,Gen *padre, Arreglo<Gen> *hijos,
         int numHijos)
{
  cruzarGenes(madre,padre,hijos,numHijos);
```

```
  int i;
  for(i=0;i<numHijos;i++)
  {
    hijos->dato(i)->ag(AG);
  }
}


////////////////////////////////////////////////////////////////////
////////////////////  OPERADORES       ////////////////////
////////////////////////////////////////////////////////////////////

void OperadorProbabilidadProporcional::asignarProbabilidad
         (int descendente)
{
  Pob->ordenar(descendente);
  Pob->sumaObjetivo();
  int i,tam;
  double SumaObjetivo=0.0;
  tam=Pob->Generacion.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    if(!descendente)
    {
      if(Pob->Generacion.dato(i)->Objetivo!=0)
      {
        SumaObjetivo+=1.0/Pob->Generacion.dato(i)->Objetivo;
      }
    }else
    {
      SumaObjetivo+=Pob->Generacion.dato(i)->Objetivo;
    }
  }
  if(SumaObjetivo==0.0)
  {
    tam=Pob->Generacion.GetItemsInContainer();
    for(i=0;i<tam;i++)
    {
      Pob->Generacion.dato(i)->Probabilidad=1.0/(double)tam;
    }
  }else
  {
    int flag=0;
    tam=Pob->Generacion.GetItemsInContainer();
    for(i=0;i<tam;i++)
    {
```

```
      if(!descendente)
      {
        if(Pob->Generacion.dato(i)->Objetivo!=0)
        {
          Pob->Generacion.dato(i)->Probabilidad=(1.0/Pob->
        Generacion.dato(i)
           ->Objetivo)/SumaObjetivo;
        }else
        {
          Pob->Generacion.dato(i)->Probabilidad=1.0;
          flag++;
        }
      }else
      {
        Pob->Generacion.dato(i)->Probabilidad=Pob->Generacion.dato(i)
          ->Objetivo/SumaObjetivo;
      }
    }
    if(flag!=0)
    {
      for(i=0;i<tam;i++)
      {
        if(Pob->Generacion.dato(i)->Probabilidad==1.0)
        {
          Pob->Generacion.dato(i)->Probabilidad=1.0/(double)flag;
        }else
        {
          Pob->Generacion.dato(i)->Probabilidad=0.0;
        }
      }
    }
  }
}

void OperadorSeleccionEstocasticaRemplazo::seleccionar()
{
  int i,j,tam;
  double *angulo;
  double azar;
  double suma=0;
  double angulomenor,angulomayor;

  tam=Pob->Generacion.GetItemsInContainer();
  angulo=new double[tam];

  for(i=0;i<tam;i++)
```

```
{
  suma=suma+Pob->Generacion.dato(i)->Probabilidad;
  angulo[i]=suma;
}
Arreglo<Individuo> *Npob;
Npob=new Arreglo<Individuo>;
for(i=0;i<tam;i++)
{
  azar=(double)((double)rand()/(double)RAND_MAX);
  for(j=0;j<tam;j++)
  {
    if(j==0)
    {
      angulomenor=0.0;
    }else
    {
      angulomenor=angulo[j-1];
    }

    if(j==tam-1)
    {
      angulomayor=1.0;
    }else
    {
      angulomayor=angulo[j];
    }
    if((azar>angulomenor)&&(azar<=angulomayor))
    {
      Individuo *ind;
      ind=Pob->Modelo->crearCopia();
      ind->copiar(Pob->Generacion.dato(j));
      Npob->Add(ind);
      j=tam;
    }
  }
}
for(i=0;i<tam;i++)
{
  if(i<Npob->GetItemsInContainer())
  {
    Pob->Generacion.dato(i)->copiar(Npob->dato(i));
  }else
  {
    Pob->Generacion.dato(i)->copiar(Npob->dato(Npob->
       GetItemsInContainer()-1));
  }
```

```
  }

  Npob->FlushDestroy();
  delete Npob;
  delete[] angulo;
}

void OperadorParejasAleatoria::asignarParejas()
{
  int i,tam,tamciclo;
  int pareja;
  double azar;
  tam=Pob->Generacion.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    Pob->Generacion.dato(i)->Pareja=NULL;
            //  NULL significa sin pareja,
            //  otro valor es el puntero de la pareja
  }
  if(tam%2==0)
  {
    tamciclo=tam;
  }else
  {
    tamciclo=tam-1;
  }
  for(i=0;i<tamciclo;i++)
  {
    if(Pob->Generacion.dato(i)->Pareja==NULL)
    {
      int loop=0;
      do
      {
        azar=(double)((double)rand()/(double)RAND_MAX);
        pareja=(int)((tamciclo)*azar);
        loop++;
        if(Pob->Generacion.dato(pareja)->Pareja==NULL&&pareja!=i)
        {
          Pob->Generacion.dato(i)->Pareja=Pob->Generacion.dato(pareja);
          Pob->Generacion.dato(pareja)->Pareja=Pob->Generacion.dato(i);
        }
        if(loop>3000)
        {
          Pob->Generacion.dato(i)->Pareja=Pob->Generacion.dato(tam-1);
        }
      }while(Pob->Generacion.dato(i)->Pareja==NULL);
```

```
    }
  }
}

void OperadorReproduccionCrucePlano::reproducir()
{
  int i,j,tam,tamGenoma;
  tamGenoma=Pob->Modelo->Genoma.GetItemsInContainer();
  tam=Pob->Generacion.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    if(Pob->Generacion.dato(i)->Pareja!=NULL)
    {
      Individuo *Padre,*Madre;
      int puntoCruce;
      double azar;
      Padre=Pob->Generacion.dato(i);
      Madre=Pob->Generacion.dato(i)->Pareja;
      azar=(double)((double)rand()/(double)RAND_MAX);
      puntoCruce=(int)((tamGenoma)*azar);
      for(j=puntoCruce;j<tamGenoma;j++)
      {
        Gen *temp;
        temp=Padre->Genoma.dato(j);
        Padre->Genoma.Detach(j);
        Padre->Genoma.AddAt(j,Madre->Genoma.dato(j));
        Madre->Genoma.Detach(j);
        Madre->Genoma.AddAt(j,temp);
      }
      Padre->Pareja=NULL;
      Madre->Pareja=NULL;
      Padre->calcularObjetivo();
      Madre->calcularObjetivo();
    }
  }
}

void OperadorReproduccionDosPadresDosHijos::reproducir()
{
  int i,j,tam,tamGenoma;
  tamGenoma=Pob->Modelo->Genoma.GetItemsInContainer();
  tam=Pob->Generacion.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    if(Pob->Generacion.dato(i)->Pareja!=NULL)
    {
```

```
      Individuo *Padre,*Madre;

      Padre=Pob->Generacion.dato(i);
      Madre=Pob->Generacion.dato(i)->Pareja;

      for(j=0;j<tamGenoma;j++)
      {
        Arreglo<Gen> Hijos;
        Gen *GenPadre,*GenMadre;
        GenPadre=Padre->Genoma.dato(j);
        GenMadre=Madre->Genoma.dato(j);
        OperadorCruce *OC;
        OC=AG->ListaOperadorCruce->dato(j);
        OC->cruzar(GenMadre,GenPadre,&Hijos,2);
        Padre->Genoma.Destroy(j);
        Madre->Genoma.Destroy(j);
        Padre->Genoma.AddAt(j,Hijos.dato(0));
        Madre->Genoma.AddAt(j,Hijos.dato(1));
        Hijos.FlushDetach();
      }

      Padre->decodificar();
      Madre->decodificar();
      Padre->calcularObjetivo();
      Madre->calcularObjetivo();

    }
  }
}
```

A.3 *Genbool.hpp*

```
#ifndef _GENBOOL_HPP
#define _GENBOOL_HPP
#endif

#ifndef _GENETICO_HPP
#include"genetico.hpp"
#endif

////////////////////////// GEN BOOL (int) ///////////////////////

class GenBool:public Gen
{
public:
```

```
  GenBool();
  ~GenBool()
  {
  }

  int Valor;  // el valor es 0 ó 1
  void crearAleatorio();
  OperadorMutacion *operadorMutacionDefecto();
  OperadorCruce *operadorCruceDefecto();
};

////////////////////////// Operadores de Mutación ///////////////////////

class OperadorMutacionBoolUniforme:public OperadorMutacion
{
public:
  OperadorMutacionBoolUniforme(){}
  ~OperadorMutacionBoolUniforme(){}

  void mutarGen(Gen *g);
};

////////////////////////// Operadores de Cruce ///////////////////////

class OperadorCruceBoolPlano:public OperadorCruce
{
public:
  OperadorCruceBoolPlano(){}
  ~OperadorCruceBoolPlano(){}

  void cruzarGenes(Gen *madre,Gen *padre, Arreglo<Gen> *hijos,
         int numHijos);
};
```

A.4 *Genbool.cpp*

```
#ifndef _GENBOOL_CPP
#define _GENBOOL_CPP
#endif

#ifndef _GENBOOL_HPP
#include"genbool.hpp"
#endif

////////////////////////// GEN BOOL (int) ///////////////////////
```

```
GenBool::GenBool()
{
  Valor=0;
}

void GenBool::crearAleatorio()
{
  double azar;
  azar=(double)((double)rand()/(double)RAND_MAX);
  if(azar<0.5)
  {
    Valor=0;
  }else
  {
    Valor=1;
  }
}

OperadorMutacion *GenBool::operadorMutacionDefecto()
{
  OperadorMutacion *OM;
  OM=new OperadorMutacionBoolUniforme;
  return OM;
}

OperadorCruce *GenBool::operadorCruceDefecto()
{
  OperadorCruce *OC;
  OC=new OperadorCruceBoolPlano;
  return OC;
}

////////////////////////// Operadores de Mutación ///////////////////////

void OperadorMutacionBoolUniforme::mutarGen(Gen *g)
{
  GenBool *gr;
  gr=(GenBool*)g;
  double azar;
  azar=(double)((double)rand()/(double)RAND_MAX);
  if(azar<0.5)
  {
    gr->Valor=0;
  }else
  {
```

```
    gr->Valor=1;
  }
}

////////////////////////// Operadores de Cruce ///////////////////////

void OperadorCruceBoolPlano::cruzarGenes(Gen *m,Gen *p, Arreglo<Gen> *h,
         int numHijos)
{
  GenBool *madre,*padre;
  madre=(GenBool*)m;
  padre=(GenBool*)p;
  Arreglo<GenBool> hijos;
  int i,tam;
  for(i=0;i<numHijos;i++)
  {
    GenBool *gr;
    gr=new GenBool;

    double azar;
    azar=(double)((double)rand()/(double)RAND_MAX);
    if(azar<0.5)
    {
      gr->Valor=madre->Valor;
    }else
    {
      gr->Valor=padre->Valor;
    }

    hijos.Add(gr);
  }

  tam=hijos.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    h->Add(hijos.dato(i));
  }
  hijos.FlushDetach();
}
```

A.5 *Genint.hpp*

```
#ifndef _GENINT_HPP
#define _GENINT_HPP
```

```
#endif

#ifndef _GENETICO_HPP
#include"genetico.hpp"
#endif

////////////////////////// GEN ENTERO (long) ///////////////////////

class GenEntero:public Gen
{
public:
  GenEntero();
  ~GenEntero()
  {
  }

  long Minimo,Maximo; // el valor por defecto está entre 0 y 1
  long Valor;
  void crearAleatorio();
  OperadorMutacion *operadorMutacionDefecto();
  OperadorCruce *operadorCruceDefecto();
};

////////////////////////// Operadores de Mutación ///////////////////////

class OperadorMutacionEnteroUniforme:public OperadorMutacion
{
public:
  OperadorMutacionEnteroUniforme(){}
  ~OperadorMutacionEnteroUniforme(){}

  void mutarGen(Gen *g);
};

class OperadorMutacionEnteroNoUniforme:public OperadorMutacion
{
public:
  OperadorMutacionEnteroNoUniforme()
  {
    B=0.5;
  }
  ~OperadorMutacionEnteroNoUniforme(){}

  double B;

  void mutarGen(Gen *g);
```

```
};

class OperadorMutacionEnteroBGA:public OperadorMutacion
{
public:
  OperadorMutacionEnteroBGA()
  {
    Factor=0.1;
  }
  ~OperadorMutacionEnteroBGA(){}

  double Factor;

  void mutarGen(Gen *g);
};

////////////////////////// Operadores de Cruce ///////////////////////

class OperadorCruceEnteroPlano:public OperadorCruce
{
public:
  OperadorCruceEnteroPlano(){}
  ~OperadorCruceEnteroPlano(){}

  void cruzarGenes(Gen *madre,Gen *padre, Arreglo<Gen> *hijos,
        int numHijos);
};

class OperadorCruceEnteroAritmetico:public OperadorCruce
{
public:
  OperadorCruceEnteroAritmetico()
  {
    Lambda=0.5;
  }
  ~OperadorCruceEnteroAritmetico(){}

  double Lambda;

  void cruzarGenes(Gen *madre,Gen *padre, Arreglo<Gen> *hijos,
         int numHijos);
};

class OperadorCruceEnteroBLX:public OperadorCruce
{
public:
```

```
  OperadorCruceEnteroBLX()
  {
    Alfa=0.25;
  }
  ~OperadorCruceEnteroBLX(){}

  double Alfa;

  void cruzarGenes(Gen *madre,Gen *padre, Arreglo<Gen> *hijos,
         int numHijos);
};

class OperadorCruceEnteroLineal:public OperadorCruce
{
public:
  OperadorCruceEnteroLineal(){}
  ~OperadorCruceEnteroLineal(){}

  void cruzarGenes(Gen *madre,Gen *padre, Arreglo<Gen> *hijos,
          int numHijos);
};

class OperadorCruceEnteroDiscreto:public OperadorCruce
{
public:
  OperadorCruceEnteroDiscreto(){}
  ~OperadorCruceEnteroDiscreto(){}

  void cruzarGenes(Gen *madre,Gen *padre, Arreglo<Gen> *hijos,
           int numHijos);
};

class OperadorCruceEnteroIntermedioExtendido:public OperadorCruce
{
public:
  OperadorCruceEnteroIntermedioExtendido()
  {
    Alfa=0.5;
  }
  ~OperadorCruceEnteroIntermedioExtendido(){}

  double Alfa;

  void cruzarGenes(Gen *madre,Gen *padre, Arreglo<Gen> *hijos,
         int numHijos);
};
```

```
class OperadorCruceEnteroHeuristico:public OperadorCruce
{
public:
  OperadorCruceEnteroHeuristico(){}
  ~OperadorCruceEnteroHeuristico(){}

  void cruzarGenes(Gen *madre,Gen *padre, Arreglo<Gen> *hijos,
         int numHijos);
};
```

A.6 *Genint.cpp*

```
#ifndef _GENINT_CPP
#define _GENINT_CPP
#endif

#ifndef _GENINT_HPP
#include "genint.hpp"
#endif

////////////////////////// GEN REAL (double) ///////////////////////

GenEntero::GenEntero()
{
  Minimo=-10;
  Maximo=10;
  Valor=0;
}

void GenEntero::crearAleatorio()
{
  double azar;
  azar=(double)((double)rand()/(double)RAND_MAX);
  Valor=(long)(Minimo+azar*(Maximo-Minimo));
  if(Valor<Minimo)Valor=Minimo;
  if(Valor>Maximo)Valor=Maximo;
}

OperadorMutacion *GenEntero::operadorMutacionDefecto()
{
  OperadorMutacion *OM;
  OM=new OperadorMutacionEnteroUniforme;
  return OM;
}
```

```
OperadorCruce *GenEntero::operadorCruceDefecto()
{
  OperadorCruce *OC;
  OC=new OperadorCruceEnteroPlano;
  return OC;
}

////////////////////////// Operadores de Mutación ///////////////////////

void OperadorMutacionEnteroUniforme::mutarGen(Gen *g)
{
  GenEntero *gr;
  gr=(GenEntero*)g;
  double azar;
  azar=(double)((double)rand()/(double)RAND_MAX);
  gr->Valor=(long)(gr->Minimo+azar*(gr->Maximo-gr->Minimo));
  if(gr->Valor<gr->Minimo)gr->Valor=gr->Minimo;
  if(gr->Valor>gr->Maximo)gr->Valor=gr->Maximo;
}

void OperadorMutacionEnteroNoUniforme::mutarGen(Gen *g)
{
  GenEntero *gr;
  gr=(GenEntero*)g;

  int t,T;
  double r,b,y,delta;

  t=AG->Generacion;
  T=AG->GeneracionMaxima;
  b=B;
  double azar;
  azar=(double)((double)rand()/(double)RAND_MAX);
  r=azar;
  azar=(double)((double)rand()/(double)RAND_MAX);
  if(azar<0.5)
  {
    y=gr->Maximo-gr->Valor;
    delta=y*(1.0-pow(r,pow((1.0-t/T),b)));
    gr->Valor=gr->Valor+delta;
  }else
  {
    y=gr->Valor-gr->Minimo;
    delta=y*(1.0-pow(r,pow((1.0-t/T),b)));
    gr->Valor=gr->Valor-delta;
```

```
  }

  if(gr->Valor<gr->Minimo){gr->Valor=gr->Minimo;}
  if(gr->Valor>gr->Maximo){gr->Valor=gr->Maximo;}
}

void OperadorMutacionEnteroBGA::mutarGen(Gen *g)
{
  GenEntero *gr;
  gr=(GenEntero*)g;
  double azar;
  double rango,alfa,gamma=0.0;
  rango=Factor*(gr->Maximo-gr->Minimo);

  int i;
  for(i=0;i<16;i++)
  {
    azar=(double)((double)rand()/(double)RAND_MAX);
    if(azar<0.06125){alfa=1.0;}else{alfa=0.0;}
    gamma+=alfa*pow(2,-1*i);
  }
  azar=(double)((double)rand()/(double)RAND_MAX);
  if(azar<0.5)
  {
    gr->Valor=gr->Valor+rango*gamma;
  }else
  {
    gr->Valor=gr->Valor-rango*gamma;
  }

  if(gr->Valor<gr->Minimo){gr->Valor=gr->Minimo;}
  if(gr->Valor>gr->Maximo){gr->Valor=gr->Maximo;}
}

////////////////////////// Operadores de Cruce ///////////////////////

void OperadorCruceEnteroPlano::cruzarGenes(Gen *m,Gen *p,
         Arreglo<Gen> *h,  int numHijos)
{
  GenEntero *madre,*padre;
  madre=(GenEntero*)m;
  padre=(GenEntero*)p;
  Arreglo<GenEntero> hijos;
  int i,tam;
  for(i=0;i<numHijos;i++)
  {
```

```
    GenEntero *gr;
    gr=new GenEntero;

    double azar;
    azar=(double)((double)rand()/(double)RAND_MAX);
    gr->Valor=madre->Valor+azar*(padre->Valor-madre->Valor);
    gr->Minimo=madre->Minimo;
      gr->Maximo=madre->Maximo;
    if(gr->Valor<gr->Minimo)gr->Valor=gr->Minimo;
    if(gr->Valor>gr->Maximo)gr->Valor=gr->Maximo;

    hijos.Add(gr);
  }

  tam=hijos.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    h->Add(hijos.dato(i));
  }
  hijos.FlushDetach();
}

void OperadorCruceEnteroAritmetico::cruzarGenes(Gen *m,Gen *p,
         Arreglo<Gen> *h, int numHijos)
{
  GenEntero *madre,*padre;
  madre=(GenEntero*)m;
  padre=(GenEntero*)p;
  Arreglo<GenEntero> hijos;
  double alfa;
  int i,tam;
  for(i=0;i<numHijos;i++)
  {
    if(i%2 == 0){alfa=Lambda;}else{alfa=1.0-Lambda;}
    GenEntero *gr;
    gr=new GenEntero;

    gr->Valor=alfa*madre->Valor+(1.0-alfa)*(padre->Valor);

    hijos.Add(gr);
  }

  tam=hijos.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    h->Add(hijos.dato(i));
```

```
  }
  hijos.FlushDetach();
}

void OperadorCruceEnteroBLX::cruzarGenes(Gen *m,Gen *p,
          Arreglo<Gen> *h, int numHijos)
{
  GenEntero *madre,*padre;
  madre=(GenEntero*)m;
  padre=(GenEntero*)p;
  Arreglo<GenEntero> hijos;
  int i,tam;
  for(i=0;i<numHijos;i++)
  {
    GenEntero *gr;
    gr=new GenEntero;

    double azar;
    azar=(double)((double)rand()/(double)RAND_MAX);
    gr->Valor=madre->Valor+azar*(padre->Valor-madre->Valor);

    hijos.Add(gr);
  }

  tam=hijos.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    h->Add(hijos.dato(i));
  }
  hijos.FlushDetach();
}

void OperadorCruceEnteroLineal::cruzarGenes(Gen *m,Gen *p,
        Arreglo<Gen> *h, int numHijos)
{
  GenEntero *madre,*padre;
  madre=(GenEntero*)m;
  padre=(GenEntero*)p;
  Arreglo<GenEntero> hijos;
  int i,tam;
  for(i=0;i<numHijos;i++)
  {
    GenEntero *gr;
    gr=new GenEntero;

    double azar;
```

```
    azar=(double)((double)rand()/(double)RAND_MAX);
    gr->Valor=madre->Valor+azar*(padre->Valor-madre->Valor);

    hijos.Add(gr);
  }

  tam=hijos.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    h->Add(hijos.dato(i));
  }
  hijos.FlushDetach();
}

void OperadorCruceEnteroDiscreto::cruzarGenes(Gen *m,Gen *p,
          Arreglo<Gen> *h, int numHijos)
{
  GenEntero *madre,*padre;
  madre=(GenEntero*)m;
  padre=(GenEntero*)p;
  Arreglo<GenEntero> hijos;
  int i,tam;
  for(i=0;i<numHijos;i++)
  {
    GenEntero *gr;
    gr=new GenEntero;

    double azar;
    azar=(double)((double)rand()/(double)RAND_MAX);
    gr->Valor=madre->Valor+azar*(padre->Valor-madre->Valor);

    hijos.Add(gr);
  }

  tam=hijos.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    h->Add(hijos.dato(i));
  }
  hijos.FlushDetach();
}

void OperadorCruceEnteroIntermedioExtendido::cruzarGenes(Gen *m,Gen *p,
        Arreglo<Gen> *h, int numHijos)
{
  GenEntero *madre,*padre;
```

```
  madre=(GenEntero*)m;
  padre=(GenEntero*)p;
  Arreglo<GenEntero> hijos;
  int i,tam;
  for(i=0;i<numHijos;i++)
  {
    GenEntero *gr;
    gr=new GenEntero;

    double azar;
    azar=(double)((double)rand()/(double)RAND_MAX);
    gr->Valor=madre->Valor+azar*(padre->Valor-madre->Valor);

    hijos.Add(gr);
  }

  tam=hijos.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    h->Add(hijos.dato(i));
  }
  hijos.FlushDetach();
}

void OperadorCruceEnteroHeuristico::cruzarGenes(Gen *m,Gen *p,
         Arreglo<Gen> *h, int numHijos)
{
  GenEntero *madre,*padre;
  madre=(GenEntero*)m;
  padre=(GenEntero*)p;
  Arreglo<GenEntero> hijos;
  int i,tam;
  for(i=0;i<numHijos;i++)
  {
    GenEntero *gr;
    gr=new GenEntero;

    double azar;
    azar=(double)((double)rand()/(double)RAND_MAX);
    gr->Valor=madre->Valor+azar*(padre->Valor-madre->Valor);

    hijos.Add(gr);
  }

  tam=hijos.GetItemsInContainer();
  for(i=0;i<tam;i++)
```

```
  {
    h->Add(hijos.dato(i));
  }
  hijos.FlushDetach();
}
```

A.7 *Genreal.hpp*

```
#ifndef _GENREAL_HPP
#define _GENREAL_HPP
#endif

#ifndef _GENETICO_HPP
#include"genetico.hpp"
#endif

////////////////////////// GEN REAL (double) ///////////////////////

class GenReal:public Gen
{
public:
  GenReal();
  ~GenReal()
  {
  }

  double Minimo,Maximo; // el valor por defecto está entre 0 y 1
  double Valor;
  void crearAleatorio();
  OperadorMutacion *operadorMutacionDefecto();
  OperadorCruce *operadorCruceDefecto();
};

////////////////////////// Operadores de Mutación ///////////////////////

class OperadorMutacionRealUniforme:public OperadorMutacion
{
public:
  OperadorMutacionRealUniforme(){}
  ~OperadorMutacionRealUniforme(){}

  void mutarGen(Gen *g);
};

class OperadorMutacionRealNoUniforme:public OperadorMutacion
```

```
{
public:
  OperadorMutacionRealNoUniforme()
  {
    B=0.5;
  }
  ~OperadorMutacionRealNoUniforme(){}

  double B;

  void mutarGen(Gen *g);
};

class OperadorMutacionRealBGA:public OperadorMutacion
{
public:
  OperadorMutacionRealBGA()
  {
    Factor=0.1;
  }
  ~OperadorMutacionRealBGA(){}

  double Factor;

  void mutarGen(Gen *g);
};

////////////////////////// Operadores de Cruce ///////////////////////

class OperadorCruceRealPlano:public OperadorCruce
{
public:
  OperadorCruceRealPlano(){}
  ~OperadorCruceRealPlano(){}

  void cruzarGenes(Gen *madre,Gen *padre, Arreglo<Gen> *hijos,
         int numHijos);
};

class OperadorCruceRealAritmetico:public OperadorCruce
{
public:
  OperadorCruceRealAritmetico()
  {
    Lambda=0.5;
  }
```

```
  ~OperadorCruceRealAritmetico(){}

  double Lambda;

  void cruzarGenes(Gen *madre,Gen *padre, Arreglo<Gen> *hijos,
        int numHijos);
};

class OperadorCruceRealBLX:public OperadorCruce
{
public:
  OperadorCruceRealBLX()
  {
    Alfa=0.25;
  }
  ~OperadorCruceRealBLX(){}

  double Alfa;

  void cruzarGenes(Gen *madre,Gen *padre, Arreglo<Gen> *hijos,
        int numHijos);
};

class OperadorCruceRealLineal:public OperadorCruce
{
public:
  OperadorCruceRealLineal(){}
  ~OperadorCruceRealLineal(){}

  void cruzarGenes(Gen *madre,Gen *padre, Arreglo<Gen> *hijos,
        int numHijos);
};

class OperadorCruceRealDiscreto:public OperadorCruce
{
public:
  OperadorCruceRealDiscreto(){}
  ~OperadorCruceRealDiscreto(){}

  void cruzarGenes(Gen *madre,Gen *padre, Arreglo<Gen> *hijos,
          int numHijos);
};

class OperadorCruceRealIntermedioExtendido:public OperadorCruce
{
public:
```

```
  OperadorCruceRealIntermedioExtendido()
  {
    Alfa=0.5;
  }
  ~OperadorCruceRealIntermedioExtendido(){}

  double Alfa;

  void cruzarGenes(Gen *madre,Gen *padre, Arreglo<Gen> *hijos,
         int numHijos);
};

class OperadorCruceRealHeuristico:public OperadorCruce
{
public:
  OperadorCruceRealHeuristico(){}
  ~OperadorCruceRealHeuristico(){}

  void cruzarGenes(Gen *madre,Gen *padre, Arreglo<Gen> *hijos,
         int numHijos);
};
```

A.8 *Genreal.cpp*

```
#ifndef _GENREAL_CPP
#define _GENREAL_CPP
#endif

#ifndef _GENREAL_HPP
#include "genreal.hpp"
#endif

////////////////////////// GEN REAL (double) ///////////////////////

GenReal::GenReal()
{
  Minimo=0.0;
  Maximo=1.0;
  Valor=0.5;
}

void GenReal::crearAleatorio()
{
  double azar;
  azar=(double)((double)rand()/(double)RAND_MAX);
```

```
  Valor=Minimo+azar*(Maximo-Minimo);
  if(Valor<Minimo){Valor=Minimo;}
  if(Valor>Maximo){Valor=Maximo;}
}

OperadorMutacion *GenReal::operadorMutacionDefecto()
{
  OperadorMutacion *OM;
  OM=new OperadorMutacionRealUniforme;
  return OM;
}

OperadorCruce *GenReal::operadorCruceDefecto()
{
  OperadorCruce *OC;
  OC=new OperadorCruceRealPlano;
  return OC;
}


////////////////////////// Operadores de Mutación ///////////////////////

void OperadorMutacionRealUniforme::mutarGen(Gen *g)
{
  GenReal *gr;
  gr=(GenReal*)g;
  double azar;
  azar=(double)((double)rand()/(double)RAND_MAX);
  gr->Valor=gr->Minimo+azar*(gr->Maximo-gr->Minimo);
  if(gr->Valor<gr->Minimo){gr->Valor=gr->Minimo;}
  if(gr->Valor>gr->Maximo){gr->Valor=gr->Maximo;}
}

void OperadorMutacionRealNoUniforme::mutarGen(Gen *g)
{
  GenReal *gr;
  gr=(GenReal*)g;

  int t,T;
  double r,b,y,delta;

  t=AG->Generacion;
  T=AG->GeneracionMaxima;
  b=B;
  double azar;
  azar=(double)((double)rand()/(double)RAND_MAX);
```

```
  r=azar;
  azar=(double)((double)rand()/(double)RAND_MAX);
  if(azar<0.5)
  {
    y=gr->Maximo-gr->Valor;
    delta=y*(1.0-pow(r,pow((1.0-t/T),b)));
    gr->Valor=gr->Valor+delta;
  }else
  {
    y=gr->Valor-gr->Minimo;
    delta=y*(1.0-pow(r,pow((1.0-t/T),b)));
    gr->Valor=gr->Valor-delta;
  }

  if(gr->Valor<gr->Minimo){gr->Valor=gr->Minimo;}
  if(gr->Valor>gr->Maximo){gr->Valor=gr->Maximo;}
}

void OperadorMutacionRealBGA::mutarGen(Gen *g)
{
  GenReal *gr;
  gr=(GenReal*)g;
  double azar;
  double rango,alfa,gamma=0.0;
  rango=Factor*(gr->Maximo-gr->Minimo);

  int i;
  for(i=0;i<16;i++)
  {
    azar=(double)((double)rand()/(double)RAND_MAX);
    if(azar<0.06125){alfa=1.0;}else{alfa=0.0;}
    gamma+=alfa*pow(2,-1*i);
  }
  azar=(double)((double)rand()/(double)RAND_MAX);
  if(azar<0.5)
  {
    gr->Valor=gr->Valor+rango*gamma;
  }else
  {
    gr->Valor=gr->Valor-rango*gamma;
  }

  if(gr->Valor<gr->Minimo){gr->Valor=gr->Minimo;}
  if(gr->Valor>gr->Maximo){gr->Valor=gr->Maximo;}
}
```

```
////////////////////// Operadores de Cruce ////////////////////

void OperadorCruceRealPlano::cruzarGenes(Gen *m,Gen *p,
        Arreglo<Gen> *h, int numHijos)
{
  GenReal *madre,*padre;
  madre=(GenReal*)m;
  padre=(GenReal*)p;
  Arreglo<GenReal> hijos;
  int i,tam;
  for(i=0;i<numHijos;i++)
  {
    GenReal *gr;
    gr=new GenReal;

    double azar;
    azar=(double)((double)rand()/(double)RAND_MAX);
    gr->Valor=madre->Valor+azar*(padre->Valor-madre->Valor);
    gr->Minimo=madre->Minimo;
    gr->Maximo=madre->Maximo;
    if(gr->Valor<gr->Minimo){gr->Valor=gr->Minimo;}
    if(gr->Valor>gr->Maximo){gr->Valor=gr->Maximo;}

    hijos.Add(gr);
  }

  tam=hijos.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    h->Add(hijos.dato(i));
  }
  hijos.FlushDetach();
}

void OperadorCruceRealAritmetico::cruzarGenes(Gen *m,Gen *p,
         Arreglo<Gen> *h, int numHijos)
{
  GenReal *madre,*padre;
  madre=(GenReal*)m;
  padre=(GenReal*)p;
  Arreglo<GenReal> hijos;
  double alfa;
  int i,tam;
  for(i=0;i<numHijos;i++)
  {
    if(i%2 == 0){alfa=Lambda;}else{alfa=1.0-Lambda;}
```

```
    GenReal *gr;
    gr=new GenReal;

    gr->Valor=alfa*madre->Valor+(1.0-alfa)*(padre->Valor);

    hijos.Add(gr);
  }

  tam=hijos.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    h->Add(hijos.dato(i));
  }
  hijos.FlushDetach();
}

void OperadorCruceRealBLX::cruzarGenes(Gen *m,Gen *p,
          Arreglo<Gen> *h, int numHijos)
{
  GenReal *madre,*padre;
  madre=(GenReal*)m;
  padre=(GenReal*)p;
  Arreglo<GenReal> hijos;
  int i,tam;
  for(i=0;i<numHijos;i++)
  {
    GenReal *gr;
    gr=new GenReal;

    double azar;
    azar=(double)((double)rand()/(double)RAND_MAX);
    gr->Valor=madre->Valor+azar*(padre->Valor-madre->Valor);

    hijos.Add(gr);
  }

  tam=hijos.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    h->Add(hijos.dato(i));
  }
  hijos.FlushDetach();
}

void OperadorCruceRealLineal::cruzarGenes(Gen *m,Gen *p,
         Arreglo<Gen> *h, int numHijos)
```

```
{
  GenReal *madre,*padre;
  madre=(GenReal*)m;
  padre=(GenReal*)p;
  Arreglo<GenReal> hijos;
  int i,tam;
  for(i=0;i<numHijos;i++)
  {
    GenReal *gr;
    gr=new GenReal;

    double azar;
    azar=(double)((double)rand()/(double)RAND_MAX);
    gr->Valor=madre->Valor+azar*(padre->Valor-madre->Valor);

    hijos.Add(gr);
  }

  tam=hijos.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    h->Add(hijos.dato(i));
  }
  hijos.FlushDetach();
}

void OperadorCruceRealDiscreto::cruzarGenes(Gen *m,Gen *p,
         Arreglo<Gen> *h, int numHijos)
{
  GenReal *madre,*padre;
  madre=(GenReal*)m;
  padre=(GenReal*)p;
  Arreglo<GenReal> hijos;
  int i,tam;
  for(i=0;i<numHijos;i++)
  {
    GenReal *gr;
    gr=new GenReal;

    double azar;
    azar=(double)((double)rand()/(double)RAND_MAX);
    gr->Valor=madre->Valor+azar*(padre->Valor-madre->Valor);

    hijos.Add(gr);
  }
```

```
  tam=hijos.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    h->Add(hijos.dato(i));
  }
  hijos.FlushDetach();
}

void OperadorCruceRealIntermedioExtendido::cruzarGenes(Gen *m,Gen *p,
          Arreglo<Gen> *h, int numHijos)
{
  GenReal *madre,*padre;
  madre=(GenReal*)m;
  padre=(GenReal*)p;
  Arreglo<GenReal> hijos;
  int i,tam;
  for(i=0;i<numHijos;i++)
  {
    GenReal *gr;
    gr=new GenReal;

    double azar;
    azar=(double)((double)rand()/(double)RAND_MAX);
    gr->Valor=madre->Valor+azar*(padre->Valor-madre->Valor);

    hijos.Add(gr);
  }

  tam=hijos.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    h->Add(hijos.dato(i));
  }
  hijos.FlushDetach();
}

void OperadorCruceRealHeuristico::cruzarGenes(Gen *m,Gen *p,
         Arreglo<Gen> *h, int numHijos)
{
  GenReal *madre,*padre;
  madre=(GenReal*)m;
  padre=(GenReal*)p;
  Arreglo<GenReal> hijos;
  int i,tam;
  for(i=0;i<numHijos;i++)
  {
```

```
    GenReal *gr;
    gr=new GenReal;

    double azar;
    azar=(double)((double)rand()/(double)RAND_MAX);
    gr->Valor=madre->Valor+azar*(padre->Valor-madre->Valor);

    hijos.Add(gr);
  }

  tam=hijos.GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    h->Add(hijos.dato(i));
  }
  hijos.FlushDetach();
}
```

Apéndice B

Código fuente de los ejemplos del capítulo 3

B.1 Codificación Real. *mainreal.cpp*

```
#ifndef _MAININT_CPP
#define _MAININT_CPP
#endif

#ifndef _GENREAL_HPP
#include "genreal.hpp"
#endif

#ifndef _GENBOOL_HPP
#include "genbool.hpp"
#endif

#ifndef _GENINT_HPP
#include "genint.hpp"
#endif

class VectorReal:public Individuo
{
public:
  VectorReal(int dim);
  ~VectorReal();

  Individuo* crearCopia();
  void copiarDetalles(Individuo *other);
  double objetivo();
  void codificar();
```

```
  void decodificar();
  void crearOperadores(Arreglo<OperadorMutacion> *LM,
         Arreglo<OperadorCruce> *LC);

  void mostrar();

  int Dimension;
  double *x;
};

VectorReal::VectorReal(int dim)
{
  Dimension=dim;
  x=new double[Dimension];
  for(int i=0;i<Dimension;i++)
  {
    GenReal *gen;
    gen= new GenReal;
    gen->Minimo=-2.0*Dimension;
    gen->Maximo=2.0*Dimension+1;
    Genoma.Add(gen);
  }
}

VectorReal::~VectorReal()
{
  Genoma.FlushDestroy();
  delete[] x;
}


Individuo* VectorReal::crearCopia()
{
  Individuo *Ind;
  Ind=new VectorReal(Dimension);
  return Ind;
}
/*
void VectorReal::copiarDetalles(Individuo *other)
{
  VectorReal *VR;
  VR=(VectorReal*)other;

  Dimension=VR->Dimension;
  Genoma.FlushDestroy();
  delete[] x;
```

```
  x=new long[Dimension];
  int i;
  for(i=0;i<Dimension;i++)
  {
    x[i]=VR->x[i];
    GenReal *gen;
    gen= new GenReal;
    Genoma.Add(gen);
  }

  codificar();
}
*/
void VectorReal::copiarDetalles(Individuo *other)
{
// Supone que los dos objetos son de la misma Dimensión
  VectorReal *VR;
  VR=(VectorReal*)other;
  for(int i=0;i<Dimension;i++)
  {
    x[i]=VR->x[i];
  }
  codificar();
}

void VectorReal::crearOperadores(Arreglo<OperadorMutacion> *LM,
         Arreglo<OperadorCruce> *LC)
{
  OperadorMutacion *OM;
  OperadorCruce *OC;
  OM=new OperadorMutacionRealUniforme;
  OC=new OperadorCruceRealPlano;
  for(int i=0;i<Dimension;i++)
  {
    LM->Add(OM);
    LC->Add(OC);
  }
}

void VectorReal::codificar()
{
  for(int i=0;i<Dimension;i++)
  {
    GenReal *g;
    g=(GenReal*)Genoma.dato(i);
    g->Valor=x[i];
```

```
  }
}

void VectorReal::decodificar()
{
  for(int i=0;i<Dimension;i++)
  {
    GenReal *g;
    g=(GenReal*)Genoma.dato(i);
    x[i]=g->Valor;
  }
}

double VectorReal::objetivo()
{
  double res=0.0;
  for(int i=0;i<Dimension;i++)
  {
    res+=((x[i]-(i))*(x[i]-(i)));
  }
  return sqrt(res);
}

void VectorReal::mostrar()
{
  calcularObjetivo();
  for(int i=0;i<Dimension;i++)
  {
    cout << x[i] << " ";
  }
  cout << ":" << Objetivo << "\n";
}

void main()
{
// Unas variables auxiliares:
  int Dim,Tamano,Generacion;
  cout << "Ejemplo de Aplicación de los Algoritmos Genéticos:\n\n";
  cout << "Dimensión del problema: ";cin >> Dim;
  cout << "Tamaño de la Población: ";cin >> Tamano;
  cout << "Número máximo de generaciones: ";cin >> Generacion;

// Definicion del algoritmo, y asignacion del modelo
// En esta forma se define el problema a optimizar:
// En este Ejemplo, "VectorReal" es una clase que
// hereda las propiedades de "Individuo"
```

```
  VectorReal *Mod;
  Mod=new VectorReal(Dim);

  AlgoritmoGenetico MiAg;
  MiAg.modelo(Mod);
  MiAg.TamanoPoblacion=Tamano;
  MiAg.GeneracionMaxima=Generacion;
   MiAg.IntervaloSalvar=1;

// Ejecución del Algoritmo:
// Ejecución contínua inhabilitada
//  cout << "Optimizando...\n";
//  MiAg.optimizar();

// Utilizacion Paso a Paso del Algoritmo
  int c=0;
  MiAg.iniciarOptimizacion();
  do
  {
    MiAg.iterarOptimizacion();
    // aqui pueden emplearse los resultados de la iteracion
    // para cualquier cosa, por ejemplo:
    c++;gotoxy(1,7);cout << c << ":";
    VectorReal *VR;
    VR=(VectorReal*)MiAg.MejorEnLaHistoria;
    VR->mostrar();
  }while(!MiAg.parada());
  MiAg.finalizarOptimizacion();
  cout << "\n";

  MiAg.GeneracionActual.ordenar(0);
  int casos;
  if (Tamano<5){casos=Tamano;}else{casos=5;}
  cout << "Estos son los mejores " << casos << " individuos:\n";
  for(int i=0;i<casos;i++)
  {
    VectorReal *VR;
    VR=(VectorReal*)MiAg.GeneracionActual.Generacion.dato(i);
    VR->mostrar();
  }

   MiAg.mostrarMedidas();
}
```

B.2 Codificación Híbrida. *mainhibr.cpp*

```
#ifndef _MAININT_CPP
#define _MAININT_CPP
#endif

#ifndef _GENREAL_HPP
#include "genreal.hpp"
#endif

#ifndef _GENBOOL_HPP
#include "genbool.hpp"
#endif

#ifndef _GENINT_HPP
#include "genint.hpp"
#endif

class VectorHibrido:public Individuo
{
public:
  VectorHibrido(int dim);
  ~VectorHibrido();

  Individuo* crearCopia();
  void copiarDetalles(Individuo *other);
  void crearOperadores(Arreglo<OperadorMutacion> *LM,
         Arreglo<OperadorCruce> *LC);
  double objetivo();
  void codificar();
  void decodificar();

  void mostrar();

  int Dimension;
  double *x;
  long K;
  int B;
};

VectorHibrido::VectorHibrido(int dim)
{
  Dimension=dim;
  x=new double[Dimension];
  for(int i=0;i<Dimension;i++)
  {
```

```
    GenReal *gen;
    gen= new GenReal;
    gen->Minimo=-2.0*Dimension;
    gen->Maximo=2.0*Dimension+1;
    Genoma.Add(gen);
  }
  GenEntero *genE;
  genE= new GenEntero;
  genE->Minimo=-10;
  genE->Maximo=10;
  Genoma.Add(genE);

  GenBool *genB;
  genB= new GenBool;
  Genoma.Add(genB);
}

VectorHibrido::~VectorHibrido()
{
  Genoma.FlushDestroy();
  delete[] x;
}


Individuo* VectorHibrido::crearCopia()
{
  Individuo *Ind;
  Ind=new VectorHibrido(Dimension);
  return Ind;
}

void VectorHibrido::crearOperadores(Arreglo<OperadorMutacion> *LM,
          Arreglo<OperadorCruce> *LC)
{
  OperadorMutacion *OM;
  OperadorCruce *OC;
  OM=new OperadorMutacionRealUniforme;
  OC=new OperadorCruceRealPlano;
  for(int i=0;i<Dimension;i++)
  {
    LM->Add(OM);
    LC->Add(OC);
  }

  OM= new OperadorMutacionEnteroUniforme;
  LM->Add(OM);
```

```
  OC= new OperadorCruceEnteroPlano;
  LC->Add(OC);

  OM= new OperadorMutacionBoolUniforme;
  LM->Add(OM);
  OC= new OperadorCruceBoolPlano;
  LC->Add(OC);
}

void VectorHibrido::copiarDetalles(Individuo *other)
{
// Supone que los dos objetos son de la misma Dimensión
  VectorHibrido *VR;
  VR=(VectorHibrido*)other;
  for(int i=0;i<Dimension;i++)
  {
    x[i]=VR->x[i];
  }
  K=VR->K;
  B=VR->B;
  codificar();
}


void VectorHibrido::codificar()
{
  for(int i=0;i<Dimension;i++)
  {
    GenReal *g;
    g=(GenReal*)Genoma.dato(i);
    g->Valor=x[i];
  }
  GenEntero *gE;
  gE=(GenEntero*)Genoma.dato(i);
  gE->Valor=K;
  i++;
  GenBool *gB;
  gB=(GenBool*)Genoma.dato(i);
  gB->Valor=B;
}

void VectorHibrido::decodificar()
{
  for(int i=0;i<Dimension;i++)
  {
    GenReal *g;
```

```
    g=(GenReal*)Genoma.dato(i);
    x[i]=g->Valor;
  }
  GenEntero *gE;
  gE=(GenEntero*)Genoma.dato(i);
  K=gE->Valor;
  i++;
  GenBool *gB;
  gB=(GenBool*)Genoma.dato(i);
  B=gB->Valor;
}

double VectorHibrido::objetivo()
{
  double res=0.0;
  for(int i=0;i<Dimension;i++)
  {
    res+=((x[i]-(i))*(x[i]-(i)));
  }
  res+=(double)(K*K)+(double)B;
  return sqrt(res);
}

void VectorHibrido::mostrar()
{
  calcularObjetivo();
  for(int i=0;i<Dimension;i++)
  {
    cout << x[i] << " ";
  }
  cout << K << " " << B << " ";
  cout << ":" << Objetivo << "\n";
}

void main()
{
// Unas variables auxiliares:
  int Dim,Tamano,Generacion;
  cout << "Ejemplo de Aplicación de los Algoritmos Genéticos:\n\n";
  cout << "Dimensión del problema: ";cin >> Dim;
  cout << "Tamaño de la Población: ";cin >> Tamano;
  cout << "Número máximo de generaciones: ";cin >> Generacion;

// Definicion del algoritmo, y asignacion del modelo
// En esta forma se define el problema a optimizar:
// En este Ejemplo, "VectorHibrido" es una clase que
```

```
// hereda las propiedades de "Individuo"

  VectorHibrido *Mod;
  Mod=new VectorHibrido(Dim);

  AlgoritmoGenetico MiAg;
  MiAg.modelo(Mod);
  MiAg.TamanoPoblacion=Tamano;
  MiAg.GeneracionMaxima=Generacion;
   MiAg.IntervaloSalvar=1;

// Ejecución del Algoritmo:
// Ejecución contínua inhabilitada
//  cout << "Optimizando...\n";
//  MiAg.optimizar();

// Utilizacion Paso a Paso del Algoritmo
  int c=0;
  MiAg.iniciarOptimizacion();
  do
  {
    MiAg.iterarOptimizacion();
    // aqui pueden emplearse los resultados de la iteracion
    // para cualquier cosa, por ejemplo:
    c++;gotoxy(1,7);cout << c << ":";
    VectorHibrido *VR;
    VR=(VectorHibrido*)MiAg.MejorEnLaHistoria;
    VR->mostrar();
  }while(!MiAg.parada());
  MiAg.finalizarOptimizacion();
  cout << "\n";

  MiAg.GeneracionActual.ordenar(0);
  int casos;
  if (Tamano<5){casos=Tamano;}else{casos=5;}
  cout << "Estos son los mejores " << casos << " individuos:\n";
  for(int i=0;i<casos;i++)
  {
    VectorHibrido *VR;
    VR=(VectorHibrido*)MiAg.GeneracionActual.Generacion.dato(i);
    VR->mostrar();
  }

   MiAg.mostrarMedidas();
}
```

Apéndice C

Código fuente de los ejemplos del capítulo 4

C.1 *Fuzzy.hpp*

```
#ifndef __FUZZY_HPP
#define __FUZZY_HPP
#endif

#ifndef __IOSTREAM_H
#include <iostream.h>
#endif

#ifndef __STRIGN_H
#include<string.h>
#endif

#ifndef __STDIO_H
#include<stdio.h>
#endif

#ifndef __MATH_H
#include<math.h>
#endif

#ifndef __ARREGLOS_HPP
#include "arreglos.hpp"
#endif

typedef int BOOL;
```

```
class ConjuntoDifuso
{
public:
  ConjuntoDifuso(){}
  virtual ~ConjuntoDifuso(){}
  void minimo(float min)
  {
    Minimo=min;
  }
  float minimo()
  {
    return Minimo;
  }
  void maximo(float max)
  {
    Maximo=max;
  }
  float maximo()
  {
    return Maximo;
  }
  int identificador()
  {
    return Identificador;
  }
  int numeroPuntosClaves()
  {
    return NumeroPuntosClaves;
  }
  virtual void puntosClaves(float *puntos)=0;
  virtual void nuevoPuntoClave(int punto, float x)=0;
  virtual float pertenencia(float x)=0;
  virtual float centroAltura()=0;
  BOOL operator==(const ConjuntoDifuso& other)
  {
    return (( Minimo == other.Minimo)&
    ( Maximo == other.Maximo) );
  }
protected:
  float Minimo;
  float Maximo;
  int NumeroPuntosClaves;
  int Identificador;
};
```

```
class ConjuntoL: public ConjuntoDifuso
{
public:
  ConjuntoL(float min, float pcor, float max)
  {
    Minimo=min;
    PrimerCorte=pcor;
    Maximo=max;
    NumeroPuntosClaves=2;
    Identificador=0;
  }
  ~ConjuntoL(){}
  float pertenencia(float);
  void puntosClaves(float *puntos)
  {
    puntos[0]=primerCorte();
    puntos[1]=maximo();
  }
  void nuevoPuntoClave(int punto, float x);
  float centroAltura()
  {
    return((Minimo+PrimerCorte)/2.0);
  }
protected:
  float PrimerCorte;
  void primerCorte(float pcor)
  {
    PrimerCorte=pcor;
  }
  float primerCorte()
  {
    return PrimerCorte;
  }
};

class ConjuntoTriangulo: public ConjuntoDifuso
{
public:
  ConjuntoTriangulo(float min, float pcor, float max)
  {
    Minimo=min;
    PrimerCorte=pcor;
    Maximo=max;
    NumeroPuntosClaves=3;
    Identificador=1;
  }
```

```
  ~ConjuntoTriangulo(){}
  float pertenencia(float);
  void puntosClaves(float *puntos)
  {
    puntos[0]=minimo();
    puntos[1]=primerCorte();
    puntos[2]=maximo();
  }
  void nuevoPuntoClave(int punto, float x);
  float centroAltura()
  {
    return(PrimerCorte);
  }
protected:
  float PrimerCorte;
  void primerCorte(float pcor)
  {
    PrimerCorte=pcor;
  }
  float primerCorte()
  {
    return PrimerCorte;
  }
};

class ConjuntoPi: public ConjuntoDifuso
{
public:
  ConjuntoPi(float min, float pcor, float scor, float max)
  {
    Minimo=min;
    PrimerCorte=pcor;
    SegundoCorte=scor;
    Maximo=max;
    NumeroPuntosClaves=4;
    Identificador=2;
  }
  ~ConjuntoPi(){}
  float pertenencia(float x);
  void puntosClaves(float *puntos)
  {
    puntos[0]=minimo();
    puntos[1]=primerCorte();
    puntos[2]=segundoCorte();
    puntos[3]=maximo();
  }
```

```
  void nuevoPuntoClave(int punto, float x);
  float centroAltura()
  {
    return((PrimerCorte+SegundoCorte)/2.0);
  }
protected:
  float PrimerCorte;
  float SegundoCorte;
  void primerCorte(float pcor)
  {
    PrimerCorte=pcor;
  }
  float primerCorte()
  {
    return PrimerCorte;
  }
  void segundoCorte(float scor)
  {
    SegundoCorte=scor;
  }
  float segundoCorte()
  {
    return SegundoCorte;
  }
};

class ConjuntoGamma: public ConjuntoDifuso
{
public:
  ConjuntoGamma(float min, float pcor,  float max)
  {
    Minimo=min;
    PrimerCorte=pcor;
    Maximo=max;
    NumeroPuntosClaves=2;
    Identificador=3;
  }
  ~ConjuntoGamma(){}
  float pertenencia(float);
  void puntosClaves(float *puntos)
  {
    puntos[0]=minimo();
    puntos[1]=primerCorte();
  }
  void nuevoPuntoClave(int punto, float x);
  float centroAltura()
```

```
  {
    return((Maximo+PrimerCorte)/2.0);
  }
protected:
  float PrimerCorte;
  void primerCorte(float pcor)
  {
    PrimerCorte=pcor;
  }
  float primerCorte()
  {
    return PrimerCorte;
  }
};

class ConjuntoZ: public ConjuntoDifuso
{
public:
  ConjuntoZ(float min, float pcor, float max)
  {
    Minimo=min;
    PrimerCorte=pcor;
    Maximo=max;
    NumeroPuntosClaves=2;
    Identificador=4;
  }
  ~ConjuntoZ(){}
  float pertenencia(float);
  void puntosClaves(float *puntos)
  {
    puntos[0]=primerCorte();
    puntos[1]=maximo();
  }
  void nuevoPuntoClave(int punto, float x);
  float centroAltura()
  {
    return((Minimo+PrimerCorte)/2.0);
  }
protected:
  float PrimerCorte;
  void primerCorte(float pcor)
  {
    PrimerCorte=pcor;
  }
  float primerCorte()
  {
```

```
    return PrimerCorte;
  }
};

class ConjuntoCampana: public ConjuntoDifuso
{
public:
  ConjuntoCampana(float min, float pcor, float max)
  {
    Minimo=min;
    PrimerCorte=pcor;
    Maximo=max;
    NumeroPuntosClaves=3;
    Identificador=5;
  }
  ~ConjuntoCampana(){}
  float pertenencia(float);
  void puntosClaves(float *puntos)
  {
    puntos[0]=minimo();
    puntos[1]=primerCorte();
    puntos[2]=maximo();
  }
  void nuevoPuntoClave(int punto, float x);
  float centroAltura()
  {
    return(PrimerCorte);
  }
protected:
  float PrimerCorte;
  void primerCorte(float pcor)
  {
    PrimerCorte=pcor;
  }
  float primerCorte()
  {
    return PrimerCorte;
  }
};

class ConjuntoS: public ConjuntoDifuso
{
public:
  ConjuntoS(float min, float pcor, float max)
  {
    Minimo=min;
```

```
    PrimerCorte=pcor;
    Maximo=max;
    NumeroPuntosClaves=2;
    Identificador=6;
  }
  ~ConjuntoS(){}
  float pertenencia(float);
  void puntosClaves(float *puntos)
  {
    puntos[0]=minimo();
    puntos[1]=primerCorte();
  }
  void nuevoPuntoClave(int punto, float x);
  float centroAltura()
  {
    return((Maximo+PrimerCorte)/2.0);
  }
protected:
  float PrimerCorte;
  void primerCorte(float pcor)
  {
    PrimerCorte=pcor;
  }
  float primerCorte()
  {
    return PrimerCorte;
  }
};

class ConjuntoPiCampana: public ConjuntoDifuso
{
public:
  ConjuntoPiCampana(float min, float pcor, float scor, float max)
  {
    Minimo=min;
    PrimerCorte=pcor;
    SegundoCorte=scor;
    Maximo=max;
    NumeroPuntosClaves=4;
    Identificador=7;
  }
  ~ConjuntoPiCampana(){}
  float pertenencia(float);
  void puntosClaves(float *puntos)
  {
    puntos[0]=minimo();
```

```
    puntos[1]=primerCorte();
    puntos[2]=segundoCorte();
    puntos[3]=maximo();
  }
  void nuevoPuntoClave(int punto, float x);
  float centroAltura()
  {
    return((PrimerCorte+SegundoCorte)/2.0);
  }
protected:
  float PrimerCorte;
  float SegundoCorte;
  void primerCorte(float pcor)
  {
    PrimerCorte=pcor;
  }
  float primerCorte()
  {
    return PrimerCorte;
  }
  void segundoCorte(float scor)
  {
    SegundoCorte=scor;
  }
  float segundoCorte()
  {
    return SegundoCorte;
  }
};


class ConjuntoSinglenton: public ConjuntoDifuso
{
public:
  ConjuntoSinglenton(float pi, float de)
  {
    Pico=pi;
    Delta=de;
    Minimo=Pico-Delta/2;
    Maximo=Pico+Delta/2;
    NumeroPuntosClaves=2;
    Identificador=8;
  }
  ~ConjuntoSinglenton(){}
  float pertenencia(float);
  void puntosClaves(float *puntos)
```

```
  {
    puntos[0]=minimo();
    puntos[1]=maximo();
  }
  void nuevoPuntoClave(int punto, float x);
  float centroAltura()
  {
    return(Pico);
  }
protected:
  float pico(){return Pico;}
  float delta(){return Delta;}
  float Delta;
  float Pico;
};

inline float ConjuntoL::pertenencia(float x)
{
  float ux;
  if(x<minimo())
    ux=1;
  if(x<primerCorte()&&x>=minimo())
    ux=1;
  if(x<maximo()&&x>=primerCorte())
    ux=(maximo()-x)/(maximo()-primerCorte());
  if(x>=maximo())
    ux=0;
  if(ux<0.0001)
    ux=0;
  return ux ;
}


inline float ConjuntoTriangulo::pertenencia(float x)
{
  float ux;
  if(x<minimo())
    ux=0;
  if(x<primerCorte()&&x>=minimo())
    ux=(x-minimo())/(primerCorte()-minimo());
  if(x<maximo()&&x>=primerCorte())
    ux=(maximo()-x)/(maximo()-primerCorte());
  if(x>=maximo())
    ux=0;
  if(ux<0.0001)
    ux=0;
```

```
  return ux ;
}


inline float ConjuntoPi::pertenencia(float x)
{
  float ux;
  if(x<minimo())
    ux=0;
  if(x<primerCorte()&&x>=minimo())
    ux=(x-minimo())/(primerCorte()-minimo());
  if(x<segundoCorte()&&x>=primerCorte())
    ux=1;
  if(x<maximo()&&x>=segundoCorte())
    ux=(maximo()-x)/(maximo()-segundoCorte());
  if(x>=maximo())
    ux=0;
  if(ux<0.0001)
    ux=0;
  return ux ;
}


inline float ConjuntoGamma::pertenencia(float x)
{
  float ux;
  if(x<minimo())
    ux=0;
  if(x<primerCorte()&&x>=minimo())
    ux=(x-minimo())/(primerCorte()-minimo());
  if(x<maximo()&&x>=primerCorte())
    ux=1;
  if(x>=maximo())
    ux=1;
  if(ux<0.0001)
    ux=0;
  return ux ;
}

inline float ConjuntoZ::pertenencia(float x)
{
  float ux;
  if(x<minimo())
    ux=1;
  if(x<primerCorte())
    ux=1;
```

```
  if(x<(primerCorte()+maximo())/2&&x>=primerCorte())
    {
      ux=(x-primerCorte())/(maximo()-primerCorte());
      ux=1-2*ux*ux;
    }
  if(x<maximo()&&x>=(primerCorte()+maximo())/2)
    {
      ux=(x-maximo())/(maximo()-primerCorte());
      ux=2*ux*ux;
    }
  if(x>=maximo())
    ux=0;
  if(ux<0.0001)
    ux=0;
  return ux ;
}

inline float ConjuntoCampana::pertenencia(float x)
{
  float ux;
  if(x<minimo())
    ux=0;
  if(x<(primerCorte()+minimo())/2&&x>=minimo())
    {
      ux=(x-minimo())/(primerCorte()-minimo());
      ux=2*ux*ux;
    }
  if(x<primerCorte()&&x>=(primerCorte()+minimo())/2)
    {
      ux=(x-primerCorte())/(primerCorte()-minimo());
      ux=1-2*ux*ux;
    }
  if(x<(primerCorte()+maximo())/2&&x>=primerCorte())
    {
      ux=(x-primerCorte())/(maximo()-primerCorte());
      ux=1-2*ux*ux;
    }
  if(x<maximo()&&x>=(primerCorte()+maximo())/2)
    {
      ux=(x-maximo())/(maximo()-primerCorte());
      ux=2*ux*ux;
    }
  if(x>=maximo())
    ux=0;
  if(ux<0.0001)
    ux=0;
```

```
  return ux ;
}


inline float ConjuntoS::pertenencia(float x)
{
  float ux;
  if(x<minimo())
    ux=0;
  if(x<(primerCorte()+minimo())/2&&x>=minimo())
    {
      ux=(x-minimo())/(primerCorte()-minimo());
      ux=2*ux*ux;
    }
  if(x<primerCorte()&&x>=(primerCorte()+minimo())/2)
    {
      ux=(x-primerCorte())/(primerCorte()-minimo());
      ux=1-2*ux*ux;
    }
  if(x>=primerCorte())
    ux=1;
  if(ux<0.0001)
    ux=0;
  return ux ;
}

inline float ConjuntoPiCampana::pertenencia(float x)
{
  float ux;
  if(x<minimo())
    ux=0;
  if(x<(primerCorte()+minimo())/2&&x>=minimo())
    {
      ux=(x-minimo())/(primerCorte()-minimo());
      ux=2*ux*ux;
    }
  if(x<primerCorte()&&x>=(primerCorte()+minimo())/2)
    {
      ux=(x-primerCorte())/(primerCorte()-minimo());
      ux=1-2*ux*ux;
    }
  if(x<segundoCorte()&&x>=primerCorte())
    ux=1;
  if(x<(segundoCorte()+maximo())/2&&x>=segundoCorte())
    {
      ux=(x-segundoCorte())/(maximo()-segundoCorte());
```

```
      ux=1-2*ux*ux;
    }
  if(x<maximo()&&x>=(segundoCorte()+maximo())/2)
    {
      ux=(x-maximo())/(maximo()-segundoCorte());
      ux=2*ux*ux;
    }
  if(x>=maximo())
    ux=0;
  if(ux<0.0001)
    ux=0;
  return ux ;
}


inline float ConjuntoSinglenton::pertenencia(float x)
{
  float ux;
  if(x<minimo())
    ux=0;
  if(x<maximo()&&x>=minimo())
    ux=1;
  if(x>=maximo())
    ux=0;
  if(ux<0.0001)
    ux=0;
  return ux ;
}



////////////////////////////////////////

class Difusor:public ConjuntoDifuso
{
public:
  Difusor()
  {
    NumeroPuntos=1;
  }
  ~Difusor()
  {}
  void numeroPuntos(int num)
  {
    if(num<1)
      num=1;
```

```
    NumeroPuntos=num;
    dx=(maximo()-minimo())/(num+1);
  }
  void ancho(float soporte)
  {
    float centro,mn,mx;
    centro=(maximo()+minimo())/2;
    mn=centro-soporte/2;
    mx=centro+soporte/2;
    minimo(mn);
    maximo(mx);
    numeroPuntos(NumeroPuntos);
  }
  int numeroPuntos()
  {
    return NumeroPuntos;
  }
  float intervalo()
  {
    return dx;
  }
  float centro()
  {
    return Centro;
  }
  float centroAltura(){return 0.0;}
  virtual float pertenencia(float)=0;
  virtual float minimo()=0;
  virtual float maximo()=0;
  virtual void minimo(float mn)=0;
  virtual void maximo(float mx)=0;
  virtual void entrada(float)=0;
  void puntosClaves(float* ){}
  void nuevoPuntoClave(int, float ){}
protected:
  int NumeroPuntos;
  float dx;
  float Centro;
};

class DifusorTriangulo:
public virtual ConjuntoTriangulo,public virtual Difusor
{
public:
  DifusorTriangulo(float x, float me, float ma):
         ConjuntoTriangulo(x-me,x,x+ma)
```

```
  {
    Centro=x;
    menos=me;
    mas=ma;
  }
  ~DifusorTriangulo()
  {
  }
  void entrada(float x1)
  {
    ConjuntoTriangulo::minimo(x1-menos);
    ConjuntoTriangulo::maximo(x1+mas);
    ConjuntoTriangulo::primerCorte(x1);
    Centro=x1;
  }
  float pertenencia(float x)
  {
    return ConjuntoTriangulo::pertenencia(x);
  }
  float minimo()
  {
    return ConjuntoTriangulo::minimo();
  }
  float maximo()
  {
    return ConjuntoTriangulo::maximo();
  }
  float primerCorte()
  {
    return ConjuntoTriangulo::primerCorte();
  }
  void primerCorte(float pc)
  {
    ConjuntoTriangulo::primerCorte(pc);
  }
  void minimo(float mn)
  {
    ConjuntoTriangulo::minimo(mn);
  }
  void maximo(float mx)
  {
    ConjuntoTriangulo::maximo(mx);
  }
protected:
  float mas;
  float menos;
```

```
};

class DifusorPi:
public virtual ConjuntoPi,public virtual Difusor
{
public:
  DifusorPi(float x, float me1, float me2, float ma1, float ma2):
         ConjuntoPi(x-me1,x-me2,x+ma1,x+ma2)
  {
    Centro=x;
    menos1=me1;
    menos2=me2;
    mas1=ma1;
    mas2=ma2;
  }
  ~DifusorPi()
  {
  }
  void entrada(float x1)
  {
    ConjuntoPi::minimo(x1-menos1);
    ConjuntoPi::maximo(x1+mas2);
    ConjuntoPi::primerCorte(x1-menos2);
    ConjuntoPi::segundoCorte(x1+mas1);
    Centro=x1;
  }
  float pertenencia(float x)
  {
    return ConjuntoPi::pertenencia(x);
  }
  float minimo()
  {
    return ConjuntoPi::minimo();
  }
  float maximo()
  {
    return ConjuntoPi::maximo();
  }
  float primerCorte()
  {
    return ConjuntoPi::primerCorte();
  }
  float segundoCorte()
  {
    return ConjuntoPi::segundoCorte();
  }
```

```
  void primerCorte(float pc)
  {
    ConjuntoPi::primerCorte(pc);
  }
  void segundoCorte(float sc)
  {
    ConjuntoPi::segundoCorte(sc);
  }
  void minimo(float mn)
  {
    ConjuntoPi::minimo(mn);
  }
  void maximo(float mx)
  {
    ConjuntoPi::maximo(mx);
  }
protected:
  float mas1;
  float mas2;
  float menos1;
  float menos2;
};

class DifusorCampana:
public virtual ConjuntoCampana,public virtual Difusor
{
public:
  DifusorCampana(float x, float me, float ma):
         ConjuntoCampana(x-me,x,x+ma)
  {
    Centro=x;
    menos=me;
    mas=ma;
  }
  ~DifusorCampana()
  {
  }
  void entrada(float x1)
  {
    ConjuntoCampana::minimo(x1-menos);
    ConjuntoCampana::maximo(x1+mas);
    ConjuntoCampana::primerCorte(x1);
    Centro=x1;
  }
  float pertenencia(float x)
  {
```

```
    return ConjuntoCampana::pertenencia(x);
  }
  float minimo()
  {
    return ConjuntoCampana::minimo();
  }
  float maximo()
  {
    return ConjuntoCampana::maximo();
  }
  float primerCorte()
  {
    return ConjuntoCampana::primerCorte();
  }
  void primerCorte(float pc)
  {
    ConjuntoCampana::primerCorte(pc);
  }
  void minimo(float mn)
  {
    ConjuntoCampana::minimo(mn);
  }
  void maximo(float mx)
  {
    ConjuntoCampana::maximo(mx);
  }
protected:
  float mas;
  float menos;
};

class DifusorPiCampana:
public virtual ConjuntoPiCampana,public virtual Difusor
{
public:
  DifusorPiCampana(float x, float me1, float me2, float ma1, float ma2):
         ConjuntoPiCampana(x-me1,x-me2,x+ma1,x+ma2)
  {
    Centro=x;
    menos1=me1;
    menos2=me2;
    mas1=ma1;
    mas2=ma2;
  }
  ~DifusorPiCampana()
  {
```

```
  }
  void entrada(float x1)
  {
    ConjuntoPiCampana::minimo(x1-menos1);
    ConjuntoPiCampana::maximo(x1+mas2);
    ConjuntoPiCampana::primerCorte(x1-menos2);
    ConjuntoPiCampana::segundoCorte(x1+mas1);
    Centro=x1;
  }
  float pertenencia(float x)
  {
    return ConjuntoPiCampana::pertenencia(x);
  }
  float minimo()
  {
    return ConjuntoPiCampana::minimo();
  }
  float maximo()
  {
    return ConjuntoPiCampana::maximo();
  }
  float primerCorte()
  {
    return ConjuntoPiCampana::primerCorte();
  }
  float segundoCorte()
  {
    return ConjuntoPiCampana::segundoCorte();
  }
  void primerCorte(float pc)
  {
    ConjuntoPiCampana::primerCorte(pc);
  }
  void segundoCorte(float sc)
  {
    ConjuntoPiCampana::segundoCorte(sc);
  }
  void minimo(float mn)
  {
    ConjuntoPiCampana::minimo(mn);
  }
  void maximo(float mx)
  {
    ConjuntoPiCampana::maximo(mx);
  }
protected:
```

```
  float mas1;
  float mas2;
  float menos1;
  float menos2;
};


class DifusorSinglenton:
public virtual ConjuntoSinglenton, public virtual Difusor
{
public:
  DifusorSinglenton(float x, float de):ConjuntoSinglenton(x,de)
  {
    Centro=x;
  }
  ~DifusorSinglenton()
  {
  }
  void entrada(float x1)
  {
    ConjuntoSinglenton::minimo(x1-delta()/2);
    ConjuntoSinglenton::maximo(x1+delta()/2);
    Centro=x1;
  }
  float pertenencia(float x)
  {
    return ConjuntoSinglenton::pertenencia(x);
  }
  float minimo()
  {
    return ConjuntoSinglenton::minimo();
  }
  float maximo()
  {
    return ConjuntoSinglenton::maximo();
  }
  void minimo(float mn)
  {
    ConjuntoSinglenton::minimo(mn);
  }
  void maximo(float mx)
  {
    ConjuntoSinglenton::maximo(mx);
  }
protected:
};
```

```
////////////////////////////////////

class Variable
{
public:
  Variable(int num=10)
  {
    Conjuntos = new ListaConjuntos(num);
    DifusorEntrada= new DifusorTriangulo(0.0,0.1,0.1);
    NombreVariable=0;
  }
  ~Variable();
  Difusor *difusorEntrada()
  {
    return DifusorEntrada;
  }
  void difusorEntrada(Difusor *dif)
  {
    delete DifusorEntrada;
    DifusorEntrada=dif;
  }
  char* nombreVariable()
  {
    return NombreVariable;
  }
  char* nombreVariable(char* s)
  {
    delete[] NombreVariable;
    NombreVariable=new char[strlen(s)+1];
    strcpy(NombreVariable,s);
    return NombreVariable;
  }
  void adicionarConjuntos(ConjuntoDifuso* cd)
  {
    Conjuntos->Add(cd);
  }
  void eliminarConjuntos(int cd)
  {
    Conjuntos->Destroy(cd);
  }
  void limpiarListaConjuntos();
  int numeroConjuntos()
  {
    return Conjuntos->GetItemsInContainer();
  }
```

```
  void numeroIntervalos(int num)
  {
    NumeroIntervalos=num;
    Intervalo=(rangoMaximo()-rangoMinimo())/(NumeroIntervalos);
  }
  int numeroIntervalos()
  {
    return NumeroIntervalos;
  }
  float intervalo()
  {
    return Intervalo;
  }

  ConjuntoDifuso* conjunto(int conj);
  float pertenencia(ConjuntoDifuso*, float);
  float pertenencia(int, float);
  float pertenenciaDifusor(float);
  float rangoMinimo(){return RangoMinimo;}
  void rangoMinimo(float rm){RangoMinimo=rm;}
  float rangoMaximo(){return RangoMaximo;}
  void rangoMaximo(float rm){RangoMaximo=rm;}
  BOOL operator==(const Variable& other)
  {
    return ( strcmp( NombreVariable,other.NombreVariable)&
    ( RangoMinimo == other.RangoMinimo)&
    ( RangoMaximo == other.RangoMaximo)&
    ( DifusorEntrada == other.DifusorEntrada)&
    ( Conjuntos == other.Conjuntos) );
  }
protected:
  typedef Arreglo<ConjuntoDifuso> ListaConjuntos;

  ListaConjuntos *Conjuntos;
  Difusor *DifusorEntrada;
  float RangoMinimo;
  float RangoMaximo;
  int NumeroIntervalos;
  float Intervalo;
  char *NombreVariable;
};

class Universo
{
public:
  Universo(int num=10)
```

```
{
  Variables=new ListaVariables(num);
}
~Universo();
void adicionarVariable(Variable *var)
{
  Variables->Add(var);
}
void eliminarVariable(int var)
{
  Variables->Destroy(var);
}
int numeroVariables()
{
  return Variables->GetItemsInContainer();
}
Variable *variable(int numVar);
void limpiarListaVariables();
Difusor *difusor(int num);
float intervaloDifusor(int numVar)
{
  Difusor *dif;
  dif=difusor(numVar);
  return dif->intervalo();
}
void numeroPuntosDifusor(int numVar,int numPuntos)
{
  Difusor *dif;
  dif=difusor(numVar);
  dif->numeroPuntos(numPuntos);
}
int numeroPuntosDifusor(int numVar)
{
  Difusor *dif;
  dif=difusor(numVar);
  return dif->numeroPuntos();
}
float pertenenciaDifusor(int numVar, float x)
{
  Variable *var;
  var=variable(numVar);
  return pertenenciaDifusor(var,x);
}
char *nombreVariable(int numVar)
{
  Variable *var;
```

```
  var=variable(numVar);
  return nombreVariable(var);
}
void nombreVariable(char *nom,int numVar)
{
  Variable *var;
  var=variable(numVar);
  nombreVariable(nom,var);
}
void entradaReal(int numVar, float x)
{
  Difusor *dif;
  dif=difusor(numVar);
  entradaReal(dif,x);
}
void entradaReal(float *ent);
int numeroConjuntosEnVariable(int numVar)
{
  Variable *var;
  var=variable(numVar);
  return numeroConjuntosEnVariable (var);
}
int numeroIntervalosEnVariable(int numVar)
{
  Variable *var;
  var=variable(numVar);
  return numeroIntervalosEnVariable (var);
}
void numeroIntervalosEnVariable(int numVar,int intervalos)
{
  Variable *var;
  var=variable(numVar);
  var->numeroIntervalos(intervalos);
}
float intervaloEnVariable(int numVar)
{
  Variable *var;
  var=variable(numVar);
  return intervaloEnVariable (var);
}
ConjuntoDifuso *conjuntoEnVariable(int numVar, int numCon)
{
  Variable *var;
  var=variable(numVar);
  return conjuntoEnVariable(var,numCon);
}
```

```
float rangoMinimoVariable(int numVar)
{
  Variable *var;
  var=variable(numVar);
  return rangoMinimoVariable(var);
}
void rangoMinimoVariable(float rm,int numVar)
{
  Variable *var;
  var=variable(numVar);
  rangoMinimoVariable(rm,var);
}
float rangoMaximoVariable(int numVar)
{
  Variable *var;
  var=variable(numVar);
  return rangoMaximoVariable(var);
}
void rangoMaximoVariable(float rm,int numVar)
{
  Variable *var;
  var=variable(numVar);
  rangoMaximoVariable(rm,var);
}
float minimoEnConjunto(int numVar, int numCon)
{
  ConjuntoDifuso *cd;
  cd=conjuntoEnVariable(numVar, numCon);
  return minimoEnConjunto(cd);
}
float maximoEnConjunto(int numVar, int numCon)
{
  ConjuntoDifuso *cd;
  cd=conjuntoEnVariable(numVar, numCon);
  return maximoEnConjunto(cd);
}
float minimoEnDifusor(int numVar)
{
  Difusor *dif;
  dif=difusor(numVar);
  return minimoEnDifusor(dif);
}
float maximoEnDifusor(int numVar)
{
  Difusor *dif;
  dif=difusor(numVar);
```

```
    return maximoEnDifusor(dif);
  }
  float pertenenciaVariable(int numVar, int numCon, float x)
  {
    ConjuntoDifuso *cd;
    cd=conjuntoEnVariable(numVar, numCon);
    return pertenenciaVariable(cd,x);
  }
  float centroEnDifusor(int numVar)
  {
    float x;
    x=difusor(numVar)->centro();;
    return x;
  }

protected:
  typedef Arreglo<Variable> ListaVariables;

  ListaVariables *Variables;

private:
  Difusor *difusor(Variable *var)
  {
    return var->difusorEntrada();
  }
  float pertenenciaDifusor(Variable *var, float x)
  {
    return var->pertenenciaDifusor(x);
  }
  char *nombreVariable(Variable *var)
  {
    return var->nombreVariable();
  }
  void nombreVariable(char *nom,Variable *var)
  {
    var->nombreVariable(nom);
  }
  void entradaReal(Difusor *dif, float x)
  {
    dif->entrada(x);
  }
  void entradaReal(Variable *var,float x)
  {
    Difusor *dif;
    dif=var->difusorEntrada();
    dif->entrada(x);
```

```
}
int numeroConjuntosEnVariable(Variable *var)
{
  return var->numeroConjuntos();
}
int numeroIntervalosEnVariable(Variable *var)
{
  return var->numeroIntervalos();
}
float intervaloEnVariable(Variable *var)
{
  return var->intervalo();
}
ConjuntoDifuso *conjuntoEnVariable(Variable *var, int numCon)
{
  return var->conjunto(numCon);
}
float rangoMinimoVariable(Variable *var)
{
  return var->rangoMinimo();
}
void rangoMinimoVariable(float rm,Variable *var)
{
  var->rangoMinimo(rm);
}
float rangoMaximoVariable(Variable *var)
{
  return var->rangoMaximo();
}
void rangoMaximoVariable(float rm,Variable *var)
{
  var->rangoMaximo(rm);
}
float minimoEnConjunto(Variable *var, int numCon)
{
  ConjuntoDifuso *cd;
  cd=conjuntoEnVariable(var, numCon);
  return minimoEnConjunto(cd);
}
float minimoEnConjunto(ConjuntoDifuso *cd)
{
  return cd->minimo();
}
float maximoEnConjunto(Variable *var, int numCon)
{
  ConjuntoDifuso *cd;
```

```
    cd=conjuntoEnVariable(var, numCon);
    return maximoEnConjunto(cd);
  }
  float maximoEnConjunto(ConjuntoDifuso *cd)
  {
    return cd->maximo();
  }
  float minimoEnDifusor(Variable *var)
  {
    Difusor *dif;
    dif=difusor(var);
    return minimoEnDifusor(dif);
  }
  float minimoEnDifusor(Difusor *dif)
  {
    return dif->minimo();
  }
  float maximoEnDifusor(Variable *var)
  {
    Difusor *dif;
    dif=difusor(var);
    return maximoEnDifusor(dif);
  }
  float maximoEnDifusor(Difusor *dif)
  {
    return dif->maximo();
  }
  float pertenenciaVariable(Variable *var, int numCon, float x)
  {
    ConjuntoDifuso *cd;
    cd=conjuntoEnVariable(var, numCon);
    return pertenenciaVariable(cd,x);
  }
  float pertenenciaVariable(ConjuntoDifuso *cd,float x)
  {
    return cd->pertenencia(x);
  }
};


class Norma
{
public:
  Norma()
  {
  }
```

```
  ~Norma()
  {
  }
  virtual float ToSNorm()=0;
  virtual float opera(float, float)=0;
protected:
};

class T_Norma:public Norma
{
public:
  T_Norma(){}
  ~T_Norma(){}
  float ToSNorm()
  {
    return 1.0;
  }
  virtual float opera(float, float)=0;
protected:
};

class S_Norma:public Norma
{
public:
  S_Norma(){}
  ~S_Norma(){}
  float ToSNorm()
  {
    return 0.0;
  }
  virtual float opera(float, float)=0;
protected:
};

//////////////////// T_Norma

class Producto:public T_Norma
{
public:
  Producto()
  {
  }
  ~Producto(){}
  float opera(float x, float y)
  {
    float z;
```

```
    z=x*y;
    return z;
  }
protected:
};

class Minimo:public T_Norma
{
public:
  Minimo()
  {
  }
  ~Minimo(){}
  float opera(float x, float y)
  {
    float z;
    if(x<y)
    {
      z=x;
    }else
    {
      z=y;
    }
    return z;
  }
protected:
};

class ProductoAcotado:public T_Norma
{
public:
  ProductoAcotado()
  {
  }
  ~ProductoAcotado(){}
  float opera(float x, float y)
  {
    float z;
    z=x+y-1;
    if(z<0)
    {
      z=0;
    }
    return z;
  }
protected:
```

```
};

class ProductoDrastico:public T_Norma
{
public:
  ProductoDrastico()
  {
  }
  ~ProductoDrastico(){}
  float opera(float x, float y)
  {
    float z;
    if(y==1)
    {
      z=x;
    }
    if(x==1)
    {
      z=y;
    }
    if(x<1&&y<1)
    {
      z=0;
    }
    return z;
  }
protected:
};

class FamiliaTp:public T_Norma
{
public:
  FamiliaTp()
  {
    p=1;
  }
  FamiliaTp(float parametro)
  {
    p=parametro;
  }
  ~FamiliaTp(){}
  float opera(float x, float y)
  {
    float z;
    z=pow(1-x,p)+pow(1-y,p)-pow(1-x,p)*pow(1-y,p);
    z=1-pow(z,(1/p));
```

```
    return z;
  }
protected:
  float p;
};

class FamiliaHp:public T_Norma
{
public:
  FamiliaHp()
  {
    p=1;
  }
  FamiliaHp(float parametro)
  {
    p=parametro;
  }
  ~FamiliaHp(){}
  float opera(float x, float y)
  {
    float z;
    z=p-(1-p)*(x+y-x*y);
    z=x*y/z;
    return z;
  }
protected:
  float p;
};

class FamiliaFp:public T_Norma
{
public:
  FamiliaFp()
  {
    p=2;
  }
  FamiliaFp(float parametro)
  {
    p=parametro;
  }
  ~FamiliaFp(){}
  float opera(float x, float y)
  {
    float z;
    z=1+(pow(p,x)-1)*(pow(p,y)-1)/(p-1);
    z=log(z)/log(p);
```

```
    return z;
  }
protected:
  float p;
};

class FamiliaYp:public T_Norma
{
public:
  FamiliaYp()
  {
    p=1;
  }
  FamiliaYp(float parametro)
  {
    p=parametro;
  }
  ~FamiliaYp(){}
  float opera(float x, float y)
  {
    float z;
    z=pow(1-x,p)+pow(1-y,p);
    z=pow(z,(1/p));
    if(z>1)
    {
      z=1;
    }
    z=1-z;
    return z;
  }
protected:
  float p;
};

class FamiliaAp:public T_Norma
{
public:
  FamiliaAp()
  {
    p=1;
  }
  FamiliaAp(float parametro)
  {
    p=parametro;
  }
  ~FamiliaAp(){}
```

```
  float opera(float x, float y)
  {
    float z;
    z=x;
    if(y>z)
    {
      z=y;
    }
    if(p>z)
    {
      z=p;
    }
    z=x*y/z;
    return z;
  }
protected:
  float p;
};



///////////////// S-Normas

class Maximo:public S_Norma
{
public:
  Maximo()
  {
  }
  ~Maximo(){}
  float opera(float x, float y)
  {
    float z;
    if(x>y)
    {
      z=x;
    }else
    {
      z=y;
    }
    return z;
  }
protected:
};

class SumaAcotada:public S_Norma
```

```
{
public:
  SumaAcotada()
  {
  }
  ~SumaAcotada(){}
  float opera(float x, float y)
  {
    float z;
    z=x+y;
    if(z>1)
    {
      z=1;
    }
    return z;
  }
protected:
};

class SumaDrastica:public S_Norma
{
public:
  SumaDrastica()
  {
  }
  ~SumaDrastica(){}
  float opera(float x, float y)
  {
    float z;
    if(y==0)
    {
      z=x;
    }
    if(x==0)
    {
      z=y;
    }
    if(x>0&&y>0)
    {
      z=1;
    }
    return z;
  }
protected:
};
```

```
class FamiliaSp:public S_Norma
{
public:
  FamiliaSp()
  {
    p=1;
  }
  FamiliaSp(float parametro)
  {
    p=parametro;
  }
  ~FamiliaSp(){}
  float opera(float x, float y)
  {
    float z;
    z=x+y+p*x*y;
    if(z>1)
    {
      z=1;
    }
    return z;
  }
protected:
  float p;
};


class Implicacion
{
public:
  Implicacion()
  {
  }
  ~Implicacion()
  {
  }
  virtual float implica(float,float)=0;
  virtual float defecto()=0;
protected:
};

class ImplicacionT_Norma:public Implicacion
{
public:
  ImplicacionT_Norma(){}
  ~ImplicacionT_Norma(){}
```

```
  float defecto()
  {
    return 0;
  }
  virtual float implica(float x, float y)=0;
protected:
};

class ImplicacionIf_Then:public Implicacion
{
public:
  ImplicacionIf_Then(){}
  ~ImplicacionIf_Then(){}
  float defecto()
  {
    return 1;
  }
  virtual float implica(float x, float y)=0;
protected:
};


class ImplicacionProducto:public ImplicacionT_Norma
{
public:
  ImplicacionProducto()
  {
  }
  ~ImplicacionProducto()
  {
  }
  float implica(float x, float y)
  {
    float z;
    z=x*y;
    return z;
  }
protected:
};

class ImplicacionMinimo:public ImplicacionT_Norma
{
public:
  ImplicacionMinimo()
  {
  }
```

```
  ~ImplicacionMinimo()
  {
  }
  float implica(float x, float y)
  {
    float z;
    if(x<y)
      z=x;
    else
      z=y;
    return z;
  }
protected:
};

class ImplicacionKleenDienes:public ImplicacionIf_Then
{
public:
  ImplicacionKleenDienes()
  {
  }
  ~ImplicacionKleenDienes()
  {
  }
  float implica(float x, float y)
  {
    float rel;
    x=1-x;
    if(x>y)
      rel=x;
    else
      rel=y;
    return(rel);
  }
protected:
};


class ImplicacionLukasiewicz:public ImplicacionIf_Then
{
public:
  ImplicacionLukasiewicz()
  {
  }
  ~ImplicacionLukasiewicz()
  {
```

```
  }
  float implica(float x, float y)
  {
    float rel;
    rel=1-x+y;
    if(rel>1)
      rel=1;
    return(rel);
  }
protected:
};


class ImplicacionZadeh:public ImplicacionIf_Then
{
public:
  ImplicacionZadeh()
  {
  }
  ~ImplicacionZadeh()
  {
  }
  float implica(float x, float y)
  {
    float rel;
    if(x<y)
      rel=x;
    else
      rel=y;
    x=1-x;
    if(rel<x)
      rel=x;
    return(rel);
  }
protected:
};


class ImplicacionEstocastica:public ImplicacionIf_Then
{
public:
  ImplicacionEstocastica()
  {
  }
  ~ImplicacionEstocastica()
  {
```

```
  }
  float implica(float x, float y)
  {
    float rel;
    rel=x*y;
    x=1-x;
    if(rel<x)
      rel=x;
    return(rel);
  }
protected:
};


class ImplicacionGoguen:public ImplicacionIf_Then
{
public:
  ImplicacionGoguen()
  {
  }
  ~ImplicacionGoguen()
  {
  }
  float implica(float x, float y)
  {
    float rel;
    if(x>0.00001)
      rel=y/x;
    else
      rel=1000;
    if(rel>1)
      rel=1;
    return(rel);
  }
protected:
};


class ImplicacionGodel:public ImplicacionIf_Then
{
public:
  ImplicacionGodel()
  {
  }
  ~ImplicacionGodel()
  {
```

```
  }
  float implica(float x, float y)
  {
    float rel;
    if(x<=y)
      rel=1;
    else
      rel=y;
    return(rel);
  }
protected:
};


class ImplicacionAguda:public ImplicacionIf_Then
{
public:
  ImplicacionAguda()
  {
  }
  ~ImplicacionAguda()
  {
  }
  float implica(float x, float y)
  {
    float rel;
    if(x<=y)
      rel=1;
    else
      rel=0;
    return(rel);
  }
protected:
};

//////////////////////////////////////

class Regla
{
public:
  Regla(int numEntradas, int numSalidas)
  {
    Antecedente=new int[numEntradas];
    Consecuente=new int[numSalidas];
    Modificadores=new float[numEntradas];
  }
```

```
  ~Regla()
  {
    delete[] Antecedente;
    delete[] Consecuente;
    delete[] Modificadores;
  }
  int conjuntoEntrada(int numVar)
  {
    return Antecedente[numVar];
  }
  void conjuntoEntrada(int numVar,int numCon)
  {
    Antecedente[numVar]=numCon;
  }
  int conjuntoSalida(int numVar)
  {
    return Consecuente[numVar];
  }
  void conjuntoSalida(int numVar,int numCon)
  {
    Consecuente[numVar]=numCon;
  }
  float modificador(int numVar)
  {
    return Modificadores[numVar];
  }
  void modificador(int numVar,float modif)
  {
    Modificadores[numVar]=modif;
  }
  void certeza(float cer)
  {
    Certeza=cer;
  }
  float certeza()
  {
    return Certeza;
  }
  BOOL operator==(const Regla& other)
  {
    return ( ( Antecedente == other.Antecedente)&
    ( Consecuente == other.Consecuente)&
    ( Modificadores == other.Modificadores) );
  }

protected:
```

```
  int *Antecedente;
  int *Consecuente;
  float *Modificadores;

   float Certeza;
};


class MaquinaInferencia
{
public:
  MaquinaInferencia(Universo *ent,Universo *sal,int numReg);
  ~MaquinaInferencia();
  Implicacion* implicacion()
  {
    return Implicaciones;
  }
  void implicacion(Implicacion *imp)
  {
    delete Implicaciones;
    Implicaciones=imp;
  }
  Norma *and()
  {
    return And;
  }
  void and(Norma *nor)
  {
    delete And;
    And=nor;
  }
  Norma *composicion()
  {
    return Composicion;
  }
  void composicion(Norma *nor)
  {
    delete Composicion;
    Composicion=nor;
  }

  Regla *regla(int numRegla);
  void conjuntoEntrada(int numRegla, int numVar, int numCon)
  {
    regla(numRegla)->conjuntoEntrada(numVar,numCon);
  }
```

```
ConjuntoDifuso *conjuntoEntrada(int numRegla, int numVar)
{
  return Entradas->conjuntoEnVariable(numVar,regla(numRegla)
       ->conjuntoEntrada(numVar));
}
int numConjuntoEntrada(int numRegla, int numVar)
{
  return regla(numRegla)->conjuntoEntrada(numVar);
}
void modificador(int numRegla, int numVar, float modif)
{
  regla(numRegla)->modificador(numVar,modif);
}
float modificador(int numRegla, int numVar)
{
  return regla(numRegla)->modificador(numVar);
}
void conjuntoSalida(int numRegla, int numVar, int numCon)
{
  regla(numRegla)->conjuntoSalida(numVar,numCon);
}
ConjuntoDifuso *conjuntoSalida(int numRegla, int numVar)
{
  return Salidas->conjuntoEnVariable(numVar,regla(numRegla)
      ->conjuntoSalida(numVar));
}
int numConjuntoSalida(int numRegla, int numVar)
{
  return regla(numRegla)->conjuntoSalida(numVar);
}
Universo *entradas()
{
  return Entradas;
}
Universo *salidas()
{
  return Salidas;
}
int numeroReglas()
{
  return NumeroReglas;
}
int numeroEntradas()
{
  return NumeroEntradas;
}
```

```
  int numeroSalidas()
  {
    return NumeroSalidas;
  }
  void limpiarMaquinaInferencia();
  void adicionarRegla(Regla *reg)
  {
    BaseReglas->Add(reg);
    NumeroReglas++;
  }
  void eliminarRegla(int num)
  {
    BaseReglas->Destroy(num);
    NumeroReglas--;
  }
  void actualizarEntradas(float *ent);
  float pertenenciaDifusores(float *ent);
  float pertenenciaImplicacion(int numSal,int numRegla, float *ent,
        float sal);
  float pertenenciaConsecuente(int numSal,int numRegla,float sal);
  float pertenenciaAntecedente(int numRegla,float *ent);
  float pertenenciaComposicion(int numVar, int numRegla,float sal);
  int activarRegla(int numregla);
  void desocuparBaseReglas();
  void EntrenaUniversoFijo(float *antecedente, float *consecuente);
  void EntrenaUniversoVariable(float *antecedente, float *consecuente);
  void llenarRegla(Regla *rg,float *antec, float *consec);
  int compararAntec(Regla *rg1, Regla *rg2);
//protected:
  typedef Arreglo<Regla> ListaReglas;

  ListaReglas *BaseReglas;
  Universo *Entradas;
  Universo *Salidas;

  Implicacion *Implicaciones;
  Norma *Composicion;
  Norma *And;

  int NumeroReglas;
  int NumeroEntradas;
  int NumeroSalidas;
};

class Concresor
{
```

```
public:
  Concresor(MaquinaInferencia *maq, int numVar, Norma *nor)
  {
    Motor=maq;
    NumeroVariable=numVar;
    Conjuncion=nor;
  }
  ~Concresor()
  {
  }
  int numeroVariable()
  {
    return NumeroVariable;
  }
  MaquinaInferencia *motor()
  {
    return Motor;
  }
  void motor(MaquinaInferencia *maq)
  {
    Motor=maq;
  }
  float defecto()
  {
    return Motor->implicacion()->defecto();
  }
  Norma *conjuncion()
  {
    return Conjuncion;
  }
  void conjuncion(Norma *nor)
  {
    Conjuncion=nor;
  }
  virtual float salidaConcreta(float *ent)=0;
  BOOL operator==(const Concresor& other)
  {
    return ( ( Motor == other.Motor)&
    ( NumeroVariable == other.NumeroVariable)&
    ( Conjuncion == other.Conjuncion) );
  }
protected:
  MaquinaInferencia *Motor;
  int NumeroVariable;
  Norma *Conjuncion;
};
```

```
class PrimerMaximo:public Concresor
{
public:
  PrimerMaximo(MaquinaInferencia *maq, int numVar, Norma *nor):
        Concresor(maq,numVar,nor)
  {}
  ~PrimerMaximo(){}
  float salidaConcreta(float *ent);
protected:
};

class UltimoMaximo:public Concresor
{
public:
  UltimoMaximo(MaquinaInferencia *maq, int numVar, Norma *nor):
         Concresor(maq,numVar,nor)
  {}
  ~UltimoMaximo(){}
  float salidaConcreta(float *ent);
protected:
};

class MediaDeMaximos:public Concresor
{
public:
  MediaDeMaximos(MaquinaInferencia *maq, int numVar, Norma *nor):
         Concresor(maq,numVar,nor)
  {}
  ~MediaDeMaximos(){}
  float salidaConcreta(float *ent);
protected:
};

class CentroDeGravedad:public Concresor
{
public:
  CentroDeGravedad(MaquinaInferencia *maq, int numVar, Norma *nor):
         Concresor(maq,numVar,nor)
  {}
  ~CentroDeGravedad(){}
  float salidaConcreta(float *ent);
protected:
};
```

```
class Altura:public Concresor
{
public:
  Altura(MaquinaInferencia *maq, int numVar, Norma *nor):
          Concresor(maq,numVar,nor)
  {}
  ~Altura(){}
  float salidaConcreta(float *ent);
protected:
};

class BloqueConcrecion
{
public:
  BloqueConcrecion(MaquinaInferencia *maq)
  {
    Motor=maq;
    int numSal;
    numSal=Motor->numeroSalidas()+1;
    Concresores=new ListaConcresores(numSal);
  }
  ~BloqueConcrecion()
  {
    limpiarListaConcresores();
    delete Concresores;
  }
  void adicionarConcresor(Concresor* conc)
  {
    Concresores->Add(conc);
  }
  int numeroConcresores()
  {
    return Motor->salidas()->numeroVariables();
  }
  void eliminarConcresor(int conc)
  {
    Concresores->Destroy(conc);
  }
  Norma *conjuncion()
  {
    return concresor(0)->conjuncion();
  }
  void conjuncion(Norma *nor);
  void limpiarListaConcresores();
  void motor(MaquinaInferencia* maq);
  MaquinaInferencia *motor()
```

```
  {
    return Motor;
  }
  Concresor *concresor(int numSal);
  float salidaConcreta(int numSal,float *ent)
  {
    Concresor *conc;
    conc=concresor(numSal);
    return conc->salidaConcreta(ent);
  }
  void salidaConcreta(float *ent,float *sal);
  void autodefinirBloqueConcrecion(MaquinaInferencia* maq,
          Norma *conjuncion);
protected:
  typedef Arreglo<Concresor> ListaConcresores;

  ListaConcresores *Concresores;
  MaquinaInferencia *Motor;
};

class SistemaLogicaDifusa
{
public:
  SistemaLogicaDifusa()
  {
    entradas=0;
    salidas=0;
    motor=0;
    concreto=0;
  }
  ~SistemaLogicaDifusa()
  {
    delete entradas;
    delete salidas;
    delete motor;
    delete concreto;
  }
  int numeroEntradas()
  {
    return entradas->numeroVariables();
  }
  int numeroSalidas()
  {
    return salidas->numeroVariables();
  }
  char *nombreVariableEntrada(int numVar)
```

```
  {
    return entradas->nombreVariable(numVar);
  }
  char *nombreVariableSalida(int numVar)
  {
    return salidas->nombreVariable(numVar);
  }
  void calcular(float *entra,float *sale)
  {
    concreto->salidaConcreta(entra,sale);
  }
  MaquinaInferencia *Motor(){return motor;}
  void EntrenaUniversoFijo(float *antecedente, float *consecuente)
  {
    motor->EntrenaUniversoFijo(antecedente,consecuente);
  }
  void EntrenaUniversoVariable(float *antecedente, float *consecuente)
  {
    motor->EntrenaUniversoVariable(antecedente,consecuente);
  }

//protected:
  Universo *entradas;
  Universo *salidas;
  MaquinaInferencia *motor;
  BloqueConcrecion *concreto;
};
```

C.2 *Fuzzy.cpp*

```
#ifndef __FUZZY_CPP
#define __FUZZY_CPP
#endif


#ifndef __FUZZY_HPP
#include<fuzzy.hpp>
#endif


void ConjuntoL::nuevoPuntoClave(int punto, float x)
{
  switch(punto)
  {
    case 0: primerCorte(x);break;
```

```
    case 1: maximo(x);break;
    default:break;
  }
}

void ConjuntoTriangulo::nuevoPuntoClave(int punto, float x)
{
  switch(punto)
  {
    case 0: minimo(x);break;
    case 1: primerCorte(x);break;
    case 2: maximo(x);break;
    default:break;
  }
}

void ConjuntoPi::nuevoPuntoClave(int punto, float x)
{
  switch(punto)
  {
    case 0: minimo(x);break;
    case 1: primerCorte(x);break;
    case 2: segundoCorte(x);break;
    case 3: maximo(x);break;
    default:break;
  }
}

void ConjuntoGamma::nuevoPuntoClave(int punto, float x)
{
  switch(punto)
  {
    case 0: minimo(x);break;
    case 1: primerCorte(x);break;
    default:break;
  }
}

void ConjuntoZ::nuevoPuntoClave(int punto, float x)
{
  switch(punto)
  {
    case 0: primerCorte(x);break;
    case 1: maximo(x);break;
    default:break;
  }
```

```
}

void ConjuntoCampana::nuevoPuntoClave(int punto, float x)
{
  switch(punto)
  {
    case 0: minimo(x);break;
    case 1: primerCorte(x);break;
    case 2: maximo(x);break;
    default:break;
  }
}

void ConjuntoPiCampana::nuevoPuntoClave(int punto, float x)
{
  switch(punto)
  {
    case 0: minimo(x);break;
    case 1: primerCorte(x);break;
    case 2: segundoCorte(x);break;
    case 3: maximo(x);break;
    default:break;
  }
}

void ConjuntoS::nuevoPuntoClave(int punto, float x)
{
  switch(punto)
  {
    case 0: minimo(x);break;
    case 1: primerCorte(x);break;
    default:break;
  }
}

void ConjuntoSinglenton::nuevoPuntoClave(int punto, float x)
{
  switch(punto)
  {
    case 0: minimo(x);break;
    case 1: maximo(x);break;
    default:break;
  }
}

Variable::~Variable()
```

```
{
  limpiarListaConjuntos();
  delete Conjuntos;
  delete DifusorEntrada;
  delete[] NombreVariable;
}

ConjuntoDifuso* Variable::conjunto(int conj)
{
  return Conjuntos->dato(conj);
}


void Variable::limpiarListaConjuntos()
{
  int i,numCon;
  numCon=Conjuntos->GetItemsInContainer();
  for(i=0;i<numCon;i++)
  {
    eliminarConjuntos(0);
  }
}

float Variable::pertenencia(int i, float x)
{
  return pertenencia(this->conjunto(i), x);
}

float Variable::pertenencia(ConjuntoDifuso* cd, float x)
{
  float ux;
  ux=cd->pertenencia(x);
  return ux;
}

float Variable::pertenenciaDifusor(float x)
{
  float ux;
  ux=DifusorEntrada->pertenencia(x);
  return ux;
}

Universo::~Universo()
{
  limpiarListaVariables();
  delete Variables;
```

```
}

Difusor* Universo::difusor(int num)
{
  return difusor(Variables->dato(num));
}

Variable* Universo::variable(int num)
{
  return Variables->dato(num);
}

void Universo::limpiarListaVariables()
{
  int i,numVar;
  numVar=numeroVariables();
  for(i=0;i<numVar;i++)
  {
    eliminarVariable(0);
  }
}

void Universo::entradaReal(float *ent)
{
  int i;
  for(i=0;i<numeroVariables();i++)
  {
    entradaReal(i,*(ent+i));
  }
}

MaquinaInferencia::
MaquinaInferencia(Universo *ent,Universo *sal,int numReg)
{
  Entradas=ent;
  Salidas=sal;
  NumeroEntradas=Entradas->numeroVariables();
  NumeroSalidas=Salidas->numeroVariables();
  NumeroReglas=numReg;
  BaseReglas=new ListaReglas(NumeroReglas);
  Regla *rg;
  int i;
  for(i=0;i<NumeroReglas;i++)
  {
    rg=new Regla(NumeroEntradas,NumeroSalidas);
    BaseReglas->Add(rg);
```

```
  }
  Implicaciones=new ImplicacionMinimo();
  Composicion=new Minimo();
  And=new Minimo();
}

MaquinaInferencia::~MaquinaInferencia()
{
  delete Implicaciones;
  delete Composicion;
  delete BaseReglas;
  delete And;
}

Regla *MaquinaInferencia::regla(int numRegla)
{
  return BaseReglas->dato(numRegla);
}


void MaquinaInferencia::limpiarMaquinaInferencia()
{
  int i,j;
  for(i=0;i<numeroReglas();i++)
  {
    for(j=0;j<numeroEntradas();j++)
    {
      modificador(i,j,1.0);
    }
  }
  for(j=0;j<NumeroReglas;j++)
  {
    for(i=0;i<NumeroSalidas;i++)
    {
      conjuntoSalida(j,i,0);
    }
  }

}

void MaquinaInferencia::actualizarEntradas(float *ent)
{
  Entradas->entradaReal(ent);
}

float MaquinaInferencia::pertenenciaDifusores(float *ent)
```

```
{
  float uxd;
  int j;
  j=0;
  uxd=Entradas->pertenenciaDifusor(j,ent[j]);
  for(j=1;j<NumeroEntradas;j++)
  {
    uxd=And->opera(uxd,Entradas->pertenenciaDifusor(j,ent[j]));
  }
  return uxd;
}

float MaquinaInferencia::
pertenenciaImplicacion(int numSal,int numRegla, float *ent,float sal)
{
  float uxa,uxb;
  uxa=pertenenciaAntecedente(numRegla,ent);
  uxb=pertenenciaConsecuente(numSal,numRegla,sal);
  return Implicaciones->implica(uxa,uxb);
}

float MaquinaInferencia::
pertenenciaConsecuente(int numSal,int numRegla,float sal)
{
  float uxc;
  int conj;
  conj=numConjuntoSalida(numRegla,numSal);
  uxc=Salidas->pertenenciaVariable(numSal,conj,sal);
  return uxc;
}

float MaquinaInferencia::pertenenciaAntecedente(int numRegla,float *ent)
{
  float ux;
  float uxa;
  int conj;
  int j;
  j=0;
  conj=numConjuntoEntrada(numRegla,0);
  ux=Entradas->pertenenciaVariable(j,conj,ent[j]);
  if(modificador(numRegla,0)>0.0)
  {
    uxa=pow(ux,modificador(numRegla,0));
  }else
  {
    uxa=1;
```

```
  }
  for (j=1;j<NumeroEntradas;j++)
  {
    conj=numConjuntoEntrada(numRegla,j);
    ux=Entradas->pertenenciaVariable(j,conj,ent[j]);
    if(modificador(numRegla,j)!=0)
    {
      ux=pow(ux,modificador(numRegla,0));
    }else
    {
      ux=1;
    }
    uxa=And->opera(uxa,ux);
  }
  return uxa;
}

float MaquinaInferencia::
pertenenciaComposicion(int numVar, int numRegla,float sal)
{
  float ux;
  float uxa;
  float uxab;
  float comp=0;
  float *x;
  x=new float[NumeroEntradas];
  int *inter;
  inter= new int[NumeroEntradas];
  if(!activarRegla(numRegla))
  {
    comp=Implicaciones->defecto();
  }
  else{
    int casos=1;
    int i;
    for (i=0;i<NumeroEntradas;i++)
    {
      casos=casos*Entradas->numeroPuntosDifusor(i);
      inter[i]=1;
    }
    for(i=0;i<casos;i++)
    {
      int k;
      for (k=0;k<NumeroEntradas;k++)
      {
        x[k]=Entradas->minimoEnDifusor(k)+Entradas->
```

```
          intervaloDifusor(k)*inter[k];
        inter[k]=inter[k]+1;
        if(inter[k]>=Entradas->numeroPuntosDifusor(k))
        {
          inter[k]=1;
        }
      }
      uxab=pertenenciaImplicacion(numVar,numRegla,x,sal);
      uxa=pertenenciaDifusores(x);
      ux=Composicion->opera(uxa,uxab);
      if(ux>comp)
      {
        comp=ux;
      }
    }
  }
  delete[] x;
  delete[] inter;
  return comp;
}

int MaquinaInferencia::activarRegla(int numRegla)
{
  int i;
  for(i=0;i<NumeroEntradas;i++)
  {
    float bmn,bmx,cmn,cmx;
    int numCon;
    numCon=numConjuntoEntrada(numRegla,i);
    bmn=Entradas->minimoEnConjunto(i,numCon);
    bmx=Entradas->maximoEnConjunto(i,numCon);
    cmn=Entradas->minimoEnDifusor(i);
    cmx=Entradas->maximoEnDifusor(i);
    if(bmn>cmx||bmx<cmn)
      return 0;
  }
  return 1;
}

void MaquinaInferencia::desocuparBaseReglas()
{
  int i,num;
   num=numeroReglas();
  for(i=1;i<num;i++)
  {
      eliminarRegla(0);
```

```
  }
   for(i=0;i<numeroEntradas();i++)
   {
      conjuntoEntrada(0,i,0);
   }
   for(i=0;i<numeroEntradas();i++)
   {
      modificador(0,i,0);
   }
   for(i=0;i<numeroSalidas();i++)
   {
      conjuntoSalida(0,i,0);
   }
   NumeroReglas=1;
}

void MaquinaInferencia::
EntrenaUniversoFijo(float *antecedente, float *consecuente)
{
  Regla *ReglaTemporal;
  ReglaTemporal=new Regla(NumeroEntradas,NumeroSalidas);
  llenarRegla(ReglaTemporal,antecedente,consecuente);
  int preexistencia=0;
  int i;
  for(i=0;i<numeroReglas();i++)
  {
    if(compararAntec(ReglaTemporal,regla(i)))
    {
      if(ReglaTemporal->certeza()>regla(i)->certeza())
      {
        adicionarRegla(ReglaTemporal);
        eliminarRegla(i);
      }
      preexistencia=1;
      i=numeroReglas()+1;
    }
  }
  if(preexistencia==0)
  {
    adicionarRegla(ReglaTemporal);
  }
}

void MaquinaInferencia::
llenarRegla(Regla *ReglaTemporal, float *antec, float *consec)
{
```

```
int i;
float certeza=1.0;
for(i=0;i<NumeroEntradas;i++)
{
  if(antec[i]>Entradas->rangoMaximoVariable(i))
  {
    antec[i]=Entradas->rangoMaximoVariable(i);
  }
  if(antec[i]<Entradas->rangoMinimoVariable(i))
  {
    antec[i]=Entradas->rangoMinimoVariable(i);
  }
  float maxPerAnte=0;
  int j;
  for(j=0;j<Entradas->numeroConjuntosEnVariable(i);j++)
  {
    float Per;
    Per=Entradas->pertenenciaVariable(i,j,antec[i]);
    if(maxPerAnte<Per)
    {
      maxPerAnte=Per;
      ReglaTemporal->conjuntoEntrada(i,j);
    }
  }
  certeza=certeza*maxPerAnte;
  ReglaTemporal->modificador(i,1.0);
}
for(i=0;i<NumeroSalidas;i++)
{
  if(consec[i]>Salidas->rangoMaximoVariable(i))
  {
    consec[i]=Salidas->rangoMaximoVariable(i);
  }
  if(consec[i]<Salidas->rangoMinimoVariable(i))
  {
    consec[i]=Salidas->rangoMinimoVariable(i);
  }
  float maxPerCons=0;
  int j;
  for(j=0;j<Salidas->numeroConjuntosEnVariable(i);j++)
  {
    float Per;
    Per=Salidas->pertenenciaVariable(i,j,consec[i]);
    if(maxPerCons<Per)
    {
      maxPerCons=Per;
```

```
        ReglaTemporal->conjuntoSalida(i,j);
      }
    }
    certeza=certeza*maxPerCons;
  }
  ReglaTemporal->certeza(certeza);
}


int MaquinaInferencia::compararAntec(Regla *rg1, Regla *rg2)
{
  int resultado=1;
  int i;
  for(i=0;i<NumeroEntradas;i++)
  {
    if(rg1->conjuntoEntrada(i)!=rg2->conjuntoEntrada(i))
    {
      resultado=0;
    }
  }
  return resultado;
}

void MaquinaInferencia::
EntrenaUniversoVariable(float *antecedente, float *consecuente)
{
  ConjuntoDifuso* conj;
  int i;
  for(i=0;i<NumeroEntradas;i++)
  {
    int tipoConjunto;
    tipoConjunto=Entradas->variable(i)->conjunto(0)->identificador();
    int numeroPuntos;
    numeroPuntos=Entradas->variable(i)->conjunto(0)->
         numeroPuntosClaves();
    float *puntos;
    puntos=new float[numeroPuntos];
    Entradas->variable(i)->conjunto(0)->puntosClaves(puntos);
    switch(tipoConjunto)
    {
      case 0 : conj=
            new ConjuntoL(puntos[0],puntos[1],puntos[2]);
            break;
      case 1 : conj=
            new ConjuntoTriangulo(puntos[0],puntos[1],puntos[2]);
            break;
```

```
      case 2 : conj=
            new ConjuntoPi(puntos[0],puntos[1],puntos[2],puntos[3]);
            break;
      case 3 : conj=
            new ConjuntoGamma(puntos[0],puntos[1],puntos[2]);
            break;
      case 4 : conj=
            new ConjuntoZ(puntos[0],puntos[1],puntos[2]);
            break;
      case 5 : conj=
            new ConjuntoCampana(puntos[0],puntos[1],puntos[2]);
            break;
      case 6 : conj=
            new ConjuntoS(puntos[0],puntos[1],puntos[2]);
            break;
      case 7 : conj=
            new ConjuntoPiCampana(puntos[0],puntos[1],
                 puntos[2],puntos[3]);
            break;
      case 8 : conj=
            new ConjuntoSinglenton(puntos[0],puntos[1]);
            break;
      default:break;
    }
    float delta;
    delta=*(antecedente+i)-conj->centroAltura();
    int j;
    for(j=0;j<numeroPuntos;j++)
    {
      conj->nuevoPuntoClave(j,puntos[j]+delta);
    }
    Entradas->variable(i)->adicionarConjuntos(conj);
  }

  for(i=0;i<NumeroSalidas;i++)
  {
    int tipoConjunto;
    tipoConjunto=Salidas->variable(i)->conjunto(0)->identificador();
    int numeroPuntos;
    numeroPuntos=Salidas->variable(i)->conjunto(0)->numeroPuntosClaves();
    float *puntos;
    puntos=new float[numeroPuntos];
    Salidas->variable(i)->conjunto(0)->puntosClaves(puntos);
    switch(tipoConjunto)
    {
      case 0 : conj=
```

```
            new ConjuntoL(puntos[0],puntos[1],puntos[2]);
            break;
      case 1 : conj=
            new ConjuntoTriangulo(puntos[0],puntos[1],puntos[2]);
            break;
      case 2 : conj=
            new ConjuntoPi(puntos[0],puntos[1],puntos[2],puntos[3]);
            break;
      case 3 : conj=
            new ConjuntoGamma(puntos[0],puntos[1],puntos[2]);
            break;
      case 4 : conj=
            new ConjuntoZ(puntos[0],puntos[1],puntos[2]);
            break;
      case 5 : conj=
            new ConjuntoCampana(puntos[0],puntos[1],puntos[2]);
            break;
      case 6 : conj=
            new ConjuntoS(puntos[0],puntos[1],puntos[2]);
            break;
      case 7 : conj=
            new ConjuntoPiCampana(puntos[0],puntos[1],
              puntos[2],puntos[3]);
            break;
      case 8 : conj=
            new ConjuntoSinglenton(puntos[0],puntos[1]);
            break;
      default:break;
    }
    float delta;
    delta=*(consecuente+i)-conj->centroAltura();
    int j;
    for(j=0;j<numeroPuntos;j++)
    {
      conj->nuevoPuntoClave(j,puntos[j]+delta);
    }
    Salidas->variable(i)->adicionarConjuntos(conj);
  }
  Regla *rg;
  rg=new Regla(NumeroEntradas,NumeroSalidas);
  for(i=0;i<NumeroEntradas;i++)
  {
    rg->conjuntoEntrada(i,Entradas->variable(i)->numeroConjuntos()-1);
    rg->modificador(i,1.0);
  }
  for(i=0;i<NumeroSalidas;i++)
```

```
  {
    rg->conjuntoSalida(i,Salidas->variable(i)->numeroConjuntos()-1);
  }
  rg->certeza(1.0);
  adicionarRegla(rg);
}


float PrimerMaximo::salidaConcreta(float *ent)
{
  float con;
  float concreto=0;
  float y;
  float ymax;
  ymax=Motor->salidas()->rangoMinimoVariable(NumeroVariable);
  Motor->actualizarEntradas(ent);
  int i;
  for(i=0;i<(Motor->salidas()->numeroIntervalosEnVariable
         (NumeroVariable)+1);i++)
  {
    con=defecto();
    y=Motor->salidas()->rangoMinimoVariable(NumeroVariable)+
         i*Motor->salidas()->intervaloEnVariable(NumeroVariable);
    int j;
    for(j=0;j<Motor->numeroReglas();j++)
    {
      float temp;
      temp=Motor->pertenenciaComposicion(NumeroVariable,j,y);
      con=Conjuncion->opera(con,temp);
    }
    if(concreto<con)
    {
      concreto=con;
      ymax=y;
    }
  }
  return ymax;
}


float UltimoMaximo::salidaConcreta(float *ent)
{
  float con;
  float concreto=0;
  float y;
  float ymax;
```

```
  ymax=Motor->salidas()->rangoMinimoVariable(NumeroVariable);
  Motor->actualizarEntradas(ent);
  int i;
  for(i=0;i<(Motor->salidas()->numeroIntervalosEnVariable
         (NumeroVariable)+1);i++)
  {
    con=defecto();
    y=Motor->salidas()->rangoMinimoVariable(NumeroVariable)+
         i*Motor->salidas()->intervaloEnVariable(NumeroVariable);
    int j;
    for(j=0;j<Motor->numeroReglas();j++)
    {
      float temp;
      temp=Motor->pertenenciaComposicion(NumeroVariable,j,y);
      con=Conjuncion->opera(con,temp);
    }
    if(concreto<=con)
    {
      concreto=con;
      ymax=y;
    }
  }
  return ymax;
}


float MediaDeMaximos::salidaConcreta(float *ent)
{
  float con;
  float concreto1=0;
  float concreto2=0;
  float y;
  float ymax;
  float ymax1,ymax2;
  ymax=Motor->salidas()->rangoMinimoVariable(NumeroVariable);
  ymax1=ymax;
  ymax2=ymax;
  Motor->actualizarEntradas(ent);
  int i;
  for(i=0;i<(Motor->salidas()->numeroIntervalosEnVariable
         (NumeroVariable)+1);i++)
  {
    con=defecto();
    y=Motor->salidas()->rangoMinimoVariable(NumeroVariable)+
          i*Motor->salidas()->intervaloEnVariable(NumeroVariable);
    int j;
```

```
    for(j=0;j<Motor->numeroReglas();j++)
    {
      float temp;
      temp=Motor->pertenenciaComposicion(NumeroVariable,j,y);
      con=Conjuncion->opera(con,temp);
    }
    if(concreto1<con)
    {
      concreto1=con;
      ymax1=y;
    }
    if(concreto2<=con)
    {
      concreto2=con;
      ymax2=y;
    }
  }
  ymax=(ymax1+ymax2)/2;
  return ymax;
}


float CentroDeGravedad::salidaConcreta(float *ent)
{
  float con;
  float y;
  float y1=0;
  float y2=0;
  float ymax;
  Motor->actualizarEntradas(ent);
  int i;
  for(i=0;i<(Motor->salidas()->numeroIntervalosEnVariable
         (NumeroVariable)+1);i++)
  {
    con=defecto();
    y=Motor->salidas()->rangoMinimoVariable(NumeroVariable)+
         i*Motor->salidas()->intervaloEnVariable(NumeroVariable);
    int j;
    for(j=0;j<Motor->numeroReglas();j++)
    {
      float temp;
      temp=Motor->pertenenciaComposicion(NumeroVariable,j,y);
      con=Conjuncion->opera(con,temp);
    }
    y1=y1+y*con;
    y2=y2+con;
```

```
  }
  if(fabs(y2)<0.000001)
    y2=100000.0;
  if(fabs(y1)<0.000001)
    y1=0.0;
  ymax=y1/y2;
  return ymax;
}

float Altura::salidaConcreta(float *ent)
{
  float ymax;
  float y;
   float y1,y2;
  Motor->actualizarEntradas(ent);
  int j;
    y1=0;
    y2=0;
  for(j=0;j<Motor->numeroReglas();j++)
  {
    if(Motor->activarRegla(j))
    {
      float temp;
      y=Motor->conjuntoSalida(j,NumeroVariable)->centroAltura();
      temp=Motor->pertenenciaComposicion(NumeroVariable,j,y);
      y1=y1+temp*y;
      y2=y2+temp;
    }
  }
  if(fabs(y2)<0.000001)
    y2=100000.0;
  if(fabs(y1)<0.000001)
    y1=0.0;
  ymax=y1/y2;
  return ymax;
}


void BloqueConcrecion::limpiarListaConcresores()
{
  int i,numConc;
  numConc=Motor->numeroSalidas();
  for(i=0;i<numConc;i++)
  {
    eliminarConcresor(0);
  }
```

```
}


Concresor* BloqueConcrecion::concresor(int numSal)
{
  return Concresores->dato(numSal);
}

void BloqueConcrecion::salidaConcreta(float *ent,float *sal)
{
  int numSal;
  numSal=Motor->numeroSalidas();
  int i;
  for(i=0;i<numSal;i++)
  {
    sal[i]=salidaConcreta(i,ent);
  }
}

void BloqueConcrecion::motor(MaquinaInferencia *maq)
{
  int i;
  int j;
  Motor=maq;
  j=maq->salidas()->numeroVariables();
  for(i=0;i<j;i++)
  {
    concresor(i)->motor(maq);
  }
}

void BloqueConcrecion::conjuncion(Norma *nor)
{
  int i;
  int j;
  j=Motor->salidas()->numeroVariables();
  for(i=0;i<j;i++)
  {
    concresor(i)->conjuncion(nor);
  }
}
```

C.3 Identificación difusa de sistemas. Estrategia 1

```
#include <owl/dc.h>
```

```
#include <stdio.h>
#include <string.h>
#include <math.h>

#include<stdio.h>
#include<iostream.h>
#include<cstring.h>

#include "fuzzy.hpp"
#include "genetico.hpp"
#include "genreal.hpp"
#include "recursos.rh"

class Patron
{
public:
  Patron()
  {
    Test=0;
  }
  ~Patron()
  {
    Entradas.FlushDestroy();
    Salidas.FlushDestroy();
  }

  Arreglo<float> Entradas;
  Arreglo<float> Salidas;
  int Test; // 0=Enteranamiento  1=Test

  BOOL operator==(const Patron &other)
  {
    return (Test==other.Test);
  }
};


class SLD_AG_Modificadores:public Individuo,public SistemaLogicaDifusa
{
public:
  SLD_AG_Modificadores(Arreglo<Patron> *Pat1=NULL,
         Arreglo<Patron> *Pat2=NULL,Arreglo<Patron> *Pat3=NULL
         ,int numE=3,int numS=3);
  SLD_AG_Modificadores(SLD_AG_Modificadores *Mod);
  ~SLD_AG_Modificadores()
  {
```

```
     Genoma.FlushDestroy();

      PatronesEntrenamiento=NULL;
      PatronesPrueba=NULL;
      Patrones=NULL;
  }

//////// funciones virtuales de Individuo
  Individuo* crearCopia();
  void copiarDetalles(Individuo *other);
  double objetivo();
  void codificar();
  void decodificar();
////////////////////

  Arreglo<Patron> *Patrones;
  Arreglo<Patron> *PatronesEntrenamiento;
  Arreglo<Patron> *PatronesPrueba;

  void crearSLD(int numEnt=3,int numSal=3);
  void entrenarUFijo();     // debe entrenar desde cero, con los patrones
  double objetivoTest();
  void crearGenoma();
  void grabaModificadores(char *cad);
};

SLD_AG_Modificadores::SLD_AG_Modificadores(Arreglo<Patron>
 *Pat1,Arreglo<Patron> *Pat2,Arreglo<Patron> *Pat3,
int numE,int numS)
{
  crearSLD(numE,numS);
  Patrones=Pat1;
  PatronesEntrenamiento=Pat2;
  PatronesPrueba=Pat3;
  entrenarUFijo();
  crearGenoma();
  codificar();
}

SLD_AG_Modificadores::SLD_AG_Modificadores
(SLD_AG_Modificadores *Mod)
{
  crearSLD(Mod->entradas->numeroConjuntosEnVariable(0),
         Mod->salidas->numeroConjuntosEnVariable(0));

// A partir de aqui se crean reglas, patrones y genoma
```

```
  Patrones=Mod->Patrones;
  PatronesEntrenamiento=Mod->PatronesEntrenamiento;
  PatronesPrueba=Mod->PatronesPrueba;

  motor->BaseReglas->FlushDestroy();
  motor->NumeroReglas=0;

  int i,tam;
  tam=Mod->motor->numeroReglas();
  for(i=0;i<tam;i++)
  {
    Regla *Reg;
    Reg=new Regla(Mod->numeroEntradas(),Mod->numeroSalidas());
    int j,tam2;
    tam2=numeroEntradas();
    for(j=0;j<tam2;j++)
    {
      Reg->conjuntoEntrada(j,Mod->motor->regla(i)->conjuntoEntrada(j));
      Reg->modificador(j,Mod->motor->regla(i)->modificador(j));
    }
    tam2=numeroSalidas();
    for(j=0;j<tam2;j++)
    {
      Reg->conjuntoSalida(j,Mod->motor->regla(i)->conjuntoSalida(j));
    }
    motor->adicionarRegla(Reg);
  }
  crearGenoma();
  codificar();
}

void SLD_AG_Modificadores::crearSLD(int numEnt,int numSal)
{
   ConjuntoDifuso *cd;
   Difusor *dif;
   Variable *var;
   Norma *And;
   Norma *Composicion;
   Norma *Conjuncion;
   Implicacion *Implica;
   Concresor *conc;

   entradas=new Universo(1);
   salidas=new Universo(1);
```

```
////////////// Entrada
   var=new Variable(numEnt);
   var->rangoMinimo(-1.000000);
   var->rangoMaximo(1.000000);

  float dx;
  dx=(1.0-(-1.0))/(numEnt+1);

  int i;
  cd=new ConjuntoL(-1.0, -1.0+dx, -1.0 +2*dx);
  var->adicionarConjuntos(cd);
  for (i=1;i<(numEnt-1);i++){
    cd=new ConjuntoTriangulo(-1.0 +i*dx, -1.0+(i+1)*dx, -1.0 +(i+2)*dx);
    var->adicionarConjuntos(cd);
  }
  cd=new ConjuntoGamma(1.0-2*dx, 1.0-dx, 1.0);
  var->adicionarConjuntos(cd);

   dif=new DifusorSinglenton(0.000000,0.010000);
   dif->numeroPuntos(1);
   var->difusorEntrada(dif);
   var->nombreVariable("Entrada");
   var->numeroIntervalos(50);
   entradas->adicionarVariable(var);

////////////// Salida
   var=new Variable(numSal);
   var->rangoMinimo(-.5);
   var->rangoMaximo(.5);

  dx=(.5-(-.5))/(numSal+1);

  cd=new ConjuntoL(-.5, -.5+dx, -.5 +2*dx);
  var->adicionarConjuntos(cd);
  for (i=1;i<(numSal-1);i++){
    cd=new ConjuntoTriangulo(-.5 +i*dx, -.5+(i+1)*dx, -.5 +(i+2)*dx);
    var->adicionarConjuntos(cd);
  }
  cd=new ConjuntoGamma(.5-2*dx, .5-dx, .5);
  var->adicionarConjuntos(cd);

   var->nombreVariable("Salida");
   var->numeroIntervalos(50);
   salidas->adicionarVariable(var);

//////////////
```

```
   motor=new MaquinaInferencia(entradas,salidas,1);
   And=new Minimo();
   Composicion=new Minimo();
   Implica=new ImplicacionMinimo();
   motor->and(And);
   motor->composicion(Composicion);
   motor->implicacion(Implica);

   motor->conjuntoEntrada(0,0,0);
   motor->conjuntoSalida(0,0,0);

   motor->modificador(0,0,1.000000);

   concreto=new BloqueConcrecion(motor);
   Conjuncion=new Maximo();
   conc=new Altura(motor,0,Conjuncion);
   concreto->adicionarConcresor(conc);
   concreto->motor(motor);
   concreto->conjuncion(Conjuncion);
}

void SLD_AG_Modificadores::crearGenoma()
{
   int i,tam,j,tam2;
   tam=motor->numeroReglas();
   for(i=0;i<tam;i++)
   {
    tam2=numeroEntradas();
    for(j=0;j<tam2;j++)
    {
      GenReal *g;
      g=new GenReal();
      g->Minimo=0.1;    // modificador entre 0.5 y 4.0
      g->Maximo=4.0;    // modificador entre 0.5 y 4.0
      Genoma.Add(g);
    }
   }
}

void SLD_AG_Modificadores::grabaModificadores(char *cad)
{
  FILE *arch;
  arch=fopen(cad,"wt");
  if(arch==NULL){return;}
   int i,tam,j,tam2,cont=0;
```

```
   tam=motor->numeroReglas();
   for(i=0;i<tam;i++)
   {
    tam2=numeroEntradas();
    for(j=0;j<tam2;j++)
    {
      fprintf(arch,"%lf ",((GenReal*)Genoma.dato(cont))->Valor);
      cont++;
    }
    fprintf(arch,"\n");
   }
   fclose(arch);
}

Individuo* SLD_AG_Modificadores::crearCopia()
{
//  SLD_AG_Modificadores *Ind;
  Individuo *Ind;
  Ind=new SLD_AG_Modificadores(this);
  return Ind;
}

void SLD_AG_Modificadores::copiarDetalles(Individuo *other)
{

  SLD_AG_Modificadores *SLD;
  SLD=(SLD_AG_Modificadores*)other;

  Genoma.FlushDestroy();
  int i,j,tam,tam2;
  tam=SLD->motor->numeroReglas();
  for(i=0;i<tam;i++)
  {
    tam2=numeroEntradas();
    for(j=0;j<tam2;j++)
    {
      motor->BaseReglas->dato(i)->conjuntoEntrada(j,
         SLD->motor->BaseReglas->dato(i)->conjuntoEntrada(j));
      motor->BaseReglas->dato(i)->modificador(j,
         SLD->motor->BaseReglas->dato(i)->modificador(j));
      GenReal *g;
      g=new GenReal;
      g->Valor=motor->BaseReglas->dato(i)->modificador(j);
      Genoma.Add(g);
    }
    tam2=numeroSalidas();
```

```
    for(j=0;j<tam2;j++)
    {
      motor->BaseReglas->dato(i)->conjuntoSalida(j,
         SLD->motor->BaseReglas->dato(i)->conjuntoSalida(j));
    }
  }
//  codificar();
}

double SLD_AG_Modificadores::objetivo()
{
  float *ant,*con;
  double error,sumaerror=0.0,sumatotal=0.0;
  ant=new float[numeroEntradas()];
  con=new float[numeroSalidas()];
  int i,tam;
  tam=PatronesEntrenamiento->GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    int j,tam2;
    tam2=numeroEntradas();
    for(j=0;j<tam2;j++)
    {
      ant[j]=*PatronesEntrenamiento->dato(i)->Entradas.dato(j);
    }
    calcular(ant,con);
    error=con[0]-*PatronesEntrenamiento->dato(i)->Salidas.dato(0);
    error=fabsl(error);
    sumaerror=sumaerror+error;
    sumatotal=sumatotal+fabsl(*PatronesEntrenamiento->dato(i)
        ->Salidas.dato(0));
  }
  delete[] ant;
  delete[] con;
  return (sumaerror/sumatotal);
}

void SLD_AG_Modificadores::codificar()
{
  int i,j,tam,tam2,cont=0;
  tam=motor->numeroReglas();
  for(i=0;i<tam;i++)
  {
    tam2=numeroEntradas();
    for(j=0;j<tam2;j++)
    {
```

```
      GenReal *g;
      g=(GenReal*)Genoma.dato(cont);
      g->Valor=motor->regla(i)->modificador(j);
      cont++;
    }
  }
}

void SLD_AG_Modificadores::decodificar()
{
  int i,j,tam,tam2,cont=0;
  tam=motor->numeroReglas();
  for(i=0;i<tam;i++)
  {
    tam2=numeroEntradas();
    for(j=0;j<tam2;j++)
    {
      GenReal *g;
      g=(GenReal*)Genoma.dato(cont);
      motor->regla(i)->modificador(j,g->Valor);
      cont++;
    }
  }
}

void SLD_AG_Modificadores::entrenarUFijo()
{
  int i,tam;
  tam=PatronesEntrenamiento->GetItemsInContainer();
  float *ant,*con;
  ant=new float[numeroEntradas()];
  con=new float[numeroSalidas()];
  for(i=0;i<tam;i++)
  {
    int j,tam2;
    tam2=numeroEntradas();
    for(j=0;j<tam2;j++)
    {
      ant[j]=*PatronesEntrenamiento->dato(i)->Entradas.dato(j);
    }
    tam2=numeroSalidas();
    for(j=0;j<tam2;j++)
    {
      con[j]=*PatronesEntrenamiento->dato(i)->Salidas.dato(j);
    }
    EntrenaUniversoFijo(ant,con);
```

```
  }
  delete []ant;
  delete []con;
}

double SLD_AG_Modificadores::objetivoTest()
{
  float *ant,*con;
  double error,sumaerror=0.0,sumatotal=0.0;
  ant=new float[numeroEntradas()];
  con=new float[numeroSalidas()];
  int i,tam;
  tam=PatronesPrueba->GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    int j,tam2;
    tam2=numeroEntradas();
    for(j=0;j<tam2;j++)
    {
      ant[j]=*PatronesPrueba->dato(i)->Entradas.dato(j);
    }
    calcular(ant,con);
    error=con[0]-*PatronesPrueba->dato(i)->Salidas.dato(0);
    error=fabsl(error);
    sumaerror=sumaerror+error;
    sumatotal=sumatotal+fabsl(con[0]);
  }
  delete[] ant;
  delete[] con;
  return (sumaerror/sumatotal);
}

void crearPatrones(string Archivo,float PorcEntrena,
Arreglo<Patron> *Patrones,Arreglo<Patron> *
PatronesEntrenamiento,Arreglo<Patron> *PatronesPrueba)
{
  FILE *arch;
  arch=fopen(Archivo.c_str(),"r");
  if(arch==NULL){return;}
  float tmp;
  string strtmp;
  char c='a';
  Patron *Pat;
  Pat=new Patron;
  do
  {
```

```
    strtmp="";
    do
    {
      c=fgetc(arch);
      strtmp+=c;
    }while((c!=' ')&&(c!='\n')&&!feof(arch));
    tmp=atof(strtmp.c_str());
    float *pat;
    pat=new float;
    *pat=tmp;
    if(c==' ')
    {
      Pat->Entradas.Add(pat);
    }else
    {
      Pat->Salidas.Add(pat);
      if(Pat->Entradas.GetItemsInContainer()>0)
      {
        Patrones->Add(Pat);
      }
      Pat=new Patron;
    }
  }while(!feof(arch));

  int contador=0,tamEntrena;
  if(PorcEntrena<0.0){PorcEntrena=0.0;}
  if(PorcEntrena>1.0){PorcEntrena=1.0;}
  tamEntrena=PorcEntrena*Patrones->GetItemsInContainer();

  randomize();

  do
  {
    double azar;
    azar=(double)((double)rand()/(double)RAND_MAX);
    int item;
    item=azar*Patrones->GetItemsInContainer();
    if(Patrones->dato(item)->Test==0)
    {
      Patrones->dato(item)->Test=1;
      contador++;
    }
  }while(contador<tamEntrena);

  int i,tam;
  tam=Patrones->GetItemsInContainer();
```

```
  for(i=0;i<tam;i++)
  {
    if(Patrones->dato(i)->Test==1)
    {
      PatronesEntrenamiento->Add(Patrones->dato(i));
    }else
    {
      PatronesPrueba->Add(Patrones->dato(i));
    }
  }
}

void main()
{
  int Indiv,Gener,NumEntra,NumSal;
  cout << "Número de Conjuntos de Entrada: ";
  cin >> NumEntra;
  cout << "Número de Conjuntos de Salida: ";
  cin >> NumSal;
  cout << "Número de Generaciones: ";
  cin >> Gener;
  cout << "Número de Individuos por Generación: ";
  cin >> Indiv;

  SLD_AG_Modificadores *PredictorRio;
  Arreglo<Patron> *Patrones;
  Arreglo<Patron> *PatronesEntrenamiento;
  Arreglo<Patron> *PatronesPrueba;

  Patrones=new Arreglo<Patron>;
  PatronesEntrenamiento=new Arreglo<Patron>;
  PatronesPrueba=new Arreglo<Patron>;

  crearPatrones("datos.txt",.7,Patrones,PatronesEntrenamiento,
         PatronesPrueba);

  string str;
  char cad[200],cad2[200];
  str="Patrones creados: ";
  sprintf(cad,"%d - ",Patrones->GetItemsInContainer());
  str+=cad;
  sprintf(cad,"%d - ",PatronesEntrenamiento->GetItemsInContainer());
  str+=cad;
  sprintf(cad,"%d \n",PatronesPrueba->GetItemsInContainer());
  str+=cad;
  printf(str.c_str());
```

```
  float Error;
  PredictorRio=new SLD_AG_Modificadores(Patrones,
         PatronesEntrenamiento, PatronesPrueba,NumEntra,NumSal);

  sprintf(cad,"Reglas creadas:%d \n Genes: %d \nEntradas: %d\n",
         PredictorRio->motor->numeroReglas(),PredictorRio->
         Genoma.GetItemsInContainer(),PredictorRio->numeroEntradas());
  str=cad;
  printf(str.c_str());

  Error=PredictorRio->objetivo();
  sprintf(cad,"Error en entrenamiento:%lf \n",Error);
  str=cad;
  printf(str.c_str());

  Error=PredictorRio->objetivoTest();
  sprintf(cad,"Error en prueba:%lf \n",Error);
  str=cad;
  printf(str.c_str());

//  EL ALgoritmo Genético comienza aqui

  AlgoritmoGenetico MiAg;

  MiAg.TamanoPoblacion=Indiv;
  MiAg.GeneracionMaxima=Gener;
  MiAg.IntervaloSalvar=1;

  MiAg.modelo(PredictorRio);

  printf("iniciando algoritmo...\n");
  printf("Iteración: 0 Objetivo: %lf ObjetivoTest: %lf\n",
        PredictorRio->objetivo(),PredictorRio->objetivoTest());
  int c=0;
  MiAg.iniciarOptimizacion();
  do
  {
    sprintf(cad,"Iteración: %d ",c+1);
    printf(cad);
    MiAg.iterarOptimizacion();
    c++;
    sprintf(cad2,"Objetivo: %lf ObjetivoTest: %lf\n",
        MiAg.MejorEnLaHistoria->Objetivo,((SLD_AG_Modificadores*)
        MiAg.MejorEnLaHistoria)->objetivoTest());
    printf(cad2);
```

```
}while(!MiAg.parada());
MiAg.finalizarOptimizacion();

Error=MiAg.MejorEnLaHistoria->objetivo();
sprintf(cad,"Error en entrenamiento:%lf \n",Error);
str=cad;
printf(str.c_str());

Error=((SLD_AG_Modificadores*)MiAg.MejorEnLaHistoria)->objetivoTest();
sprintf(cad,"Error en prueba:%lf \n Fin de programa",Error);
str=cad;
printf(str.c_str());

((SLD_AG_Modificadores*)MiAg.MejorEnLaHistoria)->
       grabaModificadores("modif.txt");

////////////// y=f(u)

FILE *arch1,*arch2;

arch1=fopen("antes.txt","wt");
arch2=fopen("despues.txt","wt");
int i,tam;
tam=Patrones->GetItemsInContainer();
float Entra[2],Sale1[2],Sale2[2];
for(i=0;i<tam;i++)
{
  Entra[0]=*Patrones->dato(i)->Entradas.dato(0);
  PredictorRio->calcular(Entra,Sale1);
  ((SLD_AG_Modificadores*)MiAg.MejorEnLaHistoria)->calcular
      (Entra,Sale2);
  fprintf(arch1,"%f %f\n",Entra[0],Sale1[0]);
  fprintf(arch2,"%f %f\n",Entra[0],Sale2[0]);
}
fclose(arch1);
fclose(arch2);

///////////////y=f(x)

arch1=fopen("fantes.txt","wt");
arch2=fopen("fdespues.txt","wt");
float t;
for(t=-1.0;t<1.00001;t=t+0.01)
{
  Entra[0]=t;
  PredictorRio->calcular(Entra,Sale1);
```

```
    ((SLD_AG_Modificadores*)MiAg.MejorEnLaHistoria)->calcular
        (Entra,Sale2);
    fprintf(arch1,"%f %f\n",Entra[0],Sale1[0]);
    fprintf(arch2,"%f %f\n",Entra[0],Sale2[0]);
  }
  fclose(arch1);
   fclose(arch2);

  PatronesEntrenamiento->FlushDetach();
  PatronesPrueba->FlushDetach();
  Patrones->FlushDestroy();

  delete PatronesEntrenamiento;
  delete PatronesPrueba;
  delete Patrones;

  delete PredictorRio;
}
```

C.4 Identificación difusa de sistemas. Estrategia 2

```
#include <stdio.h>
#include <string.h>
#include <math.h>

#include<stdio.h>
#include<iostream.h>
#include<cstring.h>

#include "fuzzy.hpp"
#include "genetico.hpp"
#include "genreal.hpp"
#include "genint.hpp"
#include "recursos.rh"

class Patron
{
public:
  Patron()
  {
    Test=0;
  }
  ~Patron()
  {
    Entradas.FlushDestroy();
```

```
    Salidas.FlushDestroy();
  }

  Arreglo<float> Entradas;
  Arreglo<float> Salidas;
  int Test; // 0=Enteranamiento  1=Test

  BOOL operator==(const Patron &other)
  {
    return (Test==other.Test);
  }
};


class SLD_AG_Univ_Regla_Modif:public Individuo,public SistemaLogicaDifusa
{
public:
  SLD_AG_Univ_Regla_Modif(Arreglo<Patron> *Pat1=NULL,
  Arreglo<Patron> *Pat2=NULL,Arreglo<Patron> *Pat3=NULL,
  int numE=3,int numS=3);
  SLD_AG_Univ_Regla_Modif(SLD_AG_Univ_Regla_Modif *Mod);
  ~SLD_AG_Univ_Regla_Modif()
  {
     Genoma.FlushDestroy();

      PatronesEntrenamiento=NULL;
      PatronesPrueba=NULL;
      Patrones=NULL;
  }

//////// funciones virtuales de Individuo
  Individuo* crearCopia();
  void copiarDetalles(Individuo *other);
  double objetivo();
  void codificar();
  void decodificar();
////////////////////

  Arreglo<Patron> *Patrones;
  Arreglo<Patron> *PatronesEntrenamiento;
  Arreglo<Patron> *PatronesPrueba;

  float *Puntos;
  int NumeroPuntos;
  float DX;
```

```
  void crearSLD(int numEnt=3,int numSal=3);
  void entrenarUFijo();     // debe entrenar desde cero, con los patrones
  double objetivoTest();
  void crearGenoma();
  void grabaModificadores(char *cad);
};

SLD_AG_Univ_Regla_Modif::SLD_AG_Univ_Regla_Modif
(Arreglo<Patron> *Pat1,Arreglo<Patron> *Pat2,Arreglo<Patron>
*Pat3,int numE,int numS)
{
  crearSLD(numE,numS);
  Patrones=Pat1;
  PatronesEntrenamiento=Pat2;
  PatronesPrueba=Pat3;
//  entrenarUFijo();
  crearGenoma();
  codificar();
}

SLD_AG_Univ_Regla_Modif::SLD_AG_Univ_Regla_Modif
(SLD_AG_Univ_Regla_Modif *Mod)
{
  crearSLD(Mod->entradas->numeroConjuntosEnVariable(0),
  Mod->salidas->numeroConjuntosEnVariable(0));

// A partir de aqui se crean reglas, patrones y genoma

  Patrones=Mod->Patrones;
  PatronesEntrenamiento=Mod->PatronesEntrenamiento;
  PatronesPrueba=Mod->PatronesPrueba;

  int i,tam;

/////////////// Copiar Univ. de entrada
  tam=Mod->entradas->numeroConjuntosEnVariable(0);
  for(i=0;i<tam;i++)
  {
    ConjuntoDifuso *cdFuente,*cdDestino;
    cdFuente=Mod->entradas->conjuntoEnVariable(0,i);
    cdDestino=entradas->conjuntoEnVariable(0,i);
    float Ptos[5];
    int numPtos;
    numPtos=cdFuente->numeroPuntosClaves();
    cdFuente->puntosClaves(Ptos);
    int j;
```

```
    for(j=0;j<numPtos;j++)
    {
      cdDestino->nuevoPuntoClave(j,Ptos[j]);
    }
  }

////////////copiar base de reglas

  motor->BaseReglas->FlushDestroy();
  motor->NumeroReglas=0;

  tam=Mod->motor->numeroReglas();
  for(i=0;i<tam;i++)
  {
    Regla *Reg;
    Reg=new Regla(Mod->numeroEntradas(),Mod->numeroSalidas());
    int j,tam2;
    tam2=numeroEntradas();
    for(j=0;j<tam2;j++)
    {
      Reg->conjuntoEntrada(j,Mod->motor->regla(i)->conjuntoEntrada(j));
      Reg->modificador(j,Mod->motor->regla(i)->modificador(j));
    }
    tam2=numeroSalidas();
    for(j=0;j<tam2;j++)
    {
      Reg->conjuntoSalida(j,Mod->motor->regla(i)->conjuntoSalida(j));
    }
    motor->adicionarRegla(Reg);
  }

  crearGenoma();
  codificar();
}

void SLD_AG_Univ_Regla_Modif::crearSLD(int numEnt,int numSal)
{
   ConjuntoDifuso *cd;
   Difusor *dif;
   Variable *var;
   Norma *And;
   Norma *Composicion;
   Norma *Conjuncion;
   Implicacion *Implica;
   Concresor *conc;
```

```
   entradas=new Universo(1);
   salidas=new Universo(1);

////////////// Entrada
   var=new Variable(numEnt);
   var->rangoMinimo(-1.000000);
   var->rangoMaximo(1.000000);

  DX=(1.0-(-1.0))/(2*numEnt-1);

  NumeroPuntos=2*(numEnt-1);
  Puntos=new float[NumeroPuntos];

  int i;

  for(i=0;i<NumeroPuntos;i++)
  {
    Puntos[i]=-1.0+(i+1)*DX;
  }

  cd=new ConjuntoL(-1.0, -1.0+DX, -1.0 +2*DX);
  var->adicionarConjuntos(cd);
  for (i=1;i<(numEnt-1);i++){
    cd=new ConjuntoPi(-1.0 +(2*i-1)*DX, -1.0+(2*i+0)*DX,-1.0+
         (2*i+1)*DX, -1.0 +(2*i+2)*DX);
    var->adicionarConjuntos(cd);
  }
  cd=new ConjuntoGamma(1.0-2*DX, 1.0-DX, 1.0);
  var->adicionarConjuntos(cd);

   dif=new DifusorSinglenton(0.000000,0.010000);
   dif->numeroPuntos(1);
   var->difusorEntrada(dif);
   var->nombreVariable("Entrada");
   var->numeroIntervalos(50);
   entradas->adicionarVariable(var);

////////////// Salida
   var=new Variable(numSal);
   var->rangoMinimo(-.5);
   var->rangoMaximo(.5);

  float dx;
  dx=(.5-(-.5))/(numSal+1);

  cd=new ConjuntoL(-.5, -.5+dx, -.5 +2*dx);
```

```
  var->adicionarConjuntos(cd);
  for (i=1;i<(numSal-1);i++){
    cd=new ConjuntoTriangulo(-.5 +i*dx, -.5+(i+1)*dx, -.5 +(i+2)*dx);
    var->adicionarConjuntos(cd);
  }
  cd=new ConjuntoGamma(.5-2*dx, .5-dx, .5);
  var->adicionarConjuntos(cd);

   var->nombreVariable("Salida");
   var->numeroIntervalos(50);
   salidas->adicionarVariable(var);

   ////////////////

   motor=new MaquinaInferencia(entradas,salidas,1);
   And=new Minimo();
   Composicion=new Minimo();
   Implica=new ImplicacionMinimo();
   motor->and(And);
   motor->composicion(Composicion);
   motor->implicacion(Implica);

//////////////

  motor->BaseReglas->FlushDestroy();
  motor->NumeroReglas=0;
   for (i=0;i<numEnt;i++)
  {
    Regla *rg;
    rg=new Regla(numEnt,numSal);
    motor->adicionarRegla(rg);
    motor->conjuntoEntrada(i,0,i);
    motor->conjuntoSalida(i,0,0);
    motor->modificador(i,0,1.000000);
  }

   concreto=new BloqueConcrecion(motor);
   Conjuncion=new Maximo();
   conc=new Altura(motor,0,Conjuncion);
   concreto->adicionarConcresor(conc);
   concreto->motor(motor);
   concreto->conjuncion(Conjuncion);
}

void SLD_AG_Univ_Regla_Modif::crearGenoma()
{
```

```
   int i,tam,j,tam2;

  ////////////// Universo Entrada

  tam=entradas->numeroConjuntosEnVariable(0);
  tam=2*(tam-1);
  for(j=0;j<tam;j++)
  {
    GenReal *g;
    g=new GenReal();
    g->Minimo=DX/(-2.0);
    g->Maximo=DX/2.0;
    Genoma.Add(g);
  }

   ////////////// Consecuentes

   tam=motor->numeroReglas();
   for(i=0;i<tam;i++)
   {
    GenEntero *g;
    g=new GenEntero();
    g->Minimo=0;
    g->Maximo=salidas->numeroConjuntosEnVariable(0);
    Genoma.Add(g);
   }
   ////////////// Modificadores

   tam=motor->numeroReglas();
   for(i=0;i<tam;i++)
   {
    tam2=numeroEntradas();
    for(j=0;j<tam2;j++)
    {
      GenReal *g;
      g=new GenReal();
      g->Minimo=0.1;    // modificador entre 0.1 y 4.0
      g->Maximo=4.0;    // modificador entre 0.1 y 4.0
      Genoma.Add(g);
    }
   }
}

void SLD_AG_Univ_Regla_Modif::grabaModificadores(char *cad)
{
  FILE *arch;
```

```
  arch=fopen(cad,"wt");
  if(arch==NULL){return;}
   int i,tam,j;

  tam=NumeroPuntos;
  for(i=0;i<NumeroPuntos;i++)
  {
    fprintf(arch,"%f %lf\n",Puntos[i],((GenReal*)Genoma.dato(i))->Valor);
  }

  tam=entradas->numeroConjuntosEnVariable(0);
  for(i=0;i<tam;i++)
  {
    float Pts[5];
    int numPtos;
    ConjuntoDifuso *cd;
    cd=entradas->conjuntoEnVariable(0,i);
    numPtos=cd->numeroPuntosClaves();
    cd->puntosClaves(Pts);
    for(j=0;j<numPtos;j++)
    {
      fprintf(arch,"%f ",Pts[j]);
    }
    fprintf(arch,"\n");
  }

   tam=motor->numeroReglas();
   for(i=0;i<tam;i++)
   {
    fprintf(arch,"%d %f \n",motor->numConjuntoSalida(i,0),
        motor->modificador(i,0));
  }
   fclose(arch);
}

Individuo* SLD_AG_Univ_Regla_Modif::crearCopia()
{
//  SLD_AG_Univ_Regla_Modif *Ind;
  Individuo *Ind;
  Ind=new SLD_AG_Univ_Regla_Modif(this);
  return Ind;
}

void SLD_AG_Univ_Regla_Modif::copiarDetalles(Individuo *other)
{
```

```
SLD_AG_Univ_Regla_Modif *SLD;
SLD=(SLD_AG_Univ_Regla_Modif*)other;

Genoma.FlushDestroy();
int i,j,tam,tam2;

//////////////  Copiar Entrada
tam=SLD->entradas->numeroConjuntosEnVariable(0);
for(i=0;i<tam;i++)
{
  ConjuntoDifuso *cdFuente,*cdDestino;
  cdFuente=SLD->entradas->conjuntoEnVariable(0,i);
  cdDestino=entradas->conjuntoEnVariable(0,i);
  float Ptos[5];
  int numPtos;
  numPtos=cdFuente->numeroPuntosClaves();
  cdFuente->puntosClaves(Ptos);
  int j;
  for(j=0;j<numPtos;j++)
  {
    cdDestino->nuevoPuntoClave(j,Ptos[j]);
  }
}


//////////////  Copiar Reglas

tam=SLD->motor->numeroReglas();
for(i=0;i<tam;i++)
{
  tam2=numeroEntradas();
  for(j=0;j<tam2;j++)
  {
    motor->BaseReglas->dato(i)->conjuntoEntrada(j,SLD->
       motor->BaseReglas->dato(i)->conjuntoEntrada(j));
    motor->BaseReglas->dato(i)->modificador(j,SLD->motor->
       BaseReglas->dato(i)->modificador(j));
  }
  tam2=numeroSalidas();
  for(j=0;j<tam2;j++)
  {
    motor->BaseReglas->dato(i)->conjuntoSalida(j,SLD->motor->
       BaseReglas->dato(i)->conjuntoSalida(j));
  }
}
crearGenoma();
```

```
  codificar();
}

double SLD_AG_Univ_Regla_Modif::objetivo()
{
  float *ant,*con;
  double error,sumaerror=0.0,sumatotal=0.0;
  ant=new float[numeroEntradas()];
  con=new float[numeroSalidas()];
  int i,tam;
  tam=PatronesEntrenamiento->GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    int j,tam2;
    tam2=numeroEntradas();
    for(j=0;j<tam2;j++)
    {
      ant[j]=*PatronesEntrenamiento->dato(i)->Entradas.dato(j);
    }
    calcular(ant,con);
    error=con[0]-*PatronesEntrenamiento->dato(i)->Salidas.dato(0);
    error=fabsl(error);
    sumaerror=sumaerror+error;
    sumatotal=sumatotal+fabsl(*PatronesEntrenamiento->dato(i)
        ->Salidas.dato(0));
  }
  delete[] ant;
  delete[] con;
  return (sumaerror/sumatotal);
}

void SLD_AG_Univ_Regla_Modif::codificar()
{
  int i,j,tam,tam2,cont=0;
  GenReal *gr;
  GenEntero *gi;

  //////////////Universo de Entrada

  float Pts[5];
  entradas->conjuntoEnVariable(0,0)->puntosClaves(Pts);
  gr=(GenReal*)Genoma.dato(cont);
  gr->Valor=-Puntos[0]+Pts[0];
  cont++;
  gr=(GenReal*)Genoma.dato(cont);
  gr->Valor=-Puntos[1]+Pts[1];
```

```
  cont++;

  tam=entradas->numeroConjuntosEnVariable(0);
  for(i=1;i<tam-1;i++)
  {
    entradas->conjuntoEnVariable(0,i)->puntosClaves(Pts);
    gr=(GenReal*)Genoma.dato(cont);
    gr->Valor=-Puntos[2*i+0]+Pts[2];
    cont++;
    gr=(GenReal*)Genoma.dato(cont);
    gr->Valor=-Puntos[2*i+1]+Pts[3];
    cont++;
  }

  ////////////////////// Consecuentes

  tam=motor->numeroReglas();
  for(i=0;i<tam;i++)
  {
    gi=(GenEntero*)Genoma.dato(cont);
    gi->Valor=motor->regla(i)->conjuntoSalida(0);
    cont++;
  }

  //////////////////  Modificadores

  tam=motor->numeroReglas();
  for(i=0;i<tam;i++)
  {
    tam2=numeroEntradas();
    for(j=0;j<tam2;j++)
    {
      gr=(GenReal*)Genoma.dato(cont);
      gr->Valor=motor->regla(i)->modificador(j);
      cont++;
    }
  }
}

void SLD_AG_Univ_Regla_Modif::decodificar()
{
  int i,j,tam,tam2,cont=0;

  GenReal *gr;
  GenEntero *gi;
```

```
//////////////Universo de Entrada

float a,b,c,d;
gr=(GenReal*)Genoma.dato(cont);
a=Puntos[0]+gr->Valor;
cont++;
gr=(GenReal*)Genoma.dato(cont);
b=Puntos[1]+gr->Valor;
cont++;
entradas->conjuntoEnVariable(0,0)->nuevoPuntoClave(0,a);
entradas->conjuntoEnVariable(0,0)->nuevoPuntoClave(1,b);

tam=entradas->numeroConjuntosEnVariable(0);
for(i=1;i<tam-1;i++)
{
  gr=(GenReal*)Genoma.dato(cont);
  c=Puntos[2*i+0]+gr->Valor;
  cont++;
  gr=(GenReal*)Genoma.dato(cont);
  d=Puntos[2*i+1]+gr->Valor;
  cont++;
  entradas->conjuntoEnVariable(0,i)->nuevoPuntoClave(0,a);
  entradas->conjuntoEnVariable(0,i)->nuevoPuntoClave(1,b);
  entradas->conjuntoEnVariable(0,i)->nuevoPuntoClave(2,c);
  entradas->conjuntoEnVariable(0,i)->nuevoPuntoClave(3,d);
  a=c;b=d;
}
entradas->conjuntoEnVariable(0,tam-1)->nuevoPuntoClave(0,a);
entradas->conjuntoEnVariable(0,tam-1)->nuevoPuntoClave(1,b);

//////////////////// Consecuentes

tam=motor->numeroReglas();
for(i=0;i<tam;i++)
{
  gi=(GenEntero*)Genoma.dato(cont);
  if(gi->Valor<0)gi->Valor=0;
  if(gi->Valor>(salidas->numeroConjuntosEnVariable(0)-1))
       gi->Valor=salidas->numeroConjuntosEnVariable(0)-1;
  motor->regla(i)->conjuntoSalida(0,(int)gi->Valor);
  cont++;
}
////////////////////  Modificadores

tam=motor->numeroReglas();
for(i=0;i<tam;i++)
```

```
  {
    tam2=numeroEntradas();
    for(j=0;j<tam2;j++)
    {
      GenReal *gr;
      gr=(GenReal*)Genoma.dato(cont);
      motor->regla(i)->modificador(j,gr->Valor);
      cont++;
    }
  }

}

void SLD_AG_Univ_Regla_Modif::entrenarUFijo()
{
  int i,tam;
  tam=PatronesEntrenamiento->GetItemsInContainer();
  float *ant,*con;
  ant=new float[numeroEntradas()];
  con=new float[numeroSalidas()];
  for(i=0;i<tam;i++)
  {
    int j,tam2;
    tam2=numeroEntradas();
    for(j=0;j<tam2;j++)
    {
      ant[j]=*PatronesEntrenamiento->dato(i)->Entradas.dato(j);
    }
    tam2=numeroSalidas();
    for(j=0;j<tam2;j++)
    {
      con[j]=*PatronesEntrenamiento->dato(i)->Salidas.dato(j);
    }
    EntrenaUniversoFijo(ant,con);
  }
  delete []ant;
  delete []con;
}

double SLD_AG_Univ_Regla_Modif::objetivoTest()
{
  float *ant,*con;
  double error,sumaerror=0.0,sumatotal=0.0;
  ant=new float[numeroEntradas()];
  con=new float[numeroSalidas()];
  int i,tam;
```

```
  tam=PatronesPrueba->GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    int j,tam2;
    tam2=numeroEntradas();
    for(j=0;j<tam2;j++)
    {
      ant[j]=*PatronesPrueba->dato(i)->Entradas.dato(j);
    }
    calcular(ant,con);
    error=con[0]-*PatronesPrueba->dato(i)->Salidas.dato(0);
    error=fabsl(error);
    sumaerror=sumaerror+error;
    sumatotal=sumatotal+fabsl(con[0]);
  }
  delete[] ant;
  delete[] con;
  return (sumaerror/sumatotal);
}

void crearPatrones(string Archivo,float PorcEntrena,Arreglo<Patron>
 *Patrones,Arreglo<Patron> *PatronesEntrenamiento,Arreglo<Patron>
 *PatronesPrueba)
{
  FILE *arch;
  arch=fopen(Archivo.c_str(),"r");
  if(arch==NULL){return;}
  float tmp;
  string strtmp;
  char c='a';
  Patron *Pat;
  Pat=new Patron;
  do
  {
    strtmp="";
    do
    {
      c=fgetc(arch);
      strtmp+=c;
    }while((c!=' ')&&(c!='\n')&&!feof(arch));
    tmp=atof(strtmp.c_str());
    float *pat;
    pat=new float;
    *pat=tmp;
    if(c==' ')
    {
```

```
      Pat->Entradas.Add(pat);
    }else
    {
      Pat->Salidas.Add(pat);
      if(Pat->Entradas.GetItemsInContainer()>0)
      {
        Patrones->Add(Pat);
      }
      Pat=new Patron;
    }
  }while(!feof(arch));

  int contador=0,tamEntrena;
  if(PorcEntrena<0.0){PorcEntrena=0.0;}
  if(PorcEntrena>1.0){PorcEntrena=1.0;}
  tamEntrena=PorcEntrena*Patrones->GetItemsInContainer();

  randomize();

  do
  {
    double azar;
    azar=(double)((double)rand()/(double)RAND_MAX);
    int item;
    item=azar*Patrones->GetItemsInContainer();
    if(Patrones->dato(item)->Test==0)
    {
      Patrones->dato(item)->Test=1;
      contador++;
    }
  }while(contador<tamEntrena);

  int i,tam;
  tam=Patrones->GetItemsInContainer();
  for(i=0;i<tam;i++)
  {
    if(Patrones->dato(i)->Test==1)
    {
      PatronesEntrenamiento->Add(Patrones->dato(i));
    }else
    {
      PatronesPrueba->Add(Patrones->dato(i));
    }
  }
}
```

```
void main()
{
  int Indiv,Gener,NumEntra,NumSal;
  cout << "Número de Conjuntos de Entrada: ";
  cin >> NumEntra;
  cout << "Número de Conjuntos de Salida: ";
  cin >> NumSal;
  cout << "Número de Generaciones: ";
  cin >> Gener;
  cout << "Número de Individuos por Generación: ";
  cin >> Indiv;

  SLD_AG_Univ_Regla_Modif *PredictorRio;
  Arreglo<Patron> *Patrones;
  Arreglo<Patron> *PatronesEntrenamiento;
  Arreglo<Patron> *PatronesPrueba;

  Patrones=new Arreglo<Patron>;
  PatronesEntrenamiento=new Arreglo<Patron>;
  PatronesPrueba=new Arreglo<Patron>;

  crearPatrones("datos.txt",.7,Patrones,PatronesEntrenamiento,
       PatronesPrueba);

  string str;
  char cad[200],cad2[200];
  str="Patrones creados: ";
  sprintf(cad,"%d - ",Patrones->GetItemsInContainer());
  str+=cad;
  sprintf(cad,"%d - ",PatronesEntrenamiento->GetItemsInContainer());
  str+=cad;
  sprintf(cad,"%d \n",PatronesPrueba->GetItemsInContainer());
  str+=cad;
  printf(str.c_str());

  float Error;
  PredictorRio=new SLD_AG_Univ_Regla_Modif(Patrones,
      PatronesEntrenamiento, PatronesPrueba,NumEntra,NumSal);

  sprintf(cad,"Reglas creadas:%d \n Genes: %d \nEntradas: %d\n",
   PredictorRio->motor->numeroReglas(),PredictorRio->Genoma.
   GetItemsInContainer(),PredictorRio->numeroEntradas());
  str=cad;
  printf(str.c_str());

  Error=PredictorRio->objetivo();
```

```
  sprintf(cad,"Error en entrenamiento:%lf \n",Error);
  str=cad;
  printf(str.c_str());

  Error=PredictorRio->objetivoTest();
  sprintf(cad,"Error en prueba:%lf \n",Error);
  str=cad;
  printf(str.c_str());

//  EL ALgoritmo Genético comienza aqui

  AlgoritmoGenetico MiAg;

  MiAg.TamanoPoblacion=Indiv;
  MiAg.GeneracionMaxima=Gener;
  MiAg.IntervaloSalvar=1;

  MiAg.modelo(PredictorRio);

  printf("iniciando algoritmo...\n");
  printf("Iteración: 0 Objetivo: %lf ObjetivoTest: %lf\n",
       PredictorRio->objetivo(),PredictorRio->objetivoTest());
  int c=0;
  MiAg.iniciarOptimizacion();
  do
  {
    sprintf(cad,"Iteración: %d ",c+1);
    printf(cad);
    MiAg.iterarOptimizacion();
    c++;
    sprintf(cad2,"Objetivo: %lf ObjetivoTest: %lf\n",MiAg.
     MejorEnLaHistoria->Objetivo,((SLD_AG_Univ_Regla_Modif*)
     MiAg.MejorEnLaHistoria)->objetivoTest());
    printf(cad2);
  }while(!MiAg.parada());
  MiAg.finalizarOptimizacion();

  Error=MiAg.MejorEnLaHistoria->objetivo();
  sprintf(cad,"Error en entrenamiento:%lf \n",Error);
  str=cad;
  printf(str.c_str());

  Error=((SLD_AG_Univ_Regla_Modif*)MiAg.MejorEnLaHistoria)
       ->objetivoTest();
  sprintf(cad,"Error en prueba:%lf \n Fin de programa",Error);
  str=cad;
```

```
printf(str.c_str());

((SLD_AG_Univ_Regla_Modif*)MiAg.MejorEnLaHistoria)->
     grabaModificadores("modif.txt");

////////////// y=f(u)

FILE *arch1,*arch2;

arch1=fopen("antes.txt","wt");
arch2=fopen("despues.txt","wt");
int i,tam;
tam=Patrones->GetItemsInContainer();
float Entra[2],Sale1[2],Sale2[2];
for(i=0;i<tam;i++)
{
  Entra[0]=*Patrones->dato(i)->Entradas.dato(0);
  PredictorRio->calcular(Entra,Sale1);
  ((SLD_AG_Univ_Regla_Modif*)MiAg.MejorEnLaHistoria)->
      calcular(Entra,Sale2);
  fprintf(arch1,"%f %f\n",Entra[0],Sale1[0]);
  fprintf(arch2,"%f %f\n",Entra[0],Sale2[0]);
}
fclose(arch1);
fclose(arch2);

///////////////y=f(x)

arch1=fopen("fantes.txt","wt");
arch2=fopen("fdespues.txt","wt");
float t;
for(t=-1.0;t<1.00001;t=t+0.01)
{
  Entra[0]=t;
  PredictorRio->calcular(Entra,Sale1);
  ((SLD_AG_Univ_Regla_Modif*)MiAg.MejorEnLaHistoria)->
     calcular(Entra,Sale2);
  fprintf(arch1,"%f %f\n",Entra[0],Sale1[0]);
  fprintf(arch2,"%f %f\n",Entra[0],Sale2[0]);
}
fclose(arch1);
 fclose(arch2);

PatronesEntrenamiento->FlushDetach();
PatronesPrueba->FlushDetach();
Patrones->FlushDestroy();
```

```
  delete PatronesEntrenamiento;
  delete PatronesPrueba;
  delete Patrones;

  delete PredictorRio;
}
```

Apéndice D

Función *Arreglo<>*

```
#ifndef __ARREGLOS_HPP
#define __ARREGLOS_HPP
#endif

template<class T>  //esta clase almacena apuntadores
class Arreglo
{
public:
  Arreglo(int sz=10)
  {
    init(sz);
  }
  virtual ~Arreglo(void)
  {
    FlushDestroy();
    delete [] data;
  }
  void Add(T* dato);
  void AddAt(int loc,T* dato);
  int size(void) const;
  int GetItemsInContainer(void) const;
  T **ptrdato(int index);
  T *dato(int index);
  T *datoConst(int index) const;
  int buscar(T *d);
  int buscarPtr(T *d);
  void Destroy(int index);
  void Detach(int index);
  void FlushDestroy();
  void FlushDetach();
private:
```

```
  int n;
  int items;
  T **data;
  void init(int sz);
};

template <class T>
void Arreglo<T>::init (int sz)
{
  if(sz<1)
  {
    sz=1;
  }
  n=sz;
  items=0;
  data=new T*[n];
}

template <class T>
int Arreglo<T>::size (void ) const
{
  return (n);
}

template <class T>
int Arreglo<T>::GetItemsInContainer (void ) const
{
  return (items);
}

template <class T>
T **Arreglo<T>::ptrdato (int index)
{
  if (index <0){index=0;}
  if (index >items){index=items;}
  return (&data[index]);
}

template <class T>
T *Arreglo<T>::dato (int index)
{
  if (index <0){index=0;}
  if (index >items){index=items;}
  return (data[index]);
}
```

```
template <class T>
T *Arreglo<T>::datoConst (int index) const
{
  if (index <0){index=0;}
  if (index >items){index=items;}
  return (data[index]);
}

template <class T>
int Arreglo<T>::buscar (T *d)
{
  int ind=-1;
  int i;
  for(i=0;i<items;i++)
  {
    if(*data[i]==*d)
    {
      ind=i;
      i=items;
    }
  }
  return (ind);
}

template <class T>
int Arreglo<T>::buscarPtr (T *d)
{
  if(d==NULL){return -1;}
  int ind=-1;
  int i;
  for(i=0;i<items;i++)
  {
    if(data[i]==d)
    {
      ind=i;
      i=items;
    }
  }
  return (ind);
}

template <class T>
void Arreglo<T>::Add(T* dato)
{
  items++;
  if(items>n)
```

```
  {
    T** temp;
    temp=new T*[items];
    n=items;
    int i;
    for(i=0;i<items-1;i++)
    {
      temp[i]=data[i];
    }
    delete [] data;
    data=temp;
  }
  data[items-1]=dato;
}

template <class T>
void Arreglo<T>::AddAt(int loc,T* dato)
{
  if(loc<0){loc=0;}
  if(loc>items){loc=items;}
  items++;

  T** temp;
  temp=new T*[items];
  n=items;
  int i;
  for(i=0;i<loc;i++)
  {
    temp[i]=data[i];
  }
  temp[loc]=dato;
  for(i=loc+1;i<items;i++)
  {
    temp[i]=data[i-1];
  }
  delete [] data;
  data=temp;
}

template <class T>
void Arreglo<T>::Destroy (int index)
{
  T* temp;
  temp=dato(index);
  int i;
  for(i=index;i<items-1;i++)
```

```
  {
    data[i]=data[i+1];
  }
  items--;
  delete temp;
}

template <class T>
void Arreglo<T>::Detach (int index)
{
  int i;
  for(i=index;i<items-1;i++)
  {
    data[i]=data[i+1];
  }
  items--;
}

template <class T>
void Arreglo<T>::FlushDestroy()
{
  int i,tam;
  tam=items;
  for(i=0;i<tam;i++)
  {
    Destroy(0);
  }
}

template <class T>
void Arreglo<T>::FlushDetach()
{
  int i,tam;
  tam=items;
  for(i=0;i<tam;i++)
  {
    Detach(0);
  }
}
```

Bibliografía

[1] O. Cordón and F. Herrera. Identification of linguistic fuzzy models by means of genetic algorithms. In H. Hellendoorn and D. Driankov, editors, *Fuzzy Model Identification. Selected Approaches*, pages 215–250. Springer-Verlag, Berlin, 1997.

[2] O. Cordón, F. Herrera, and M. Lozano. A classified review on the combination fuzzy logic-genetic algorithms bibliography: 1989-1995. In E. Sanchez, T. Shibata, and L. Zadeh, editors, *Genetic Algorithms and Fuzzy Logic Systems. Soft Computing Perspectives*. World Scientific, 1996.

[3] Adenso Diaz, editor. *Optimización Heurística y Redes Neuronales*. Editorial Paraninfo, Madrid, 1996.

[4] Adenso Diaz and Fan T. Tseng. Introducción a las técnicas heurísticas. In Adenso Diaz, editor, *Optimización Heuríatica Y Redes Neuronales*, chapter 1, pages 19–36. editorial paraninfo, Madrid, 1996.

[5] Driankov, Dimiter, et al. *An Introduction to Fuzzy Control*. Springer Verlag, 1993.

[6] Óscar G. Duarte. UNFUZZY: Software para el análisis, diseño, simulación e implementación de sistemas de lógica difusa. Master's thesis, Universidad Nacional de Colombia, Bogotá, 1997.

[7] Óscar G. Duarte. *Técnicas Difusas En la Evaluación de Impacto Ambiental*. PhD thesis, Universidad de Granada, Granada, 2000.

[8] H. Hellendoorn and D. Driankov, editors. *Fuzzy Model Identification. Selected Approaches*. Springer-Verlag, Belin, 1997.

[9] F. Herrera and J.L. Verdegay, editors. *Genetic Algorithms and Soft Computing*, volume 8 of *Studies in Fuzziness and Soft Computing*. Physica-Verlag, 1996.

[10] J.H. Holland. *Adaptation in Natural and Artificial Systems*. The MIT Press, London, 1975.

[11] George Klir and Bo Yuan. *Fuzzy Sets and Fuzzy Logic*. Prentice Hall, 1995.

[12] Manuel Laguna and Pablo Moscato. Moscatto, pablo. In Adenso Diaz, editor, *Optimización Heurística Y Redes Neuronales*, pages 67–104. Editorial Paraninfo, Madrid, 1996.

[13] Chuen Chien Lee. Fuzzy logic in control systems: Fuuzy logic controller-part. *IEEE Transactions on Systems, Man, and Cibernetics*, 20(3):404–418, 1990.

[14] Chuen Chien Lee. Fuzzy logic in control systems: Fuzzy logic controller-PartII. *IEEE Transactions on Systems, Man, and Cibernetics*, 20(3):419–435, 1990.

[15] Ljung. *System Identification ???* Prentice Hall ???, 1996.

[16] Manuel Lozano. *Aplicación de Técnicas Basadas En Lógica Difusa Para la Mejora Del Comportamiento de Los Algoritmos Genéticos con Codificación Real.* PhD thesis, Universidad de Granada, Granada, 1996.

[17] José.J Martínez and Sergio A.. Rojas. *Introducción a la Informática Evolutiva. Un Nuevo Enfoque Para Resolver Problemas de Ingeniería.* Facultad de Ingeniería. Universidad Nacional de Colombia, 1999.

[18] Jerry Mendel. Fuzzy logic systems for engineering: A tutorial. *Proceedings of the IEEE*, 83(3):345–377, 1995.

[19] Kevin Passino. Fuzzy control. In William S. Levine, editor, *The Control Handbook*, pages 1001–1017. CRC Press and IEEE Press, 1996.

[20] V. Vergara and C. Moraga. Optimization of fuzzy models by global numeic optimization. In H. Hellendoorn and D. Driankov, editors, *Fuzzy Model Identification. Selected Approaches*, pages 251–277. Springer-Verlag, Berlin, 1997.

[21] Li-Xin Wang. Fuzzy systems are universal approximators. *Proceedings of the IEEE International Conference on Fuzzy Systems*, pages 1163–1170, 1992.

[22] Li-Xin Wang. *Adaptative Fuzzy Systems and Control. Design and Stability Analysis.* Prentice Hall, 1994.

[23] Li-Xin Wang and Jerry Mendel. Generating fuzzy rules by learning from examples. *Proceeding of the 1991 International Symposium on Intelligent Control*, pages 263–268, 1991.

[24] Li-Xin Wang and Jerry Mendel. Back propagation fuzzy systems as nonlinear dynamic systems identifiers. *Proceeding of the IEEE International Conference on Fuzzy Systems*, pages 1409–1418, 1992.

[25] Li-Xin Wang and Jerry Mendel. Fuzzy basis functions, universal approximation, and orthogonal least-squares learning. *IEEE Transactions on Neural Networks*, 3(5):807–814, 1992.

[26] L.A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.

[27] L.A. Zadeh. The concept of a linguistic variable and its application to approximate reasoning. part I. *Information Sciences*, 8:199–240, 1975.

[28] L.A. Zadeh. The concept of a linguistic variable and its application to approximate reasoning. part II. *Information Sciences*, 8:301–375, 1975.

[29] L.A. Zadeh. The concept of a linguistic variable and its application to approximate reasoning. part III. *Information Sciences*, 9:43–80, 1975.