Subsecretaría de Educación Superior
Dirección General de Educación Superior Tecnológica
Instituto Tecnológico de La Paz

INSTITUTO TECNOLÓGICO DE LA PAZ
DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN
MAESTRÍA EN SISTEMAS COMPUTACIONALES

# Batería de operadores en lenguaje ensamblador para algoritmos genéticos

## T E S I S

QUE PARA OBTENER EL GRADO DE
MAESTRO EN SISTEMAS COMPUTACIONALES

PRESENTA:

JORGE BRAVO ESPINOZA

DIRECTOR DE TESIS:

DR. MARCO ANTONIO CASTRO LIERA

MIEMBROS DEL JURADO:

MC. JESÚS ANTONIO CASTRO, UNAM.
MATI. LUIS ARMANDO CÁRDENAS FLORIDO, ITESM.
MSC. ILIANA CASTRO LIERA, ITLP.

LA PAZ, BAJA CALIFORNIA SUR, MÉXICO, DICIEMBRE 2013.

# RESUMEN

Los compiladores, así como sus capacidades, han ido evolucionando a lo largo del tiempo. Actualmente una de las facetas más importantes de dicha evolución involucra la optimización que pueden lograr sobre el código de máquina generado, con el fin de que se ejecute lo más rápido posible. Si bien los compiladores actuales logran un nivel de optimización aceptable, presentan todavía algunas deficiencias en este rubro.

Esta tesis propone el empleo del lenguaje ensamblador como estrategia de optimización de código. Se presenta una aplicación, escrita completamente en lenguaje C, que implementa un algoritmo genético a ejecutar en una máquina paralela virtual. De la aplicación se seleccionan algunas rutinas para ser optimizadas manualmente, esto es, para ser escritas en lenguaje ensamblador. Se establecen las estrategias para la comprobación de confiabilidad de las rutinas programadas y para la comparación de eficiencia entre la versión original y la optimizada manualmente. Se incluyen también los resultados obtenidos y las conclusiones a que se ha llegado.

# ABSTRACT

Compilers and their capabilities have evolved over time; currently one of the most important facets of compiler evolution involves automatic code optimization, in order to generate machine-code that run as fast as possible. While current compilers achieve an acceptable level of optimization, still have some shortcomings in this area.

This thesis proposes the use of assembly language as code optimization strategy. An application, written entirely in C language, that implements a genetic algorithm to run on a virtual parallel machine is presented. Some routines of the application are selected to be optimized manually, that is, to be written in assembly language. Strategies for checking reliability of programmed routines and for efficiency comparison between the original version and the manually optimized one are set. The results obtained and the conclusions that have been reached are also included.

**AGRADECIMIENTOS**

*A mi director de tesis, Marco Castro, por el gran apoyo que me brindó y por su paciencia para conmigo.*

**ÍNDICE**

Capítulo 1. Introducción.....5
1.1 Problemas de optimización.....6
1.2 Algoritmos bioinspirados.....6
1.3 Programación de algoritmos bioinspirados.....7
1.4 Objetivo general.....8
1.5 Objetivos específicos.....9

Capítulo 2. Optimización de código.....10
2.1 Lenguajes de alto nivel vs lenguaje ensamblador.....11
2.2 Estrategias generales de optimización de código.....13
2.3 El compilador *GNU C* y la optimización de código.....16
2.3.1 Plegado de constantes (*Constant folding*).....17
2.3.2 Propagación de constantes (*Constant propagation*).....17
2.3.3 Eliminación de código muerto (*Dead Code Elimination*).....18
2.3.4 Eliminación de subexpresiones comunes (*Common Subexpression Elimination*).....22
2.3.5 Variables de inducción (*Strength Reduction using Induction Variable*).....28

Capítulo 3. Algoritmos genéticos.....31
3.1 Descripción.....32
3.2 Operadores genéticos.....33
3.3 Algoritmos genéticos distribuidos con migración (AG-DM).....34

Capítulo 4. Batería de operadores en lenguaje ensamblador.....36
4.1 Aplicación a considerar.....37

4.2 Estructura general de la aplicación AG-DM.....40
4.3 Herramientas y plataforma de ejecución.....41
4.3.1 Software de base.....41
4.3.2 Cluster.....41
4.3.3 Habilitación del cluster con PVM.....42

4.4 Programación de rutinas.....46
4.4.1 Funciones de aptitud.....47
4.4.2 Generador de números pseudo-aleatorios .....69
4.4.3 Operadores AG .....82
4.4.4 Funciones auxiliares.....98
4.4.5 Estructura de la aplicación AG-DM optimizada.....107

4.5 Pruebas de confiabilidad a las rutinas programadas.....109
4.6 Estrategia de comparación de tiempos de ejecución.....110

Capítulo 5. Resultados y trabajo futuro.....111
5.1 Tiempos de ejecución en el cluster PVM.....112
5.2 Conclusiones.....114
5.3 Trabajo futuro.....116

Apéndice A. Módulos en lenguaje C del algoritmo AG-DM original.....118
A.1 Módulo maestro.....119
A.2 Módulo esclavo y operadores.....125
A.3 Funciones de aptitud.....132
A.3.1 Ackley.....132
A.3.2 Rastrigin.....133
A.3.3 Schwefel.....134
A.3.4 Weierstrass.....135

Apéndice B. Módulos en lenguaje C del algoritmo AG-DM optimizado.....137
B.1 Módulo maestro.....138
B.2 Módulo esclavo.....143

Apéndice C. Módulos en lenguaje ensamblador del algoritmo AG-DM Optimizado.....149
C.1 Función GeneraPoblacion().....150
C.2 Función OrdenaPoblacion().....152
C.3 Función CopiaPoblacion().....155
C.4 Operador Seleccion().....155
C.5 Operador Cruce().....158
C.6 Operador Mutacion().....161

Apéndice D. Funciones de aptitud en lenguaje ensamblador del algoritmo AG-DM Optimizado.....165
D.1 Rastrigin().....166
D.2 Ackley().....167
D.3 Schwefel().....169
D.4 Weierstrass().....170

Apéndice E. Código en lenguaje ensamblador para el generador de números pseudoaleatorios MT19937.....174

Apéndice F. Resultados de las pruebas de confiabilidad a los módulos programados en lenguaje ensamblador.....181
F.1 Ackley().....182
F.2 Rastrigin().....183

F.3 Schwefel()....184
F.4 Weierstrass()....185
F.5 Operador Seleccion()....186
F.6 Operador Cruce()....188
F.7 Operador Mutacion()....190
F.8 Función GeneraPoblacion()....192
F.9 Función OrdenaPoblacion()....194

Referencias....196

# Capítulo 1

## Introducción

## 1.1 Problemas de optimización

Un problema tiene diferentes connotaciones dependiendo de la perspectiva desde la que se analice. A pesar de ello, todos los problemas comparten una misma estructura genérica gracias a la cual pueden estudiarse de manera global. En general, los problemas están compuestos por tres factores comunes: una situación inicial, un estado u objetivo que se desea alcanzar y un conjunto de limitaciones o restricciones. Cuando se trata de solucionar un problema, lo que se busca es obtener una solución que satisfaga los objetivos, cumpliendo las restricciones impuestas.

Los problemas de optimización son una clase específica de problemas con los que nos enfrentamos a diario para hacer más eficiente el uso de algún recurso, hallar el camino más corto entre dos puntos, distribuir uniformemente un recurso entre varios consumidores, etcétera. Un problema de optimización, en general, se formula como la minimización o maximización de una función objetivo que relaciona las variables relevantes del dominio del problema y durante el proceso de optimización se busca el conjunto de valores que minimicen o maximicen dicha función. Las variables representan el dominio del problema y la función objetivo caracteriza el objetivo propio de éste. Al igual que en cualquier problema, las variables se someten a restricciones que garantizan soluciones aceptables dentro de su contexto.

## 1.2 Algoritmos bioinspirados

Los algoritmos bioinspirados son una familia de algoritmos en el campo de la inteligencia artificial que son útiles para la solución de problemas muy complejos de optimización y de búsqueda no lineal. Los miembros de esta clase de problemas van desde los clásicos desafíos matemáticos a problemas reales del mundo científico y de ingeniería, como el problema del agente viajero (TSP, *Travelling Salesman Problem*),

problemas de búsqueda en espacios dinámicos, multimodales o desconocidos, entre otros. En general, se pueden usar estos métodos para abordar problemas sobre los cuales poco se sabe a priori y que de otro modo sería difícil su solución.

Los algoritmos bioinspirados presentan una estructura general conocida como heurística, que puede aplicarse a los distintos problemas, facilitando así las tareas de diseño e implementación de soluciones [1].

Uno de los tipos de algoritmos bioinspirados más populares son los algoritmos genéticos. Se caracterizan por representar las soluciones al problema que abordan en forma de cadenas de bits. Entre las razones que hacen que este tipo de algoritmos bioinspirados suela ser uno de los estudiados con más detalle están su eficiencia y sencillez de implementación.

## 1.3 Programación de algoritmos bioinspirados

En la actualidad, la programación de algoritmos bioinspirados se hace mediante lenguajes de programación de alto nivel como C, C++, C#, Java y MatLab. Inclusive pueden modelarse utilizando paquetes de software prediseñados como ECJ [2], Lil-GP [3], Beagle [4] y EO [5].

La principal desventaja de los lenguajes de alto nivel es que el código ejecutable generado por los compiladores no es el más eficiente. Aunque se ha avanzado en el desarrollado de compiladores con generación de código optimizado [6], no se ha logrado obtener productos (compiladores) completamente confiables en cuanto al nivel de optimización que alcanzan [7]. Ni hablar de los lenguajes de alto nivel que utilizan intérpretes, que ofrecen una velocidad de ejecución aun más lenta.

Dada la gran cantidad de datos que debe procesar un algoritmo bioinspirado y también la gran cantidad (millones) de operaciones que debe realizar, se hace necesario emplear técnicas de programación que proporcionen la máxima eficiencia posible. Un aspecto muy relevante relacionado con la eficiencia, es el tiempo de ejecución, por tanto, el objetivo es lograr la obtención de una solución aceptable en el menor tiempo posible.

Un paradigma de programación que se ha venido empleando últimamente para optimizar la velocidad de ejecución es la computación paralela. Ejemplos del uso de esta técnica son los clusters (como *pvm* y *mpi*) y el lenguaje CUDA para las tarjetas GP-GPU de Nvidia [8,9].

## 1.4 Objetivo general

En el presente trabajo se propone el empleo del lenguaje ensamblador como herramienta de programación híbrida de algoritmos bioinspirados.

Se pretende demostrar que el empleo del lenguaje ensamblador junto con un lenguaje de alto nivel para la programación de algoritmos bioinspirados puede incrementar significativamente su eficiencia en el tiempo de ejecución.

La propuesta para aumentar la eficiencia en el tiempo de ejecución de los algoritmos bioinspirados consiste en programar en lenguaje ensamblador los módulos cuya ejecución representa los puntos críticos en cuanto a consumo de tiempo de procesamiento.

Se pretende el desarrollo de operadores en lenguaje ensamblador para ser utilizados en una aplicación de optimización de funciones mediante algoritmos genéticos. El producto será un conjunto de módulos ensamblados y listos para ser ligados con el programa

principal escrito en lenguaje C para el ambiente Linux. Además, serán puestos a disposición de la comunidad en calidad de software libre, de tal manera que puedan ser modificados y actualizados para aplicaciones con necesidades específicas.

## 1.5 Objetivos específicos

Los objetivos específicos que se pretende alcanzar en el presente proyecto son los siguientes:

- Analizar el código, escrito en lenguaje C, de la aplicación *AG-DM (Algoritmos Genéticos Distribuidos con Migración)* [10] para determinar cuáles son los módulos críticos cuya optimización manual pudiera proveer mayor velocidad de ejecución.
- Programar en lenguaje ensamblador los módulos pertinentes.
- Evaluar el desempeño de las rutinas optimizadas manualmente para asegurar que los resultados que arrojan son totalmente confiables.
- Comparar la eficiencia del programa original desarrollado en lenguaje C contra la eficiencia de la aplicación optimizada manualmente.

# Capítulo 2

## Optimización de código

## 2.1 Lenguajes de alto nivel vs Lenguaje ensamblador

En los primeros días de la revolución de las computadoras personales, el software de alto rendimiento era escrito en lenguaje ensamblador. Conforme el tiempo pasó, los compiladores con optimización para lenguajes de alto nivel se fueron perfeccionando y sus autores empezaron a declarar que el desempeño del código generado por compilador era casi tan bueno que el código optimizado manualmente con lenguaje ensamblador. Tales proclamas marcaron el comienzo del ascenso de los lenguajes de alto nivel para el desarrollo de aplicaciones para PC, dictando la sentencia de muerte para el lenguaje ensamblador. Muchos programadores de compiladores empezaron a citar números como: "mi compilador logra un 90% de la velocidad del ensamblador, por lo tanto es una locura utilizar el lenguaje ensamblador". El problema es que nunca se molestaron en escribir versiones optimizadas manualmente con lenguaje ensamblador de sus aplicaciones para verificar sus proclamas. A menudo, sus suposiciones sobre el desempeño de sus compiladores están equivocadas [11].

Sin embargo, los autores de compiladores con optimización no estaban del todo equivocados. Bajo condiciones apropiadas, un compilador con optimización puede producir código casi tan bueno como el optimizado manualmente con lenguaje ensamblador. Pero, para lograr esos niveles de desempeño, el código en lenguaje de alto nivel tiene que ser escrito en una manera apropiada. Para escribir código en lenguaje de alto nivel de esta manera se requiere de un sólido entendimiento acerca de cómo las computadoras operan y ejecutan el software [11].

Cuando los primeros programadores empezaron a renunciar al lenguaje ensamblador en favor del uso de lenguajes de alto nivel, entendían de manera general las ramificaciones a bajo nivel de las sentencias en lenguaje de alto nivel que estaban utilizando y elegían esas sentencias apropiadamente. Desafortunadamente, la generación de programadores que los siguieron no tuvo el beneficio de dominar el lenguaje ensamblador. Por tanto, no

estuvieron en posición de elegir sabiamente las sentencias y las estructuras de datos que los compiladores de lenguajes de alto nivel tradujeran eficientemente a código de máquina. Sus aplicaciones, si hubieran sido medidas contra el desempeño de un programa optimizado manualmente con lenguaje ensamblador, seguramente avergonzarían a quien escribió el compilador [11].

Los programadores veteranos, quienes reconocieron este problema, ofrecieron un consejo sabio a los nuevos programadores: "si quieres aprender cómo escribir buen código en lenguaje de alto nivel, necesitas aprender lenguaje ensamblador". Sabiendo lenguaje ensamblador, un programador tendrá la habilidad para considerar las implicaciones de bajo nivel de su código y puede tomar decisiones informadas sobre la mejor manera de escribir código en un lenguaje de alto nivel [11].

Sin duda, la principal ventaja que proporcionan los lenguajes de alto nivel es que permiten escribir programas en un tiempo corto respecto al requerido cuando se programa con lenguaje ensamblador. Sin embargo, esa ventaja se pierde cuando se trata de programar aplicaciones en las que se requiere la mayor eficiencia posible en tiempo de ejecución. Por otro lado, la principal desventaja del lenguaje ensamblador es que requiere la inversión de mucho tiempo de programación. Sin embargo, el código generado puede ser más eficiente (dependiendo de la habilidad y experiencia del programador) que el generado por los compiladores.

Si al programar algoritmos con alta exigencia de procesamiento, como es el caso de los algoritmos bioinspirados, consideramos que el código obtenido con la programación de módulos en lenguaje ensamblador puede proveer mayor velocidad de ejecución que con el generado por un compilador para una versión completa en lenguaje de alto nivel, bien vale la pena invertir en el tiempo de programación necesario.

## 2.2 Estrategias generales de optimización de código

La optimización de código puede verse desde dos perspectivas. Por un lado, puede entenderse como la escritura de código empleando la menor cantidad de instrucciones posible; por otro lado, puede verse como la escritura de programas que provean la mayor velocidad de ejecución posible. En el presente documento entenderemos como optimización de código a la segunda perspectiva.

En la optimización código intervienen varios elementos, entre los que podemos mencionar:

a). Experiencia y habilidad del programador al escribir el código fuente.

Un programador experimentado, que tiene conocimiento acerca de la manera en que el compilador genera el código, puede optimizarlo tomando ciertas decisiones. Dos ejemplos: si necesita una variable de 8 bits, puede decidir usar una variable de 16 bits, sacrificando memoria, pero buscando una alineación de 16 bits en los datos, que proveen mayor velocidad para la CPU. Cuando necesita implementar un ciclo, puede decidir entre usar una estructura `for` o una `do..while`, dependiendo de las características del módulo; una u otra estructura de control le proporcionará mayor velocidad de ejecución.

b). Compilador con capacidades de optimización.

Como se ha mencionado, existen compiladores con capacidades de optimización automática. Por ejemplo, el compilador C del proyecto GNU (gcc) proporciona cuatro niveles de optimización, del 0 al 3 [6]. El nivel 0 es compilación sin optimización, esto es, el código generado es un fiel reflejo del código fuente escrito en C por el programador. Los niveles 1, 2 y 3 proporcionan optimización de código. Sin embargo,

no se garantiza que el código resultante sea un fiel reflejo del código fuente [7]. Al emplear estos niveles de optimización, queda a responsabilidad del programador verificar que el código resultante proporcione los resultados deseados.

Como ejemplo ilustrativo, consideremos el siguiente segmento de programa fuente en C:

```
Suma=0;
for (i=0; i < 100; i++)
{
    Suma+=i;
}
```

Al compilar sin optimización este código fuente con el compilador gcc, se obtiene la siguiente versión en lenguaje ensamblador como código de salida:

```
    mov DWORD PTR [esp+24], 0
    mov DWORD PTR [esp+28], 0
    jmp .L2
.L3:
    mov eax, DWORD PTR [esp+28]
    add DWORD PTR [esp+24], eax
    add DWORD PTR [esp+28], 1
.L2:
    cmp DWORD PTR [esp,28], 99
    jle .L3
```

Como puede observarse, el código generado es un fiel reflejo del programa en C. El operando `[esp+24]` es la variable `Suma`; el operando `[esp+28]` es la variable `i`. El problema con este código generado es que contiene demasiadas referencias a memoria. El acceso a memoria es una operación muy costosa en cuanto al consumo de ciclos de CPU. Por lo tanto, aunque este código es fiel reflejo del programa en C, es bastante ineficiente.

Al compilar el código fuente del ejemplo con nivel de optimización 1, el compilador obtiene el siguiente código de salida:

```
     mov eax, 0
.L2:
     add eax, 1
     cmp eax, 100
     jne .L2
     mov DWORD PTR [esp+4], 4950
```

Como se aprecia, este código generado está más optimizado que el anterior. Ya no existe la variable `i` como contador del ciclo, sino que se emplea el registro EAX de la CPU. Eso proporciona mayor velocidad de ejecución. Sin embargo, dentro del ciclo ya no se ejecutan instrucciones relevantes, salvo el incremento del contador. La línea de código interesante aquí es la última. `[esp+4]` es la variable `Suma`, en la que se almacena directamente el valor 4950. Se puede observar entonces que antes de generar el código de salida, el compilador determina que al concluir el ciclo, la variable `Suma` terminará con el valor 4950, entonces "saca" del ciclo ese cálculo y lo reemplaza con la asignación directa del valor final.

Si se compila ese código fuente con nivel de optimización 2, el compilador obtiene:

```
mov DWORD PTR [esp+4], 4950
```

Puede verse que la capacidad de optimización del compilador en este nivel es bastante buena. El código generado en este ejemplo es, además de corto, mucho más rápido.

Ahora, el código fuente compilado en este ejemplo es bastante simple y resulta sencillo para el compilador optimizarlo. Cuando los programas son más complejos, el poder de optimización del compilador ya no es tan eficiente.

c). Escritura de los módulos críticos en lenguaje ensamblador (optimización manual).

Consiste en realizar un análisis al programa en lenguaje de alto nivel a optimizar y determinar cuáles son los módulos que consumen más tiempo de ejecución. Entonces, pueden escribirse esos módulos en lenguaje ensamblador y ligarlos al programa principal durante la compilación.

## 2.3 El compilador GNU C y la optimización de código

El compilador GNU C es un compilador sofisticado de lenguaje C que emplea una gran cantidad de técnicas de optimización para producir código eficiente, algunas de las cuales han sido descritas y analizadas por Joshi [12] con el fin de tener un mejor entendimiento sobre lo que es la optimización y cómo ésta puede incrementar la eficiencia del código objeto generado.

Un compilador es un programa que lee un programa fuente escrito en un lenguaje de alto nivel y lo traduce, normalmente, a código de máquina. Si el compilador tiene capacidades de optimización, entonces "optimiza" el código de máquina de tal manera que requiera menos tiempo para ejecutarse, menos memoria o algunas veces ambas características. Por supuesto, independientemente del tipo de optimización que el compilador efectúe, ésta no deberá afectar la lógica del programa; esto es, la optimización debe preservar el significado del programa.

A continuación se describen las técnicas de optimización de código más representativas implementadas en el compilador C del proyecto GNU.

### 2.3.1 Plegado de constantes (*Constant folding*)

El plegado de constantes consiste en simplificar expresiones en las que hay valores constantes. Supongamos la sentencia `x = 45 * 88;` en un programa en C. Un compilador sin optimización generará el código para multiplicar 45 por 88 y almacenar el resultado en la variable `x`. Un compilador con optimización detecta que tanto el valor 45 como el 88 son constantes, por lo tanto, el producto será también una constante. Entonces, determinará que 45*88 es 3960 y entonces genera el código que simplemente almacena el valor 3960 en la variable `x`.

### 2.3.2 Propagación de constantes (*Constant propagation*)

Esta técnica consiste en sustituir en las expresiones los valores de constantes conocidas, o bien, de funciones preincorporadas (*built-in functions)* aplicadas a valores constantes. En este caso entendemos por constantes a las variables cuyo valor no cambia en cierta sección de código. Por ejemplo, considerando la siguiente sección de código:

```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

Propagando la variable *x* se obtiene:

```
int x = 14;
int y = 7 - 14 / 2;
return y * (28 / 14 + 2);
```

Continuando con la propagación se puede obtener:

```
int x = 14;
int y = 0;
return 0;
```

### 2.3.3 Eliminación de código muerto (*Dead Code Elimination*)

Esta técnica de optimización consiste en eliminar el código que no afecta los resultados del programa. Con esta eliminación se obtienen dos beneficios: por un lado se reduce el tamaño del programa y por el otro, al evitar la ejecución de instrucciones irrelevantes, se reduce el tiempo de ejecución. El código muerto comprende aquellas secciones que nunca serán ejecutadas, o bien, el código que contiene declaración de variables muertas, esto es, variables que son irrelevantes en el programa.

Considerando el código optimizado del ejemplo anterior de propagación de variables:

```
int x = 14;
int y = 0;
return 0;
```

Se observa que la declaración de las variables `x` y `y` es irrelevante, por lo que al aplicar la eliminación de código muerto, la optimización puede generar:

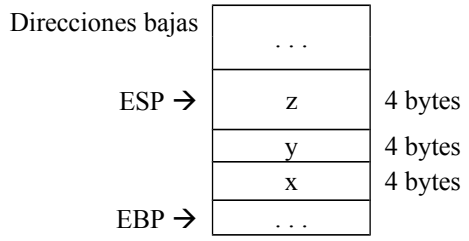
```
return 0;
```

Cabe mencionar que en los ejemplos anteriores se ha ilustrado la técnica de propagación de constantes usando código en lenguaje C. Sin embargo, el compilador en realidad efectúa la optimización durante la traducción a código intermedio. Se presenta a continuación otro ejemplo de las dos técnicas de optimización descritas previamente para ilustrar el proceso de optimización con el código en lenguaje ensamblador generado. En el ejemplo se presenta el código original en C, el código en lenguaje ensamblador generado sin optimización y la versión generada con optimización de nivel 2.

```
01: /* prueba2.c */
02: /* Ejemplo de propagación de constantes */
03: #include <stdio.h>
04:
05: int main()
06: {
07:     int x, y, z;
08:     x = 10;
09:     y = x + 45;
10:     z = y + 4;
11:     printf("El valor de z = %d", z);
12:     return 0;
13: }
14: /* fin de prueba2.c */
15:
16: /* ---------------------------------------------------------------- */
17: /* Código en lenguaje ensamblador sin optimización */
18:     .file   "prueba2.c"
19:     .version    "01.01"
20: gcc2_compiled.:
21: .section    .rodata
22: .LC0:
23:     .string "El valor de z = %d"
24: .text
25:     .align 4
26: .globl main
27:     .type    main,@function
28: main:
29:     pushl %ebp          /* guarda EBP en la pila */
30:     movl %esp,%ebp      /* EBP = ESP */
31:     subl $12,%esp       /* 3 variables x 4 bytes */
32:
33:     /* x = 10; */
34:     movl $10,-4(%ebp)   /* x = 10. x está en el tope de la pila */
35:
36:     /* y = x + 45; */
37:     movl -4(%ebp),%edx  /* EDX = x */
38:     addl $45,%edx       /* EDX = EDX + 45 */
39:     movl %edx,-8(%ebp)  /* y = EDX. y es el segundo de la pila */
40:
41:     /* z = y + 4 */
42:     movl -8(%ebp),%edx  /* EDX = y */
43:     addl $4,%edx        /* EDX = EDX + 4 */
44:     movl %edx,-12(%ebp) /* z = EDX. z es el tercero de la pila */
45:
46:     /* printf("El valor de z = ", z); */
47:     movl -12(%ebp),%eax    /* EAX = z */
48:     pushl %eax             /* EAX(=z) primer parámetro de printf */
49:     pushl $.LC0            /* segundo parámetro de printf */
50:     call printf
51:     addl $8,%esp           /* limpiar la pila */
52:
53:     /* return 0; */
54:     xorl %eax,%eax         /* regresa 0 */
55:     jmp .L1
56:     .p2align 4,,7
57: .L1:
```

```
58:     leave
59:     ret
60: .Lfe1:
61:     .size    main,.Lfe1-main
62:     .ident  "GCC: (GNU) egcs-2.91.66 Linux (egcs-1.1.2 release)"
63: /* fin del código en lenguaje ensamblador sin optimización */
64: /* ------------------------------------------------------------- */
65:
66: /* ------------------------------------------------------------- */
67: /* Código en lenguaje ensamblador generado optimizado en nivel 2*/
68:
69:     .file   "prueba2.c"
70:     .version    "01.01"
71: gcc2_compiled.:
72: .section    .rodata
73: .LC0:
74:     .string "El valor de z = %d"
75: .text
76:     .align 4
77: .globl main
78:     .type    main,@function
79: main:
80:     pushl %ebp              /* Guarda EBP en la pila */
81:     movl %esp,%ebp          /* EBP = ESP */
82:
83:     /* Por propagación de constante, z será siempre 59 */
84:     /* printf("El valor de z = %d", z); */
85:     pushl $59               /* primer parámetro de printf     */
86:     pushl $.LC0             /* Segundo parámetro de printf    */
87:     call printf
88:                             /* no limpiar pila, está saliendo */
89:     /* return 0; */
90:     xorl %eax,%eax
91:     leave
92:     ret
93: .Lfe1:
94:     .size    main,.Lfe1-main
95:     .ident  "GCC: (GNU) egcs-2.91.66 Linux (egcs-1.1.2 release)"
96: /* fin del código en lenguaje ensamblador optimizado */
97: /* ------------------------------------------------------------- */
```

Se observa que en la función `main()` se definen tres variables locales cuyo espacio de almacenamiento se reserva en la pila. Para acceder a estas variables se utiliza el modo de direccionamiento indexado mediante el registro EBP. Por lo tanto, la primera instrucción guarda el valor actual de EBP en la pila. Luego se copia el apuntador de la pila ESP hacia EBP. Debido a que se requiere reservar espacio para tres variables de cuatro bytes cada una, se resta 12 a ESP, esto es, se hace crecer la pila en doce bytes. La variable `x` es el elemento que estará en el fondo de la pila con desplazamiento de -4 respecto a ESP, como se muestra en el siguiente diagrama:




El desplazamiento para la variable `y` es de -8 respecto a EBP y para la variable `z` es de -12. Ahora, consideremos la asignación `x=10`. Esa sentencia se implementa en la línea 34 con la instrucción movl $10,-4(%ebp). El modo de direccionamiento indexado se escribe de la forma <desplazamiento>(<registro base>). La instrucción anterior copiará el valor constante 10 hacia la dirección (EBP-4), esto es, hacia la dirección de la variable x en la pila. De manera similar, -8(%ebp) se usa para acceder a la variable `y`, mientras que, -12(%ebp) se usa para acceder a la variable `z`. Las líneas 37, 38 y 39 evalúan `x+45` y asignan el resultado a `y`. Para hacer eso, el valor de `x` primero se copia al registro EDX. A él se suma el valor 45 y el resultado, que está en EDX, se copia a `y`. El código para la instrucción `z=y+4` es similar. En la línea 47, los parámetros para la función `printf()` son almacenados en la pila. El último parámetro (`z`) se guarda primero y después la dirección de la cadena. En la línea 51 se limpia la pila sumando 8 a ESP.

Ahora, al observar el programa, cuando se evalúa `x+45`, el valor de `x` siempre será 10, dada la asignación `x=10`. Por lo tanto, `x+45` siempre se avaluará en 55. Entonces la sentencia siempre asigna el valor 55 a la variable `y`. De manera similar, al evaluar `y+4`, el resultado siempre será 59.

El código en lenguaje ensamblador generado con optimización se muestra de la línea 69 a la 95. Se puede notar que en la línea 85 el compilador almacena en la pila directamente el valor 59 y enseguida la dirección de la cadena y luego llama a la función

`printf()`. Esto es, con el plegado de constantes el compilador evalúa las expresiones en el programa sólo una vez y determina el valor final en el código generado. Una cosa interesante más que hay que observar es que, después del plegado de constantes, no se necesitan las variables `x`, `y` y `z`. Por lo tanto no se asigna espacio para ellas en la pila, reduciendo con esto los requerimientos de memoria para el programa. Esto resalta el hecho de que una optimización puede conducir a otra. En el caso de este ejemplo el plegado de constantes conlleva una reducción en el tiempo de ejecución y a la vez una reducción en los requerimientos de espacio.

### 2.3.4 Eliminación de subexpresiones comunes (*Common Subexpression Elimination*)

Muchas veces sucede que la misma expresión es evaluada en diferentes lugares del mismo programa y los valores de los operandos en la expresión no cambian entre las evaluaciones de la misma. Por ejemplo, un programa puede evaluar `a*b` al inicio y al final. Si los valores de `a` y `b` no cambian entre las evaluaciones de `a*b`, entonces, en lugar de evaluar esa expresión de nuevo al final, se puede conservar el resultado de la evaluación realizada al inicio en alguna variable temporal y usarla al final. Esto ayuda a eliminar cómputos redundantes en el programa. Consideremos el siguiente programa ejemplo:

```
001: /* prueba3.c */
002: /* Eliminación de subexpresiones communes y propagación de
        constantes*/
003: #include <stdio.h>
004:
005: int main()
006: {
007:     int a, b;
008:     int x, y, z;
009:     scanf("%d %d", &a, &b);
010:     x = a * b;
011:
012:     if(b >= 4)
013:     {
014:         y = a * b;
```

```
015:         z = 0;
016:     }
017:     else
018:     {
019:         z = a * b * 4;
020:         y = 0;
021:     }
022:
023:     printf("x = %d, y = %d, z = %d\n", x, y, z);
024:     return 0;
025: }
026: /* Fin de prueba3.c */
027:
028: /* ------------------------------------------------------------ */
029: /* Código no optimizado generado en lenguaje ensamblador */
030:     .file   "prueba3.c"
031:     .version    "01.01"
032: gcc2_compiled.:
033: .section    .rodata
034: .LC0:
035:     .string "%d %d"
036: .LC1:
037:     .string "x = %d, y = %d, z = %d\n"
038: .text
039:     .align 4
040: .globl main
041:     .type    main,@function
042: main:
043:     pushl %ebp              /* guardar EBP */
044:     movl %esp,%ebp          /* EBP = ESP */
045:     subl $20,%esp           /* Crear espacio para 5 variables */
046:
047:     /* scanf("%d %d". &a, &b); */
048:     leal -8(%ebp),%eax
049:     pushl %eax              /* guardar la dirección de b */
050:     leal -4(%ebp),%eax
051:     pushl %eax              /* guardar la dirección de a */
052:     pushl $.LC0             /* guardar la dirección de la cadena */
053:     call scanf
054:     addl $12,%esp           /* limpiar la pila después de scanf */
055:
056:     /* x = a * b; */
057:     movl -4(%ebp),%eax      /* EAX = a */
058:     imull -8(%ebp),%eax     /* EAX = EAX * b = a * b */
059:     movl %eax,-12(%ebp)     /* x = EAX = a * b */
060:
061:     /* if( b >= 4)... */
062:     cmpl $3,-8(%ebp)        /* compare b with 3 */
063:     jle .L2                 /* .L2, es la parte para el else */
064:
065:     /* y = a * b; */
066:     movl -4(%ebp),%eax      /* EAX = a */
067:     imull -8(%ebp),%eax     /* EAX = EAX * b = a * b */
068:     movl %eax,-16(%ebp)     /* y = EAX = a * b */
069:     /* z = 0; */
070:     movl $0,-20(%ebp)
071:     jmp .L3                 /* brincar la parte del else */
```

```
072:
073:     .p2align 4,,7
074: .L2:
075:     /* Aquí inicia la parte para el else */
076:
077:     /* z = a * b * 4; */
078:     movl -4(%ebp),%eax       /* EAX = a */
079:     imull -8(%ebp),%eax      /* EAX = EAX * b = a * b */
080:     leal 0(,%eax,4),%edx     /* EDX = EAX*4 + 0 */
081:     movl %edx,-20(%ebp)      /* z = EDX */
082:     /* y = 0; */
083:     movl $0,-16(%ebp)
084: .L3:
085:     /* termina la estructura if..else  */
086:
087:     /* printf("x = %d, y = %d, z = %d\n", x, y, x); */
088:     movl -20(%ebp),%eax
089:     pushl %eax               /* guardar la dirección de z */
090:     movl -16(%ebp),%eax
091:     pushl %eax               /* guardar la dirección de y */
092:     movl -12(%ebp),%eax
093:     pushl %eax               /* guardar la dirección de  x */
094:     pushl $.LC1              /* dirección de la cadena */
095:     call printf
096:     addl $16,%esp            /* limpiar la pila después de printf */
097:
098:     /* return 0 */
099:     xorl %eax,%eax
100:     jmp .L1
101:     .p2align 4,,7
102: .L1:
103:     leave
104:     ret
105: .Lfe1:
106:     .size    main,.Lfe1-main
107:     .ident  "GCC: (GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2
release)"
108: /* Fin del código en lenguaje ensamblador no optimizado */
109: /* ------------------------------------------------------------ */
110:
111: /* ------------------------------------------------------------ */
112: /* Código optimizado generado en lenguaje ensamblador */
113:     .file   "prueba3.c"
114:     .version    "01.01"
115: gcc2_compiled.:
116: .section    .rodata
117: .LC0:
118:     .string "%d %d"
119: .LC1:
120:     .string "x = %d, y = %d, z = %d\n"
121: .text
122:     .align 4
123: .globl main
124:     .type    main,@function
125: main:
126:     pushl %ebp               /* guardar EBP */
127:     movl %esp,%ebp           /* EBP = ESP */
```

```
128:     subl $8,%esp             /* espacio para 2 variables en pila */
129:
130:     /* scanf("%d %d", &a, &b); */
131:     leal -4(%ebp),%eax
132:     pushl %eax               /* guardar la dirección de b */
133:     leal -8(%ebp),%eax
134:     pushl %eax               /* guardar la dirección de a */
135:     pushl $.LC0              /* dirección de la cadena */
136:     call scanf
137:
138:     /* x = a * b; */
139:     movl -4(%ebp),%eax       /* EAX = b */
140:     movl %eax,%edx           /* EDX = EAX = b */
141:     imull -8(%ebp),%edx      /* EDX = EDX * a = b * a = a * b */
142:
143:     addl $12,%esp            /* limpiar la pila */
144:     /* if( b >= 4).... */
145:     cmpl $3,%eax             /* compara EAX = b con 3 */
146:     jle .L17                 /* la parte del else es .L17 */
147:
148:                             /* y está en ECX, z en EAX, x en EDX */
149:     /* y = a * b; */
150:     movl %edx,%ecx
151:     /* z = 0; */
152:     xorl %eax,%eax
153:     jmp .L18                 /* saltar la parte del else */
154:     .p2align 4,,7
155: .L17:
156:     /* z = a * b * 4; */
157:     leal 0(,%edx,4),%eax     /* LEA EAX, [EDX*4]+0 */
158:     /* y = 0; */
159:     xorl %ecx,%ecx
160: .L18:
161:     pushl %eax               /* guardar el valor de z */
162:     pushl %ecx               /* guardar el valor de y */
163:     pushl %edx               /* guardar el valor de x */
164:     pushl $.LC1              /* guardar dirección de la cadena */
165:     call printf
166:     /* no es necesario limpiar la pila después de este printf */
167:
168:     /* return 0; */
169:     xorl %eax,%eax
170:     leave
171:     ret
172: .Lfe1:
173:     .size    main,.Lfe1-main
174:     .ident  "GCC: (GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2
release)"
175: /* fin del código optimizado en lenguaje ensamblador */
176: /* ------------------------------------------------------------ */
```

El listado muestra el código en C original y el código en lenguaje ensamblador generado por el compilador; la versión no optimizada y una versión optimizada.

El programa contiene un ejemplo de una subexpresión común. En la línea 10, la expresión `a*b` es evaluada por primera vez y luego es evaluada de nuevo en las líneas 14 y 19. Las dos últimas evaluaciones de `a*b` son redundantes debido a que los valores de `a` y `b` no cambian después de la primera evaluación. Entonces, estas dos evaluaciones son subexpresiones comunes que pueden ser eliminadas.

De la línea 30 a la 108 se muestra el código en lenguaje ensamblador no optimizado generado por el compilador. Se observa en primer término que se reserva espacio en la pila para las cinco variables. Respecto a la subexpresión `a*b`, se observa que se evalúa por primera vez en las líneas 57 y 58, luego se evalúa de nuevo en las líneas 66 y 67, y por tercera vez en las líneas 78 y 79.

De la línea 113 a la 176 se muestra el código en lenguaje ensamblador generado con optimización. En esta versión lo primero que se puede notar es que en la pila se reserva espacio sólo para dos variables (`a` y `b`), esto es, se requieren sólo 8 bytes de la pila contra los 20 bytes reservados en la versión no optimizada. Estas variables tienen un significado especial en el programa, debido a que se requiere conocer su dirección de memoria porque son utilizadas en la función `scanf()`, mientras que las variables guardadas en registros no pueden tener una dirección de memoria (como el compilador a decidido tratar a las variables `x`, `y` y `z`). Los registros que el compilador ha seleccionado para almacenar estas otras variables son: EDX para `x`, ECX para `y` y EAX para `z`. Las sentencias 139 a la 141 evalúan `a*b` y el resultado se almacena en el registro EDX (la variable `x`).

Luego inicia la sentencia `if`. En la parte `if`, la expresión `a*b` debe ser reevaluada, por lo que esperaríamos que el compilador utilizara el resultado previamente almacenado en el registro EDX. Eso es justamente lo que el compilador realiza. En lugar de reevaluar la expresión, se copia el contenido de EDX hacia ECX (la variable `y`). En la parte

correspondiente al `else`, está la subexpresión común `z=a*b*4`. Entonces, dado que el resultado de `a*b` está almacenado en EDX, sólo es necesario multiplicar el contenido de este registro por 4 y guardar el resultado en EAX (la variable `z`). Esto podría implementarse con las instrucciones:

```
movl %edx, %eax          /* EAX = EDX  = a * b */
imull $4, %eax           /* EAX = EAX * 4 */
```

Otra opción sería emplear una instrucción de desplazamiento (shift) para hacer la multiplicación:

```
movl %edx, %eax          /* EAX = EDX  = a * b */
shll $2, %eax            /* EAX = EAX * 4 */
```

Sin embargo el compilador utiliza un truco para optimizar el código. Usa la instrucción:

```
leal 0(,%edx,4), %eax
```

Esta instrucción aprovecha las capacidades de escalado e indexación de los procesadores 80386 y posteriores. En la instrucción anterior el registro EDX es considerado como registro índice, el valor 4 como la escala y el valor 0 como un desplazamiento. Se calcula la dirección efectiva y se almacena el resultado en el registro EAX. Con esto, el registro EAX obtiene el valor `EDX=(a*b)*4+0`. O sea, `a*b*4`. La instrucción `leal` se ejecuta en dos ciclos del CPU, mientras que la instrucción `shll` toma alrededor de siete ciclos.

Se puede observar que la eliminación de subexpresiones comunes puede ahorrar gran cantidad de instrucciones y a la vez ahorrar espacio en el código resultante. Se puede entender que para efectos de optimización, el compilador analiza cómo cambian las variables, cómo debe evaluar las expresiones y cuáles registros puede utilizar para

almacenar variables. Este tipo de análisis que el compilador lleva a cabo sobre un programa fuente es conocido como análisis de flujo de datos.

### 2.3.5 Variables de inducción (*Strength Reduction using Induction Variable*)

Este tipo de optimización consiste en reemplazar operaciones por otras menos costosas. Por ejemplo, en un programa que contiene la evaluación $x^2$ es mucho más eficiente si se multiplica x*x en lugar de hacer una llamada a la función de potencia. Un tipo de estructura de control en el que este tipo de optimización puede ser usada es en los ciclos. Muchas veces, en un ciclo una variable cambia sincronizadamente con el contador del ciclo. Tal variable puede ser entonces considerada como una variable de inducción. Estas variables permiten al compilador aplicar optimización, como se muestra en el siguiente ejemplo:

```
01: /* prueba5.c */
02: /* Ejemplo de eliminación de variables de inducción */
03: int main()
04: {
05:     int i, j;
06:
07:     for(i = 0; i < 10; i++)
08:     {
09:         j = i * 7;
10:         printf("i = %d, j = %d", i, j);
11:     }
12:     return 0;
13: }
14: /* fin de prueba5.c */
15:
16: /* ---------------------------------------------------------- */
17: /* Código en lenguaje ensamblador optimizado */
18:     .file   "prueba5.c"
19:     .version    "01.01"
20: gcc2_compiled.:
21: .section    .rodata
22: .LC0:
23:     .string "i = %d, j = %d"
24: .text
25:     .align 4
26: .globl main
27:     .type    main,@function
28: main:
29:     pushl %ebp                  /* guardar EBP en la pila */
30:     movl %esp,%ebp              /* ESP = EBP */
```

```
31:
32:     pushl %esi                  /* ESI contendrá 'j' */
33:     pushl %ebx                  /* EBX contendrá 'i' */
34:     xorl %ebx,%ebx              /* i y j se inician en 0 */
35:     xorl %esi,%esi
36:     .p2align 4,,7
37: .L5:
38:     /* printf("i = %d, j = %d", i, j); */
39:     pushl %esi                  /* guarder valor de j */
40:     pushl %ebx                  /* guarder valor de i */
41:     pushl $.LC0                 /* guarder dirección de la cadena */
42:     call printf
43:     addl $12,%esp               /* limpiar la pila */
44:
45:     /* En lugar de j = i * 7, es más eficiente j = j + 7 */
46:     addl $7,%esi
47:     incl %ebx                   /* i++ */
48:     cmpl $9,%ebx                /* si i <= 9, repetir el ciclo */
49:     jle .L5
50:
51:     /* return 0; */
52:     xorl %eax,%eax
53:     leal -8(%ebp),%esp
54:     popl %ebx
55:     popl %esi
56:     leave
57:     ret
58: .Lfe1:
59:     .size    main,.Lfe1-main
60:     .ident  "GCC: (GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2
release)"
61:
62: /* Fin del código en lenguaje ensamblador optimizado */
63: /* ------------------------------------------------------------ */
```

En este ejemplo, `i` es el contador del ciclo mientras que `j` es la variable de inducción, ya que siempre cambia sincronizadamente con la variable `i`. En el código en lenguaje ensamblador generado, vemos que el compilador ha decidido utilizar registros para almacenar tanto `i` como `j`, `i` en EBX y `j` en ESI. En la línea 34 se inicia EBX (=`i`) a cero, como se indica en la inicialización del ciclo `for`. El compilador puede detectar que en la primera pasada `j` será iniciada en cero. Con esto, se inicia ESI en cero en la línea 35. El ciclo inicia en la línea 37 y sigue hasta la línea 49. Dado que `j` ya tiene su valor asignado, lo primero que realiza el compilador dentro del ciclo es llamar a la función `printf()`. En este punto, el compilador ha ajustado el orden de las sentencias dentro del ciclo para efectuar una optimización por variables de inducción. Una vez

analizado el programa, el compilador revisa la parte interna el ciclo. El valor de `i` siempre se incrementa en 1 y `j`, que es la variable de inducción, siempre incrementa su valor en 7. Por lo tanto, en lugar de multiplicar `i` por 7 (que es muy costoso), solamente suma 7 al valor contenido en `j` antes de volver a la siguiente iteración.

Este ejemplo muestra cómo una operación de multiplicación costosa ha sido reemplazada por una suma, que es más barata en términos de tiempo de ejecución.

# Capítulo 3

## Algoritmos genéticos

## 3.1 Descripción

Los algoritmos genéticos proporcionan una técnica de búsqueda heurística inspirada en la evolución de las especies. David E. Goldberg [13] los define de la siguiente manera: "Los algoritmos genéticos son algoritmos de búsqueda basados en los mecanismos de la selección natural y la genética natural."

El material genético de un individuo se conoce como el genotipo, mientras que su manifestación como un organismo se conoce como el fenotipo. La selección natural opera solamente sobre el fenotipo, pero el genotipo es el vehículo definitivo de la herencia. Las células contienen una clase de moléculas conocidas como proteínas, cuya forma, concentración y comportamiento determinan las propiedades de la célula. Por ejemplo, las células de cabello y las células de músculo son diferentes ya que están compuestas de diferentes proteínas. La definición de proteínas específicas depende de otra molécula conocida como ADN (ácido desoxirribonucleico). El ADN es el material genético transmitido entre generaciones y la información genética está contenida en los genes. Un gen es un segmento de la cadena ADN [14].

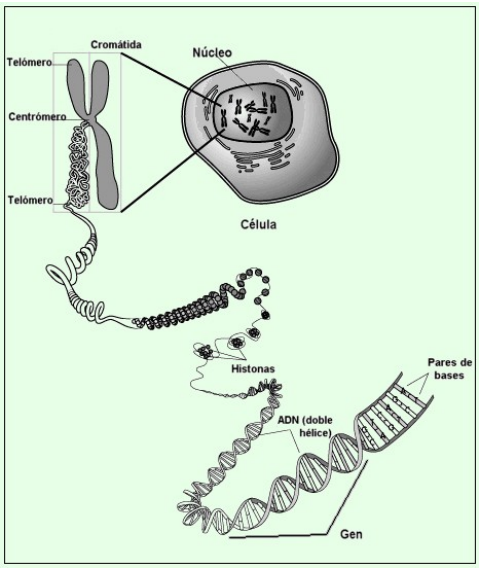


Figura 4.1 Representación de una célula, cromosoma, ADN y gen.

Los algoritmos genéticos fueron propuestos por John Holland [15] en la universidad de Michigan.

El algoritmo genético usualmente trabaja a nivel genotípico representado por una cadena de valores binarios, donde cada posición representa un gen. Para conocer el nivel fenotípico de un individuo se debe utilizar un proceso de decodificación.

Operación de un Algoritmo Genético típico:

- Generar una población inicial aleatoria de "n" individuos.
- Evaluar cada solución (individuo).
- Seleccionar los individuos más aptos de acuerdo a su valor de la función de aptitud.
- Aplicar operadores de variación (cruce, mutación) para generar los descendientes.
- Evaluar a los descendientes.
- Seleccionar a los individuos que sobrevivirán para la próxima generación.
- Iterar hasta que se cumpla una condición de paro.

## 3.2 Operadores genéticos

Los operadores genéticos son las funciones que se aplican a los individuos de una población. El resultado de la aplicación de los operadores determina las características que tendrán los individuos de generaciones futuras. Los operadores genéticos que se aplican típicamente son:

**Selección.** Este operador determina cuáles individuos de una población conformarán la siguiente generación. La estrategia de selección y cantidad de individuos seleccionados dependerá de las necesidades de la aplicación con la que se trabaja. La selección de individuos está en relación directa con la **aptitud** de cada individuo; esta aptitud se determina mediante la **función de aptitud**.

**Cruce (cruza, cruzamiento).** Consiste en obtener un individuo nuevo a partir de dos individuos padres. La cantidad de cruzamientos a realizar en una población es un parámetro que debe ser determinado dependiendo de las características de la aplicación.

**Mutación**. Este operador se aplica aleatoriamente. La mutación consiste en modificar el ADN de un individuo (normalmente consiste en cambiar algún bit del valor del individuo) e incorporarlo a la siguiente generación.

### 3.3 Algoritmos genéticos distribuidos con migración (AG-DM)

Como su nombre lo sugiere, esta técnica de programación de algoritmos genéticos se basa en la computación paralela. En el caso de un algoritmo genético tradicional, en el que se define una sola población, puede presentarse el problema de que se confundan "regiones sub-óptimas" con los resultados óptimos que deben obtenerse y con ello, converger prematuramente.

Para solventar el problema de convergencia prematura, se emplean los AG-DM, en los que se define más de una población, y con eso, el algoritmo puede descartar aquellas poblaciones que queden atrapadas en regiones sub-óptimas y considerar como resultado óptimo a la población que proporcione el mejor resultado.

Esta estrategia de paralelización consiste en utilizar un conjunto estático de sub-poblaciones independientes y utilizar un operador genético adicional de migración, mediante la cual, cada cierto número de generaciones se da el intercambio de individuos entre las poblaciones, como una forma de compartir material genético entre las mismas. Esto hace necesario la introducción de un conjunto de parámetros migratorios que determinen la periodicidad del proceso de migración, la cantidad de individuos a migrar en cada ocasión y las poblaciones destino de dichos individuos [16].

En este tipo de algoritmo, se denomina periodo migratorio al número de generaciones que se ejecutan en cada procesador antes del intercambio de material genético con los demás, mientras que a la cantidad de individuos que se seleccionan para migrar cada vez, se le denomina razón de migración [16].

# Capítulo 4

## Batería de operadores en lenguaje ensamblador

## 4.1 Aplicación a considerar

Como se mencionó en la exposición de objetivos, se ha decidido en el presente proyecto realizar la optimización manual sobre la aplicación "Algoritmos Genéticos Distribuidos con Migración" (AG-DM) [10]. Esta aplicación está programada completamente en lenguaje C, implementa un algoritmo de optimización de variables continuas y ha sido objeto de estudio en otros trabajos [10] [17]. Trata sobre la minimización de funciones consideradas en la batería de pruebas CEC05 [18]. Aunque en el trabajo relativo a la presente tesis se programaron y probaron diez de las funciones de la batería CEC05, se decidió realizar las corridas y análisis de tiempos de ejecución para las cuatro que se mencionan a continuación, debido a que sobre ellas contábamos con resultados de trabajos previos [8] [17].

**Función de Ackley**

$$F(\mathbf{x}) = -20\exp(-0.2\sqrt{\frac{1}{D}\sum_{i=1}^{D}\mathbf{x}_i^2}) - \exp(\frac{1}{D}\sum_{i=1}^{D}\cos(2\pi\mathbf{x}_i)) + 20 + e \qquad (1)$$

D es el número de dimensiones, el espacio de búsqueda está entre -32 y 32 para cada dimensión y sus propiedades son: multimodal, rotable, desplazabe, no separable y escalable. Presenta una gran cantidad de óptimos locales y el óptimo global está en la frontera. Una representación para dos dimensiones se muestra en la siguiente figura:

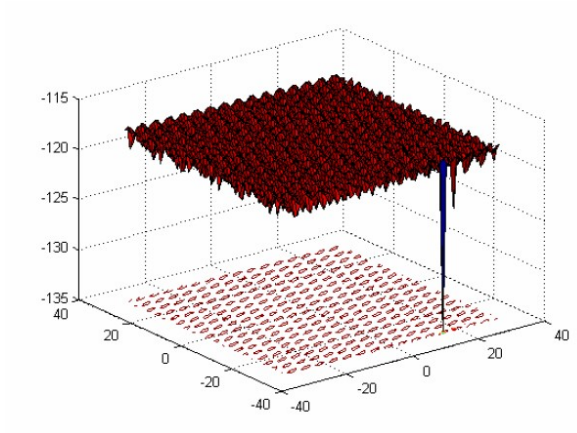



Figura 5.1.1 Representación de la función de Ackley

**Función de Rastrigin**

$$F(\mathbf{x}) = \sum_{i=1}^{D} (\mathbf{x}_i^{\,2} - 10\cos(2\pi \mathbf{x}_i) + 10) \tag{2}$$

D es el número de dimensiones, el espacio de búsqueda está entre -5 y 5 para cada dimensión y sus propiedades son: multimodal, desplazabe, separable, escalable y presenta gran cantidad de óptimos locales. Una representación para dos dimensiones se muestra en la siguiente figura:

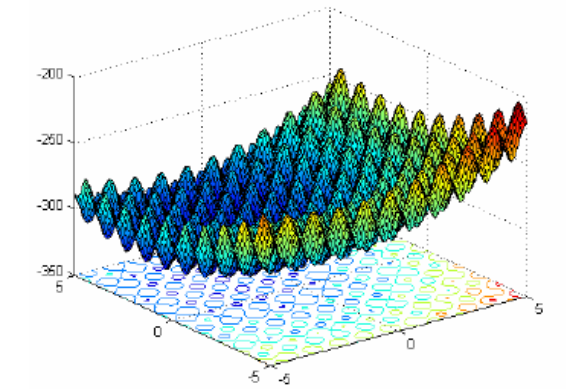



Figura 5.1.2 Representación de la función de Rastrigin

**Función de Schwefel**

$$F(\mathbf{x}) = \sum_{i=1}^{D} (\sum_{j=1}^{i} \mathbf{x}_j)^2 \tag{3}$$

D es el número de dimensiones, el espacio de búsqueda está entre -100 y 100 para cada dimensión y sus propiedades son: unimodal, desplazabe, no separable y escalable. Una representación para dos dimensiones se muestra en la siguiente figura:

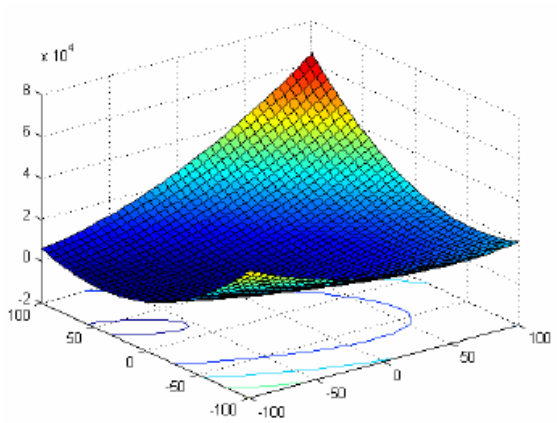



Figura 5.1.3 Representación de la función de Schwefel

**Función de Weierstrass**

$$F(\mathbf{x}) = \sum_{i=1}^{D} (\sum_{k=0}^{k\max} [a^k \cos(2\pi b^k (\mathbf{x}_i + 0.5))]) - D \sum_{k=0}^{k\max} [a^k \cos(2\pi b^k \cdot 0.5)] \tag{4}$$

D es el número de dimensiones, a=0.5, b=3, Kmax=20, el espacio de búsqueda está entre -0.5 y 0.5 para cada dimensión y sus propiedades son: multimodal, desplazabe, rotable, no separable, escalable, con gran cantidad de óptimos locales y continua pero diferenciable sólo en algunos intervalos. Una representación para dos dimensiones se muestra en la siguiente figura:

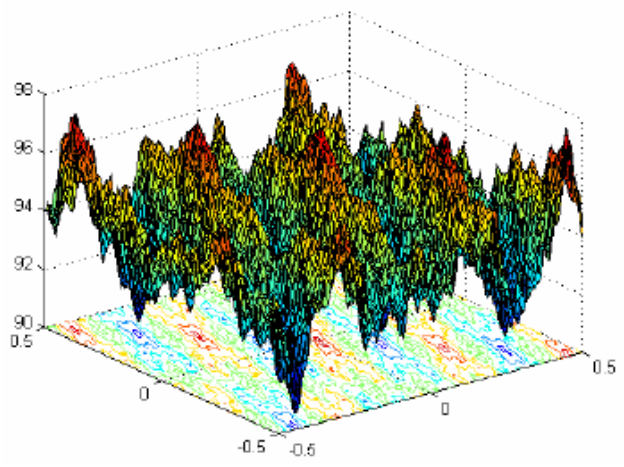



Figura 5.1.4 Representación de la función de Weierstrass

## 4.2 Estructura general de la aplicación AG-DM

La aplicación AG-DM está conformada por los siguientes archivos fuente:

- `agdm.c.` Este es el módulo principal de la aplicación. Contiene la definición de los parámetros globales para la operación del algoritmo genético y la función main() es el proceso maestro, encargado de la creación de los procesos esclavos en el cluster. Una vez creados los esclavos, el proceso maestro les envía los parámetros a considerar en el AG. Posteriormente, recolecta los resultados generados por los esclavos. Este proceso también genera el archivo de reporte con los resultados generados en la ejecución.

- `agdmesc.c.` Este archivo contiene el código que deben ejecutar los esclavos. Recibe los parámetros AG del proceso maestro, inicia el generador de números pseudoaleatorios, genera aleatoriamente la primera población de individuos y entra al ciclo principal en el que se ejecutan los operadores genéticos (selección, cruce y mutación). Posteriormente hace el intercambio de algunos individuos con otros procesos esclavos y finalmente envía al proceso maestro la información relativa al mejor individuo generado.

- `scaffer.h, ackley.h, esferica.h, rastrigin.h, schwefel.h, schwefel-noise.h, griewank.h, rosenbrock.h, weierstrass.h, eliptica.h y quadric.h.` Cada uno de estos archivos contiene el código que implementa a cada una de las funciones de aptitud seleccionadas para su estudio.

- `aptitud.h.` Contiene las directivas de inclusión de los archivos con las funciones de aptitud. Antes de compilar la aplicación, hay que dejar sin comentario la directiva de inclusión de la función a analizar y comentar todas las demás directivas.

- `Makefile.` Contiene los comandos requeridos para la compilación de la aplicación. Además, traslada los archivos ejecutables (agdm y agdmesc) al directorio de PVM desde el que serán ejecutados.

## 4.3 Herramientas y plataforma de ejecución

Para realizar la edición de código, generación de los ejecutables y las pruebas de eficiencia de la aplicación AG-DM originalmente escrita en lenguaje C contra la versión optimizada manualmente (C + ASM), se empleó la plataforma que se describe a continuación.

### 4.3.1 Software de base

- Sistema operativo: Linux Fedora 17
- Entorno paralelo: PVM (Parallel Virtual Machine)
- Editor de textos: GEDIT
- Compilador: GNU C

- Ensamblador: NASM

### 4.3.2 Cluster

| | |
|---|---|
| Características de cada PC: | Marca Dell con procesador Pentium Core 2 Duo de 2.6 Ghz y 2 GB de RAM |
| Número de PCs: | 14 |
| Núcleos: | 28 |
| Sistema operativo: | Linux Fedora 17, 32 bits |
| Máquina virtual: | PVM (*Parallel Virtual Machine*) |

### 4.3.3 Habilitación del cluster con PVM

Para instalar la última versión de PVM en Fedora 17, debemos ejecutar en una terminal con privilegios de usuario *root* el siguiente comando:

```
#yum -y install pvm pvm-gui
```

Tras lo cual debemos agregar al archivo .bashrc las líneas:

```
PVM_ROOT=/usr/share/pvm3
PVM_ARCH=LINUXI386
PVM_RSH=/usr/bin/ssh
PVM_ALLOW_ROOT=yes

export PVM_ROOT PVM_ARCH PVM_RSH PVM_ALLOW_ROOT
```

Debemos además agregar al archivo /etc/hosts una lista de los nombres de los nodos del cluster y su respectiva dirección IP, para el presente trabajo:

```
10.0.0.1  maquina1
10.0.0.2  maquina2
10.0.0.3  maquina3
10.0.0.4  maquina4
10.0.0.5  maquina5
10.0.0.6  maquina6
10.0.0.7  maquina7
```

```
10.0.0.8  maquina8
10.0.0.9  maquina9
10.0.0.10 maquina10
10.0.0.11 maquina11
10.0.0.12 maquina12
10.0.0.13 maquina13
10.0.0.14 maquina14
```

Con el propósito de facilitar la administración del cluster, se recomienda habilitar en el nodo maestro, el acceso a los nodos esclavos vía ssh sin password, usando únicamente la autentificación por llave pública.

Una vez iniciada nuestra sesión como usuario *root*, primeramente deberemos generar la llave pública en el servidor "Maestro".

```
ssh-keygen -t rsa
```

En pantalla deberá aparecer un mensaje parecido al siguiente:

```
Generating public/private rsa key pair.
Enter file in which to save the key (/home/rob/.ssh/id_rsa):
```

Presionamos la tecla  Enter para dejar los archivos por default para las llaves.

A continuación se nos pide una frase de password, la cual dejaremos en blanco para efectos de poder entrar sin password a los servidores (pero autentificando la sesión con la llave privada que solo tendremos nosotros)

```
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/rob/.ssh/id_rsa.
Your public key has been saved in /home/rob/.ssh/id_rsa.pub.
The key fingerprint is:
a6:5c:c3:fb:19:94:0d:06:a1:a6:29:58:fa:80:0a:bc
usuario@localhost
```

Para dejar la frase en blanco solo hay que pulsar la tecla Enter en un par de ocasiones.

Con lo anterior se han creado dos archivos en el directorio .ssh, id_rsa que es su llave privada e id_rsa.pub que es la llave pública.

Para usar este par de llaves en un servidor basta con agregar la llave pública en su archvio .ssh/authorized_keys2. Por ejemplo, si sólo se va a acceder de nuestro equipo, podemos ejecutar:

```
ssh server "mkdir .ssh; chmod 0700 .ssh"
scp .ssh/id_rsa.pub server:.ssh/authorized_keys2
```

La primera instrucción crea el directorio .ssh en la cuenta del usuario en el servidor "esclavo", la segunda instrucción, copia nuestra llave pública sobre la lista de llaves autorizadas en dicho servidor.

De aqui en adelante podremos acceder al servidor mediante ssh, en la cuenta con la que estemos trabajando en el servidor "maestro" sin necesidad de teclear el password.

La seguridad de este método será tan buena como protegida se encuentre nuestra llave privada.
Para mas información puede consultarse el excelente libro "Linux Server Hacks" de Rob Flickenger publicado por O`Reilly [19].

Adicionalmente, se elaboraron dos procesos por lote, el primero (llamado cluster) permite ejecutar una instrucción en todos los esclavos del cluster:

```
maquinas=`cat ~/cluster.lst`
for HOST in $maquinas
  do
    ssh $HOST $*
  done
```

El archivo cluster.lst deberá encontrarse en el directorio casa del usuario del cluster y contener un listado de los esclavos, por ejemplo:

```
maquina2
maquina3
maquina4
maquina5
maquina6
maquina7
maquina8
maquina9
maquina10
maquina11
maquina12
maquina14
```

El segundo proceso por lotes (llamado copia), que permite copiar un archivo a todos los esclavos del cluster, es muy útil para distribuir los archivos ejecutables una vez que han sido compilados.

```
maquinas=`cat ~/cluster.lst`
for HOST in $maquinas
do
     scp $1 $HOST:$2
done
```

Este comando puede invocarse dentro del archivo Makefile, como se muestra en el siguiente ejemplo:

```
agdma4: agdma4esc agdma4.c aptitud.asm mt19937.asm agdmoper.asm
definiciones.h deffunciones.h scaffer.inc ackley.inc
esferica.inc rastrigin.inc schwefel.inc schwefel-noise.inc
griewank.inc rosenbrock.inc weierstrass.inc eliptica.inc
quadric.inc Makefile
     gcc agdma4.c -I $(PVM_ROOT)/include -L $(PVM_ROOT)/lib/$
(PVM_ARCH) -lm -lpvm3 -o agdma4
     copia agdma4  $(PVM_ROOT)/bin/$(PVM_ARCH)
     mv agdma4 $(PVM_ROOT)/bin/$(PVM_ARCH)
agdma4esc: agdma4esc.c aptitud.asm mt19937.asm agdmoper.asm
definiciones.h deffunciones.h scaffer.inc ackley.inc
esferica.inc rastrigin.inc schwefel.inc schwefel-noise.inc
griewank.inc rosenbrock.inc weierstrass.inc eliptica.inc
quadric.inc Makefile
     nasm -f elf aptitud.asm
     nasm -f elf mt19937.asm
     nasm -f elf agdmoper.asm
     gcc agdma4esc.c -I $(PVM_ROOT)/include -L $(PVM_ROOT)/lib/$
(PVM_ARCH) -lm -lpvm3 -o agdma4esc aptitud.o mt19937.o
agdmoper.o

     copia agdma4esc  $(PVM_ROOT)/bin/$(PVM_ARCH)
     mv agdma4esc $(PVM_ROOT)/bin/$(PVM_ARCH)
```

Para arrancar el cluster se deberá ejecutar la siguiente serie de comandos:

```
#cluster service iptables stop
#cluster init 3
#pvm cluster.lst
pvm> quit
```

El primer comando detiene el servicio de cortafuegos *iptables* en los esclavos.

El segundo comando pone a los esclavos en el nivel de ejecución (*runlevel*) 3 en el cual se ejecutan todos los servicios de red pero no el ambiente gráfico.

El tercer comando inicia las librerías de pvm y agrega las mismas a los nodos esclavos (que se encuentran listados en el archivo cluster.lst).

El cuarto comando permite salir de la terminal de pvm, pero se mantiene el cluster en funcionamiento.

Una vez que se ha terminado de trabajar con el cluster, hay que apagarlo mediante la ejecución de los siguientes comandos:

```
#pvm
pvm> halt
#cluster shutdown -h now
```

## 4.4 Programación de rutinas

Con base en el resultado del análisis realizado al programa original escrito en lenguaje C, se decidió programar en lenguaje ensamblador, esto es, aplicar optimización manual a las rutinas cuya estructura y participación en el algoritmo pueden proveer, una vez optimizadas, alguna mejora en la eficiencia en cuanto al tiempo de ejecución.

La selección de las rutinas a optimizar se realizó basándonos en dos criterios:

El primer criterio es el relativo a la cantidad de veces que se ejecuta la rutina. Por ejemplo, al ejecutar la aplicación se observa que la función de aptitud y la rutina de generación de números pseudo-aleatorios participan en varios cientos de millones de veces durante una corrida.

El segundo criterio es el relativo a la posibilidad de optimización de instrucciones y estructuras contenidas en las rutinas. Por ejemplo, reducir el acceso a variables de memoria mediante la utilización de registros del CPU, implementar las instrucciones matemáticas de manera más eficiente utilizando de la mejor manera posible las instrucciones del coprocesador matemático, escribir las estructuras iterativas de manera que se ejecuten más rápido, etc.

En las siguientes secciones se presentan las estrategias de optimización aplicadas a cada una de las rutinas seleccionadas. Cabe mencionar que en la descripción de las rutinas, se presentan sólo las secciones de código relevantes. Para revisar el código fuente completo de cada uno de los módulos programados, pueden consultarse los apéndices A al E de este documento.

### 4.4.1 Funciones de aptitud

Como se mencionó, cada una de las funciones de aptitud se ejecuta varios cientos de millones de veces en cada corrida del algoritmo; por tanto, se espera que al optimizarlas se logre una reducción en el tiempo de ejecución. Cada función requiere diferente complejidad en cuanto a las instrucciones requeridas para su implementación, por lo que la optimización aplicada a cada una representará diferentes niveles de mejora en la eficiencia.

En este capítulo no se hace referencia a los niveles de optimización obtenidos con la aplicación de las estrategias, sino que se hace solamente una descripción de la programación de las rutinas y los puntos atacables en cada una de ellas. Para conocer los resultados de las pruebas y comparaciones de eficiencia, consulte el capítulo 5 (Resultados y trabajo futuro).

Enseguida se describe la optimización aplicada a cada una de las funciones. Antes cabe mencionar que, en el caso de las funciones de aptitud, en todas ellas está declarada la variable `evaluaciones`, que se utiliza para contar las veces que se ejecuta la función de aptitud y su valor final (en una corrida del proceso esclavo) es enviado al proceso maestro para ser incluido en los datos de los reportes. Es por esta razón que la variable `evaluaciones` estará fuera de consideración en el tema de optimización. En la

versión de lenguaje ensamblador la variable también estará declarada y su incremento se hace con la instrucción `inc dword [evaluaciones]`.

Es pertinente mencionar también que, dado que las funciones de aptitud reciben como parámetro la dirección de inicio del arreglo que representa al individuo a evaluar, el código en lenguaje ensamblador para las funciones de aptitud inician con las instrucciones `push ebp`, `mov ebp,esp` y `mov esi,[ebp+8]`, para apuntar el registro `ESI` al arreglo. Estas tres instrucciones están también fuera de consideración en la optimización.

**Ackley**

El código en lenguaje C para esta función (1) es el siguiente:

```
01: #define FUNCION "ackley"
02: #define PI 3.14159265358979323846264338327950 29
03: #define E  2.71828182845904523536028747135266 25
04: #define XMAX 32
05:
06: long evaluaciones = 0;
07:
08: double aptitud(double x[DIMENSIONES])
09: {
10:     int i;
11:     double sum1, sum2, res;
12:     evaluaciones++;
13:     sum1 = 0.0;
14:     sum2 = 0.0;
15:     for (i=0; i < DIMENSIONES; i++)
16:     {
17:         sum1 += x[i]*x[i];
18:         sum2 += cos(2.0*PI*x[i]);
```

```
19:     }
20:     sum1 = -0.2*sqrt(sum1/DIMENSIONES);
21:     sum2 /= DIMENSIONES;
22:     res = 20.0 + E - 20.0*exp(sum1) - exp(sum2);
23:     return (-res);
24: }
```

Está declarada la variable `i` para ser utilizada como contador del ciclo y como índice al arreglo que representa al individuo al que se le está calculando la aptitud. En la versión en lenguaje ensamblador se omitirá la declaración de esta variable y en su lugar se utilizará algún registro de la CPU. También están declaradas las variables `sum1` y `sum2` para ir acumulando los resultados de algunas operaciones y la variable `res` para almacenar el valor final de la aptitud, que será devuelto por la función.

En la versión optimizada puede omitirse la declaración de las tres variables. El caso de las variables `sum1` y `sum2` es el más crítico en esta función ya que se realizan cálculos con ellas y se almacenan datos en ellas dentro del ciclo (lineas 15 a la 19). Además, también dentro del ciclo se emplea el operando `x[i]`; por tanto, en cada iteración hay al menos seis accesos a memoria, lo que es muy costoso en términos del consumo de ciclos de la CPU.

En lugar de emplear las variables `sum1` y `sum2`, en la versión optimizada se puede utilizar la pila del coprocesador matemático (también conocido como Unidad de Punto Flotante o *Floating-Point Uint, FPU*) para contener los valores correspondientes y con eso evitar los accesos a memoria.

El caso de la variable `res` es más sencillo de resolver. Basta con dejar en el tope de la pila de la FPU el resultado final de la aptitud para que sea considerado como el valor devuelto por la función, ya que el compilador de C espera que los valores de punto flotante devueltos por una función estén en el tope de la pila de la FPU.

Una vez aplicada la optimización, el código en lenguaje ensamblador resultante para esta función es el siguiente:

```
01:;acley.inc
03:;Función 8, Pag 11 CEC05
04:
05: section .data
06:        Veinte  dd 20
07:  MenosPuntoDos dq -0.2
08:             E  dq 2.7182818284590452353602874713526625
09:
10: segment .text
11: Aptitud:
12:      push ebp
13:      mov ebp,esp
14:      mov esi,[ebp+8]             ;ESI = Apuntador al arreglo
15:
16:      inc dword [Evaluaciones]
17:
18:      fild dword [Veinte]          ;20
19:      fld ST0                      ;20, 20
20:      fld qword [E]                ;E, 20, 20
21:      faddp ST2,ST0                ;20, 20+E
22:
23:      ;Calcular sum1
24:      mov ecx,dword [Dimensiones]
25:      fldz                         ;0, 20, 20+E
26: CicloSuma1:
27:      jecxz Salir1
28:      dec ecx
29:
30:      fld qword [esi + ecx * 8]    ;x[i], 0, 20, 20+E
31:      fld ST0                      ;x[i], x[i], 0, 20, 20+E
32:      fmulp                        ;x[i]*x[i], 0, 20, 20+E
33:      faddp                        ;x[i]*x[i], 20, 20+E
34:                        ;--> en cada iteración va acumulando la suma
```

```
35:                                ;    sum1 += x[i]*x[i];
36:      jmp CicloSuma1
37: Salir1:
38:      fidiv dword [Dimensiones]    ;sum1/DIMENSIONES,20,20+E
39:      fsqrt                        ;sqrt(sum1/DIMENSIONES),20,20+E
40:      fmul qword [MenosPuntoDos]  ;-0.2*sqrt(sum1/DIMENSIONES),20,20+E
41:                                   ;sum1, 20, 20+E
42:      ;exp(sum1)
43:      ;e^sum1 = 2 ^ (sum1*Lg(E))
44:      ;nota: Lg es Logaritmo base 2
45:      fldl2e                       ;Lg(E), sum1, 20, 20+E
46:      fmulp                        ;sum1*Lg(E), 20, 20+E
47:      ;En los comentarios se escribirá 2^x la expresión x=sum1*Lg(E)
48:      fld ST0                      ;x, x, 20, 20+E
49:      frndint                      ;int(x), x, 20, 20+E
50:      fxch                         ;x, int(x), 20, 20+E
51:      fsub ST0,ST1                 ;frac(x), int(x), 20, 20+E
52:      f2xm1                        ;(2^(frac(x)))-1, int(x), 20, 20+E
53:      fld1                         ;1,(2^(frac(x)))-1,int(x),20,20+E
54:      faddp                        ;2^(frac(x)), int(x), 20, 20+E
55:      fscale                       ;2^x, int(x), 20, 20+E
56:      fstp ST1                     ;2^x, 20, 20+E
57:                                   ;exp(sum1), 20, 20+E
58:      fmulp                        ;exp(sum1)*20, 20+E
59:      fsubp                        ;20+E - 20*exp(sum1)
60:
61:      ;Calcular sum2
62:      mov ecx,dword [Dimensiones]
63:      fldz                         ;0, 20+E - 20*exp(sum1)
64: CicloSuma2:
65:      jecxz Salir2
66:      dec ecx
67:
68:      fldpi                        ;Pi, 0, 20+E - 20*exp(sum1)
69:      fmul qword [esi + ecx * 8]   ;Pi*x[i], 0, 20+E - 20*exp(sum1)
70:      fld ST0              ;Pi*x[i], Pi*x[i], 0, 20+E - 20*exp(sum1)
71:      faddp                      ;2.0*Pi*x[i], 0, 20+E - 20*exp(sum1)
72:      fcos                    ;cos(2.0*Pi*x[i]),0,20+E - 20*exp(sum1)
```

```
73:      faddp                      ;sum2, 20+E - 20*exp(sum1)
74:      jmp CicloSuma2
75: Salir2:
76:      fidiv dword [Dimensiones]  ;sum2/DIMENSIONES,20+E - 20*exp(sum1)
77:                                   ;sum2, 20+E - 20*exp(sum1)
78:      ;exp(sum2)
79:      ;e^sum2 = 2 ^ (sum2*Lg(E))
80:      fldl2e                       ;Lg(E), sum2, 20+E - 20*exp(sum1)
81:      fmulp                        ;sum2*Lg(E), 20+E - 20*exp(sum1)
82:      ;En los comentarios se escribirá 2^x la expresión x=sum1*Lg(E)
83:      fld ST0                      ;x, x, 20+E - 20*exp(sum1)
84:      frndint                      ;int(x), x, 20+E - 20*exp(sum1)
85:      fxch                         ;x, int(x), 20+E - 20*exp(sum1)
86:      fsub ST0,ST1            ;frac(x), int(x), 20+E - 20*exp(sum1)
87:      f2xm1             ;(2^(frac(x)))-1, int(x), 20+E - 20*exp(sum1)
88:      fld1           ;1, (2^(frac(x)))-1, int(x), 20+E - 20*exp(sum1)
89:      faddp                  ;2^(frac(x)), int(x), 20+E - 20*exp(sum1)
90:      fscale                 ;2^x, int(x), 20+E - 20*exp(sum1)
91:      fstp ST1               ;2^x, 20+E - 20*exp(sum1)
92:                             ;exp(sum2), 20+E - 20*exp(sum1)
93:      fsubp                  ;20+E - 20*exp(sum1) - exp(sum2)
94:      fchs                   ;-res
95:      leave
96:      ret
```

En el código original en C hay sólo un ciclo, en el que se acumulan los valores para las variables `sum1` y `sum2`, mientras que en esta versión optimizada hay dos ciclos. En el primero (`CicloSuma1`), se acumula el resultado para lo que sería la variable `sum1` y en el segundo (`CicloSuma2`), se acumula el resultado para lo que sería la variable `sum2`.

Como se comentó anteriormente, en lugar de emplear variables de memoria se utiliza la pila de la FPU para ir acumulando estos dos resultados. Esa es la razón por la que se implementan dos ciclos. Una vez finalizado el primero, el resultado para `sum1` queda

en el tope de la pila y antes de iniciar el segundo se calcula `20+E - 20*exp(sum1)`. Una vez terminado el segundo ciclo, el resultado para `sum2` queda en el tope de la pila y el resto de las instrucciones realizan el cálculo final de la aptitud `-(20+E - 20*exp(sum1) - exp(sum2))`.

Como puede observarse, en cada uno de los ciclos hay sólo una referencia a memoria por cada iteración (para cargar el operando `[esi+ecx*8]`con la instrucción `fld qword [esi+ecx*8]`). Este acceso a memoria no puede ser eliminado, ya que se utiliza para leer los genes del individuo al que se le está calculando la aptitud. Se ha utilizado el registro ECX como contador del ciclo y en cuanto al índice al arreglo, se ha empleado el registro ESI como base y el registro ECX como desplazamiento (multiplicándolo por 8, ya que cada gen del individuo es de 64 bits).

Aunque en este código se sacrifica memoria al declarar tres variables auxiliares (`Veinte, MenosPuntoDos y E`), esperamos que su utilización no influya significativamente en el tiempo de ejecución ya que el acceso a ellas se realiza fuera de los ciclos.

**Rastrigin**

El código en lenguaje C para esta función (2) es el siguiente:

```
01: #define FUNCION "rastrigin"
02: #define PI 3.14159265358979323846264338327950 29
03: #define E  2.71828182845904523536028747135266 25
04: #define XMAX 5     //Máximo valor para x
05:
06: long evaluaciones = 0;
07:
08: double aptitud(double x[DIMENSIONES])
```

```
09: {
10:     int i;
11:     double res;
12:     evaluaciones++;
13:     res = 0.0;
14:     for (i=0; i < DIMENSIONES; i++)
15:     {
16:         res += (x[i]*x[i] - 10.0 * cos(2.0*PI*x[i]) + 10.0);
17:     }
18:     return (-res);
19: }
```

Están declaradas las variables `i` y `res`. La variable `i` como contador del ciclo y como índice al arreglo que representa al individuo. La variable `res` se utiliza para acumular el valor de aptitud. Igual que en el caso de la función anterior, para optimizar esta función se eliminará la variable `i` y en su lugar se utilizará un registro de la CPU. También se eliminará la variable `res` y el valor de aptitud se dejará en el tope de la pila de la FPU.

La instrucción que está en el ciclo implica al menos cuatro accesos a memoria. Al eliminar la variable `res` se pueden reducir estos accesos; sin embargo, la constante 10 puede representar accesos adicionales a memoria al emplear la FPU.

El código en lenguaje ensamblador producto de la optimización de esta función es el siguiente:

```
01: ;rastrigin.inc
02:;Función 9, Pag 12 CEC05
03:
04: section .data
05:        Diez  dq 10.0
06:
07: segment .text
```

```
08: Aptitud:
09:      push ebp
10:      mov ebp,esp
11:      mov esi,[ebp+8]      ;ESI = Apuntador al arreglo
12:
13:      inc dword [Evaluaciones]
14:
15:      mov ecx,dword [Dimensiones]
16:      fldz                        ;ST0 = 0
17: CicloSuma:
18:      jecxz Salir
19:      dec ecx
20:
21:      fldpi                       ;Pi, 0
22:      fmul qword [esi + ecx * 8]  ;Pi*x[i], 0
23:      fadd ST0,ST0                ;2.0*Pi*x[i], 0
24:      fcos                        ;cos(2.0*Pi*x[i]), 0
25:      fmul qword [Diez]           ;10*cos(2.0*Pi*x[i]), 0
26:      fld qword [esi + ecx * 8]   ;x[i], 10*cos(2.0*Pi*x[i]), 0
27:      fmul ST0,ST0                ;x[i]*x[i], 10*cos(2.0*Pi*x[i]), 0
28:      fsubrp                      ;x[i]*x[i] - 10*cos(2.0*Pi*x[i]), 0
29:      fadd qword [Diez]      ;x[i]*x[i] - 10*cos(2.0*Pi*x[i]) + 10.0,0
30:      faddp                    ;x[i]*x[i] - 10*cos(2.0*Pi*x[i]) + 10.0
31:                               ;--> Va acumulando la suma
32:      jmp CicloSuma
33: Salir:
34:      fchs
35:      leave
36:      ret
```

Se ha utilizado el registro ECX como contador del ciclo y como desplazamiento en el índice al arreglo (el registro ESI se usa como base). En el ciclo hay cuatro accesos a memoria, dos accesos al arreglo del individuo y dos accesos a la variable auxiliar `Diez`.

El hecho de que se tengan cuatro accesos a memoria dentro del ciclo hace pensar que esta función puede optimizarse aún más. Y en efecto, el código puede reescribirse para

eliminar tres accesos a memoria y dejarlo con sólo uno. Esta segunda versión es la siguiente:

```
01: ;rastrigin2.inc
02: ;Función 9, Pag 12 CEC05
03:
04: section .data
05:        Diez  dw 10
06:
07: segment .text
08: Aptitud:
09:      push ebp
10:      mov ebp,esp
11:      mov esi,[ebp+8]           ;ESI = Apuntador al arreglo
12:
13:      inc dword [Evaluaciones]
14:
15:      mov ecx, dword [Dimensiones]
16:      fldz                      ;0
17:      fild dword [Diez]         ;10,0
18: CicloSuma:
19:      jecxz Salir
20:      dec ecx
21:
22:      fld qword [esi + ecx * 8] ;x[i],10,0
23:      fld ST0                   ;x[i],x[i],10,0
24:      fldpi                     ;pi,x[i],x[i],10,0
25:      fmulp                     ;pi*x[i],x[i],10,0
26:      fadd ST0,ST0              ;2.0*pi*x[i],x[i],10,0
27:      fcos                      ;cos(2.0*pi*x[i]),x[i],10,0
28:      fmul ST0,ST2              ;10*cos(2.0*pi*x[i]),x[i],10,0
29:      fld ST1                   ;x[i],10*cos(2.0*pi*x[i]),x[i],10,0
30:      fmulp ST2,ST0             ;10*cos(2.0*pi*x[i]),x[i]*x[i],10,0
31:      fsubp                     ;x[i]*x[i]-10*cos(2.0*pi*x[i]),10,0
32:      fadd ST0,ST1          ;x[i]*x[i]-10*cos(2.0*pi*x[i])+10.0,10,0
33:      faddp ST2,ST0 ;10,(x[i]*x[i]-10*cos(2.0*pi*x[i])+10.0)+Acumulado
34:                    ;--> Se va acumulando la suma (res)
35:      jmp CicloSuma
```

```
36: Salir:
37:      fxch                      ;res,10
38:      fchs                      ;-res,10
39:      ffree ST1                 ;-res
40:      leave
41:      ret
```

En esta segunda versión optimizada para la función de Rastrigin hay un solo acceso a memoria dentro del ciclo. En primera instancia puede pensarse que esta versión debería ser más rápida que la primera; sin embargo, al ejecutar las pruebas de comparación entre las dos versiones, nos damos cuenta que esta segunda versión es más lenta que la primera (están casi empatadas, pero la segunda versión siempre es más lenta). Esto significa que el hecho de eliminar accesos a memoria, no necesariamente se traduce en ganancias en velocidad de ejecución.

Dentro del ciclo de esta segunda versión hay dos instrucciones más para la FPU y al salir del mismo, hay dos instrucciones (`fxch` y `ffree`) adicionales. Estas instrucciones adicionales, más el tiempo de acceso a operandos de pila del coprocesador, limitan el tiempo de ejecución de esta función respecto a la versión anterior.

Se antoja entonces la reprogramación de una tercera versión de optimización, en la que se reduzcan los accesos a memoria del operando de genes del individuo (al arreglo). Esta tercera versión involucra un acceso al arreglo de genes y dos accesos al operando `Diez`. El código es el siguiente:

```
01: ;rastrigin3.inc
02: ;Función 9, Pag 12 CEC05
03:
04: section .data
05:        Diez  dq 10.0
06:
07: segment .text
```

```
08: Aptitud:
09:      push ebp
10:      mov ebp,esp
11:      mov esi,[ebp+8]      ;ESI = Apuntador al arreglo
12:
13:      inc dword [Evaluaciones]
14:
15:      mov ecx, dword [Dimensiones]
16:      fldz                      ;0
17: CicloSuma:
18:      jecxz Salir
19:      dec ecx
20:
21:      fld qword [esi + ecx * 8] ;x[i],0
22:      fldpi                     ;pi,x[i],0
23:      fmul ST0,ST1              ;pi*x[i],x[i],0
24:      fadd ST0,ST0              ;2.0*pi*x[i],x[i],0
25:      fcos                      ;cos(2.0*pi*x[i]),x[i],0
26:      fmul qword [Diez]         ;10*cos(2.0*pi*x[i]),x[i],0
27:      fld ST1                   ;x[i],10*cos(2.0*pi*x[i]),x[i],0
28:      fmulp ST2,ST0             ;10*cos(2.0*pi*x[i]),x[i]*x[i],0
29:      fsubp                     ;x[i]*x[i]-10*cos(2.0*pi*x[i]),0
30:      fadd qword [Diez]         ;x[i]*x[i]-10*cos(2.0*pi*x[i])+10.0,0
31:      faddp          ;(x[i]*x[i]-10*cos(2.0*pi*x[i])+10.0)+Acumulado
32:                     ;--> Se va acumulando la suma (res)
33:      jmp CicloSuma
34: Salir:
35:      fchs                      ;-res
36:      leave
37:      ret
```

Esta tercera versión tiene una instrucción más dentro del ciclo que la primera. Fuera del ciclo tienen las mismas instrucciones.

Al comparar la eficiencia de la primera versión contra la tercera, se aprecia que hay una ganancia en el tiempo de ejecución de esta última versión. Esto se explica por el uso

que la CPU hace de la memoria caché. Puede observarse que el operando de variable llamado `Diez`, se emplea dos veces en cada iteración del ciclo, sin embargo, su uso no implica accesos adicionales a la RAM ya que es una variable que no cambia su valor; por tanto, es un operando que reside en la memoria caché, lo que contribuye a la disminución del tiempo de acceso a ese dato.

Esta tercera versión es la que elegiremos para competir con la versión original escrita en C. Veremos en el capítulo 5 (Resultados y trabajo futuro) si este código es más rápido que el generado por el compilador.

**Schwefel**

El código en lenguaje C para esta función (3) es el siguiente:

```
01: #define FUNCION "schwefel"
02: #define PI 3.14159265358979323846264338327950 29
03: #define E  2.71828182845904523536028747135266 25
04:
05: #define XMAX 100
06:
07: long evaluaciones = 0;
08: double aptitud(double x[DIMENSIONES])
09: {
10:     int i, j;
11:     long double sum1, sum2;
12:     sum1 = 0.0;
13:     evaluaciones++;
14:     for (i=0; i < DIMENSIONES; i++)
15:     {
16:         sum2 = 0.0;
17:         for (j=0; j<=i; j++)
18:         {
19:             sum2 += x[j];
20:         }
```

```
21:         sum1 += sum2*sum2;
22:     }
23:     return (-sum1);
24: }
```

Este código implementa fielmente la definición de la función de schwefel. Así como en la definición de la función hay una sumatoria anidada a otra externa, en la implementación hay un ciclo anidado a otro externo. Sin embargo, al analizar detenidamente el código, podemos darnos cuenta que en realidad no se requiere el ciclo interno para implementar la función. El código puede ser modificado de manera tal que sin tener ciclos anidados se implemente la función para que arroje los resultados correctos. Por tanto, un primer paso de optimización de código puede ser realizado en lenguaje C.

Enseguida se presenta el código en C con la primera optimización realizada.

```
01: #define FUNCION "schwefel"
02: #define PI 3.14159265358979323846264338327950 29
03: #define E  2.71828182845904523536028747135266 25
04: #define XMAX 100
05:
06: long evaluaciones = 0;
07: double aptitud(double x[DIMENSIONES])
08: {
09:    int i;
10:    long double sum1=0.0, sum2=0.0;
11:    evaluaciones++;
12:    for (i=0; i<DIMENSIONES; i++)
13:    {
14:        sum2 += x[i];
15:        sum1 += sum2*sum2;
16:    }
17:    return (-sum1);
18: }
```

Puede verse que se ha eliminado el ciclo interno y se han reorganizado ciertas instrucciones relativas a la variable `sum2` para que la implementación refleje lo que la función indica.

Una vez optimizada la versión en C, puede ser escrita la versión en lenguaje ensamblador, tratando de optimizar aún más el tiempo de ejecución.

El código en lenguaje ensamblador para esta función es el siguiente:

```
01: ;schwefel.inc
02: ;Función 2, Pag 5
03:
04: segment .text
05: Aptitud:
06:      push ebp
07:      mov ebp,esp
08:      mov esi,[ebp+8]              ;ESI = Apuntador al arreglo
09:
10:      inc dword [Evaluaciones]
11:
12:      mov ecx,dword [Dimensiones]
13:      xor eax,eax                  ;i = 0
14:      fldz                         ;sum2=0.0
15:      fldz                         ;sum1=0.0,sum2
16: CicloSuma:
17:      fld qword [esi + eax * 8]    ;x[i],sum1,sum2
18:      faddp ST2,ST0                ;sum1,sum2+=x[i]
19:      fld ST1                      ;sum2,sum1,sum2
20:      fmul ST0,ST0                 ;sum2*sum2,sum1,sum2
21:      faddp                        ;sum1+=sum2*sum2,sum2
22:      inc eax
23:      cmp eax,ecx
```

```
24:      jl CicloSuma
25:      ffree ST1                    ;sum
26:      fchs                         ;-sum1
27:      leave
28:      ret
```

En esta versión optimizada se ha descartado el uso de las variables `i`, `sum1` y `sum2`. Se utiliza el registro `ECX` como contador del ciclo, el registro `EAX` como índice al arreglo y para almacenar los datos de `sum1` y `sum2` se emplea la pila del coprocesador.

La función es muy sencilla, por lo que incluye un reducido número de instrucciones. Veremos si la optimización representa ahorro en el tiempo de ejecución.

**Weierstrass**

El código fuente en lenguaje C para esta función (4) es el siguiente:

```
01: #define FUNCION "weierstrass"
02:
03: #define PI 3.14159265358979323846264338327950 29
04: #define E  2.71828182845904523536028747135266 25
05: #define XMAX 1
06:
07: long evaluaciones = 0;
08: double aptitud(double x[DIMENSIONES])
09: {
10:     int i, j;
11:     double res;
12:     double sum;
13:     double a, b;
14:     int k_max;
15:     evaluaciones++;
16:     a = 0.5;
```

```
17:     b = 3.0;
18:     k_max = 20;
19:     res = 0.0;
20:     for (i=0; i<DIMENSIONES; i++)
21:     {
22:         sum = 0.0;
23:         for (j=0; j<=k_max; j++)
24:         {
25:             sum += pow(a,j)*cos(2.0*PI*pow(b,j)*(x[i]+0.5));
26:         }
27:         res += sum;
28:     }
29:     sum = 0.0;
30:     for (j=0; j<=k_max; j++)
31:     {
32:         sum += pow(a,j)*cos(2.0*PI*pow(b,j)*(0.5));
33:     }
34:     res-=DIMENSIONES*sum;
35:     return (-res);
36: }
```

Esta función es la más compleja de las cuatro evaluadas. Hay un ciclo anidado (en el que se calcula la variable `sum`) dentro de otro en el que se acumulan valores para la variable `res`. También hay otro ciclo en el que se acumulan valores para la variable `sum`, que posteriormente es acumulada como resultado final en la variable `res`.

Esta implementación contiene instrucciones muy interesantes y susceptibles a ser optimizadas de manera significativa, sobre todo considerando que hay cuatro llamadas a la función pow() realizadas dentro de ciclos.

Al analizar el código podemos darnos cuenta que se pueden eliminar algunas llamadas a la función pow(). Esto es, se pueden sacar de los ciclos las llamadas a la función para los casos en que la potencia es 0 y 1. Entonces, una primera optimización puede realizarse sobre el código fuente en C. Esta versión optimizada es la siguiente:

```
01: #define FUNCION "weierstrass"
02:
03: #define PI 3.14159265358979323846264338327950 29
04: #define E  2.71828182845904523536028747135266 25
05:
06: #define XMAX 0.5
07:
08: long evaluaciones = 0;
09: //funcion de aptitud
10: double aptitud(double x[DIMENSIONES])
11: {
12:     int i, j;
13:     double res;
14:     double sum;
15:
16:     evaluaciones++;
17:     res = 0.0;
18:     for (i=0; i < DIMENSIONES; i++)
19:     {
20:         //Fuera del ciclo calcular potencias para j=0 y j=1 evitando
21:         //dos llamadas a la función pow()
22:         sum = cos(2.0*PI*(x[i]+0.5)) + 0.5 * cos(6.0*PI*(x[i]+0.5));
23:         for (j=2; j <= 20; j++)
24:         {
25:             sum += pow(0.5,j)*cos(2.0*PI*pow(3,j)*(x[i]+0.5));
26:         }
27:         res += sum;
28:     }
29:     //Fuera del ciclo calcular potencias para j=0 y j=1 evitando
30:     //dos llamadas a la función pow()
31:     sum = cos(PI) + 0.5 * cos(3.0*PI);
32:     for (j=2; j <= 20; j++)
33:     {
34:         sum += pow(0.5,j) * cos(2.0*PI*pow(3,j)*0.5);
35:     }
36:     res-=DIMENSIONES*sum;
37:     return (-res);
```

```
38: }
```

En esta versión optimizada en C puede verse que los ciclos inician con el contador j en 2. Por tanto, se están eliminando cuatro llamadas a la función pow().

Después de realizar las ejecuciones de comparación de eficiencia entre las dos versiones en C, resultó que la segunda versión es más rápida que la primera en alrededor de 5%. Por lo tanto, para la optimización en lenguaje ensamblador consideraremos a la segunda versión en C.

En la versión optimizada escrita en lenguaje ensamblador también se realizarán los cálculos de las potencias para 0 y 1 fuera de los ciclos. Además, los cálculos de potencia que están dentro de los ciclos también se optimizarán debido a que, si bien el coprocesador tiene instrucciones para la elevación de una base a cualquier potencia, en el caso de esta función se pueden utilizar instrucciones alternativas de manera tal que la operación de potencia resulte más rápida.

El código optimizado en lenguaje ensamblador para esta función es el siguiente:

```
001: ;weierstrass.inc
002: ;Función 11, Pag 14 CEC05
003:
004: section .data
005:    a dq 0.5
006:    b dq 3.0
007:
008: segment .text
009: Aptitud:
010:      push ebp
011:      mov ebp,esp
012:      mov esi,[ebp+8]      ;ESI = Apuntador al arreglo
013:
014:      inc dword [Evaluaciones]
```

```
015:
016:      fldz                          ;res=0
017:      mov ecx,dword [Dimensiones]
018:      xor eax,eax
019: CicloDim:
020:   ;Fuera del ciclo el cálculo de para j=0 y j=1 evitando los pow()
021:   ;sum = cos(2.0*PI*(x[i]+0.5)) + 0.5 * cos(6.0*PI*(x[i]+0.5));
022:
023:      ;Para j=0: sum = cos(2.0*PI*(x[i]+0.5))
024:      fldz
025:      fld qword [esi + eax*8]
026:      fld qword [a]
027:      fadd ST1,ST0
028:      fldpi
029:      fadd ST0,ST0
030:      fmul ST0,ST2
031:      fcos
032:      faddp ST3,ST0
033:
034:      ;Para j=1: sum+= 0.5 * cos(6.0*PI*(x[i]+0.5));
035:      fld qword [b]
036:      fldpi
037:      fmul ST0,ST3
038:      fld ST1
039:      fadd ST0,ST0
040:      fmulp
041:      fcos
042:      fmul ST0,ST2
043:      ;Para j=0 y j=1: sum = cos(2.0*PI*(x[i]+0.5)) + 0.5 *
044:      ;cos(6.0*PI*(x[i]+0.5));
045:      faddp ST4,ST0
046:      fld ST0
047:
048:      ;En la pila quedan preparados los valores para ir
049:      ;calculando las las potencias:
050:      ; b, pow(b,j), pow(a,j), x[i]+0.5, sum, res
051:
052:      mov ebx,2
```

```
053: CicloK1:
054:      ;iterativamente: sum +=
055:      ;pow(a,j)*cos(2.0*PI*pow(b,j)*(x[i]+0.5));
056:      fmul ST1,ST0
057:      fld qword [a]
058:      fmulp ST3,ST0
059:      fldpi
060:      fadd ST0,ST0
061:      fmul ST0,ST2
062:      fmul ST0,ST4
063:      fcos
064:      fmul ST0,ST3
065:      faddp ST5,ST0
066:      inc bx
067:      cmp bx,21
068:      jl CicloK1    ;for (j=0; j <= k_max; j++)
069:
070:    ;Ajustar la pila para la siguiente iteración a CicloDim
071:    ;Actualmente la pila contiene: ;b, pow(b,j), pow(a,j), x[i]+0.5,
072:      ;sum, res, libre, libre
073:      fld ST5
074:      fadd ST0,ST5
075:      ffree ST6
076:      ffree ST5
077:      ffree ST4
078:      ffree ST3
079:      ffree ST2
080:      ffree ST1
081:      inc eax
082:      cmp eax,ecx
083:      jl CicloDim     ;for (i=0; i<DIMENSIONES; i++)
084:
085:    ;Fuera del ciclo el cálculo de para j=0 y j=1 evitando los pow()
086:    ;sum = cos(PI) + 0.5 * cos(3.0*PI);
087:      fldz
088:      fld qword [b]
089:      fld qword [a]
090:      fld ST0
```

```
091:      fldpi
092:      fmul ST0,ST3
093:      fcos
094:      fmul ST0,ST1
095:      fldpi
096:      fcos
097:      faddp
098:      faddp ST4,ST0
099:
100:       mov ebx,2     ;j=2 ----  para el resto del ciclo for (j=0; j <=
101:                    ;k_max; j++)
102: CicloK2:
103:      ;Iterativamente: sum += pow(a,j) * cos(2.0*PI*pow(b,j)*0.5);
104:      fmul ST1,ST0
105:      fld qword [b]
106:      fmulp ST3,ST0
107:      fldpi
108:      fadd ST0,ST0
109:      fmul ST0,ST3
110:      fmul ST0,ST1
111:      fcos
112:      fmul ST0,ST2
113:      faddp ST4,ST0
114:      inc bx
115:      cmp bx,21
116:      jl CicloK2    ;for (j=0; j <= k_max; j++)
117:
118:      ;Ajustar la pila
119:      fld ST4
120:      fld ST4
121:      ffree ST6
122:      ffree ST5
123:      ffree ST4
124:      ffree ST3
125:      ffree ST2
126:      fild dword [Dimensiones]
127:      fmulp
128:      fsubp
```

```
129:
130:        fchs
131:        leave
132:        ret
```

En lugar de emplear la instrucción de potencia del coprocesador, en esta versión se emplea la instrucción de multiplicación (`fmul`, que es más rápida). La base se mantiene permanentemente en la pila y el exponente se acumula en cada vuelta del ciclo. Esto es, el resultado de la potencia anterior está en la pila, luego se carga la base y se multiplica por esa potencia anterior. El resultado de la multiplicación queda en la pila como la potencia calculada y en la siguiente vuelta es acumulada nuevamente.

En este código optimizado se utiliza prácticamente al máximo la pila de la FPU. Al salir de cada uno de los dos ciclos (etiquetas `CicloK1` y `CicloK2`) se tiene que emplear la instrucción `ffree` para liberar la pila, operación que es requerida por el compilador C de manera tal que en la siguiente llamada no se produzca un error de desbordamiento.

Aunque el código en esta versión optimizada con lenguaje ensamblador es extenso, confiamos que en las corridas de comparación de eficiencia resulte más rápida que la versión original en C.

### 4.4.2 Generador de números pseudo-aleatorios

Dadas las características de la aplicación AG-DM, se requiere de un generador de números pseudo-aleatorios confiable. Se emplea el algoritmo MT19937©, conocido como *Mersenne Twister*. Es un generador de números pseudo-aleatorios desarrollado por Matsumoto y Nishimura [20,21]. Está basado en una matriz de recurrencia lineal sobre un campo binario y provee una rápida generación de números pseudo-aleatorios

de muy alta calidad. Su nombre deriva del hecho de que la longitud de periodo se elige de un número primo de Mersenne, esa longitud de periodo es de $2^{19937} - 1$.

Este generador está incluido en la biblioteca científica del proyecto GNU (*GSL, GNU Scientific Library*), que provee un gran número de rutinas matemáticas y es software libre bajo la licencia pública general de GNU [22].

Las tres rutinas del generador de números pseudo-aleatorios consideradas para optimización en el presente proyecto son las siguientes:

***gsl_rng_set(const gsl_rng * r)***. Esta función establece la semilla para el generador.

***gsl_rng_uniform*(*const gsl_rng * r*)**. Esta función devuelve un número pseudo-aleatorio entero de 32 bits sin signo.

***gsl_rng_uniform_pos(const gsl_rng * r)***. Esta función devuelve un número pseudo-aleatorio positivo de punto flotante de doble precisión, uniformemente distribuido en el intervalo (0,1). Esto es, excluye tanto el valor 0.0 como el valor 1.0.

Cabe mencionar que, aunque para realizar las llamadas al generador en la biblioteca GSL se emplean los nombres previamente descritos, en la implementación del algoritmo los nombres originales de las funciones son: `mt_set()`, `mt_get()` y `mt_get_double()` respectivamente.

En la versión en lenguaje ensamblador se identifican las funciones con los nombres empleados para las llamadas a GSL y se definen como globales a la aplicación:

```
global gsl_rng_uniform
global gsl_rng_uniform_pos
```

```
global gsl_rng_set
```

El generador requiere de la definición y declaración de las siguientes constantes, variables y estructuras:

```
01: #define N 624   /* Period parameters */
02: #define M 397
03:
04: /* most significant w-r bits */
05: static const unsigned long UPPER_MASK = 0x80000000UL;
06:
07: /* least significant r bits */
08: static const unsigned long LOWER_MASK = 0x7fffffffUL;
09:
10: typedef struct
11: {
12:     unsigned long mt[N];
13:     int mti;
14: }
15: mt_state_t;
```

La definición de esas estructuras en la versión en lenguaje ensamblador se realiza de la siguiente manera:

```
01: %define MASCARA_SUP 80000000h     ;Bits w-r mas significativos
02: %define MASCARA_INF 7fffffffh     ;Bits r menos significativos
03:
04: section .data
05:     DivisorDouble dq 4294967296.0
06:
07: section .bss
08:      mt resd 624
09:      mti resd 1
10:      ValorUniform resd 1
```

En la versión optimizada no existen las constantes N y M ya que se emplean esos valores como operandos inmediatos. Se define una variable de 64 bits (`DivisorDouble`), requerida por el generador. Además se reserva memoria para variables no inicializadas: `mt` es un arreglo de 624 valores de 32 bits, `mti` es un valor de 32 bits (utilizado como índice al arreglo) y `ValorUniform` es un valor también de 32 bits.

A continuación se presenta el código en lenguaje C de cada función, así como el código en lenguaje ensamblador correspondiente a la optimización manual aplicada a cada una de ellas.

**gsl_rng_set(*const gsl_rng * r*)**

El código en C para esta función es el siguiente:

```
01: static void mt_set (void *vstate, unsigned long int s)
02: {
03:     mt_state_t *state = (mt_state_t *) vstate;
04:    int i;
05:
06:     if (s == 0)
07:         s = 4357;   /* the default seed is 4357 */
08:     state->mt[0]= s & 0xffffffffUL;
09:     for (i = 1; i < N; i++)
10:     {
11:         /* See Knuth's "Art of Computer Programming" Vol. 2, 3rd
12:          Ed. p.106 for multiplier. */
13:         state->mt[i] = (1812433253UL * (state->mt[i-1] ^ (state->mt[i-1]
                                                                >> 30)) + i);
14:         state->mt[i] &= 0xffffffffUL;
15:     }
```

```
16:     state->mti = i;
17: }
```

Los puntos atacables para optimización de esta función son: la variable `s`, que es el valor para la semilla; la variable `i`, que es el contador del ciclo; pero sobre todo, el uso de apuntadores al arreglo `mt` y a la variable `mti`. Estos apuntadores son requeridos por la biblioteca GSL, debido a que incluye una gran cantidad de generadores de números pseudo-aleatorios y tienen la finalidad de proveer uniformidad para realizar las llamadas a las funciones de cada generador.

La versión optimizada se muestra a continuación.

```
01: gsl_rng_set:
02:      push ebp
03:      mov ebp,esp
04:      mov edx, dword [ebp+8]       ;EDX = Semilla
05:
06:      ;if (s == 0) s = 4357;   /* the default seed is 4357 */
07:      cmp edx,0
08:      jne SigueSet
09:      mov edx,4357
10: SigueSet:
11:      ;state->mt[0]= s & 0xffffffffUL;
12:      ;and edx,0ffffffffh     ;**** usado solo en cpu x86-64
13:      mov dword [mt],edx
14:
15:      ;for (i = 1; i < N; i++)
16:      mov esi,1
17: CicloSet:
18:      ; state->mt[i] = (1812433253UL * (state->mt[i-1] ^ (state->mt[i-1]
19:      ;>> 30)) + i);
20:      mov ebx,[mt - 4 + esi * 4]
21:      mov eax,ebx
22:      shr ebx,30
23:      xor eax,ebx
```

```
24:      mov ebx,1812433253
25:      mul ebx
26:      add eax,esi
27:
28:      ; state->mt[i] &= 0xffffffffUL;
29:      ;and eax,0ffffffffh         ;**** usado solo en cpu x86-64
30:      mov [mt + esi * 4],eax
31:      inc esi
32:      cmp esi,624
33:      jl CicloSet
34:
35:      ;state->mti = i
36:      mov dword [mti],624  ;esi
37:      leave
38:     ret
```

En lugar de usar la variable `s`, se emplea el registro EDX. El contador del ciclo es el registro ESI y en el caso de los apuntadores, simplemente no se utilizan. Se leen y almacenan datos en las variables `mt` y `mti` directamente.

**gsl_rng_uniform(*const gsl_rng * r*)**

```
01: static inline unsigned long mt_get (void *vstate)
02: {
03:    mt_state_t *state = (mt_state_t *) vstate;
04:    unsigned long k ;
05:    unsigned long int *const mt = state->mt;
06:    #define MAGIC(y) (((y)&0x1) ? 0x9908b0dfUL : 0)
07:    if (state->mti >= N)
08:    {   /* generate N words at one time */
09:        int kk;
10:        for (kk = 0; kk < N - M; kk++)
11:        {
```

```
12:            unsigned long y = (mt[kk] & UPPER_MASK) | (mt[kk + 1] &
13:                                LOWER_MASK);
14:            mt[kk] = mt[kk + M] ^ (y >> 1) ^ MAGIC(y);
15:         }
16:         for (; kk < N - 1; kk++)
17:         {
18:            unsigned long y = (mt[kk] & UPPER_MASK) | (mt[kk + 1] &
19:                               LOWER_MASK);
20:            mt[kk] = mt[kk + (M - N)] ^ (y >> 1) ^ MAGIC(y);
21:         }
22:
23:         {
24:            unsigned long y = (mt[N - 1] & UPPER_MASK) | (mt[0] &
25:                                LOWER_MASK);
26:            mt[N - 1] = mt[M - 1] ^ (y >> 1) ^ MAGIC(y);
27:         }
28:         state->mti = 0;
29:     }
30:     /* Tempering */
31:     k = mt[state->mti];
32:     k ^= (k >> 11);
33:     k ^= (k << 7) & 0x9d2c5680UL;
34:     k ^= (k << 15) & 0xefc60000UL;
35:     k ^= (k >> 18);
36:     state->mti++;
37:     return k;
38: }
```

La optimización para esta función también tratará sobre la eliminación del uso de apuntadores y de variables contadores en ciclos. También se usarán operandos inmediatos en lugar de las constantes N y M. Además, se aprecia la definición de una función de operador unario llamado MAGIC, la cual puede ser optimizada también. La versión en lenguaje ensamblador es la siguiente:

```
001: gsl_rng_uniform:
002:      cmp dword [mti],624
```

```
003:      jl Tempering
004:
005:      ;Generar 624 words
006:      ;for (kk = 0; kk < N - M; kk++)
007:      xor esi,esi
008: CicloN_M:
009:      ;unsigned long y = (mt[kk] & UPPER_MASK) | (mt[kk + 1] &
010:      ;LOWER_MASK);
011:      mov eax,[mt + esi * 4]
012:      mov ebx,[mt + 4 + esi * 4]
013:      and eax,MASCARA_SUP
014:      and ebx,MASCARA_INF
015:      or eax,ebx            ;eax es y
016:      ;mt[kk] = mt[kk + M] ^ (y >> 1) ^ MAGIC(y);
017:      mov ebx,eax
018:      and ebx,1
019:      neg ebx
020:      and ebx,09908b0dfh
021:      shr eax,1
022:      xor eax,ebx
023:      xor eax,[mt + 1588 + esi * 4]   ;1588 = M * 4  (397*4)
024:
025:      mov [mt + esi * 4],eax
026:
027:      inc esi
028:      cmp esi,227
029:      jl CicloN_M
030:
031:      ;for (; kk < N - 1; kk++)
032: CicloN_1:
033:      ;unsigned long y = (mt[kk] & UPPER_MASK) | (mt[kk + 1] &
034:      ;LOWER_MASK);
035:      mov eax,[mt + esi * 4]
036:      mov ebx,[mt + 4 + esi * 4]
037:      and eax,MASCARA_SUP
038:      and ebx,MASCARA_INF
039:      or eax,ebx            ;eax es y
040:      ;mt[kk] = mt[kk + (M - N)] ^ (y >> 1) ^ MAGIC(y);
```

```
041:      mov ebx,eax
042:      shr eax,1
043:      and ebx,1
044:      neg ebx
045:      and ebx,09908b0dfh
046:      xor eax,ebx
047:      mov ebx,[mt - 908 + esi * 4]   ;908 = (M - N) * 4=(397 - 624) * 4
048:      xor eax,ebx
049:
050:      mov [mt + esi * 4],eax
051:
052:      inc esi
053:      cmp esi,623
054:      jl CicloN_1
055:
056:    ;unsigned long y = (mt[N - 1] & UPPER_MASK) | (mt[0] & LOWER_MASK);
057:      mov eax,[mt + 2492]
058:      mov ebx,[mt]
059:      and eax,MASCARA_SUP
060:      and ebx,MASCARA_INF
061:      or eax,ebx            ;eax es y
062:      ;mt[N - 1] = mt[M - 1] ^ (y >> 1) ^ MAGIC(y);
063:      mov ebx,eax
064:      shr eax,1
065:      and ebx,1
066:      neg ebx
067:      and ebx,09908b0dfh
068:      xor eax,ebx
069:      mov ebx,[mt + 1584]
070:      xor eax,ebx
071:      mov [mt + 2492],eax
072:
073:      ;state->mti = 0
074:      mov dword [mti],0
075: Tempering:
076:      ;k = mt[state->mti]
077:      mov ecx,[mti]
078:      mov ebx,[mt + ecx * 4]
```

```
079:      ;k ^= (k >> 11)
080:      mov eax,ebx
081:      shr ebx,11
082:      xor eax,ebx
083:      ;k ^= (k << 7) & 0x9d2c5680UL
084:      mov ebx,eax
085:      shl ebx,7
086:      and ebx,09d2c5680h
087:      xor eax,ebx
088:      ;k ^= (k << 15) & 0xefc60000UL
089:      mov ebx,eax
090:      shl ebx,15
091:      and ebx,0efc60000h
092:      xor eax,ebx
093:      ;k ^= (k >> 18)
094:      mov ebx,eax
095:      shr ebx,18
096:      xor eax,ebx
097:
098:      ;state->mti++
099:      inc dword [mti]
100:      ret
```

En esta versión se usa el registro ESI como contador en los ciclos y en lugar de emplear constantes definidas, se emplean operandos inmediatos. Respecto a la función MAGIC, ésta se implementa con tres instrucciones, que se emplean en tres puntos de la rutina (lineas 18..20, 43..45 y 65..67):

```
and ebx,1
neg ebx
and ebx,09908b0dfh
```

Donde el registro EBX se emplea en lugar de la variable `y`.

El número pseudo-aleatorio generado queda almacenado en el registro EAX, ya que así lo exige el compilador de C para el caso de valores enteros que deben ser devueltos por funciones.

**gsl_rng_uniform_pos(*const gsl_rng * r*)**

```
1: static double
2: mt_get_double (void * vstate)
3: {
4:   return mt_get (vstate) / 4294967296.0 ;
5: }
```

Esta function consiste simplemente en llamar a la función `mt_get()`, esto es, a la función `gsl_rng_uniform()` de GSL y el valor devuelto es dividido entre un valor inmediato. La versión optimizada es la siguiente:

```
001: gsl_rng_uniform_pos:
002:      pushad
003:      cmp dword [mti],624
004:      jl Tempering_pos
005:
006:      ;Generar 624 words
007:      ;for (kk = 0; kk < N - M; kk++)
008:      xor esi,esi
009: CicloN_M_pos:
010:      ;unsigned long y = (mt[kk] & UPPER_MASK) | (mt[kk + 1] &
011:      ;LOWER_MASK);
012:      mov eax,[mt + esi * 4]
013:      mov ebx,[mt + 4 + esi * 4]
014:      and eax,MASCARA_SUP
015:      and ebx,MASCARA_INF
016:      or eax,ebx            ;eax es y
017:      ;mt[kk] = mt[kk + M] ^ (y >> 1) ^ MAGIC(y);
```

```
018:      mov ebx,eax
019:      and ebx,1
020:      neg ebx
021:      and ebx,09908b0dfh
022:      shr eax,1
023:      xor eax,ebx
024:      xor eax,[mt + 1588 + esi * 4]   ;1588 = M * 4  (397*4)
025:
026:      mov [mt + esi * 4],eax
027:
028:      inc esi
029:      cmp esi,227
030:      jl CicloN_M_pos
031:
032:      ;for (; kk < N - 1; kk++)
033: CicloN_1_pos:
034:     ;unsigned long y = (mt[kk] & UPPER_MASK) | (mt[kk + 1] &
035:     ;LOWER_MASK);
036:     mov eax,[mt + esi * 4]
037:     mov ebx,[mt + 4 + esi * 4]
038:     and eax,MASCARA_SUP
039:     and ebx,MASCARA_INF
040:     or eax,ebx            ;eax es y
041:     ;mt[kk] = mt[kk + (M - N)] ^ (y >> 1) ^ MAGIC(y);
042:     mov ebx,eax
043:     shr eax,1
044:     and ebx,1
045:     neg ebx
046:     and ebx,09908b0dfh
047:     xor eax,ebx
048:     mov ebx,[mt - 908 + esi * 4]   ;908 = (M - N) * 4 =  (397 - 624) *
04
049:     xor eax,ebx
050:
051:     mov [mt + esi * 4],eax
052:
053:      inc esi
054:      cmp esi,623
```

```
055:      jl CicloN_1_pos
056:
057:     ;unsigned long y = (mt[N - 1] & UPPER_MASK) | (mt[0] & LOWER_MASK);
058:      mov eax,[mt + 2492]
059:      mov ebx,[mt]
060:      and eax,MASCARA_SUP
061:      and ebx,MASCARA_INF
062:      or eax,ebx            ;eax es y
063:      ;mt[N - 1] = mt[M - 1] ^ (y >> 1) ^ MAGIC(y);
064:      mov ebx,eax
065:      shr eax,1
066:      and ebx,1
067:      neg ebx
068:      and ebx,09908b0dfh
069:      xor eax,ebx
070:      mov ebx,[mt + 1584]
071:      xor eax,ebx
072:      mov [mt + 2492],eax
073:
074:      ;state->mti = 0
075:      mov dword [mti],0
076:
077: Tempering_pos:
078:      ;k = mt[state->mti]
079:      mov ecx,[mti]
080:      mov ebx,[mt + ecx * 4]
081:      ;k ^= (k >> 11)
082:      mov eax,ebx
083:      shr ebx,11
084:      xor eax,ebx
085:      ;k ^= (k << 7) & 0x9d2c5680UL
086:      mov ebx,eax
087:      shl ebx,7
088:      and ebx,09d2c5680h
089:      xor eax,ebx
090:      ;k ^= (k << 15) & 0xefc60000UL
091:      mov ebx,eax
092:      shl ebx,15
```

```
093:      and ebx,0efc60000h
094:      xor eax,ebx
095:      ;k ^= (k >> 18)
096:      mov ebx,eax
097:      shr ebx,18
098:      xor eax,ebx
099:
100:      ;state->mti++
101:      inc dword [mti]
102:
103:      mov [ValorUniform],eax
104:      fild qword [ValorUniform]
105:      fdiv qword [DivisorDouble]
106:      popad
107:      ret
```

En esta versión se repite todo el código de la función *gsl_rng_uniform()* con el fin de evitar las llamadas a ella, con lo que se ahorra el tiempo que el CPU dedicaría al almacenamiento en la pila de los descriptores de página y su posterior recuperación. Esto es, se están evitando los accesos a memoria implicados en las llamadas a la función.

Antes de finalizar, esta función divide el valor resultante de las instrucciones relativas al módulo *gsl_rng_uniform()* entre el valor `DivisorDouble`, quedando el resultado en el tope de la pila de la FPU, ya que así lo exige el compilador de C para el caso de valores de punto flotante que deben ser devueltos por funciones.

### 4.4.3 Operadores AG

Aunque en la aplicación AG-DM los operadores genéticos no implican una ejecución tan exhaustiva como las funciones de aptitud y el generador de números pseudo-aleatorios, su optimización puede proveer algún ahorro en el tiempo de

ejecución. La optimización aplicada a los tres operadores AG se describe a continuación.

**Seleccion()**

El código en C para esta función es el siguiente:

```
01: void seleccion(void)
02: {
03:    int i,j,k,l;
04:    //Copiar a psig los ELITISMO individuos con mejor aptitud
05:    for(i=0;i < ELITISMO;i++)
06:   {
07:      for(j=0;j < DIMENSIONES;j++)
08:         psig[i].x[j] = p[i].x[j];
09:      psig[i].ap = p[i].ap;
10:   }
11:
12:   //Selecciona por sorteo el resto de los individuos
13:   for(i=ELITISMO;i < TamPoblacion;i++)
14:   {
15:      //Elige aleatoriamente al individuo k
16:      k = UNI*TamPoblacion;
17:      //El individo k "competirá" con otros TamTorneo individuos
18:      //seleccionados aleatoriamente
19:      for(j=1;j < TamTorneo;j++)
20:      {
21:         l = UNI*TamPoblacion;
22:         if(p[k].ap < p[l].ap)
23:           k = l;
24:      }
25:      //En k queda el ganador del sorteo
26:      for(j=0;j < DIMENSIONES;j++)
27:         psig[i].x[j] = p[k].x[j];
28:      psig[i].ap = p[k].ap;
```

```
29:   }
30:   //Al final, en psig quedan los individuos seleccionados
31: } //void seleccion(void)
```

Esta función tiene dos ciclos principales (lineas 5..10 y 13..29). El primero es para copiar los individuos elitistas (que tienen mejor aptitud) a la siguiente generación. En un ciclo anidado (lineas 7 y 8) se copian los genes del individuo y luego se copia la aptitud (linea 9). En el segundo ciclo se copian a la siguiente generación individuos que son seleccionados aleatoriamente y por torneo. Se selecciona aleatoriamente un individuo y en un ciclo anidado se seleccionan aleatoriamente otros que compiten para ser seleccionados de acuerdo a su aptitud (lineas 16..24). El individuo seleccionado es copiado a la siguiente generación mediante otro ciclo anidado al segundo en cuestión (lineas 26..28).

Puede verse que en esta función se emplea el símbolo UNI. Este símbolo involucra una llamada al generador de números pseudo-aleatorios *gsl_rng_uniform_pos*(), ya que en el archivo fuente agdm.c se encuentra la directiva `#define UNI gsl_rng_uniform_pos(r)`. La función `Selección` realiza en los ciclos múltiples llamadas al generador (lineas 16 y 21), por tanto, la optimización realizada al generador contribuye a la optimización de este operador de selección.
La versión optimizada para este operador es la siguiente:

```
001: Seleccion:
002:      push ebp
003:      mov ebp,esp
004:      sub esp,100h         ;Espacio para uso local de la pila
005:      mov esi,[ebp+8]      ;ESI = Apuntador al inicio de la poblacion
006:      mov edi,[ebp+12]    ;EDI = Apuntador a una poblacion auxiliar (en
007:                           ;ésta quedarán los seleccionados)
008:
009:      ;Modificar el registro de control (word) para que la instrucción
010:      ;FISTP tome el valor truncado
```

```
011:      ;Esto es necesario ya que en C, al combinar int con double, los
012:      ;valores deben truncarse.
013:      fstcw word [AuxDW]       ;Leer el registro de control
014:      or word [AuxDW],0c00h    ;Poner en 1 los bits 10 y 11
015:     fldcw word [AuxDW]       ;Sobreescribir el registro de control
016:
017:      ;//Copiar a psig los ELITISMO individuos con mejor aptitud
018:      ;Calcular cantidad de doubles en los elitistas
019:      mov eax, dword [Dimensiones]
020:      inc eax              ;incluir la aptitud
021:      mul dword [Elitismo]
022:      shl eax,1            ;EAX=EAX*2
023:      mov ecx,eax          ;ECX = doubles en los elitistas * 2
024:      push edi
025:      push esi
026:      cld
027:      rep movsd
028:      pop esi       ;ESI=apuntador original al inicio de la poblacion
029:      pop ecx      ;ECX=EDI original al inicio de la poblacion auxiliar
030:
031:      ;//Selecciona por sorteo el resto de los individuos
032:      ;Bytes Por Individuo
033:      mov eax, dword [Dimensiones]
034:      inc eax          ;incluir la aptitud
035:      shl eax,3     ;EAX=EAX * 8 bytes en cada double
036:                    ;= Bytes por individuo
037:      push eax
038:      ;Calcular bytes totales en la poblacion
039:      ;EAX * TamPoblacion
040:      mul dword [TamPoblacion]
041:      ;EAX = Total de bytes en la poblacion
042:      add eax,ecx           ;EAX=limite para el ciclo Selecciona
043:      pop ebx               ;EBX = Bytes por individuo
044:      add edi,ebx
045:      dec edi          ;EDI apunta al último byte de la aptitud del
046:                       ;individuo Elitismo+1
047:
048: Selecciona:
```

```
049:      push eax          ;EAX=limite para el ciclo Selecciona
050:      push esi          ;ESI=apuntador al inicio de la poblacion
051:
052:      ;k = UNI*TamPoblacion;
053:      call gsl_rng_uniform_pos    ;UNI
054:      fimul dword [TamPoblacion]  ;k=UNI*TamPoblacion
055:      fabs                        ;Valor absoluto de k
056:      fistp dword [AuxDD]         ;
057:      mov eax,dword [AuxDD]
058:      mul ebx                     ;k*Bytes por individuo
059:      add esi,eax                 ;ESI apunta al individuo k
060:      add esi,ebx
061:      sub esi,8                 ;ESI apunta a la aptitud del individuo k
062:      ;termina k = UNI*TamPoblacion;
063:
064:      pop edx           ;EDX=apuntador al inicio de la poblacion
065:      push edx
066:
067:      mov ecx, dword [TamTorneo]
068:      dec ecx
069: CicloTorneo:
070:      push edx          ;EDX=apuntador al inicio de la poblacion
071:      ;l = UNI*TamPoblacion;
072:      call gsl_rng_uniform_pos    ;UNI
073:      fimul dword [TamPoblacion]  ;l=UNI*TamPoblacion
074:      fabs                        ;Valor absoluto de l
075:      fistp dword [AuxDD]         ;
076:      mov eax,dword [AuxDD]
077:      mul ebx                     ;l*Bytes por individuo
078:      pop edx
079:      add edx,eax                 ;EDX apunta al individuo l
080:      add edx,ebx
081:      sub edx,8                   ;EAX apunta a la aptitud del individuo
082:      ;termina l = UNI*TamPoblacion;
083:      fld qword [esi]             ;Carga p[k].ap
084:      fcomp qword [edx]           ;compara p[k].ap con p[l].ap
085:      fstsw ax                ;Carga en ax los bits de estado de la FPU
086:      sahf                       ;Guarda AH en el registro de banderas
```

```
087:      jnb FinIf2                  ;if(p[k].ap >= p[l].ap), salta
088:      mov esi,edx                 ;k = l
089:   FinIf2:
090:      loop CicloTorneo
091:
092:
093:      ;Copiar a psig el individuo ganador del sorteo
094:      add esi,7
095:      push edi
096:      std
097:      mov ecx, dword [Dimensiones]
098:      inc ecx
099:      shl ecx,1
100:      rep movsd
101:      pop edi
102:
103:      add edi,ebx     ;Avanzar EDI hacia el sig individuo
104:      pop esi
105:      pop eax
106:      cmp edi,eax
107:      jl Selecciona:
108:      leave
109:      ret
```

Al inicio de la rutina se reservan 256 bytes extras en la pila para uso local (linea 4). Esto es debido a que se requiere del resguardo y recuperación de valores de algunos registros de la CPU (con instrucciones push y pop), además de que también se realizan llamadas al generador de números pseudo-aleatorios (función `gsl_rng_uniform_pos`, con la instrucción call), que también requieren de espacio en la pila para el resguardo y recuperación de descriptores de página.

En esta versión, en lugar de implementar un ciclo para copiar los individuos elitistas a la siguiente generación, se utiliza la instrucción `rep movsd` (linea 27), que aunque no deja de ser una instrucción iterativa, es de bajo nivel y la CPU la ejecuta más rápido que un ciclo tradicional. Para el caso de la selección del resto de los individuos, se ha

implementado un ciclo (`Selecciona`) con otro anidado (`CicloTorneo`). Una vez seleccionado el individuo, éste es copiado a la siguiente generación utilizando también la instrucción iterativa `rep movsd` (linea 100).

**Cruce()**

```
01: void cruce(void)
02: {
03:    double NuevoX1,NuevoX2;
04:    int i,j,k,d;
05:    float lambda;
06:    lambda = UNI;  //Obtener el factor de ponderación
07:    for(i=0;i < CuantosCruzar;i++)
08:    {
09:       j=UNI*TamPoblacion;
10:       k=UNI*TamPoblacion;
11:       for(d=0;d < DIMENSIONES;d++)
12:       {
13:          NuevoX1 = psig[j].x[d]*lambda+psig[k].x[d]*(1-lambda);
14:          NuevoX2 = psig[j].x[d]*(1-lambda)+psig[k].x[d]*lambda;
15:          psig[j].x[d] = NuevoX1;
16:          psig[k].x[d] = NuevoX2;
17:       }
18:       psig[j].ap = aptitud(psig[j].x);
19:       psig[k].ap = aptitud(psig[k].x);
20:    }
21: } //void cruce(void)
```

Hay un ciclo principal (lineas 7..20) en el que se realizan tantos cruzamientos como indique la variable `CuantosCruzar`. Se seleccionan aleatoriamente dos individuos a ser cruzados (lineas 9 y 10) y en un ciclo anidado (lineas 11..17) se realiza el cruzamiento (modificando los genes). Posteriormente se realiza el cálculo de la aptitud a los individuos resultantes del cruzamiento.

Se realizan múltiples llamadas al generador de números pseudo-aleatorios (mediante el símbolo UNI). Considerando que el generador está optimizado con lenguaje ensamblador, se optimiza también la función de cruzamiento. Sobre todo las llamadas que se realizan dentro del ciclo principal.

También en el ciclo principal se realizan múltiples llamadas a la función de aptitud, que está implementada con ensamblador. Estas llamadas también proveen optimización a la función de cruce.

La versión optimizada se presenta a continuación:

```
001: Cruce:
002:      push ebp
003:      mov ebp,esp
004:      sub esp,100h         ;Espacio para uso local de la pila
005:      mov esi,[ebp+8]      ;ESI = Apuntador al arreglo de la poblacion
006:
007:      ;Modificar el registro de control (word) para que la instrucción
008:      ;FISTP tome el valor del tope truncado
009:      ;Esto es necesario ya que en C, al combinar int con double, los
010:      ;valores deben truncarse.
011:      fstcw word [AuxDW]       ;Leer el registro de control
012:      or word [AuxDW],0c00h    ;Poner en 1 los bits 10 y 11
013:      fldcw word [AuxDW]       ;Sobreescribir el registro de control
014:
015:      mov edi,esi     ;ESI y EDI apuntan a psig (a la poblacion)
016:
017:      mov eax, dword [Dimensiones]
018:      inc eax               ;incluir la aptitud
019:      shl eax,3             ;EAX=EAX * 8 bytes en cada double
020:                            ; = Bytes por individuo
021:      mov ebx,eax           ;EBX = Bytes por individuo
022:
023:      call gsl_rng_uniform_pos    ;UNI
```

```
024:      fstp qword [AuxDQ]          ;lambda = UNI;
025:      ;for(i=0;i < CuantosCruzar;i++)
026:      fld qword [CuantosCruzar]
027:      fistp dword [AuxDD]         ;poner CuantosCruzar a entero
028:
029:      ;for(i=0;i < CuantosCruzar;i++)
030:      mov ecx, dword [AuxDD]
031: CicloCuantosCruzar:
032:      push ecx
033:      push esi           ;Guardar apuntadores al inicio de la poblacion
034:      push edi           ;
035:      ;Calcular j=UNI*TamPoblacion
036:      call gsl_rng_uniform_pos    ;UNI
037:      fimul dword [TamPoblacion]  ;UNI*TamPoblacion
038:      fistp dword [AuxDD]         ;j=UNI*TamPoblacion
039:      mov eax,dword [AuxDD]
040:      mul ebx                     ;j*Bytes por individuo
041:      add esi,eax                 ;ESI apunta al individuo j
042:      ;Calcular k=UNI*TamPoblacion
043:      call gsl_rng_uniform_pos    ;UNI
044:      fimul dword [TamPoblacion]  ;UNI*TamPoblacion
045:      fistp dword [AuxDD]         ;k=UNI*TamPoblacion
046:      mov eax,dword [AuxDD]
047:      mul ebx                     ;k*Bytes por individuo
048:      add edi,eax                 ;EDI apunta al individuo k
049:
050:      push esi                    ;Guardar apuntador al individuo j
051:      push edi                    ;Guardar apuntador al individuo k
052:      mov ecx, dword [Dimensiones]
053: CicloNuevosX:
054:      ;NuevoX1 = psig[j].x[d]*lambda+psig[k].x[d]*(1-lambda);
055:      fld qword [AuxDQ]           ;lamda
056:      fmul qword [esi]            ;psig[j].x[d]*lambda
057:      fld1                        ;1, psig[j].x[d]*lambda
058:      fsub qword [AuxDQ]          ;1-lambda, psig[j].x[d]*lambda
059:      fmul qword [edi]     ;psig[k].x[d]*(1-lambda), psig[j].x[d]*lambda
060:      faddp     ;NuevoX1 = psig[k].x[d]*(1-lambda) + psig[j].x[d]*lambda
061:
```

```
062:      ;NuevoX2 = psig[j].x[d]*(1-lambda)+psig[k].x[d]*lambda;
063:      fld1                        ;1, NuevoX1
064:      fsub qword [AuxDQ]          ;1-lamda, NuevoX1
065:      fmul qword [esi]            ;psig[j].x[d]*(1-lambda), NuevoX1
066:      fld qword [AuxDQ]     ;lambda, psig[j].x[d]*(1-lambda), N   uevoX1
067:      fmul qword [edi]    ;psig[k].x[d]*lambda, psig[j].x[d]*(1-lambda),
068:                          ;NuevoX1
069:      faddp    ;NuevoX2 = psig[k].x[d]*lambda + psig[j].x[d]*(1-lambda),
070:               ;NuevoX1
071:      fstp qword [edi]            ;psig[k].x[d] = NuevoX2;
072:      fstp qword [esi]            ;psig[j].x[d] = NuevoX1;
073:
074:      add esi,8
075:      add edi,8
076:      loop CicloNuevosX
077:
078:      pop edi        ;Restaurar apuntador al individuo k
079:      pop esi        ;Restaurar apuntador al individuo j
080:
081:      ;psig[j].ap = Aptitud(psig[j].x);
082:      pushad
083:      push esi       ;Enviar parámetro apuntador al individuo j
084:      call Aptitud
085:      add esp,4
086:      popad
087:      add esi,ebx
088:      sub esi,8
089:      fstp qword [esi]            ;psig[j].ap = Aptitud(psig[j].x)
090:
091:      ;psig[k].ap = Aptitud(psig[k].x);
092:      pushad
093:      push edi       ;Enviar parámetro apuntador al individuo k
094:      call Aptitud
095:      add esp,4
096:      popad
097:      add edi,ebx
098:      sub edi,8
099:      fstp qword [edi]            ;psig[k].ap = Aptitud(psig[k].x)
```

```
100:
101:      pop edi           ;Restaurar apuntadores al inicio de la poblacion
102:      pop esi           ;
103:      pop ecx           ;Restaurar contador del ciclo
104:      dec ecx
105:      jecxz FinCicloCuantosCruzar
106:      jmp CicloCuantosCruzar
107: FinCicloCuantosCruzar:
108:      leave
109:      ret
```

También en este caso se reserva espacio extra en la pila para el resguardo y recuperación de valores almacenados en registros de la CPU y para el llamado y retorno del generador de números pseudo-aleatorios y de la función de aptitud.

El ciclo principal está implementado entre las etiquetas `CicloCuantosCruzar` y `FinCicloCuantosCruzar`, de la linea 31 a la 107, empleando el registro ECX como contador. El cruzamiento de los individuos se hace en un ciclo implementado con la estructura loop y la etiqueta `CicloNuevosX` (lineas 53..76) donde el contador es también el registro ECX.

Las llamadas al generador de pseudo-aleatorios se hace con la instrucción `call gsl_rng_uniform_pos` (lineas 23, 36 y 43) y respecto al cálculo de la aptitud de los nuevos individuos, éste se realiza mediante la instrucción `call Aptitud`, en las lineas 84 y 94. La optimización implementada en lenguaje ensamblador a estas dos funciones deberá contribuir a la disminución del tiempo de ejecución de este operador de cruce.

Puede verse que la llamada a la función de aptitud implica instrucciones que consumen tiempo adicional de CPU (lineas 82..86 y 92..96):

```
pushad
push esi
call Aptitud
add esp,4
popad
```

Sin embargo, no hay manera de realizar la llamada sin resguardar los valores de todos los registros, debido a que se está haciendo desde la rutina en ensamblador. Posterior al regreso de la llamada se ajusta la pila en cuatro bytes, que fueron los utilizados para el envío del parámetro (ESI).

El caso de las llamadas al generador de números pseudo-aleatorios (lineas 23, 36 y 43) es tratado de manera diferente. No hay resguardo ni restauración de los registros de la CPU antes y después de la llamada, debido a que dentro de la función se realizan estas operaciones. Las instrucciones para el resguardo del registro ESI y al ajuste de ESP no son necesarios, ya que esta función no recibe parámetros.

Se observa que después de cada llamada hay instrucciones del coprocesador matemático, aprovechando la ventaja de que el generador deja en el tope de la pila de la FPU el valor pseudo-aleatorio generado.

**Mutacion()**

```
01: void mutacion(int g)
02: {
03:   float m,h;
04:   int i,gen;
05:   for (i=ELITISMO; i < TamPoblacion; i++) //pueden mutar solo los que no
06:                                           //son los elitistas
07:   {
```

```
08:     m = UNI; //Para decidir si el individuo muta en base a ProbMutacion
09:     if (m <= ProbMutacion)
10:     {
11:        gen = UNI*DIMENSIONES; //Para decidir cual gen mutar x[0],
12:                               //x[1]..x[gen]
13:        h = UNI;               //Para saber si incrementa o decrementa
14:        if (h < 0.5)
15:            psig[i].x[gen] = psig[i].x[gen] +
                        (UNI*(XMAX - psig[i].x[gen]))*(1-g/MaxGeneraciones);
16:        else
17:            psig[i].x[gen] = psig[i].x[gen] -
                          (UNI*(psig[i].x[gen]+XMAX))*(1-g/MaxGeneraciones);
18:        psig[i].ap = aptitud(psig[i].x);
19:     }
20:   }
21: }  //void mutacion(int g)
```

Hay un ciclo principal en el que se aprecia la llamada al generador de números pseudo-aleatorios (mediante el símbolo UNI) en cinco lineas, sin embargo, cuatro de estas llamadas están condicionadas aleatoriamente contra la variable `ProbMutacion` (lineas 8 y 9), por tanto, la cantidad de veces que se llama al generador no es la misma en cada vuelta del ciclo, tampoco en cada ejecución del operador. Habremos entonces de analizar la varianza en los tiempos de ejecución durante las corridas de prueba para determinar cual es la influencia de la aleatoriedad en las llamadas al generador en la eficiencia de la aplicación, tanto en la versión original como en la optimizada.

También dentro del ciclo se llama a la función de aptitud. Igual que en los casos de los operadores anteriores, la optimización aplicada a las funciones UNI y aptitud contribuye a la optimización del operador.

Por otro lado, una adecuada selección de instrucciones en lenguaje ensamblador para implementar el resto de las instrucciones, puede incrementar la optimización.

La versión en lenguaje ensamblador es la siguiente:

```
001: Mutacion:
002:      push ebp
003:      mov ebp,esp
004:      sub esp,100h         ;Espacio para uso local de la pila
005:      mov esi,[ebp+8]      ;ESI = Apuntador al arreglo de la poblacion
006:      mov edx,dword [ebp+12]     ;EDX = g   (como dword ya que g es int)
007:
008:      push edx     ;respaldar g
009:      ;Respaldar apuntador al inicio de la poblacion
010:      mov ecx,esi
011:      ;Calcular cantidad de doubles en los elitistas
012:      ;Bytes Por Individuo
013:      mov eax, dword [Dimensiones]
014:      inc eax     ;incluir la aptitud
015:      shl eax,3   ;EAX=EAX*8 bytes en cada double = Bytes por individuo
016:      push eax
017:      mul dword [Elitismo]  ;EAX=bytes en los elititistas
018:      add esi,eax    ;ESI apunta al primer individuo después de ELITISMO
019:      pop ebx        ;EBX = Bytes por individuo
020:      ;Calcular bytes totales en la poblacion
021:      ;(Bytes por individuo * TamPoblacion)
022:      mov eax,ebx
023:      mul dword [TamPoblacion]
024:      ;EAX = Total de bytes en la poblacion
025:      add eax,ecx           ;EAX=limite para el ciclo Selecciona
026:
027:      pop edx       ;restaurar g
028:      ;for (i=ELITISMO;i < TamPoblacion;i++)
029: CicloMutacion:
030:      push eax
031:      push esi             ;Guardar apuntador al individuo i
032:
033:      call gsl_rng_uniform_pos    ;m=UNI
034:      ;if(m <= ProbMutacion)
035:      fcomp qword [ProbMutacion]
036:      fstsw ax                ;Carga en ax los bits de estado de la FPU
```

```
037:      sahf                    ;Guarda AH en el registro de banderas
038:      ja SigueCicloMutacion   ;Si m > ProbMutacion, salta al fin del if
039:      push esi             ;Otra vez
040:
041:      ;gen = UNI*DIMENSIONES; //Para decidir cual gen mutar x[0],
042:      ;                       //x[1]....x[gen]
043:      call gsl_rng_uniform_pos    ;UNI
044:      fimul dword [Dimensiones]   ;UNI*Dimensiones
045:      fistp dword [AuxDD]         ;gen=UNI*Dimensiones
046:      mov eax,dword [AuxDD]
047:      shl eax,3  ;EAX=EAX*8 bytes en cada double = Apunta al gen a mutar
048:      add esi,eax                 ;ESI apunta al gen a mutar
049:
050:      mov dword [AuxDD],edx       ;AuxDD = g
051:
052:      call gsl_rng_uniform_pos    ;h = UNI; //Para saber si incrementa o
053:                                  ;          decrementa
054:      ;if (h < 0.5)
055:      fcomp qword [PuntoCinco]
056:      fstsw ax              ;Carga en ax los bits de estado de la FPU
057:      sahf                  ;Guarda AH en el registro de banderas
058:      jae SigueDecrementa       ;Si h >= 0.5, salta al else
059:      ;psig[i].x[gen] = psig[i].x[gen]+(UNI*(XMAX-psig[i].x[gen]))*
060:      ;                                         (1-g/MaxGeneraciones);
061:      fild dword [Xmax]           ;XMAX
062:      fsub qword [esi]            ;XMAX-psig[i].x[gen]
063:      call gsl_rng_uniform_pos    ;UNI, (XMAX-psig[i].x[gen])
064:      fmulp                       ;(UNI*(XMAX-psig[i].x[gen]))
065:      fild dword [AuxDD]          ;g, (UNI*(XMAX-psig[i].x[gen]))
066:      fidiv dword [MaxGeneraciones] ;g/MaxGeneraciones,
067:                                    ;(UNI*(XMAX-psig[i].x[gen]))
068:      fld1                    ;1, g/MaxGeneraciones,
069:                              ;(UNI*(XMAX-psig[i].x[gen]))
070:      fsubrp                  ;(1-g/MaxGeneraciones),
071:                              ;(UNI*(XMAX-psig[i].x[gen]))
072:      fmulp           ;(UNI*(XMAX-psig[i].x[gen]))*(1-g/MaxGeneraciones)
073:      fadd qword [esi]  ;psig[i].x[gen]+(UNI*(XMAX-psig[i].x[gen]))*(1-
074:                        ;                        g/MaxGeneraciones)
```

```
075:      ;Termina cálculo: psig[i].x[gen]+(UNI*(XMAX-psig[i].x[gen]))*(1-
076:      ;                                          g/MaxGeneraciones);
077:      jmp MutCalculaAptitud
078: SigueDecrementa:
079:      ;else
080:      ;psig[i].x[gen] = psig[i].x[gen]-(UNI*(psig[i].x[gen]+XMAX))*(1-
081:      ;                                          g/MaxGeneraciones);
082:      fild dword [Xmax]           ;XMAX
083:      fadd qword [esi]            ;psig[i].x[gen]+XMAX
084:      call gsl_rng_uniform_pos    ;UNI, (psig[i].x[gen]+XMAX)
085:      fmulp                       ;(UNI*(psig[i].x[gen]+XMAX))
086:      fild dword [AuxDD]          ;g, (UNI*(psig[i].x[gen]+XMAX))
087:      fidiv dword [MaxGeneraciones] ;g/MaxGeneraciones,
088:                                    ;(UNI*(psig[i].x[gen]+XMAX))
089:      fld1                        ;1, g/MaxGeneraciones,
090:                                  ;(UNI*(psig[i].x[gen]+XMAX))
091:      fsubrp                      ;(1-g/MaxGeneraciones),
092:                                  ;(UNI*(psig[i].x[gen]+XMAX))
093:      fmulp           ;(UNI*(psig[i].x[gen]+XMAX))*(1-g/MaxGeneraciones)
094:      fsubr qword [esi]  ;psig[i].x[gen]-
095:                      ;(UNI*(psig[i].x[gen]+XMAX))*(1-g/MaxGeneraciones)
096:      ;Termina cálculo: psig[i].x[gen]-(UNI*(psig[i].x[gen]+XMAX))*(1-
097:      ;                                              g/MaxGeneraciones);
098: MutCalculaAptitud:
099:      ;psig[i].x[gen] = Resultado del cálculo anterior
100:      ;(Guardar el gen mutado)
101:      fstp qword [esi]
102:      pop esi             ;Restaurar apuntador al individuo i
103:
104:      ;psig[i].ap = Aptitud(psig[i].x);
105:      pushad
106:      push esi            ;Enviar parámetro apuntador al individuo i
107:      call Aptitud
108:      add esp,4
109:      popad
110:      add esi,ebx
111:      sub esi,8
112:      fstp qword [esi]    ;psig[i].ap = Aptitud(psig[i].x)
```

```
113:
114: SigueCicloMutacion:
115:      pop esi
116:      pop eax
117:      add esi,ebx         ;Avanzar ESI al sig individuo
118:      cmp esi,eax
119:      jl CicloMutacion
120:      leave
121:      ret
```

Igual que en el caso de los operadores anteriores, se reserva espacio extra en la pila para el resguardo y recuperación de valores almacenados en registros del CPU y para el llamado y retorno del generador de números pseudo-aleatorios y de la función de aptitud.

Como contador en el ciclo principal se emplea el registro ESI, comparándolo como condicón de salida contra el registro EAX (lineas 118 y 119), que contiene el total de bytes en la población. En cuanto a otras instrucciones optimizadas, pueden comentarse el avance del índice al arreglo de genes y la operación `1-g/MaxGeneraciones`. El avance del índice al arreglo de genes debe realizarse multiplicando EAX por ocho (lineas 15 y 47). En lugar de emplear una instrucción de multiplicación, se emplea una instrucción más rápida (`shl EAX,3`). Para realizar la resta en la operación `1-g/MaxGeneraciones`, se emplea la instrucción `fsbrp`, con lo que se evita el uso de la instrucción `fxch`, posterior a la `fld1`, como sucedería en una resta convencional.

Las llamadas al generador de números pseudo-aleatorios (lineas 33, 43, 52, 63 y 84) y a la función de aptitud (lineas 105..109) están implementadas de la misma manera que el operador de cruzamiento.

### 4.4.4 Funciones auxiliares

**GeneraPoblacion()**

```
01:   //--- Generación aleatoria de la primera población ---
02:   for(i=0;i < TamPoblacion;i++)
03:   {
04:     for(d=0;d < DIMENSIONES;d++)     //Generar un valor aleatorio para
05:                                      //cada dimensión del individuo
06:     {
07:        #ifdef ASIMETRICO
08:          p[i].x[d] = UNI*XMAX;       //UNI distribución uniforme (0,1)
09:        #else
10:          if(UNI < 0.5)
11:             p[i].x[d] = UNI*XMAX;
12:          else
13:             p[i].x[d] = -(UNI*XMAX);
14:        #endif
15:      }
16:      p[i].ap = aptitud(p[i].x);    //Calcular la aptitud inicial
17:                                    //del individuo
18:   }
```

En la versión en C, la generación de la población inicial no es una función como tal, sino que el código está incluido en la función main() del proceso esclavo. En la versión optimizada está implementada como una función a la que se le envía como parámetro la dirección de inicio del arreglo de la población.

Hay un ciclo principal (lineas 2..18) en el que se almacenan en el arreglo de la población los individuos generados (tantos como indica la variable `TamPoblacion`). En el ciclo anidado se generan aleatoriamente los genes para cada individuo. La directiva de la linea 7 aplica sólo para el caso de la función de aptitud de Griewank, por tanto, para el caso de las cuatro funciones de aptitud contempladas en este documento consideraremos sólo

la ejecucion de las instrucciones de las lineas 10..13. Una vez terminado el ciclo anidado se calcula la aptitud del individuo recién creado.

Se ejecuta el generador de números pseudo-aleatorios tantas veces como el número de dimensiones que tiene cada individuo, mientras que la función de aptitud se ejecuta tantas veces como lo indica el tamaño de la población.

La versión optimizada es la siguiente:

```
01: GeneraPoblacion:
02:      push ebp
03:      mov ebp,esp
04:      sub esp,100h         ;Espacio para uso local de la pila
05:      mov esi,[ebp+8]      ;ESI = Apuntador a la poblacion completa
06:
07:      mov eax, dword [Dimensiones]
08:      mov ebx,eax  ;ebx=Dimensiones
09:      inc eax      ;eax=Dimensiones+1 (1 de la aptitud)
10:      shl eax,3            ;EAX=EAX * 8 bytes en cada double
11:      mov edx,eax  ;edx = Bytes por individuo
12:      mov eax, dword [TamPoblacion]
13:      dec eax    ;Para que procese el rango 0..TamPoblacion-1
14:      cmp dword [Asimetrico],1
15:      jl CicloPobNA
16: CicloPobA:
17:      push eax
18:      xor ecx,ecx
19:   CicloDimA:
20:      ;p[i].x[d] = UNI*XMAX;     //UNI distribucion uniforme (0,1)
21:      call gsl_rng_uniform_pos   ;UNI
22:      fimul dword [Xmax]         ;UNI*XMAX
23:      fstp qword [esi + ecx*8]   ;p[i].x[d] = UNI*XMAX
24:      inc ecx
25:      cmp ecx,ebx
26:      jl CicloDimA
27:      pushad
```

```
28:      push esi                    ;ESI=Apuntador al individuo
29:      call Aptitud
30:      add esp,4
31:      popad
32:      fstp qword [esi + ecx*8]    ;p[i].ap = Aptitud(p[i].x);
33:      add esi,edx
34:      pop eax
35:      dec eax
36:      jns CicloPobA
37:      jmp FinGenPoblacion
38:
39: CicloPobNA:
40:      push eax
41:      xor ecx,ecx
42:   CicloDimNA:
43:      ;if(UNI < 0.5)
44:      ;   p[i].x[d] = UNI*XMAX;
45:      ;else
46:      ;   p[i].x[d] = -(UNI*XMAX);
47:      call gsl_rng_uniform_pos    ;UNI
48:      fimul dword [Xmax]          ;UNI*XMAX
49:      call gsl_rng_uniform_pos    ;UNI, UNI*XMAX
50:      fcomp qword [PuntoCinco]    ;Compara UNI con 0.5
51:      fstsw ax                  ;Carga en ax los bits de estado de la FPU
52:      sahf                      ;Guarda AH en el registro de banderas
53:      jb SigueIf                ;Si UNI < 0.5, salta
54:      fchs                      ;-(UNI*XMAX)
55: SigueIf:
56:      fstp qword [esi + ecx*8]    ;p[i].x[d] = UNI*XMAX
57:      inc ecx
58:      cmp ecx,ebx
59:      jl CicloDimNA
60:      pushad
61:      push esi
62:      call Aptitud
63:      add esp,4
64:      popad
65:      fstp qword [esi + ecx*8]    ;p[i].ap = Aptitud(p[i].x);
```

```
66:      add esi,edx
67:      pop eax
68:      dec eax
69:      jns CicloPobNA
70: FinGenPoblacion:
71:      leave
72:      ret
```

Las optimizaciones aplicadas a esta función son las mismas que en el caso de la función Mutación(), anteriormente descrita. Las instrucciones, tanto para la CPU como para la FPU, son las normalmente generadas por el compilador durante la traducción del código en C hacia lenguaje ensamblador. Las llamadas al generador de números pseudo-aleatorios y a la función de aptitud son las que proveen la mayor parte de la optimización a esta función.

**OrdenaPoblacion()**

```
01: void OrdenaPoblacion(void)
02: {
03:    //Ordena descendentemente por aptitud
04:    //Nota: Solo ordena los primeros MIGRABLES (RazonDeMigracion)
05:    //individuos. El resto los deja desordenados
06:    int i,j,d;
07:    struct individuo Aux;
08:    for (j=0;j < RazonDeMigracion;j++)
09:    {
10:       //Buscar el mejor
11:       mejor=j;
12:       for(i=j+1;i < TamPoblacion;i++)
13:          if(p[i].ap > p[mejor].ap)
14:             mejor=i;
15:       if (mejor != j)
16:       {
17:          //Intercambiar los individuos j y mejor
```

```
18:          //Respaldar en Aux individuo mejor
19:          Aux.ap = p[mejor].ap;
20:          for(d=0;d < DIMENSIONES;d++)
21:             Aux.x[d] = p[mejor].x[d];
22:          //Copiar individuo j hacia la posicion mejor
23:          p[mejor].ap = p[j].ap;
24:          for(d=0;d < DIMENSIONES;d++)
25:             p[mejor].x[d] = p[j].x[d];
26:          //Copiar individuo que estaba en mejor hacia j
27:          p[j].ap = Aux.ap;
28:          for(d=0;d < DIMENSIONES;d++)
29:             p[j].x[d] = Aux.x[d];
30:       }
31:    }
32: } //void OrdenaPoblacion(void)
```

Esta función realiza el ordenamiento de una parte de los individuos de la población (indicada por la variable `RazonDeMigracion`). El criterio para el ordenamiento es el valor de la aptitud de cada individuo y debido a que la aptitud es devuelta por la función correspondiente que la calcula, como un número negativo, el mayor valor será cero. Es por eso que el ordenamiento se hace de manera descendente.

La función está implementada con un ciclo principal y dos estructuras de control dentro de él. Primero se busca al mejor individuo en la población (lineas 11..14) para después determinar si el individuo seleccionado es mejor que el que actualmente tiene mejor aptitud (linea 15), caso en el cual hay que hacer el intercambio de los dos individuos en la población.

Esta función realiza un acceso exhaustivo a memoria, ya que la población de individuos se encuentra ahí; por tanto, hay poca posibilidad de optimización. Los pocos puntos atacables para optimización son los contadores de los ciclos y algunas pocas instrucciones para el avance del índice entre los genes de cada individuo.

La versión en lenguaje ensamblador es la siguiente:

```
001: OrdenaPoblacion:
002:      push ebp
003:      mov ebp,esp
004:      sub esp,100h         ;Espacio para uso local de la pila
005:      mov esi,[ebp+8]      ;ESI = Apuntador al arreglo de la poblacion
006:      mov edi,[ebp+12]     ;EDI = Apuntador al individuo auxiliar
007:
008:      ;Ordena descendentemente por aptitud
009:      ;Nota: Solo ordena los primeros MIGRABLES (RazonDeMigracion)
010:      ;individuos. El resto los deja desordenados
011:      mov eax, dword [Dimensiones]
012:      inc eax              ;incluir la aptitud
013:      shl eax,3            ;EAX=EAX * 8 bytes en cada double
014:                           ;= Bytes por individuo
015:      push eax
016:      ;Calcular bytes totales en los migrables
017:      ;EAX * RazonDeMigracion
018:      mul dword [RazonDeMigracion]
019:      ;EAX = Total de bytes en los migrables
020:      add eax,esi          ;EAX=limite para el ciclo
021:      pop edx              ;EDX = Bytes por individuo
022:      ;Apuntar ESI a la aptitud del primer individuo
023:      add esi,edx
024:      sub esi,8            ;ESI apunta a la aptitud del primer individuo
025:      ;Apuntar EDI a la aptitud del individuo auxiliar
026:      add edi,edx
027:      sub edi,8         ;EDI apunta a la aptitud del individuo auxiliar
028:      ;for (j=0;j < RazonDeMigracion;j++)
029:      mov ecx,dword [TamPoblacion]
030: CicloJ:
031:      push eax
032:      dec ecx              ;Contador para el ciclo BuscaMejor
033:      push ecx
034:      push esi             ;ESI es j para control del ciclo
035:     mov ebx,esi          ;mejor=j
036:      add esi,edx          ;ESI es i=j+1
```

```
037:      ;for(i=j+1;i < TamPoblacion;i++)
038:      ;   if(p[i].ap > p[mejor].ap)
039:      ;       mejor=i;
040: BuscaMejor:
041:      fld qword [esi]             ;Carga p[i].ap
042:      fcomp qword [ebx]           ;compara p[i].ap con p[mejor].ap
043:      fstsw ax                ;Carga en ax los bits de estado de la FPU
044:      sahf                    ;Guarda AH en el registro de banderas
045:      jna FinIf1                  ;if(p[i].ap <= p[mejor].ap), salta
046:      mov ebx,esi                 ;mejor = i
047:   FinIf1:
048:      add esi,edx       ;Apuntar a la aptitud del sig individuo
049:      loop BuscaMejor
050:
051:      ;if (mejor != j)
052:      pop esi          ;Saca en ESI el apuntador j
053:      ;EBX es mejor, ESI es j
054:      cmp ebx,esi      ;mejor!=j?
055:      je SigueCicloJ   ;Si mejor == j, salta
056:
057:      ;Intercambiar los individuos j y mejor
058:
059:      ;Primero respaldar en Aux el individuo mejor
060:      ;Aux.ap = p[mejor].ap;
061:      ;for(d=0;d < DIMENSIONES;d++)
062:      ;     Aux.x[d] = p[mejor].x[d];
063:      push ebx    ;respaldar apuntador al mejor
064:      push edi
065:      mov ecx,dword [Dimensiones]
066:      inc ecx
067:  RespaldaMejor:
068:      fld qword [ebx]
069:      fstp qword [edi]
070:      sub ebx,8
071:      sub edi,8
072:      loop RespaldaMejor ;Dimesiones+1 vueltas para incluir la aptitud
073:      pop edi
074:      pop ebx      ;restaurar apuntador al mejor
```

```
075:
076:      ;//Copiar individuo j hacia la posicion mejor
077:      ;p[mejor].ap = p[j].ap;
078:      ;for(d=0;d < DIMENSIONES;d++)
079:      ;     p[mejor].x[d] = p[j].x[d];
080:      push esi         ;respalda j
081:      push ebx         ;respalda mejor
082:      mov ecx, dword [Dimensiones]
083:      inc ecx
084:  CopiaMejor:
085:      fld qword [esi] ;[PuntoCinco] ;[esi]
086:      fstp qword [ebx]
087:      sub ebx,8
088:      sub esi,8
089:      loop CopiaMejor
090:      pop ebx
091:      pop esi
092:
093:      ;//Copiar individuo Aux (que era mejor) hacia j
094:      ;p[j].ap = Aux.ap;
095:      ;for(d=0;d < DIMENSIONES;d++)
096:      ;     p[j].x[d] = Aux.x[d];
097:      push esi
098:      push edi
099:      mov ecx, dword [Dimensiones]
100:      inc ecx
101:  CopiaJ:
102:      fld qword [edi]
103:      fstp qword [esi]
104:      sub edi,8
105:      sub esi,8
106:      loop CopiaJ
107:      pop edi
108:      pop esi
109:
110: SigueCicloJ:
111:      pop ecx
112:      add esi,edx     ;Avanzar ESI hacia el sig individuo
```

```
113:      pop eax
114:      cmp esi,eax
115:      jl CicloJ
116:
117:     leave
118:      ret
```

Se aprecia que esta versión no implica una optimización significativa, debido a que los accesos a memoria deben preservarse y a que en esta función no hay llamadas a funciones ya optimizadas. Si bien en los ciclos se emplean registros de la CPU como contadores, el compilador hace la traducción de la misma manera. Por otro lado, el uso de instrucciones de la FPU es muy poca.

**CopiaPoblacion()**

```
01:  //Pasar los individuos de psig al arreglo p
02:  for(i=0;i < TamPoblacion;i++)
03:  {
04:     p[i].ap = psig[i].ap;
05:     for(d=0;d < DIMENSIONES;d++)
06:         p[i].x[d] = psig[i].x[d];
07:  }
```

En el código original, la copia de la población no está implementada en una función aparte, sino que estas instrucciones están incluidas en el ciclo principal del proceso esclavo. Hay un ciclo principal en el que se copian los individuos desde el arreglo `psig` hacia el arreglo `p`. Se copia la aptitud y en un cilo interior se copian los genes de cada individuo.

La versión en lenguaje ensamblador es la siguiente:

```
01: CopiaPoblacion:
02:      push ebp
03:      mov ebp,esp
04:      sub esp,100h        ;Espacio para uso local de la pila
05:      mov esi,[ebp+8]     ;ESI = Apuntador a la poblacion fuente
06:      mov edi,[ebp+12]    ;EDI = Apuntador a la poblacion destino
07:
08:      ;Calcular cantidad de doubles en la poblacion
09:      mov eax, dword [Dimensiones]
10:      inc eax              ;incluir la aptitud
11:      mul dword [TamPoblacion]
12:      shl eax,1      ;EAX=EAX*2
13:      mov ecx,eax    ;ECX = poblacion * 2 (ya que se copiará con movsd)
14:      cld
15:      rep movsd
16:      leave
17:      ret
```

En esta versión se realiza la copia de la población sin la implementación de estructuras de ciclo tradicionales. Se calcula la cantidad total de valores (de 32 bits) a copiar y ese resultado se pone en el registro ECX, que es el contador requerido por la instrucción `rep`, utilizada para la copia (linea 15) junto con la instrucción `movsd`.

### 4.4.5 Estructura de la aplicación AG-DM optimizada

La aplicación AG-DM optimizada está conformada por los siguientes archivos fuente:

- `agdma4.c.` Este es el módulo principal de la aplicación. La función main() es el proceso maestro, encargado de la creación de los procesos esclavos en el cluster. Una vez creados los esclavos, el proceso maestro les envía los parámetros a considerar en el AG. Posteriormente, recolecta los resultados generados por los

esclavos. Este proceso también genera el archivo de reporte con los resultados generados en la ejecución.

- `agdma4esc.c.` Este archivo contiene el código que deben ejecutar los esclavos. Recibe los parámetros AG del proceso maestro, inicia el generador de números pseudo-aleatorios, genera aleatoriamente la primera población de individuos y entra al ciclo principal en el que se ejecutan los operadores genéticos (selección, cruce y mutación) programadas en lenguaje ensamblador. Posteriormente hace el intercambio de algunos individuos con otros procesos esclavos y finalmente envía al proceso maestro la información relativa al mejor individuo generado.

- `scaffer.inc`, `ackley.inc`, `esferica.inc`, `rastrigin.inc`, `schwefel.inc`, `schwefel-noise.inc`, `griewank.inc`, `rosenbrock.inc`, `Weierstrass.inc`, `eliptica.inc` y `quadric.inc.` Cada uno de estos archivos contiene el código en lenguaje ensamblador que implementa cada una de las funciones de aptitud seleccionadas para su estudio.

- `aptitud.asm.` Contiene las directivas de inclusión de los archivos con las funciones de aptitud. Antes de compilar la aplicación, hay que dejar sin comentario la directiva de inclusión de la función a analizar y comentar todas las demás directivas.

- `Makefile.` Contiene los comandos requeridos para la compilación de la aplicación. Además, traslada los archivos ejecutables (agdma4 y agdma4esc) al directorio de PVM desde el que serán ejecutados.

- `mt19937.asm.` Contiene el código en lenguaje ensamblador para el generador de números pseudo-aleatorios.

- `agdmoper.asm.` Contiene el código en lenguaje ensamblador correspondiente a los operadores genéticos (selección, cruce y mutación). Además, contiene la definición de funciones auxiliares (generación aleatoria de la población inicial, ordenación de la población de acuerdo a la aptitud de los individuos y copia de una población hacia un arreglo destino).

- `definiciones.h.` Contiene la definición de los parámetros globales para la operación del algoritmo genético.

- `deffunciones.h.` Contiene la definición de las funciones que fueron implementadas en lenguaje ensamblador pero que serán invocadas desde los módulos escritos en lenguaje C. Esto es necesario para que el compilador pueda hacer correctamente la traducción del tipo y estructura de datos empleados como parámetros formales para la llamada de cada función.

## 4.5 Pruebas de confiabilidad a las rutinas programadas

Cada una de las rutinas programadas en lenguaje ensamblador fue sometida a pruebas de confiabilidad, esto es, se ejecutó la rutina originalmente programada en lenguaje C y se analizó el resultado obtenido. Posteriormente se ejecutó la rutina programada en lenguaje ensamblador con los mismos datos de entrada que para la rutina en lenguaje C y se analizó el resultado.

En el caso del generador de números pseudo-aleatorios, se ejecutó la versión en lenguaje C para generar 100,000,000 de números y se almacenaron los resultados en un archivo de texto. Se ejecutó la versión programada en lenguaje ensamblador con la misma semilla de entrada para generar la misma cantidad de números pseudo-aleatorios y se

almacenaron en un archivo de texto. Al comparar los archivos de texto producidos por ambas versiones, se comprobó que producen exactamente los mismos resultados.

Todas las rutinas programadas en lenguaje ensamblador resultaron ser totalmente confiables. Las tablas con los resultados de confiabilidad pueden consultarse en el apéndice F de este documento.

## 4.6 Estrategia de comparación de tiempos de ejecución

Para efectos de comparación de tiempos de ejecución, se compiló la aplicación AG-DM original escrita en lenguaje C para obtener cuatro versiones de código ejecutable, cada una de las cuales corresponde a cada uno de los cuatro niveles de optimización del compilador. Se realizaron corridas de la aplicación optimizada con lenguaje ensamblador para luego comparar el tiempo de ejecución resultante contra el tiempo de ejecución reportado por la ejecución de cada una de las versiones de la aplicación AG-DM original en lenguaje C.

Los tiempos de ejecución resultantes en cada una de las corridas pueden consultarse en el capítulo 5 (Resultados y trabajo futuro).

# Capítulo 5

# Resultados y trabajo futuro

## 5.1 Tiempos de ejecución en cluster PVM

Para efectos de comparación de los tiempos de ejecución de cada una de las versiones del algoritmo (la versión original en lenguaje C en cada uno de los cuatro niveles de optimización y la versión optimizada con lenguaje ensamblador) se realizaron corridas en el cluster PVM considerando cien muestras por función. Los tiempos de ejecución resultantes fueron almacenados en archivos para calcular los promedios en cada una de

las corridas, así como el ahorro de tiempo de cada versión respecto a las otras. Posteriormente, los resultados fueron concentrados en tablas para realizar el análisis respectivo.

Los tiempos promedio obtenidos en la ejecución de los cuatro niveles de optimización del AG-DM original en C se muestran a continuación:

**COMPARATIVO DE TIEMPOS DE EJECUCIÓN EN LOS CUATRO NIVELES DE OPTIMIZACIÓN DEL COMPILADOR**

Lenguaje: **C**
Compilador: **GNU C**
Plataforma: **Cluster de 28 núcleos, SO Linux Fedora, PVM**
Muestras: **100 por función**

| Función | Tiempo promedio (s) | | Ahorro | Tiempo promedio (s) | Ahorro | Tiempo promedio (s) | Ahorro |
|---|---|---|---|---|---|---|---|
| | Optimización 0 | Optimización 1 | 1 contra 0 | Optimización 2 | 2 contra 0 | Optimización 3 | 3 contra 0 |
| **Ackley** | **6.44447363** | **5.14930375** | **20.09%** | **5.30700430** | **17.65%** | **5.18560596** | **19.53%** |
| **Rastrigin** | **6.18582418** | **4.86810727** | **21.30%** | **4.88628345** | **21.00%** | **4.83701518** | **21.80%** |
| **Schwefel** | **9.21488234** | **4.16193503** | **54.83%** | **4.0126339** | **56.45%** | **3.95089064** | **57.12%** |
| **Weierstrass** | **34.3550889** | **34.2547874** | **0.29%** | **33.9470862** | **1.19%** | **34.319609** | **0.10%** |

Puede observarse que de las opciones de optimización del compilador, el nivel 1 es el que representa la optimización más significativa y que los niveles 2 y 3 no garantizan una mayor eficiencia. Inclusive, para el caso de las funciones Ackley y Rastrigin, el nivel 2 la disminuye. En el caso de la función Weierstrass, ninguno de los niveles provee optimización significativa. Entonces, basta con compilar la aplicación en nivel 1 de optimización para obtener una ganancia aceptable en la eficiencia.

Por otro lado, al analizar el código de cada una de las funciones de aptitud podemos darnos cuenta de que cada una tiene una complejidad particular. La función de Schwefel es la más simple y la función de Weierstrass es la más compleja de las cuatro. Tomando en cuenta esto y al observar los tiempos y porcentajes de ahorro, se puede deducir que la optimización que el compilador puede alcanzar es inversamente proporcional al grado de complejidad del código.

El comparativo de eficiencia de la versión optimizada manualmente con lenguaje ensamblador contra la versión escrita completamente en C arroja los siguientes resultados:

**COMPARATIVO DE TIEMPOS DE EJECUCIÓN C vs C+ASM**

Compilador: **GNU C**
Ensamblador: **NASM**
Plataforma: **Cluster de 28 núcleos, SO Linux Fedora, PVM**
Muestras: **100 por función**

| Función | Tiempo promedio (s) | | Ahorro |
|---|---|---|---|
| | C | C+ASM | |
| **Ackley** | **6.44447363** | **4.53656731** | **29.60%** |
| **Rastrigin** | **6.18582418** | **4.15930530** | **32.76%** |
| **Schwefel** | **9.21488234** | **5.32556821** | **42.21%** |
| **Weierstrass** | **34.35508890** | **10.09793980** | **70.61%** |

Puede observarse que la optimización manual aplicada al algoritmo AG-DM provee mayor eficiencia de ejecución que cualquiera de los niveles de optimización proporcionados por el compilador en el caso de las funciones Ackley, Rastrigin y Weierstrass. El caso de la función Weierstrass es sobresaliente, ya que la versión en lenguaje ensamblador para esta función representa alrededor del 70% de ahorro en el tiempo de ejecución, mientras que el compilador provee sólo un poco más de un punto porcentual.

El caso de la función Schwefel es especial, debido a que es una función que tiene una estructura muy sencilla y la implementación con lenguaje ensamblador no ha sido capaz de reducir el tiempo de ejecución contra la versión optimizada por el compilador. Esto último se explica por dos razones. Por un lado y como se mencionó, es una función muy simple y por tanto no hay un margen significativo de optimización. Por otro lado, hay que considerar que el compilador optimiza toda la aplicación escrita en lenguaje C, mientras que la optimización manual se aplica sólo sobre algunos módulos.

Es pertinente mecionar aquí que en el caso de la función Schwefel y para efectos de acercarnos al nivel de optimización del compilador, se podría observar el código generado automáticamente por éste. Sin embargo y debido a que el objetivo del presente trabajo es el de comparar el código generado automáticamente contra la optimización manual, no aplica definitivamente el copiar aquel código con tal de cuando menos empatar la optimización. En un futuro y para efectos de aprendizaje, se podrá observar ese código.

## 5.2 Conclusiones

En el presente trabajo se han desarrollado las rutinas y operadores genéticos propuestos desde un principio para conformar una batería que, dada su implementación, pueden fácilmente ser modificadas para ser aplicadas en otros problemas de minimización de funciones. De hecho, en el caso de las funciones de aptitud, éstas no requieren modificación alguna. También las rutinas que conforman el generador de números pseudo-aleatorios están programadas de tal manera que no requieren modificación alguna para ser utilizadas en proyectos futuros.

Una vez realizados los trabajos de comparación de eficiencia de la versión optimizada manualmente contra la versión en lenguaje C, podemos concluir que la programación de rutinas en lenguaje ensamblador representan la mejor estrategia de optimización tendiente a obtener la mayor eficiencia posible en los algoritmos.

Se analizó el código, escrito en lenguaje C, de la aplicación *AG-DM (Algoritmos Genéticos Distribuidos con Migración)* [10] y en base al análisis realizado se detectaron como rutinas críticas a las funciones de aptitud y al generador de números pseudo-aleatorios, debido a la gran cantidad de veces que se ejecutan durante una

corrida. También se detectaron como críticas y susceptibles de optimizar las rutinas correspondientes a los operadores del AG (selección, cruce y mutación), así como las funciones auxiliares GeneraPoblación, OrdenaPoblación y CopiaPoblación. Estas rutinas fueron seleccionadas considerando que todas ellas implementan ciclos para llevar a cabo las tareas. Además, algunas de estas rutinas hacen llamadas a otras que fueron escritas en lenguaje ensamblador, particularmente al generador de números pseudo-aleatorios.

Se programaron en lenguaje ensamblador los módulos elegidos, tratando de seleccionar las instrucciones pertinentes con el fin de obtener el código más optimizado. También se consideraron los requerimientos que el compilador GNUC establece para el paso de parámetros, la devolución de valores, el uso de variables globales, la organización de la pila del coprocesador matemático y la sintaxis requerida para realizar un ligado exitoso entre módulos objeto ensamblados y el programa principal escrito en C.

Una vez programados los módulos en lenguaje ensamblador, se realizaron corridas para asegurarnos de que los resultados arrojados por cada rutina fueran totalmente confiables. Para lograr eso, se realizaron corridas de cada rutina escrita en lenguaje C y se almacenaron en archivos los resultados arrojados por cada una. Posteriormente se realizaron corridas para cada uno de los módulos en ensamblador y se almacenaron también en archivos los resultados obtenidos por cada uno. Al hacer un comparativo de los resultados, se concluyó que las rutinas optimizadas manualmente son totalmente confiables.

Tal como se mencionó en el apartado 5.1 (Tiempos de ejecución en cluster PVM), se realizaron corridas para comparar la eficiencia del programa original escrito en lenguaje C contra la eficiencia de la versión optimizada manualmente con lenguaje ensamblador. Los resultados indican que de las cuatro funciones de aptitud consideradas, tres de ellas

(Ackley, Rastrigin y Weierstrass) lograron ser optimizadas de mejor manera que como lo hace el compilador en cualquiera de sus niveles de optimización.

La función de aptitud de Schwefel resultó ser mejor optimizada por el compilador que con el empleo del lenguaje ensamblador. Al observar esta función, podemos darnos cuenta de que contiene instrucciones muy sencillas, por tanto, se puede concluir que en el caso de rutinas muy simples, el uso del lenguaje ensamblador no provee un nivel significativo de ahorro en el tiempo de ejecución.

## 5.3 Tabajo futuro

Para reafirmar los resultados positivos alcanzados en este trabajo respecto a la aplicación de optimización manual como estrategia de optimización de código, se pretende en un futuro emplear esta técnica de optimización en otra aplicación de algoritmos genéticos con individuos de diferente naturaleza a los considerados en el problema abordado aquí. Tentativamente, se abordará la optimización manual de código para algoritmos genéticos en los que las poblaciones estén integradas por individuos de tipo cadena de caracteres.

Considerando que en el presente trabajo se han programado en lenguaje ensamblador las funciones de aptitud utilizadas en el problema de minimización de funciones, se pretende tanbién en un futuro realizar trabajos de optimización de código para el mismo problema, pero aplicando la técnica de optimización conocida como PSO (*Particle Swarm Optimization*) y realizar las comparaciones de eficiencia.

# Apéndice A

# Módulos en lenguaje C del algoritmo AG-DM original

## A.1 Módulo maestro

```
/*  agdm.c
    Algoritmo Genetico Distribuido con migración (PROCESO MAESTRO)

    Copyright (C) 2010 Marco Antonio Castro Liera  mcastro@itlp.edu.mx
                       Joel Artemio Morales Viscaya
                  2011 Jorge Bravo Espinoza   bravo_sistemas@live.com.mx

    This program is free software; you can redistribute it and/or modify
```

```
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
*/


//Parametros Globales del AG
#define TAMPOBLACION 500       //TAMPOBLACION Tamaño de la población
#define MAXGENERACIONES 6000   //MAXGENERACIONES Número de generaciones creadas en
cada colonia (en cada proc esclavo)
                  //Valores de prueba para f(x) a 1E-12 y parámetros a 1E-8
                          //-Función----------TAMPOBLACION-----MAXGENERACIONES
                          // ACKLEY         -      500       -      6000
                          // ELIPTICA       -      400       -     30000
                          // ESFERICA       -      500       -      6000
                          // GRIEWANK       -      400       -      8000
                          // QUADRIC        -      400       -      5000
                          // RASTRIGIN      -      500       -      6000
                          // ROSENBROCK     -
                          // SCAFFER        -
                          // SCHWEFEL       -      400       -    120000
                          // SCHWEFEL-NOISE -      400       -    120000
                          // WEIERSTRASS    -


#define TAMTORNEO 2       //TAMTORNEO Tamaño del torneo (para la selección)
#define PROBCRUCE 0.6     //PROBCRUCE Probabilidad de cruce (cantidad de
            individuos generados por cruce =MAU*PROBCRUCE)
#define PROBMUTACION 0.3       //PROBMUTACION Probabilidad de mutación
#define PERIODOMIGRATORIO 200  //PERIODOMIGRATORIO  Número de generaciones antes de
cada migración
#define RAZONMIGRACION 25      //RAZONMIGRACION Cuantos individuos migran a otra
colonia (entre los procs esclavos)
#define POBLACIONES 8          //POBLACIONES Número de colonias
```

```
//DIMENSIONES está definida en el archivo aptitud.h

//Otros Parámetros para el análisis estadístico
#define TMUESTRA 3          //TMUESTRA Cuantas veces se ejecuta el algoritmo
#define MAXERROR 0.01       //MAXERROR Error máximo permitido


#include <stdio.h>
#include <math.h>
#include "pvm3.h"       //las librerías de PVM
#include <time.h>       //Para tomar la fecha y hora formateadas

#include <gsl/gsl_rng.h>

#define UNI gsl_rng_uniform_pos(r)
gsl_rng * r;

#include "aptitud.h"      //Función de aptitud (descomentar una en el .h)


void exit(int);

//Definición de los individuos para la migración
struct individuo
{
    double x[DIMENSIONES];  //cromosoma -- DIMENSIONES está definida en el archivo
aptitud.h
    double ap;              //aptitud
};

struct individuo AuxIndividuo;        //Para recibir individuos de los esclavos

main()
{
    int mytid;                  // mi task id
    int tids[32];         // task ids esclavos
    int sendint;
    int n, nproc, numt, i, j, k, l, r, d,  msgtype, nhost, narch, prematuros;
    char salida[256];
    long evals;
    long si,ui,sf,uf;
    double sendouble;
```

```
    double result,xin,mresult,mx[DIMENSIONES],eprom,error;
    double tiempo;
    FILE *informe;
    //FILE *tiempos;
    struct timeval tv;
    struct timezone tz;
    struct pvmhostinfo *hostp;
    //Para obtener nombre del archivo informe
    time_t TiempoActual;
    struct tm *FechaHora;

    //Obtener info de dia y hora para nombrar al archivo de informe
    //Leer tiempo actual
    TiempoActual = time (NULL);
    //Convertir a tiempo local
    FechaHora = localtime (&TiempoActual);
    strftime(salida,256,"%d%b%y-%H:%M:%S.txt",FechaHora);
    informe = fopen(salida,"w");

    //Guardar info de cabecera en archivo de informe
    fprintf(informe,"Función............: %s (%d
dimensiones)\n",FUNCION,DIMENSIONES);
    fprintf(informe,"Poblaciones........: %d\n",POBLACIONES);
    fprintf(informe,"Tamaño de población: %d\n",TAMPOBLACION);
    fprintf(informe,"Generaciones.......: %d\n",MAXGENERACIONES);
    fprintf(informe,"Periodo migratorio.: %d\n",PERIODOMIGRATORIO);
    fprintf(informe,"Razón de migración.: %d\n\n",RAZONMIGRACION);


//sprintf(salida,"ag-dm-%s-%02d-%03d-%05d.txt",FUNCION,POBLACIONES,DIMENSIONES,TAMPOB
LACION);
    //informe = fopen(salida,"w");

//sprintf(salida,"t-ag-dm-%s-%02d-%03d-%05d.txt",FUNCION,POBLACIONES,DIMENSIONES,TAMP
OBLACION);
    //tiempos = fopen(salida,"w");

    prematuros = 0;
    error = 0;
    for(j=0;j < TMUESTRA;j++)
    {
       gettimeofday(&tv,&tz);
       si = tv.tv_sec;
```

```
        ui = tv.tv_usec;
        mytid = pvm_mytid(); //Agregarse a la pvm
        pvm_config(&nhost,&narch,&hostp ); //Para determinar cuantos
                                           //hosts tiene la PVM

        nproc = POBLACIONES; //Iniciar un proceso por cada población.

        /* Iniciar Esclavos */
        printf("Procesando Muestra %d------------------\n",j+1);
        numt=pvm_spawn("agdmesc", (char**)0, 0, "", nproc, tids);
        if( numt < nproc )
        {
           printf("\n Error creando esclavos, Error codes:\n");
           for( i=numt ; i < nproc ; i++ )
              printf("TID %d %d\n",i,tids[i]);
           for( i=0 ; i<nproc ; i++ )
              pvm_kill( tids[i] );
           pvm_exit();
           exit(1);
        }

        //Enviar parametros a los procesos esclavos
        msgtype = 5;
        for(i=0;i < nproc;i++)
        {
            pvm_initsend(PvmDataDefault);
            sendint = TAMPOBLACION;
            pvm_pkint(&sendint,1,1);
            sendint = MAXGENERACIONES;
            pvm_pkint(&sendint,1,1);
            sendint = TAMTORNEO;
            pvm_pkint(&sendint,1,1);
            sendint = PERIODOMIGRATORIO;
            pvm_pkint(&sendint,1,1);
            sendint = RAZONMIGRACION;
            pvm_pkint(&sendint,1,1);
            sendouble = PROBCRUCE;
            pvm_pkdouble(&sendouble,1,1);
            sendouble = PROBMUTACION;
            pvm_pkdouble(&sendouble,1,1);
            //sendint = i % 2;
            //pvm_pkint(&sendint,1,1);    //**Si i es Par mandará 0, si es impar
mandará 1
```

```
                                        //para que el proc esclavo sepa si es par o
impar
           sendint=tids[(i+1)%nproc];
           pvm_pkint(&sendint,1,1);    //**Para i=0..nproc-2 mandara i+1, para
i=nproc-1 mandará 0
                                        //para que el proc esclavo sepa hacia cual
otro esclavo
                                        //enviará sus migrables
           pvm_send(tids[i],msgtype);
       } //for(i=0;i<nproc;i++)
       //Esperar (y recibir) los resultados de los procesos esclavos
       mresult=100000;
       evaluaciones = 0;
       for(i=0;i < nproc;i++)
       {
           pvm_recv(-1,msgtype);        //El -1 indica que recibe mensaje de
cualquier proceso esclavo
           pvm_upkdouble(&result,1,1);
           pvm_upkdouble(AuxIndividuo.x,DIMENSIONES,1);
           pvm_upklong(&evals,1,1);
           //Guardar en mresult y en mx[] el mejor resultado de los que se han
recibido
           if(mresult > result)
           {
              mresult = result;
              for(d=0;d < DIMENSIONES;d++)
                 mx[d] = AuxIndividuo.x[d];
           }
           evaluaciones +=evals;
       } //for(i=0;i < nproc;i++)
       if(mresult > MAXERROR)
          prematuros++;
       else
          error += mresult;
       gettimeofday(&tv,&tz);
       sf = tv.tv_sec;
       uf = tv.tv_usec;
       sf -= si;
       if (ui > uf)
       {
          sf--;
          uf+=1000000;
       }
```

```
        uf-=ui;
        tiempo = (double) sf;
        tiempo += (double) uf/1000000;
        printf("Mejor resultado: f(x)=%.12f\n",mresult);
        printf("Tiempo (segundos):%.06f\nEvaluaciones de la función de aptitud:
%lu\n",tiempo,evaluaciones);
        printf("Mejor individuo (%d dimensiones):\n",DIMENSIONES);
        for(d=0;d < DIMENSIONES;d++)
           printf("%14.8f\n",mx[d]);
        printf("\n\n");

        //Guardar resultados en archivo de informe
        fprintf(informe,"Muestra %d------------------\n",j+1);
        fprintf(informe,"Mejor resultado: f(x)=%.12f\n",mresult);
        fprintf(informe,"Tiempo (segundos):%.06f\nEvaluaciones de la función de
aptitud: %lu\n",tiempo,evaluaciones);
        fprintf(informe,"Mejor individuo:\n");
        for(d=0;d < DIMENSIONES;d++)
           fprintf(informe,"%14.8f\n",mx[d]);
        fprintf(informe,"\n");

        pvm_exit();
    }  //for(j=0;j < TMUESTRA;j++)
    eprom = error/(TMUESTRA-prematuros);
    fprintf(informe,"Prematuros....: %d\n",prematuros);
    fprintf(informe,"Error Promedio: %.25f\n",eprom);
    fprintf(informe,"\n");
    fclose(informe);
} //main()
//Fin de agdm.c
```

## A.2 Módulo esclavo y operadores

```
/*  agdmesc.c Algoritmo Genetico Distribuido con Migracion (PROCESO ESCLAVO)

    Copyright (C) 2010 Marco Antonio Castro Liera  mcastro@itlp.edu.mx
                       Joel Artemio Morales Viscaya
                  2011 Jorge Bravo Espinoza   bravo_sistemas@live.com.mx

    This program is free software; you can redistribute it and/or modify
```

```
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
*/
#define UNI gsl_rng_uniform_pos(r)

#include <stdio.h>
#include "pvm3.h"
#include <stdlib.h>
#include <sys/time.h>
#include <unistd.h>
#include <math.h>
#include <gsl/gsl_rng.h>

#define MAXTAMPOBLACION  30000    //Máximo tamaño que puede tener el arreglo p de la
poblacion
#define ELITISMO 25               //Número de individuos con mejor aptitud a
seleccionar directamente

//XMAX Máximo valor para x. Está difinida en el archivo .h de la
//     función de aptitud correspondiente (rastrigin.h, eliptica.h, etc.)

//DIMENSIONES está definida en el archivo aptitud.h

int impar, destino,mejor=0;
int ProcMaestro, TipoMsg;

gsl_rng * r;

#include "aptitud.h"

struct individuo
{
```

```
   double x[DIMENSIONES];    //cromosoma -- DIMENSIONES está definida en el archivo
aptitud.h
   double ap;                //aptitud
};

struct individuo p[MAXTAMPOBLACION];    //Arreglo para guardar la poblacion
struct individuo psig[MAXTAMPOBLACION]; //Arreglo auxiliar de poblacion

//Variables para recibir los parámetros del maestro (ver archivo ag-dm.c)
int  TamPoblacion, MaxGeneraciones, TamTorneo, PeriodoMigratorio, RazonDeMigracion;
double ProbMutacion, ProbCruce, CuantosCruzar;

//Tercera versión de ordenamiento
//Ordena sólo una parte de la población (considerando RazonDeMigración)
//Hace la ordenación sobre el mismo arreglo p
void OrdenaPoblacion(void)
{
   //Ordena descendentemente por aptitud
   //Nota: Solo ordena los primeros MIGRABLES (RazonDeMigracion) individuos
   //      El resto los deja desordenados
   int i,j,d;
   struct individuo Aux;
   for (j=0;j < RazonDeMigracion;j++)
   {
      //Buscar el mejor
      mejor=j;
      for(i=j+1;i < TamPoblacion;i++)
         if(p[i].ap > p[mejor].ap)
            mejor=i;
      if (mejor != j)
      {
         //Intercambiar los individuos j y mejor
         //Respaldar en Aux individuo mejor
         Aux.ap = p[mejor].ap;
         for(d=0;d < DIMENSIONES;d++)
            Aux.x[d] = p[mejor].x[d];
         //Copiar individuo j hacia la posicion mejor
         p[mejor].ap = p[j].ap;
         for(d=0;d < DIMENSIONES;d++)
            p[mejor].x[d] = p[j].x[d];
         //Copiar individuo que estaba en mejor hacia j
         p[j].ap = Aux.ap;
         for(d=0;d < DIMENSIONES;d++)
```

```
            p[j].x[d] = Aux.x[d];
      }
   }
} //void OrdenaPoblacion(void)

void BuscaMejor(void)
{
   //Busca el mejor individuo y deja su posición en la variable mejor
   int i;
   mejor=0;
   for(i=1;i < TamPoblacion;i++)
      if(p[i].ap > p[mejor].ap)
         mejor=i;
} //void BuscaMejor(void)

void seleccion(void)
{
   int i,j,k,l;
   //Copiar a psig los ELITISMO individuos con mejor aptitud
   for(i=0;i < ELITISMO;i++)
   {
      for(j=0;j < DIMENSIONES;j++)
         psig[i].x[j] = p[i].x[j];
      psig[i].ap = p[i].ap;
   }

   //*** Comentar el resto de código para probar lo siguiente:
   //*** No seleccionar aquí al resto de la población
   //*** Obtener mediante cruzamiento el resto para la siguiente generación
   //Selecciona por sorteo el resto de los individuos
   for(i=ELITISMO;i < TamPoblacion;i++)
   {
      //Elige aleatoriamente al individuo k
      k = UNI*TamPoblacion;       //***Queda pendiente si k la forzamos desde
ELITISMO en adelante
      //El individo k "competirá" con otros TamTorneo individuos seleccionados
aleatoriamente
      for(j=1;j < TamTorneo;j++)    //TamTorneo es el tamaño de sorteo (2)
      {
         l = UNI*TamPoblacion;
         if(p[k].ap < p[l].ap)
           k = l;
      }
```

```
      //En k queda el ganador del sorteo
      for(j=0;j < DIMENSIONES;j++)
         psig[i].x[j] = p[k].x[j];
      psig[i].ap = p[k].ap;
   }
   //Al final, en psig quedan los individuos seleccionados

} //void seleccion(void)


//Versión original de cruzamiento
//Cruce Aritmético: Combina linealmente los vectores de cromosoma de dos padres para
generar dos hijos.
//Hijo1 = fp * Padre1 + (1 - fp) * Padre2
//Hijo2 = (1 - fp) * Padre1 + fp * Padre2
//Donde fp es un factor de ponderación aleatorio
void cruce(void)
{
   double NuevoX1,NuevoX2;
   int i,j,k,d;
   float lambda;
   lambda = UNI;  //Obtener el factor de ponderación
   for(i=0;i < CuantosCruzar;i++)
   {
      j=UNI*TamPoblacion;
      k=UNI*TamPoblacion;
      for(d=0;d < DIMENSIONES;d++)
      {
         NuevoX1 = psig[j].x[d]*lambda+psig[k].x[d]*(1-lambda);
         NuevoX2 = psig[j].x[d]*(1-lambda)+psig[k].x[d]*lambda;
         psig[j].x[d] = NuevoX1;
         psig[k].x[d] = NuevoX2;
      }
      psig[j].ap = aptitud(psig[j].x);
      psig[k].ap = aptitud(psig[k].x);
   }
} //void cruce(void)

void mutacion(int g)
{
   float m,h;
   int i,gen;
```

```
   //** original for (i=0;i < TamPoblacion;i++) //Puede mutar cualquiera de los
individuos de la población
   for (i=ELITISMO;i < TamPoblacion;i++)  //pueden mutar solo los que no son los
elitistas
   {
      m = UNI; //Para decidir si el individuo muta en base a ProbMutacion
      if(m <= ProbMutacion)
      {
         gen = UNI*DIMENSIONES; //Para decidir cual gen mutar x[0], x[1]....x[gen]
         h = UNI;               //Para saber si incrementa o decrementa
         if (h < 0.5)
            psig[i].x[gen] = psig[i].x[gen]+
(UNI*(XMAX-psig[i].x[gen]))*(1-g/MaxGeneraciones);
         else
            psig[i].x[gen] = psig[i].x[gen]-(UNI*(psig[i].x[gen]
+XMAX))*(1-g/MaxGeneraciones);
         psig[i].ap = aptitud(psig[i].x);
      }
   }
}  //void mutacion(int g)

void envia(void)
{
   int i;
   double result;
   for(i=0; i < RazonDeMigracion; i++)
   {
      pvm_initsend(PvmDataDefault);
      result = p[i].ap;
      pvm_pkdouble(&result,1,1);
      pvm_pkdouble(p[i].x,DIMENSIONES,1);
      pvm_send(destino,TipoMsg);
   }
} //void envia(void)

void recibe(void)
{
   int i,j=TamPoblacion-1;
   double result;
   for(i=0;i < RazonDeMigracion;i++)
   {
      pvm_recv(-1,TipoMsg);
      pvm_upkdouble(&result,1,1);
```

```
      p[j].ap = result;
      pvm_upkdouble(p[j].x,DIMENSIONES,1);
      j--;
   }
} //void recibe(void)

main()
{
  int MiTaskID;
  int i,j,k,d;
  int g,veces;    //generacion y ciclos
  double result;
  struct timeval tv;  //estructura para leer la hora del dia
  struct timezone tz; //estructura para leer la zona horaria

  //ingresar a la pvm
  MiTaskID = pvm_mytid();
  ProcMaestro = pvm_parent();
  TipoMsg = 5;

  //Recibir parametros del proceso maestro
  pvm_recv(ProcMaestro,TipoMsg);
  pvm_upkint(&TamPoblacion,1,1);      //Tamaño de poblacion
  pvm_upkint(&MaxGeneraciones,1,1);   //# maximo de generaciones por muestra
  pvm_upkint(&TamTorneo,1,1);         //Tamaño del torneo (cuantos individuos
compiten para ser seleccionados)
  pvm_upkint(&PeriodoMigratorio,1,1); //Cuantas generaciones habrá entre cada
migracion
  pvm_upkint(&RazonDeMigracion,1,1),  //Cuantos individuos migran
  pvm_upkdouble(&ProbCruce,1,1);      //probabilidad de cruce
  pvm_upkdouble(&ProbMutacion,1,1);   //probabilidad de mutacion
  //pvm_upkint(&impar,1,1);
  pvm_upkint(&destino,1,1);
  CuantosCruzar = TamPoblacion*ProbCruce/2;     //Cantidad de cruces por generacion,
                                                //Se divide entre dos pues hay dos
hijos por cruce
  veces = MaxGeneraciones/PeriodoMigratorio;    //Número de veces que habrá migración

  gettimeofday(&tv,&tz);         //Obtener la hora del sistema
  srandom(tv.tv_usec);
  r =gsl_rng_alloc(gsl_rng_mt19937);
  gsl_rng_set(r,random());
```

```
  evaluaciones = 0;              //Para llevar un control de las veces que se ejecuta
la función de aptitud
  //--- Generacion aleatoria de la primera población ---
  for(i=0;i < TamPoblacion;i++)
  {
    for(d=0;d < DIMENSIONES;d++)     //Generar un valor aleatorio para cada dimensión
del individuo
    {
       #ifdef ASIMETRICO
         p[i].x[d] = UNI*XMAX;       //UNI distribucion uniforme (0,1)
       #else
         if(UNI < 0.5)
            p[i].x[d] = UNI*XMAX;    //XMAX está difinida en el archivo .h de la
función de aptitud correspondiente
         else
            p[i].x[d] = -(UNI*XMAX);
       #endif
     }
     p[i].ap = aptitud(p[i].x);      //Calcular la aptitud inicial del individuo
  }
  //--- Ciclo principal del esclavo ---
  for(k=0;k < veces;k++)
  {
     for(g=0;g < PeriodoMigratorio;g++)    //Ejecutar PeriodoMigratorio iteraciones
     {
        OrdenaPoblacion();
        seleccion();                       //Los seleccionados quedan en psig
        cruce();                           //El cruce se hace sobre psig
        mutacion(g+(PeriodoMigratorio*k)); //La mutación se hace sobre psig
        //Pasar los individuos de psig al arreglo p
        for(i=0;i < TamPoblacion;i++)
        {
           p[i].ap = psig[i].ap;
           for(d=0;d < DIMENSIONES;d++)
              p[i].x[d] = psig[i].x[d];
        }
     }
     OrdenaPoblacion();
     envia();
     recibe();
  }  //for(k=0;k < veces;k++) //ciclo principal del esclavo
  BuscaMejor(); //Para enviarlo al Maestro
  pvm_initsend(PvmDataDefault);
```

```
  result = -p[mejor].ap;
  pvm_pkdouble(&result,1,1);
  pvm_pkdouble(p[mejor].x,DIMENSIONES,1);
  pvm_pklong(&evaluaciones,1,1);
  pvm_send(ProcMaestro,TipoMsg);
  gsl_rng_free(r);
  pvm_exit();
}  //main()
//Fin de agdmesc.c
```

## A.3 Funciones de aptitud

### A.3.1 Ackley

```
//Función 8, Pag 11 CEC05

#define FUNCION "ackley"

#define PI 3.14159265358979323846264338327950 29
#define E  2.71828182845904523536028747135266 25

#define XMAX 32

long evaluaciones = 0;
//funcion de aptitud
double aptitud(double x[DIMENSIONES])
  {
    int i;
    double sum1, sum2, res;
    evaluaciones++;
    sum1 = 0.0;
    sum2 = 0.0;
    for (i=0; i<DIMENSIONES; i++)
    {
        sum1 += x[i]*x[i];
        sum2 += cos(2.0*PI*x[i]);
```

```
    }
    sum1 = -0.2*sqrt(sum1/DIMENSIONES);
    sum2 /= DIMENSIONES;
    res = 20.0 + E - 20.0*exp(sum1) - exp(sum2);
    return (-res);
 }
```

### A.3.2 Rastrigin

```
//Función 9, Pag 12 CEC05

#define FUNCION "rastrigin"

#define PI 3.14159265358979323846264338327950 29
#define E  2.71828182845904523536028747135266 25

#define XMAX 5     //Máximo valor para x

long evaluaciones = 0;

//funcion de aptitud
double aptitud(double x[DIMENSIONES])
{
    int i;
    double res;
    evaluaciones++;
    res = 0.0;
    for (i=0; i < DIMENSIONES; i++)
    {
        res += (x[i]*x[i] - 10.0 * cos(2.0*PI*x[i]) + 10.0);
    }
    return (-res);
}
```

### A.3.3 Schwefel

```
//Función 2, Pag 5 CEC05

#define FUNCION "schwefel"
#define PI 3.14159265358979323846264338327950 29
#define E  2.71828182845904523536028747135266 25
#define XMAX 100

long evaluaciones = 0;
double aptitud(double x[DIMENSIONES])
{
   int i;
   long double sum1=0.0, sum2=0.0;
   evaluaciones++;
   for (i=0; i<DIMENSIONES; i++)
   {
       sum2 += x[i];
       sum1 += sum2*sum2;
   }
   return (-sum1);
}
```

## A.3.4 Weierstrass

```
//Función 11, Pag 14 CEC05

#define FUNCION "weierstrass"
```

```
#define PI 3.14159265358979323846264338327950 29
#define E  2.71828182845904523536028747135266 25

#define XMAX 1

long evaluaciones = 0;
//funcion de aptitud
double aptitud(double x[DIMENSIONES])
  {
    int i, j;
    double res;
    double sum;
    double a, b;
    int k_max;
    evaluaciones++;
    a = 0.5;
    b = 3.0;
    k_max = 20;
    res = 0.0;
    for (i=0; i<DIMENSIONES; i++)
    {
        sum = 0.0;
        for (j=0; j<=k_max; j++)
        {
            sum += pow(a,j)*cos(2.0*PI*pow(b,j)*(x[i]+0.5));
        }
        res += sum;
    }
    sum = 0.0;
    for (j=0; j<=k_max; j++)
      {
        sum += pow(a,j)*cos(2.0*PI*pow(b,j)*(0.5));
      }
    res-=DIMENSIONES*sum;
    return (-res);
}
```

# Apéndice B

# Módulos en lenguaje C del algoritmo AG-DM optimizado

## B.1 Módulo maestro

```
/*  agdma4.c
    Algoritmo Genetico Distribuido con migración (PROCESO MAESTRO)

    Copyright (C) 2010 Marco Antonio Castro Liera  mcastro@itlp.edu.mx
```

```
                    Joel Artemio Morales Viscaya
                2012 Jorge Bravo Espinoza   bravo_sistemas@live.com.mx

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301
USA
*/

#include <stdio.h>
#include <math.h>
#include "pvm3.h"       //las librerías de PVM
#include <time.h>       //Para tomar la fecha y hora formateadas

#include "definiciones.h"  //Parámetros a considerar en el AG

void exit(int);

//Definición de los individuos para la migración
struct individuo
  {
    double x[DIMENSIONES];  //cromosoma -- DIMENSIONES está definida en el
archivo aptitud.h
    double ap;              //aptitud
  };

struct individuo AuxIndividuo;        //Para recibir individuos de los
esclavos
```

```
long Evaluaciones = 0;                //Para contar las veces que se ejecuta
la función de aptitud

main()
{
    int mytid;                  // mi task id
    int tids[32];       // task ids esclavos
    int sendint;
    int n, nproc, numt, i, j, k, l, r, d,  msgtype, nhost, narch,
prematuros;
    char salida[256];
    long evals;
    long si,ui,sf,uf;
    double sendouble;
    double result,mresult,mx[DIMENSIONES],eprom,error;
    double tiempo;
    FILE *informe;
    //FILE *tiempos;
    struct timeval tv;
    struct timezone tz;
    struct pvmhostinfo *hostp;
    //Para obtener nombre del archivo informe
    time_t TiempoActual;
    struct tm *FechaHora;

    //Obtener info de dia y hora para nombrar al archivo de informe
    //Leer tiempo actual
    TiempoActual = time (NULL);
    //Convertir a tiempo local
    FechaHora = localtime (&TiempoActual);
    strftime(salida,256,"%d%b%y-%H:%M:%S.txt",FechaHora);
    informe = fopen(salida,"w");

    //Guardar info de cabecera en archivo de informe
    fprintf(informe,"Función............: %s (%d
dimensiones)\n",FUNCION,DIMENSIONES);
    fprintf(informe,"Poblaciones........: %d\n",POBLACIONES);
    fprintf(informe,"Tamaño de población: %d\n",TAMPOBLACION);
```

```
    fprintf(informe,"Generaciones.......: %d\n",MAXGENERACIONES);
    fprintf(informe,"Periodo migratorio.: %d\n",PERIODOMIGRATORIO);
    fprintf(informe,"Razón de migración.: %d\n\n",RAZONMIGRACION);


//sprintf(salida,"ag-dm-%s-%02d-%03d-%05d.txt",FUNCION,POBLACIONES,DIMENSION
ES,TAMPOBLACION);
    //informe = fopen(salida,"w");

//sprintf(salida,"t-ag-dm-%s-%02d-%03d-%05d.txt",FUNCION,POBLACIONES,DIMENSI
ONES,TAMPOBLACION);
    //tiempos = fopen(salida,"w");

    prematuros = 0;
    error = 0;
    for(j=0;j < TMUESTRA;j++)
    {
       gettimeofday(&tv,&tz);
       si = tv.tv_sec;
       ui = tv.tv_usec;
       mytid = pvm_mytid(); //Agregarse a la pvm
       pvm_config(&nhost,&narch,&hostp ); //Para determinar cuantos
                                          //hosts tiene la PVM

       nproc = POBLACIONES; //Iniciar un proceso esclavo por cada población.

       /* Iniciar Esclavos */
       printf("Procesando Muestra %d------------------\n",j+1);
       numt=pvm_spawn("agdma4esc", (char**)0, 0, "", nproc, tids);
       if( numt < nproc )
       {
          printf("\n Error creando esclavos, Error codes:\n");
          for( i=numt ; i < nproc ; i++ )
             printf("TID %d %d\n",i,tids[i]);
          for( i=0 ; i<nproc ; i++ )
             pvm_kill( tids[i] );
          pvm_exit();
          exit(1);
```

```
        }

        //Enviar parametros a los procesos esclavos
        msgtype = 5;
        for(i=0;i < nproc;i++)
        {
            pvm_initsend(PvmDataDefault);
            sendint = TAMPOBLACION;
            pvm_pkint(&sendint,1,1);
            sendint = MAXGENERACIONES;
            pvm_pkint(&sendint,1,1);
            sendint = TAMTORNEO;
            pvm_pkint(&sendint,1,1);
            sendint = PERIODOMIGRATORIO;
            pvm_pkint(&sendint,1,1);
            sendint = RAZONMIGRACION;
            pvm_pkint(&sendint,1,1);
            sendouble = PROBCRUCE;
            pvm_pkdouble(&sendouble,1,1);
            sendouble = PROBMUTACION;
            pvm_pkdouble(&sendouble,1,1);
            //sendint = i % 2;
            //pvm_pkint(&sendint,1,1);    //**Si i es Par mandará 0, si es
impar mandará 1
                                        //para que el proc esclavo sepa si es
par o impar
            sendint=tids[(i+1)%nproc];
            pvm_pkint(&sendint,1,1);    //**Para i=0..nproc-2 mandara i+1,
para i=nproc-1 mandará 0
                                        //para que el proc esclavo sepa hacia
cual otro esclavo
                                        //enviará sus migrables
            pvm_send(tids[i],msgtype);
        } //for(i=0;i<nproc;i++)
        //Esperar (y recibir) los resultados de los procesos esclavos
        mresult=100000;
        Evaluaciones = 0;
        for(i=0;i < nproc;i++)
```

```
        {
            pvm_recv(-1,msgtype);        //El -1 indica que recibe mensaje de
cualquier proceso esclavo
            pvm_upkdouble(&result,1,1);
            pvm_upkdouble(AuxIndividuo.x,DIMENSIONES,1);
            pvm_upklong(&evals,1,1);
            //Guardar en mresult y en mx[] el mejor resultado de los que se
han recibido
            if(mresult > result)
            {
               mresult = result;
               for(d=0;d < DIMENSIONES;d++)
                  mx[d] = AuxIndividuo.x[d];
            }
            Evaluaciones +=evals;
        } //for(i=0;i < nproc;i++)
        if(mresult > MAXERROR)
           prematuros++;
        else
           error += mresult;
        gettimeofday(&tv,&tz);
        sf = tv.tv_sec;
        uf = tv.tv_usec;
        sf -= si;
        if (ui > uf)
        {
           sf--;
           uf+=1000000;
        }
        uf-=ui;
        tiempo = (double) sf;
        tiempo += (double) uf/1000000;
        printf("Mejor resultado: f(x)=%.12f\n",mresult);
        printf("Tiempo (segundos):%.06f\nEvaluaciones de la función de
aptitud: %lu\n",tiempo,Evaluaciones);
        printf("Mejor individuo (%d dimensiones):\n",DIMENSIONES);
        for(d=0;d < DIMENSIONES;d++)
           printf("%14.8f\n",mx[d]);
```

```
        printf("\n\n");

        //Guardar resultados en archivo de informe
        fprintf(informe,"Muestra %d------------------\n",j+1);
        fprintf(informe,"Mejor resultado: f(x)=%.12f\n",mresult);
        fprintf(informe,"Tiempo (segundos):%.06f\nEvaluaciones de la función
de aptitud: %lu\n",tiempo,Evaluaciones);
        fprintf(informe,"Mejor individuo:\n");
        for(d=0;d < DIMENSIONES;d++)
           fprintf(informe,"%14.8f\n",mx[d]);
        fprintf(informe,"\n");

        pvm_exit();
    }  //for(j=0;j < TMUESTRA;j++)
    eprom = error/(TMUESTRA-prematuros);
    fprintf(informe,"Prematuros....: %d\n",prematuros);
    fprintf(informe,"Error Promedio: %.25f\n",eprom);
    fprintf(informe,"\n");
    fclose(informe);
} //main()
//Fin de agdma4.c
```

## B.2 Módulo esclavo

```
/*  agdma4esc.c
    Algoritmo Genetico Distribuido con Migracion (PROCESO ESCLAVO)

    Copyright (C) 2010 Marco Antonio Castro Liera  mcastro@itlp.edu.mx
                       Joel Artemio Morales Viscaya
                  2012 Jorge Bravo Espinoza   bravo_sistemas@live.com.mx

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.
```

```
    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301
USA
*/

#include <stdio.h>
#include "pvm3.h"
#include <stdlib.h>
#include <sys/time.h>
#include <unistd.h>
#include <math.h>

#include "definiciones.h"  //Parámetros a considerar en el AG

struct individuo
{
   double x[DIMENSIONES];    //cromosoma -- DIMENSIONES está definida en el
archivo aptitud.h
   double ap;                //aptitud
};

#include "deffunciones.h"

//Variables de parámetros
int Dimensiones=DIMENSIONES;
int Xmax=XMAX;
int Elitismo=ELITISMO;
/*
#ifdef ASIMETRICO
 int Asimetrico=1;
#else
 int Asimetrico=0;
```

```
#endif
*/
int Asimetrico=0;
long evaluaciones = 0;
int impar, destino, mejor=0;
int ProcMaestro, TipoMsg;

struct individuo p[MAXTAMPOBLACION];    //Arreglo para guardar la poblacion
struct individuo psig[MAXTAMPOBLACION]; //Arreglo auxiliar de poblacion
struct individuo Aux;

//Variables para recibir los parámetros del maestro (ver archivo agdma4.c)
int  TamPoblacion, MaxGeneraciones, TamTorneo, PeriodoMigratorio,
RazonDeMigracion;
double ProbMutacion, ProbCruce, CuantosCruzar;

void BuscaMejor(void)
{
   //Busca el mejor individuo y deja su posición en la variable mejor
   int i;
   mejor=0;
   for(i=1;i < TamPoblacion;i++)
      if(p[i].ap > p[mejor].ap)
         mejor=i;
} //void BuscaMejor(void)

void envia(void)
{
   int i;
   double result;
   for(i=0; i < RazonDeMigracion; i++)
   {
      pvm_initsend(PvmDataDefault);
      result = p[i].ap;
      pvm_pkdouble(&result,1,1);
      pvm_pkdouble(p[i].x,DIMENSIONES,1);
      pvm_send(destino,TipoMsg);
   }
```

```
} //void envia(void)

void recibe(void)
{
   int i,j=TamPoblacion-1;
   double result;
   for(i=0;i < RazonDeMigracion;i++)
   {
      pvm_recv(-1,TipoMsg);
      pvm_upkdouble(&result,1,1);
      p[j].ap = result;
      pvm_upkdouble(p[j].x,DIMENSIONES,1);
      j--;
   }
} //void recibe(void)

main()
{
  int MiTaskID;
  int i,j,k,d;
  int g,veces;    //generacion y ciclos
  double result;
  struct timeval tv;  //estructura para leer la hora del dia
  struct timezone tz; //estructura para leer la zona horaria

  //ingresar a la pvm
  MiTaskID = pvm_mytid();
  ProcMaestro = pvm_parent();
  TipoMsg = 5;

  //Recibir parametros del proceso maestro
  pvm_recv(ProcMaestro,TipoMsg);
  pvm_upkint(&TamPoblacion,1,1);      //Tamaño de poblacion
  pvm_upkint(&MaxGeneraciones,1,1);   //# maximo de generaciones por muestra
  pvm_upkint(&TamTorneo,1,1);         //Tamaño del torneo (cuantos
individuos compiten para ser seleccionados)
  pvm_upkint(&PeriodoMigratorio,1,1); //Cuantas generaciones habrá entre
cada migracion
```

```
  pvm_upkint(&RazonDeMigracion,1,1),  //Cuantos individuos migran
  pvm_upkdouble(&ProbCruce,1,1);      //probabilidad de cruce
  pvm_upkdouble(&ProbMutacion,1,1);   //probabilidad de mutacion
  pvm_upkint(&destino,1,1);
  CuantosCruzar = TamPoblacion*ProbCruce/2;     //Cantidad de cruces por
generacion,
                                                //Se divide entre dos pues
hay dos hijos por cruce

  veces = MaxGeneraciones/PeriodoMigratorio;    //Número de veces que habrá
migración

  gettimeofday(&tv,&tz);         //Obtener la hora del sistema
  srandom(tv.tv_usec);
  gsl_rng_set(random());
  evaluaciones = 0;              //Para llevar un control de las veces que
se ejecuta la función de aptitud
  //--- Generacion aleatoria de la primera población ---
  GeneraPoblacion(p);
  //--- Ciclo principal del esclavo ---
  for(k=0;k < veces;k++)
  {
     for(g=0;g < PeriodoMigratorio;g++)    //Ejecutar PeriodoMigratorio
iteraciones
     {
        OrdenaPoblacion(p,&Aux);
        Seleccion(p,psig);                       //Los seleccionados quedan
en psig
        Cruce(psig);                             //El cruce se hace sobre
psig
        Mutacion(psig,g+(PeriodoMigratorio*k));  //La mutación se hace sobre
psig
        //Pasar los individuos de psig al arreglo p
        CopiaPoblacion(psig,p);
     }
     OrdenaPoblacion(p,&Aux);
     envia();
     recibe();
```

```
  }  //for(k=0;k < veces;k++) //ciclo principal del esclavo
  BuscaMejor(); //Para enviarlo al Maestro
  pvm_initsend(PvmDataDefault);
  result = -p[mejor].ap;
  pvm_pkdouble(&result,1,1);
  pvm_pkdouble(p[mejor].x,DIMENSIONES,1);
  pvm_pklong(&evaluaciones,1,1);
  pvm_send(ProcMaestro,TipoMsg);
  pvm_exit();
}  //main()
//Fin de agdma4esc.c
```

# Apéndice C

## Módulos en lenguaje ensamblador del algoritmo AG-DM optimizado

## C.1 Función GeneraPoblacion()

```
extern TamPoblacion, Dimensiones, Xmax, Asimetrico, RazonDeMigracion,
Elitismo, TamTorneo
extern ProbCruce, CuantosCruzar, ProbMutacion, MaxGeneraciones
extern gsl_rng_uniform_pos, Aptitud
global GeneraPoblacion, OrdenaPoblacion, Seleccion, Cruce, Mutacion,
CopiaPoblacion

section .data
   PuntoCinco dq 0.5
        AuxDW dw 0
        AuxDD dd 0
        AuxDQ dq 0

segment .text

;***GeneraPoblacion *** Generación aleatoria de una población
****************** GENERAPOBLACION ****
;***Definición: void GeneraPoblacion(struct individuo *)
;***Recibe como parámetro el apuntador al arreglo de la población.
;***Se requiere la declaración, en el módulo en C, de las siguientes
variables globales:
;***int TamPoblacion; int Dimensiones; int Xmax; int Asimetrico;
GeneraPoblacion:
     push ebp
     mov ebp,esp
     sub esp,100h         ;Espacio para uso local de la pila
     mov esi,[ebp+8]      ;ESI = Apuntador al arreglo de la poblacion
completa

     ;for(i=0;i < TamPoblacion;i++)
     ;{
     ;  for(d=0;d < DIMENSIONES;d++)     //Generar un valor aleatorio para
cada dimensión del individuo
     ;  {
     ;     #ifdef ASIMETRICO
     ;       p[i].x[d] = UNI*XMAX;       //UNI distribucion uniforme (0,1)
     ;     #else
     ;       if(UNI < 0.5)
     ;          p[i].x[d] = UNI*XMAX;    //XMAX está difinida en el
archivo .h de la función de aptitud correspondiente
     ;       else
     ;          p[i].x[d] = -(UNI*XMAX);
     ;     #endif
     ;   }
     ;   p[i].ap = Aptitud(p[i].x);      //Calcular la aptitud inicial del
individuo
     ;}
```

```
     mov eax, dword [Dimensiones]
     mov ebx,eax  ;ebx=Dimensiones
     inc eax      ;eax=Dimensiones+1 (1 de la aptitud)
     shl eax,3            ;EAX=EAX * 8 bytes en cada double
     mov edx,eax  ;edx = Bytes por individuo
     mov eax, dword [TamPoblacion]
     dec eax    ;Para que procese el rango 0..TamPoblacion-1
     cmp dword [Asimetrico],1
     jl CicloPobNA
CicloPobA:
     push eax
     xor ecx,ecx
  CicloDimA:
     ;p[i].x[d] = UNI*XMAX;     //UNI distribucion uniforme (0,1)
     call gsl_rng_uniform_pos   ;UNI
     fimul dword [Xmax]         ;UNI*XMAX
     fstp qword [esi + ecx*8]   ;p[i].x[d] = UNI*XMAX
     inc ecx
     cmp ecx,ebx
     jl CicloDimA
     pushad
     push esi                    ;ESI=Apuntador al individuo
     call Aptitud
     add esp,4
     popad
     fstp qword [esi + ecx*8]    ;p[i].ap = Aptitud(p[i].x);
     add esi,edx
     pop eax
     dec eax
     jns CicloPobA
     jmp FinGenPoblacion

CicloPobNA:
     push eax
     xor ecx,ecx
  CicloDimNA:
     ;if(UNI < 0.5)
     ;   p[i].x[d] = UNI*XMAX;
     ;else
     ;   p[i].x[d] = -(UNI*XMAX);
     call gsl_rng_uniform_pos    ;UNI
     fimul dword [Xmax]          ;UNI*XMAX
     call gsl_rng_uniform_pos    ;UNI, UNI*XMAX
     fcomp qword [PuntoCinco]    ;Compara UNI con 0.5
     fstsw ax                    ;Carga en ax los bits de estado de la FPU
     sahf                        ;Guarda AH en el registro de banderas
     jb SigueIf                  ;Si UNI < 0.5, salta
     fchs                        ;-(UNI*XMAX)
SigueIf:
     fstp qword [esi + ecx*8]    ;p[i].x[d] = UNI*XMAX
     inc ecx
     cmp ecx,ebx
     jl CicloDimNA
     pushad
     push esi
     call Aptitud
     add esp,4
```

```
     popad
     fstp qword [esi + ecx*8]    ;p[i].ap = Aptitud(p[i].x);
     add esi,edx
     pop eax
     dec eax
     jns CicloPobNA
FinGenPoblacion:
     leave
     ret
;*** Fin GeneraPoblacion ***
```

## C.2 Función OrdenaPoblacion()

```
;***OrdenaPoblacion *** Ordenación de una población (descendentemente por
Aptitud) *** ORDENAPOBLACION ****
;***Definición: void OrdenaPoblacion(struct individuo *, struct individuo *)
;***Esta función requiere la declaración, en el módulo en C, de las
siguientes variables globales:
;***int TamPoblacion; int Dimensiones; int RazonDeMigracion;
OrdenaPoblacion:
     push ebp
     mov ebp,esp
     sub esp,100h         ;Espacio para uso local de la pila
     mov esi,[ebp+8]      ;ESI = Apuntador al arreglo de la poblacion
     mov edi,[ebp+12]     ;EDI = Apuntador al individuo auxiliar

     ;Ordena descendentemente por aptitud
     ;Nota: Solo ordena los primeros MIGRABLES (RazonDeMigracion) individuos
     ;      El resto los deja desordenados
     ;int i,j,d;
     ;for (j=0;j < RazonDeMigracion;j++)
     ;{
     ;   //Buscar el mejor
     ;   mejor=j;
     ;   for(i=j+1;i < TamPoblacion;i++)
     ;      if(p[i].ap > p[mejor].ap)
     ;         mejor=i;
     ;   if (mejor != j)
     ;   {
     ;      //Intercambiar los individuos j y mejor
     ;      //Respaldar en Aux individuo mejor
     ;      Aux.ap = p[mejor].ap;
     ;      for(d=0;d < DIMENSIONES;d++)
     ;         Aux.x[d] = p[mejor].x[d];
     ;      //Copiar individuo j hacia la posicion mejor
     ;      p[mejor].ap = p[j].ap;
     ;      for(d=0;d < DIMENSIONES;d++)
     ;         p[mejor].x[d] = p[j].x[d];
     ;      //Copiar individuo que estaba en mejor hacia j
     ;      p[j].ap = Aux.ap;
     ;      for(d=0;d < DIMENSIONES;d++)
     ;         p[j].x[d] = Aux.x[d];
     ;   }
```

```
     ;}

     ;Bytes Por Individuo
     mov eax, dword [Dimensiones]
     inc eax              ;incluir la aptitud
     shl eax,3            ;EAX=EAX * 8 bytes en cada double  = Bytes por
individuo
     push eax
     ;Calcular bytes totales en los migrables
     ;EAX * RazonDeMigracion
     mul dword [RazonDeMigracion]
     ;EAX = Total de bytes en los migrables
     add eax,esi          ;EAX=limite para el ciclo
     pop edx              ;EDX = Bytes por individuo
     ;Apuntar ESI a la aptitud del primer individuo
     add esi,edx
     sub esi,8            ;ESI apunta a la aptitud del primer individuo
     ;Apuntar EDI a la aptitud del individuo auxiliar
     add edi,edx
     sub edi,8            ;EDI apunta a la aptitud del individuo auxiliar
     ;for (j=0;j < RazonDeMigracion;j++)
     mov ecx,dword [TamPoblacion]
CicloJ:
     push eax

     dec ecx              ;Contador para el ciclo BuscaMejor
     push ecx
     push esi             ;ESI es j para control del ciclo
     mov ebx,esi          ;mejor=j
     add esi,edx          ;ESI es i=j+1
     ;for(i=j+1;i < TamPoblacion;i++)
     ;   if(p[i].ap > p[mejor].ap)
     ;       mejor=i;
BuscaMejor:
     fld qword [esi]             ;Carga p[i].ap
     fcomp qword [ebx]           ;compara p[i].ap con p[mejor].ap
     fstsw ax                    ;Carga en ax los bits de estado de la FPU
     sahf                        ;Guarda AH en el registro de banderas
     jna FinIf1                  ;if(p[i].ap <= p[mejor].ap), salta
     mov ebx,esi                 ;mejor = i
  FinIf1:
     add esi,edx       ;Apuntar a la aptitud del sig individuo
     loop BuscaMejor

     ;if (mejor != j)
     pop esi          ;Saca en ESI el apuntador j
     ;EBX es mejor, ESI es j
     cmp ebx,esi      ;mejor!=j?
     je SigueCicloJ   ;Si mejor == j, salta

     ;Intercambiar los individuos j y mejor

     ;Primero respaldar en Aux el individuo mejor
     ;Aux.ap = p[mejor].ap;
     ;for(d=0;d < DIMENSIONES;d++)
     ;     Aux.x[d] = p[mejor].x[d];
     push ebx    ;respaldar apuntador al mejor
```

```
     push edi
     mov ecx,dword [Dimensiones]
     inc ecx
 RespaldaMejor:
     fld qword [ebx]
     fstp qword [edi]
     sub ebx,8
     sub edi,8
     loop RespaldaMejor   ;El ciclo da Dimesiones+1 iteraciones para incluir
la aptitud
     pop edi
     pop ebx      ;restaurar apuntador al mejor

     ;//Copiar individuo j hacia la posicion mejor
     ;p[mejor].ap = p[j].ap;
     ;for(d=0;d < DIMENSIONES;d++)
     ;     p[mejor].x[d] = p[j].x[d];
     push esi         ;respalda j
     push ebx         ;respalda mejor
     mov ecx, dword [Dimensiones]
     inc ecx
 CopiaMejor:
     fld qword [esi] ;[PuntoCinco] ;[esi]
     fstp qword [ebx]
     sub ebx,8
     sub esi,8
     loop CopiaMejor
     pop ebx
     pop esi

     ;//Copiar individuo Aux (que era mejor) hacia j
     ;p[j].ap = Aux.ap;
     ;for(d=0;d < DIMENSIONES;d++)
     ;     p[j].x[d] = Aux.x[d];
     push esi
     push edi
     mov ecx, dword [Dimensiones]
     inc ecx
 CopiaJ:
     fld qword [edi]
     fstp qword [esi]
     sub edi,8
     sub esi,8
     loop CopiaJ
     pop edi
     pop esi

SigueCicloJ:
     pop ecx
     add esi,edx     ;Avanzar ESI hacia el sig individuo
     pop eax
     cmp esi,eax
     jl CicloJ

     leave
     ret
;*** Fin OrdenaPoblacion ***
```

## C.3 Función CopiaPoblacion()

```
;***CopiaPoblacion *** Obtiene una copia de una población
******************** COPIAPOBLACION ****
;***Definición: void CopiaPoblacion(struct individuo *, struct individuo *)
;***El primer parámetro apunta a la poblacion fuente, el segundo a la
poblacion destino.
;***Esta función requiere la declaración, en el módulo en C, de las
siguientes variables globales:
;***int TamPoblacion; int Dimensiones;
CopiaPoblacion:
     push ebp
     mov ebp,esp
     sub esp,100h         ;Espacio para uso local de la pila
     mov esi,[ebp+8]      ;ESI = Apuntador al arreglo de la poblacion fuente
     mov edi,[ebp+12]     ;EDI = Apuntador al arreglo de la poblacion
destino

     ;//Pasar los individuos de psig al arreglo p
     ;for(i=0;i < TamPoblacion;i++)
     ;{
     ;   p[i].ap = psig[i].ap;
     ;   for(d=0;d < DIMENSIONES;d++)
     ;      p[i].x[d] = psig[i].x[d];
     ;}

     ;Calcular cantidad de doubles en la poblacion
     mov eax, dword [Dimensiones]
     inc eax              ;incluir la aptitud
     mul dword [TamPoblacion]
     shl eax,1            ;EAX=EAX*2
     mov ecx,eax          ;ECX = doubles en la poblacion * 2 (porque se va a
copiar con movsd)
     cld
     rep movsd

     leave
     ret
;*** Fin CopiaPoblacion ***
```

## C.4 Operador Seleccion()

```
;***Seleccion *** Selecciona los mejores (Elitismo) individuos *********
SELECCION ****
;***Definición: void Seleccion(struct individuo *, struct individuo *)
;***El primer parámetro apunta a la poblacion, el segundo a una poblacion
auxiliar
;***Esta función requiere la declaración, en el módulo en C, de las
siguientes variables globales:
;***int TamPoblacion; int Dimensiones; int Elitismo; int TamTorneo;
Seleccion:
     push ebp
     mov ebp,esp
     sub esp,100h         ;Espacio para uso local de la pila
     mov esi,[ebp+8]      ;ESI = Apuntador al inicio de la poblacion
     mov edi,[ebp+12]     ;EDI = Apuntador a una poblacion auxiliar (en esta
quedarán los seleccionados)

     ;int i,j,k,l;
     ;//Copiar a psig los ELITISMO individuos con mejor aptitud
     ;for(i=0;i < ELITISMO;i++)
     ;{
     ;   for(j=0;j < DIMENSIONES;j++)
     ;      psig[i].x[j] = p[i].x[j];
     ;   psig[i].ap = p[i].ap;
     ;}

     ;//Selecciona por sorteo el resto de los individuos
     ;for(i=ELITISMO;i < TamPoblacion;i++)
     ;{
     ;   //Elige aleatoriamente al individuo k
     ;   k = UNI*TamPoblacion;       //***Queda pendiente si k la forzamos
desde ELITISMO en adelante
     ;   //El individo k "competirá" con otros TamTorneo individuos
seleccionados aleatoriamente
     ;   for(j=1;j < TamTorneo;j++)    //TamTorneo es el tamaño de sorteo
(2)
     ;   {
     ;      l = UNI*TamPoblacion;
     ;      if(p[k].ap < p[l].ap)
     ;        k = l;
     ;   }
     ;   //En k queda el ganador del sorteo
     ;   for(j=0;j < DIMENSIONES;j++)
     ;      psig[i].x[j] = p[k].x[j];
     ;   psig[i].ap = p[k].ap;
     ;}
     ;//Al final, en psig quedan los individuos seleccionados

     ;Modificar el registro de control (word) para que la instrucción
     ;FISTP tome el valor del tope truncado (leer solo la parte entera, no
redondear).
     ;Los bits 10 y 11 deben ponerse en 1.
     ;Esto es necesario ya que en C, al combinar int con double, los valores
deben truncarse.
     fstcw word [AuxDW]       ;Leer el registro de control
     or word [AuxDW],0c00h    ;Poner en 1 los bits 10 y 11
     fldcw word [AuxDW]       ;Escribir el word modificado en el registro de
control
```

```
     ;//Copiar a psig los ELITISMO individuos con mejor aptitud
     ;Calcular cantidad de doubles en los elitistas
     mov eax, dword [Dimensiones]
     inc eax              ;incluir la aptitud
     mul dword [Elitismo]
     shl eax,1            ;EAX=EAX*2
     mov ecx,eax          ;ECX = doubles en los elitistas * 2 (porque se va
a copiar con movsd)
     push edi
     push esi
     cld
     rep movsd
     pop esi             ;ESI=apuntador original al inicio de la poblacion
     pop ecx             ;ECX=EDI original al inicio de la poblacion
auxiliar

     ;//Selecciona por sorteo el resto de los individuos
     ;Bytes Por Individuo
     mov eax, dword [Dimensiones]
     inc eax               ;incluir la aptitud
     shl eax,3             ;EAX=EAX * 8 bytes en cada double  = Bytes por
individuo
     push eax
     ;Calcular bytes totales en la poblacion
     ;EAX * TamPoblacion
     mul dword [TamPoblacion]
     ;EAX = Total de bytes en la poblacion
     add eax,ecx           ;EAX=limite para el ciclo Selecciona
     pop ebx               ;EBX = Bytes por individuo
     add edi,ebx
     dec edi               ;EDI apunta al último byte de la aptitud del
individuo Elitismo+1

Selecciona:
     push eax          ;EAX=limite para el ciclo Selecciona
     push esi          ;ESI=apuntador al inicio de la poblacion

     ;k = UNI*TamPoblacion;
     call gsl_rng_uniform_pos    ;UNI
     fimul dword [TamPoblacion]  ;k=UNI*TamPoblacion
     fabs                        ;Valor absoluto de k
     fistp dword [AuxDD]         ;
     mov eax,dword [AuxDD]
     mul ebx                     ;k*Bytes por individuo
     add esi,eax                 ;ESI apunta al individuo k
     add esi,ebx
     sub esi,8                   ;ESI apunta a la aptitud del individuo k
     ;termina k = UNI*TamPoblacion;

     pop edx           ;EDX=apuntador al inicio de la poblacion
     push edx

     mov ecx, dword [TamTorneo]
     dec ecx
CicloTorneo:
     push edx          ;EDX=apuntador al inicio de la poblacion
```

```
     ;l = UNI*TamPoblacion;
     call gsl_rng_uniform_pos    ;UNI
     fimul dword [TamPoblacion]  ;l=UNI*TamPoblacion
     fabs                        ;Valor absoluto de l
     fistp dword [AuxDD]         ;
     mov eax,dword [AuxDD]
     mul ebx                     ;l*Bytes por individuo
     pop edx
     add edx,eax                 ;EDX apunta al individuo l
     add edx,ebx
     sub edx,8                   ;EAX apunta a la aptitud del individuo l
     ;termina l = UNI*TamPoblacion;
     fld qword [esi]             ;Carga p[k].ap
     fcomp qword [edx]           ;compara p[k].ap con p[l].ap
     fstsw ax                    ;Carga en ax los bits de estado de la FPU
     sahf                        ;Guarda AH en el registro de banderas
     jnb FinIf2                  ;if(p[k].ap >= p[l].ap), salta
     mov esi,edx                 ;k = l
  FinIf2:
     loop CicloTorneo


     ;Copiar a psig el individuo ganador del sorteo
     add esi,7
     push edi
     std
     mov ecx, dword [Dimensiones]
     inc ecx
     shl ecx,1
     rep movsd
     pop edi

     add edi,ebx     ;Avanzar EDI hacia el sig individuo
     pop esi
     pop eax
     cmp edi,eax
     jl Selecciona
     leave
     ret
;*** Fin Seleccion ***
```

## C.5 Operador Cruce()

```
;***Cruce
*****************************************************************************
** CRUCE *********
;***Cruce Aritmético: Combina linealmente los vectores de cromosoma de dos
padres para generar dos hijos.
;***Hijo1 = fp * Padre1 + (1 - fp) * Padre2
;***Hijo2 = (1 - fp) * Padre1 + fp * Padre2
;***Donde fp es un factor de ponderación aleatorio
```

```
;***Definición: void Cruce(struct individuo *)
;***El parámetro apunta a la poblacion.
;***Esta función requiere la declaración, en el módulo en C, de las
siguientes variables globales:
;***int TamPoblacion; int Dimensiones;
;***double ProbCruce, CuantosCruzar;   <-- OJO: CuantosCruzar debe ser
double
Cruce:
     push ebp
     mov ebp,esp
     sub esp,100h         ;Espacio para uso local de la pila
     mov esi,[ebp+8]      ;ESI = Apuntador al arreglo de la poblacion

     ;double NuevoX1,NuevoX2;
     ;int i,j,k,d;
     ;float lambda;
     ;lambda = UNI;  //Obtener el factor de ponderación
     ;for(i=0;i < CuantosCruzar;i++)
     ;{
     ;   j=UNI*TamPoblacion;
     ;   k=UNI*TamPoblacion;
     ;   for(d=0;d < DIMENSIONES;d++)
     ;   {
     ;      NuevoX1 = psig[j].x[d]*lambda+psig[k].x[d]*(1-lambda);
     ;      NuevoX2 = psig[j].x[d]*(1-lambda)+psig[k].x[d]*lambda;
     ;      psig[j].x[d] = NuevoX1;
     ;      psig[k].x[d] = NuevoX2;
     ;   }
     ;   psig[j].ap = Aptitud(psig[j].x);
     ;   psig[k].ap = Aptitud(psig[k].x);
     ;}

     ;Modificar el registro de control (word) para que la instrucción
     ;FISTP tome el valor del tope truncado (leer solo la parte entera, no
redondear).
     ;Los bits 10 y 11 deben ponerse en 1.
     ;Esto es necesario ya que en C, al combinar int con double, los valores
deben truncarse.
     fstcw word [AuxDW]       ;Leer el registro de control
     or word [AuxDW],0c00h    ;Poner en 1 los bits 10 y 11
     fldcw word [AuxDW]       ;Escribir el word modificado en el registro de
control

     mov edi,esi     ;ESI y EDI apuntan a psig (a la poblacion)

     mov eax, dword [Dimensiones]
     inc eax               ;incluir la aptitud
     shl eax,3             ;EAX=EAX * 8 bytes en cada double  = Bytes por
individuo
     mov ebx,eax           ;EBX = Bytes por individuo

     call gsl_rng_uniform_pos    ;UNI
     fstp qword [AuxDQ]          ;lambda = UNI;  //Obtener el factor de
ponderación

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

```
     fld qword [CuantosCruzar]
     fistp dword [AuxDD]         ;poner CuantosCruzar a entero

     ;for(i=0;i < CuantosCruzar;i++)
     mov ecx, dword [AuxDD]
CicloCuantosCruzar:
     push ecx
     push esi             ;Guardar apuntadores al inicio de la poblacion
     push edi             ;
     ;Calcular j=UNI*TamPoblacion
     call gsl_rng_uniform_pos    ;UNI
     fimul dword [TamPoblacion]  ;UNI*TamPoblacion
     fistp dword [AuxDD]         ;j=UNI*TamPoblacion
     mov eax,dword [AuxDD]
     mul ebx                     ;j*Bytes por individuo
     add esi,eax                 ;ESI apunta al individuo j
     ;Calcular k=UNI*TamPoblacion
     call gsl_rng_uniform_pos    ;UNI
     fimul dword [TamPoblacion]  ;UNI*TamPoblacion
     fistp dword [AuxDD]         ;k=UNI*TamPoblacion
     mov eax,dword [AuxDD]
     mul ebx                     ;k*Bytes por individuo
     add edi,eax                 ;EDI apunta al individuo k

     push esi                    ;Guardar apuntador al individuo j
     push edi                    ;Guardar apuntador al individuo k
     mov ecx, dword [Dimensiones]
CicloNuevosX:
     ;NuevoX1 = psig[j].x[d]*lambda+psig[k].x[d]*(1-lambda);
     fld qword [AuxDQ]           ;lamda
     fmul qword [esi]            ;psig[j].x[d]*lambda
     fld1                        ;1, psig[j].x[d]*lambda
     fsub qword [AuxDQ]          ;1-lambda, psig[j].x[d]*lambda
     fmul qword [edi]            ;psig[k].x[d]*(1-lambda),
psig[j].x[d]*lambda
     faddp                       ;NuevoX1 = psig[k].x[d]*(1-lambda) +
psig[j].x[d]*lambda
     ;fstp qword [esi]             ;psig[j].x[d] = NuevoX1;

     ;NuevoX2 = psig[j].x[d]*(1-lambda)+psig[k].x[d]*lambda;
     fld1                        ;1, NuevoX1
     fsub qword [AuxDQ]          ;1-lamda, NuevoX1
     fmul qword [esi]            ;psig[j].x[d]*(1-lambda), NuevoX1
     fld qword [AuxDQ]           ;lambda, psig[j].x[d]*(1-lambda), NuevoX1
     fmul qword [edi]            ;psig[k].x[d]*lambda,
psig[j].x[d]*(1-lambda), NuevoX1
     faddp                       ;NuevoX2 = psig[k].x[d]*lambda +
psig[j].x[d]*(1-lambda), NuevoX1
     fstp qword [edi]            ;psig[k].x[d] = NuevoX2;
     fstp qword [esi]            ;psig[j].x[d] = NuevoX1;

     add esi,8
     add edi,8
     loop CicloNuevosX

     pop edi        ;Restaurar apuntador al individuo k
     pop esi        ;Restaurar apuntador al individuo j
```

```
     ;psig[j].ap = Aptitud(psig[j].x);
     pushad
     push esi       ;Enviar parámetro apuntador al individuo j
     call Aptitud
     add esp,4
     popad
     add esi,ebx
     sub esi,8
     fstp qword [esi]            ;psig[j].ap = Aptitud(psig[j].x)

     ;psig[k].ap = Aptitud(psig[k].x);
     pushad
     push edi       ;Enviar parámetro apuntador al individuo k
     call Aptitud
     add esp,4
     popad
     add edi,ebx
     sub edi,8
     fstp qword [edi]            ;psig[k].ap = Aptitud(psig[k].x)

     pop edi           ;Restaurar apuntadores al inicio de la poblacion
     pop esi           ;
     pop ecx           ;Restaurar contador del ciclo
     dec ecx
     jecxz FinCicloCuantosCruzar
     jmp CicloCuantosCruzar
FinCicloCuantosCruzar:
     leave
     ret
;*** Fin Cruce ***
```

## C.6 Operador Mutacion()

```
;***Mutacion
*********************************************************************
MUTACION *********
;***Definición: void Mutacion(struct individuo *, int)
;***Aplica la mutación sólo a los no elitistas
;***El primer parámetro apunta a la poblacion.
;***El segundo es el valor para g
;***Esta función requiere la declaración, en el módulo en C, de las
siguientes variables globales:
;***int TamPoblacion; int Dimensiones; int Elitismo; int MaxGeneraciones;
int Xmax;
;***double ProbMutacion
Mutacion:
     push ebp
     mov ebp,esp
     sub esp,100h         ;Espacio para uso local de la pila
     mov esi,[ebp+8]      ;ESI = Apuntador al arreglo de la poblacion
```

```
     mov edx,dword [ebp+12]     ;EDX = g   (como dword ya que g es int)

     ;float m,h;
     ;int i,gen;
     ;for (i=ELITISMO;i < TamPoblacion;i++)  //pueden mutar solo los que no
son elitistas
     ;{
     ;   m = UNI; //Para decidir si el individuo muta en base a ProbMutacion
     ;   if(m <= ProbMutacion)
     ;   {
     ;      gen = UNI*DIMENSIONES; //Para decidir cual gen mutar x[0],
x[1]....x[gen]
     ;      h = UNI;               //Para saber si incrementa o decrementa
     ;      if (h < 0.5)
     ;         psig[i].x[gen] = psig[i].x[gen]+
(UNI*(XMAX-psig[i].x[gen]))*(1-g/MaxGeneraciones);
     ;      else
     ;         psig[i].x[gen] = psig[i].x[gen]-(UNI*(psig[i].x[gen]
+XMAX))*(1-g/MaxGeneraciones);
     ;      psig[i].ap = Aptitud(psig[i].x);
     ;   }
     ;}

     push edx     ;respaldar g
     ;Respaldar apuntador al inicio de la poblacion
     mov ecx,esi
     ;Calcular cantidad de doubles en los elitistas
     ;Bytes Por Individuo
     mov eax, dword [Dimensiones]
     inc eax               ;incluir la aptitud
     shl eax,3             ;EAX=EAX * 8 bytes en cada double  = Bytes por
individuo
     push eax
     mul dword [Elitismo]  ;EAX=bytes en los elititistas
     add esi,eax           ;ESI apunta al primer individuo después de
ELITISMO
     pop ebx               ;EBX = Bytes por individuo
     ;Calcular bytes totales en la poblacion (Bytes por individuo *
TamPoblacion)
     mov eax,ebx
     mul dword [TamPoblacion]
     ;EAX = Total de bytes en la poblacion
     add eax,ecx           ;EAX=limite para el ciclo Selecciona

     pop edx       ;restaurar g
     ;for (i=ELITISMO;i < TamPoblacion;i++)  //pueden mutar solo los que no
son elitistas
CicloMutacion:
     push eax
     push esi             ;Guardar apuntador al individuo i

     call gsl_rng_uniform_pos    ;m=UNI
     ;if(m <= ProbMutacion)
     fcomp qword [ProbMutacion]
     fstsw ax                    ;Carga en ax los bits de estado de la FPU
     sahf                        ;Guarda AH en el registro de banderas
     ja SigueCicloMutacion       ;Si m > ProbMutacion, salta al fin del if
```

```
     push esi             ;Otra vez

     ;gen = UNI*DIMENSIONES; //Para decidir cual gen mutar x[0],
x[1]....x[gen]
     call gsl_rng_uniform_pos    ;UNI
     fimul dword [Dimensiones]   ;UNI*Dimensiones
     fistp dword [AuxDD]         ;gen=UNI*Dimensiones
     mov eax,dword [AuxDD]
     shl eax,3                   ;EAX=EAX * 8 bytes en cada double  = Apunta
al gen a mutar
     add esi,eax                 ;ESI apunta al gen a mutar

     mov dword [AuxDD],edx       ;AuxDD = g

     call gsl_rng_uniform_pos    ;h = UNI;               //Para saber si
incrementa o decrementa
     ;if (h < 0.5)
     fcomp qword [PuntoCinco]
     fstsw ax                    ;Carga en ax los bits de estado de la FPU
     sahf                        ;Guarda AH en el registro de banderas
     jae SigueDecrementa         ;Si h >= 0.5, salta al else
     ;psig[i].x[gen] = psig[i].x[gen]+
(UNI*(XMAX-psig[i].x[gen]))*(1-g/MaxGeneraciones);
     fild dword [Xmax]           ;XMAX
     fsub qword [esi]            ;XMAX-psig[i].x[gen]
     call gsl_rng_uniform_pos    ;UNI, (XMAX-psig[i].x[gen])
     fmulp                       ;(UNI*(XMAX-psig[i].x[gen]))
     fild dword [AuxDD]          ;g, (UNI*(XMAX-psig[i].x[gen]))
     fidiv dword [MaxGeneraciones] ;g/MaxGeneraciones,
(UNI*(XMAX-psig[i].x[gen]))
     fld1                        ;1, g/MaxGeneraciones,
(UNI*(XMAX-psig[i].x[gen]))
     fsubrp                      ;(1-g/MaxGeneraciones),
(UNI*(XMAX-psig[i].x[gen]))
     fmulp                       ;
(UNI*(XMAX-psig[i].x[gen]))*(1-g/MaxGeneraciones)
     fadd qword [esi]            ;psig[i].x[gen]+
(UNI*(XMAX-psig[i].x[gen]))*(1-g/MaxGeneraciones)
     ;Termina cálculo: psig[i].x[gen]+
(UNI*(XMAX-psig[i].x[gen]))*(1-g/MaxGeneraciones);
     jmp MutCalculaAptitud
SigueDecrementa:
     ;else
     ;psig[i].x[gen] = psig[i].x[gen]-(UNI*(psig[i].x[gen]
+XMAX))*(1-g/MaxGeneraciones);
     fild dword [Xmax]           ;XMAX
     fadd qword [esi]            ;psig[i].x[gen]+XMAX
     call gsl_rng_uniform_pos    ;UNI, (psig[i].x[gen]+XMAX)
     fmulp                       ;(UNI*(psig[i].x[gen]+XMAX))
     fild dword [AuxDD]          ;g, (UNI*(psig[i].x[gen]+XMAX))
     fidiv dword [MaxGeneraciones] ;g/MaxGeneraciones, (UNI*(psig[i].x[gen]
+XMAX))
     fld1                        ;1, g/MaxGeneraciones, (UNI*(psig[i].x[gen]
+XMAX))
     fsubrp                      ;(1-g/MaxGeneraciones),
(UNI*(psig[i].x[gen]+XMAX))
```

```
     fmulp                        ;(UNI*(psig[i].x[gen]
+XMAX))*(1-g/MaxGeneraciones)
     fsubr qword [esi]            ;psig[i].x[gen]-(UNI*(psig[i].x[gen]
+XMAX))*(1-g/MaxGeneraciones)
     ;Termina cálculo: psig[i].x[gen]-(UNI*(psig[i].x[gen]
+XMAX))*(1-g/MaxGeneraciones);

MutCalculaAptitud:
     ;psig[i].x[gen] = Resultado del cálculo anterior (Guardar el gen
mutado)
     fstp qword [esi]
     pop esi              ;Restaurar apuntador al individuo i

     ;psig[i].ap = Aptitud(psig[i].x);
     pushad
     push esi             ;Enviar parámetro apuntador al individuo i
     call Aptitud
     add esp,4
     popad
     add esi,ebx
     sub esi,8
     fstp qword [esi]     ;psig[i].ap = Aptitud(psig[i].x)

SigueCicloMutacion:
     pop esi
     pop eax
     add esi,ebx          ;Avanzar ESI al sig individuo
     cmp esi,eax
     jl CicloMutacion
     leave
     ret
;*** Fin Mutacion ***
```

# Apéndice D

## Funciones de aptitud en lenguaje ensamblador del algoritmo AG-DM optimizado

## D.1 Rastrigin()

```
;rastrigin.inc
;Función 9, Pag 12 CEC05 ****************************** RASTRIGIN ********

section .data
       Diez  dq 10.0

segment .text
Aptitud:
     push ebp
     mov ebp,esp
     mov esi,[ebp+8]      ;EBX = Apuntador al arreglo

    ;int i;
    ;double res;
    ;evaluaciones++;
    ;res = 0.0;
    ;for (i=0; i < DIMENSIONES; i++)
    ;{
    ;    res += (x[i]*x[i] - 10.0 * cos(2.0*PI*x[i]) + 10.0);
    ;}
    ;return (-res);

     mov ecx,DIMENSIONES
     fldz               ;ST0 = 0
CicloSuma:
     jecxz Salir
     dec ecx

     fldpi                        ;Pi, 0
     fmul qword [esi + ecx * 8]   ;Pi*x[i], 0
     fld ST0                      ;Pi*x[i], Pi*x[i], 0
     faddp                        ;2.0*Pi*x[i], 0
     fcos                         ;cos(2.0*Pi*x[i]), 0
     fmul qword [Diez]            ;10*cos(2.0*Pi*x[i]), 0
     fld qword [esi + ecx * 8]    ;x[i], 10*cos(2.0*Pi*x[i]), 0
     fld ST0                      ;x[i], x[i], 10*cos(2.0*Pi*x[i]), 0
     fmulp                        ;x[i]*x[i], 10*cos(2.0*Pi*x[i]), 0
     fsubrp                       ;x[i]*x[i] - 10*cos(2.0*Pi*x[i]), 0
     fadd qword [Diez]            ;x[i]*x[i] - 10*cos(2.0*Pi*x[i]) + 10.0, 0
     faddp                        ;x[i]*x[i] - 10*cos(2.0*Pi*x[i]) + 10.0
;--> Va acumulando la suma
     jmp CicloSuma
```

```
Salir:
     fchs
     inc dword [evaluaciones]
     leave
     ret

;Fin de rastrigin.inc
```

## D.2 Ackley()

```
;acley.inc
;Función 8, Pag 11 CEC05 ****************************** ACKLEY    ********

section .data
     Veinte  dd 20
    PuntoDos dq 0.2
          E  dq 2.71828182845904523536028747135266 25

segment .text
Aptitud:
     push ebp
     mov ebp,esp
     mov esi,[ebp+8]      ;EBX = Apuntador al arreglo

    ;int i;
    ;double sum1, sum2, res;
    ;evaluaciones++;
    ;sum1 = 0.0;
    ;sum2 = 0.0;
    ;for (i=0; i<DIMENSIONES; i++)
    ;{
    ;    sum1 += x[i]*x[i];
    ;    sum2 += cos(2.0*PI*x[i]);
    ;}
    ;sum1 = -0.2*sqrt(sum1/DIMENSIONES);
    ;sum2 /= DIMENSIONES;
    ;res = 20.0 + E - 20.0*exp(sum1) - exp(sum2);
    ;return (-res);

     inc dword [evaluaciones]

     finit

     fild dword [Veinte]          ;20
     fadd qword [E]               ;20+E
     fild dword [Veinte]          ;20, 20+E

     ;Calcular sum1
     mov ecx,DIMENSIONES
     fldz                         ;0, 20, 20+E
CicloSuma1:
     jecxz Salir1
     dec ecx
```

```
     fld qword [esi + ecx * 8]    ;x[i], 0, 20, 20+E
     fld ST0                      ;x[i], x[i], 0, 20, 20+E
     fmulp                        ;x[i]*x[i], 0, 20, 20+E
     faddp                        ;x[i]*x[i], 20, 20+E   ;--> en cada
iteración va acumulando la suma
                                                          ;    sum1 +=
x[i]*x[i];
     jmp CicloSuma1
Salir1:
     mov dword [Veinte],DIMENSIONES
     fidiv dword [Veinte]          ;sum1/DIMENSIONES, 20, 20+E
     fsqrt                        ;sqrt(sum1/DIMENSIONES), 20, 20+E
     fmul qword [PuntoDos]        ;0.2*sqrt(sum1/DIMENSIONES), 20, 20+E
     fchs                         ;-0.2*sqrt(sum1/DIMENSIONES), 20, 20+E
                                  ;sum1, 20, 20+E
     ;exp(sum1)
     ;e^sum1 = 2 ^ (sum1*Lg(E))
     ;nota: Lg es Logaritmo base 2
     fldl2e                       ;Lg(E), sum1, 20, 20+E
     fmulp                        ;sum1*Lg(E), 20, 20+E
     ;Consideremos x=sum1*Lg(E) para expresarlo en los comentarios como 2^x
     fld ST0                      ;x, x, 20, 20+E
     frndint                      ;int(x), x, 20, 20+E
     fxch                         ;x, int(x), 20, 20+E
     fsub ST0,ST1                 ;frac(x), int(x), 20, 20+E
     f2xm1                        ;(2^(frac(x)))-1, int(x), 20, 20+E
     fld1                         ;1, (2^(frac(x)))-1, int(x), 20, 20+E
     faddp                        ;2^(frac(x)), int(x), 20, 20+E
     fscale                       ;2^x, int(x), 20, 20+E
     fstp ST1                     ;2^x, 20, 20+E
                                  ;exp(sum1), 20, 20+E
     fmulp                        ;exp(sum1)*20, 20+E
     fsubp                        ;20+E - 20*exp(sum1)

     ;Calcular sum2
     mov ecx,DIMENSIONES
     fldz                         ;0, 20+E - 20*exp(sum1)
CicloSuma2:
     jecxz Salir2
     dec ecx

     fldpi                        ;Pi, 0, 20+E - 20*exp(sum1)
     fmul qword [esi + ecx * 8]   ;Pi*x[i], 0, 20+E - 20*exp(sum1)
     fld ST0                      ;Pi*x[i], Pi*x[i], 0, 20+E - 20*exp(sum1)
     faddp                        ;2.0*Pi*x[i], 0, 20+E - 20*exp(sum1)
     fcos                         ;cos(2.0*Pi*x[i]), 0, 20+E - 20*exp(sum1)
     faddp                        ;sum2, 20+E - 20*exp(sum1)
     jmp CicloSuma2
Salir2:
     fidiv dword [Veinte]         ;sum2/DIMENSIONES, 20+E - 20*exp(sum1)
                                  ;sum2, 20+E - 20*exp(sum1)
     ;exp(sum2)
     ;e^sum2 = 2 ^ (sum2*Lg(E))
     fldl2e                       ;Lg(E), sum2, 20+E - 20*exp(sum1)
     fmulp                        ;sum2*Lg(E), 20+E - 20*exp(sum1)
     ;Consideremos x=sum2*Lg(E) para expresarlo en los comentarios como 2^x
```

```
     fld ST0                        ;x, x, 20+E - 20*exp(sum1)
     frndint                        ;int(x), x, 20+E - 20*exp(sum1)
     fxch                           ;x, int(x), 20+E - 20*exp(sum1)
     fsub ST0,ST1                   ;frac(x), int(x), 20+E - 20*exp(sum1)
     f2xm1                          ;(2^(frac(x)))-1, int(x), 20+E -
20*exp(sum1)
     fld1                           ;1, (2^(frac(x)))-1, int(x), 20+E -
20*exp(sum1)
     faddp                          ;2^(frac(x)), int(x), 20+E - 20*exp(sum1)
     fscale                         ;2^x, int(x), 20+E - 20*exp(sum1)
     fstp ST1                       ;2^x, 20+E - 20*exp(sum1)
                                    ;exp(sum2), 20+E - 20*exp(sum1)
     fsubp                          ;20+E - 20*exp(sum1) - exp(sum2)
     fchs
     leave
     ret

;Fin de acley.inc
```

## D.3 Schwefel()

```
;schwefel.inc
;Función 2, Pag 5 CEC05 ****************************** SCHWEFEL  ********

segment .text
Aptitud:
     push ebp
     mov ebp,esp
     mov esi,[ebp+8]      ;EBX = Apuntador al arreglo

     ;int i, j;
     ;long double sum1, sum2;
     ;sum1 = 0.0;
     ;evaluaciones++;
     ;for (i=0; i<DIMENSIONES; i++)
     ;{
     ;   sum2 = 0.0;
     ;   for (j=0; j<=i; j++)
     ;   {
     ;       sum2 += x[j];
     ;   }
     ;   sum1 += sum2*sum2;
     ;}
     ;return (-sum1);

     inc dword [evaluaciones]
     mov ecx,DIMENSIONES
     fldz               ;ST0 = 0     ;sum1 = 0.0;
CicloSuma1:
     jecxz Salir1
     dec ecx
     xor eax,eax
     fldz               ;ST0 = 0     ;sum2 = 0.0;
```

```
CicloSuma2:
     fld qword [esi + eax * 8]    ;x[j], sum2, sum1          ;En las demas
pasadas, en lugar de 0 sera el acumulado
     faddp                        ;x[j]+sum2, sum1
     inc eax
     cmp eax,ecx
     jle CicloSuma2
     fld ST0                      ;sum2,sum2,sum1
     fmulp                        ;sum2*sum2,sum1
     faddp                        ;sum1                  (sum1 +=
sum2*sum2;)

     jmp CicloSuma1
Salir1:
     fchs                         ;-sum1
     leave
     ret

;Fin de schwefel.inc
```

## D.4 Weierstrass()

```
;weierstrass.inc
;Función 11, Pag 14 CEC05 ****************************** WEIERSTRASS
********

section .data
   AuxJ dd 0

segment .text
Aptitud:
     push ebp
     mov ebp,esp
     mov esi,[ebp+8]      ;EBX = Apuntador al arreglo

     ;int i, j;
     ;double res;
     ;double sum;
     ;double a, b;
     ;int k_max;
     ;evaluaciones++;
     ;a = 0.5;
     ;b = 3.0;
     ;k_max = 20;
     ;res = 0.0;
     ;for (i=0; i<DIMENSIONES; i++)
     ;{
     ;   sum = 0.0;
     ;   for (j=0; j<=k_max; j++)
     ;   {
     ;       sum += pow(a,j)*cos(2.0*PI*pow(b,j)*(x[i]+0.5));
     ;   }
     ;   res += sum;
```

```
     ;}
     ;sum = 0.0;
     ;for (j=0; j<=k_max; j++)
     ;{
     ;   sum += pow(a,j)*cos(2.0*PI*pow(b,j)*(0.5));
     ;}
     ;res-=DIMENSIONES*sum;
     ;return (-res);

     inc dword [evaluaciones]
     mov ecx,DIMENSIONES
     dec ecx
     fldz                          ;ST0 = 0   ;res=0;
CicloDim:
     fldz                          ;0, 0      ;sum=0
     mov ebx,20 ;k_max
CicloK1:
     mov dword [AuxJ],ebx
     ;**Cálculo de pow(b,j) = 2^(j*Lg(b))
     ;**nota: Lg es Logaritmo base 2
     ;Cargar el exponente j
     fild dword [AuxJ]             ;j, sum, res
     ;Cargar la base b=3
     fld1                          ;1, j, sum, res
     fld1                          ;1, 1, j, sum, res
     fld1                          ;1, 1, 1, j, sum, res
     faddp                         ;2, 1, j, sum, res
     faddp                         ;3, j, sum, res
     fyl2x                         ;j*Lg(b), sum, res
     ;Consideremos en los comentarios x=j*Lg(b)
     fld ST0                       ;x, x, sum, res
     frndint                       ;int(x), x, sum, res
     fxch                          ;x, int(x), sum, res
     fsub ST0,ST1                  ;frac(x), int(x), sum, res
     f2xm1                         ;(2^(frac(x)))-1, int(x), sum, res
     fld1                          ;1, (2^(frac(x)))-1, int(x), sum, res
     faddp                         ;2^(frac(x)), int(x), sum, res
     fscale                        ;2^x, int(x), sum, res
     fstp ST1                      ;pow(b,j), sum, res
     ;**Termina cálculo de pow(b,j)

     fldpi                         ;PI, pow(b,j) sum, res
     fmulp                         ;PI*pow(b,j), sum, res
     fld ST0                       ;PI*pow(b,j), PI*pow(b,j), sum, res
     faddp                         ;2*PI*pow(b,j), sum, res
     fld1                          ;1, 2*PI*pow(b,j), sum, res
     fld1                          ;1, 1, 2*PI*pow(b,j), sum, res
     fld1                          ;1, 1, 1, 2*PI*pow(b,j), sum, res
     faddp                         ;2, 1, 2*PI*pow(b,j), sum, res
     fdivp                         ;0.5, 2*PI*pow(b,j), sum, res
     fld qword [esi + ecx*8]       ;x[i],0.5, 2*PI*pow(b,j), sum, res
     faddp                         ;x[i]+0.5, 2*PI*pow(b,j), sum, res
     fmulp                         ;2*PI*pow(b,j)*(x[i]+0.5), sum, res
     fcos                          ;cos(2*PI*pow(b,j)*(x[i]+0.5)), sum, res

     ;**Cálculo de pow(a,j) = 2^(j*Lg(a))
     ;**nota: Lg es Logaritmo base 2
```

```
     ;Cargar el exponente j
     fild dword [AuxJ]          ;j, cos(2*PI*pow(b,j)*(x[i]+0.5)), sum, res
     ;Cargar la base a=0.5
     fld1                     ;1, j, cos(2*PI*pow(b,j)*(x[i]+0.5)), sum, res
     fld1                  ;1, 1, j, cos(2*PI*pow(b,j)*(x[i]+0.5)), sum, res
     fld1               ;1, 1, 1, j, cos(2*PI*pow(b,j)*(x[i]+0.5)), sum, res
     faddp              ;2, 1, j, cos(2*PI*pow(b,j)*(x[i]+0.5)), sum, res
     fdivp              ;0.5, j, cos(2*PI*pow(b,j)*(x[i]+0.5)), sum, res
     fyl2x              ;j*Lg(a), cos(2*PI*pow(b,j)*(x[i]+0.5)), sum, res
     ;Consideremos en los comentarios x=j*Lg(a)
     fld ST0            ;x, x, cos(2*PI*pow(b,j)*(x[i]+0.5)), sum, res
     frndint            ;int(x), x, cos(2*PI*pow(b,j)*(x[i]+0.5)), sum, res
     fxch               ;x, int(x), cos(2*PI*pow(b,j)*(x[i]+0.5)), sum, res
     fsub ST0,ST1     ;frac(x),int(x),cos(2*PI*pow(b,j)*(x[i]+0.5)),sum,res
     f2xm1            ;(2^(frac(x)))-1, int(x), cos(2*PI*pow(b,j)*(x[i]
+0.5)), sum, res
     fld1             ;1, (2^(frac(x)))-1, int(x), cos(2*PI*pow(b,j)*(x[i]
+0.5)), sum, res
     faddp            ;2^(frac(x)), int(x), cos(2*PI*pow(b,j)*(x[i]+0.5)),
sum, res
     fscale           ;2^x, int(x), cos(2*PI*pow(b,j)*(x[i]+0.5)), sum, res
     fstp ST1         ;pow(a,j), cos(2*PI*pow(b,j)*(x[i]+0.5)), sum, res
     ;**Termina cálculo de pow(a,j)
     fmulp            ;pow(a,j)*cos(2*PI*pow(b,j)*(x[i]+0.5)), sum, res
     faddp            ;pow(a,j)*cos(2*PI*pow(b,j)*(x[i]+0.5)), res

     dec ebx
     jns CicloK1    ;Si >= 0, sigue en el ciclo

     faddp                          ;res+=sum

     dec ecx
     js FinCicloDim ;Si < 0, sale
     jmp CicloDim
FinCicloDim:

     fldz                           ;sum=0, res
     mov ebx,20 ;k_max
CicloK2:
     ;**Cálculo de pow(b,j) = 2^(j*Lg(b))
     ;**nota: Lg es Logaritmo base 2
     ;Cargar el exponente j
     mov dword [AuxJ],ebx
     fild dword [AuxJ]              ;j, sum, res
     ;Cargar la base b=3
     fld1                           ;1, j, sum, res
     fld1                           ;1, 1, j, sum, res
     fld1                           ;1, 1, 1, j, sum, res
     faddp                          ;2, 1, j, sum, res
     faddp                          ;3, j, sum, res
     fyl2x                          ;j*Lg(b), sum, res
     ;Consideremos en los comentarios x=j*Lg(b)
     fld ST0                        ;x, x, sum, res
     frndint                        ;int(x), x, sum, res
     fxch                           ;x, int(x), sum, res
     fsub ST0,ST1                   ;frac(x), int(x), sum, res
     f2xm1                          ;(2^(frac(x)))-1, int(x), sum, res
```

```
     fld1                          ;1, (2^(frac(x)))-1, int(x), sum, res
     faddp                         ;2^(frac(x)), int(x), sum, res
     fscale                        ;2^x, int(x), sum, res
     fstp ST1                      ;pow(b,j), sum, res
     ;**Termina cálculo de pow(b,j)

     fldpi                         ;PI, pow(b,j) sum, res
     fmulp                         ;PI*pow(b,j), sum, res
     fld ST0                       ;PI*pow(b,j), PI*pow(b,j), sum, res
     faddp                         ;2*PI*pow(b,j), sum, res
     fld1                          ;1, 2*PI*pow(b,j), sum, res
     fld1                          ;1, 1, 2*PI*pow(b,j), sum, res
     fld1                          ;1, 1, 1, 2*PI*pow(b,j), sum, res
     faddp                         ;2, 1, 2*PI*pow(b,j), sum, res
     fdivp                         ;0.5, 2*PI*pow(b,j), sum, res
     fmulp                         ;2*PI*pow(b,j)*(0.5), sum, res
     fcos                          ;cos(2*PI*pow(b,j)*(0.5)), sum, res

     ;**Cálculo de pow(a,j) = 2^(j*Lg(a))
     ;**nota: Lg es Logaritmo base 2
     ;Cargar el exponente j
     fild dword [AuxJ]             ;j, cos(2*PI*pow(b,j)*(0.5)), sum, res
     ;Cargar la base a=0.5
     fld1                       ;1, j, cos(2*PI*pow(b,j)*(0.5)), sum, res
     fld1                       ;1, 1, j, cos(2*PI*pow(b,j)*(0.5)), sum, res
     fld1                  ;1, 1, 1, j, cos(2*PI*pow(b,j)*(0.5)), sum, res
     faddp                 ;2, 1, j, cos(2*PI*pow(b,j)*(0.5)), sum, res
     fdivp                 ;0.5, j, cos(2*PI*pow(b,j)*(0.5)), sum, res
     fyl2x                 ;j*Lg(a), cos(2*PI*pow(b,j)*(0.5)), sum, res
     ;Consideremos en los comentarios x=j*Lg(a)
     fld ST0               ;x, x, cos(2*PI*pow(b,j)*(0.5)), sum, res
     frndint               ;int(x), x, cos(2*PI*pow(b,j)*(0.5)), sum, res
     fxch                  ;x, int(x), cos(2*PI*pow(b,j)*(0.5)), sum, res
     fsub ST0,ST1       ;frac(x), int(x), cos(2*PI*pow(b,j)*(0.5)), sum, res
     f2xm1              ;(2^(frac(x)))-1, int(x), cos(2*PI*pow(b,j)*(0.5)),
sum, res
     fld1               ;1, (2^(frac(x)))-1, int(x),
cos(2*PI*pow(b,j)*(0.5)), sum, res
     faddp          ;2^(frac(x)), int(x), cos(2*PI*pow(b,j)*(0.5)), sum, res
     fscale         ;2^x, int(x), cos(2*PI*pow(b,j)*(0.5)), sum, res
     fstp ST1       ;pow(a,j), cos(2*PI*pow(b,j)*(0.5)), sum, res
     ;**Termina cálculo de pow(a,j)
     fmulp          ;pow(a,j)*cos(2*PI*pow(b,j)*(0.5)), sum, res
     faddp          ;pow(a,j)*cos(2*PI*pow(b,j)*(0.5)), res

     dec ebx
     jns CicloK2   ;Si >= 0, sigue en el ciclo

     mov dword [AuxJ],DIMENSIONES
     fild dword [AuxJ]             ;DIMENSIONES, sum, res
     fmulp                         ;DIMENSIONES*sum, res
     fsubp                         ;res - DIMENSIONES*sum

     fchs                          ;-res
     leave
     ret
```

```
;Fin de weierstrass.inc
```

# Apéndice E

## Código en lenguaje ensamblador para el generador de números pseudoaleatorios MT19937

```
;  mt19937.asm

;  Implementación en lenguaje ensamblador de las rutinas de generación
;  de números pseudoaleatorios (mt19937)
;  a utilizar en el programa ag-dm-esc.c, que es el módulo esclavo
;  del Algoritmo Genetico Distribuido con migración.

;  Definiciones:

;                unsigned long gsl_rng_uniform(void)
;                double gsl_rng_uniform_pos(void)
;                static void gsl_rng_set(unsigned long int semilla);

;  Ensamblar:      nasm -f elf mt.asm

;  Copyright (C) 2011 Jorge Bravo Espinoza   bravo_sistemas@live.com.mx

;  This program is free software; you can redistribute it and/or modify
;  it under the terms of the GNU General Public License as published by
;  the Free Software Foundation; either version 2 of the License, or
;  (at your option) any later version.

;  This program is distributed in the hope that it will be useful,
;  but WITHOUT ANY WARRANTY; without even the implied warranty of
;  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
;  GNU General Public License for more details.

;  You should have received a copy of the GNU General Public License
;  along with this program; if not, write to the Free Software
;  Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301
USA

%define MASCARA_SUP 80000000h     ;Bits w-r mas significativos
%define MASCARA_INF 7fffffffh     ;Bits r menos significativos

global gsl_rng_uniform
global gsl_rng_uniform_pos
global gsl_rng_set

section .data
    DivisorDouble dq 4294967296.0
```

```
section .bss
     mt resd 624
     mti resd 1
     ValorUniform resq 1


segment .text

;
*************************************************************************
**** gsl_rng_uniform ***
gsl_rng_uniform:
     pushad
     cmp dword [mti],624
     jl Tempering

     ;Generar 624 words
     ;for (kk = 0; kk < N - M; kk++)
     xor esi,esi
CicloN_M:
     ;unsigned long y = (mt[kk] & UPPER_MASK) | (mt[kk + 1] & LOWER_MASK);
     mov eax,[mt + esi * 4]
     mov ebx,[mt + 4 + esi * 4]
     and eax,MASCARA_SUP
     and ebx,MASCARA_INF
     or eax,ebx            ;eax es y
     ;mt[kk] = mt[kk + M] ^ (y >> 1) ^ MAGIC(y);
     mov ebx,eax
     and ebx,1
     neg ebx
     and ebx,09908b0dfh
     shr eax,1
     xor eax,ebx
     xor eax,[mt + 1588 + esi * 4]   ;1588 = M * 4  (397*4)

     mov [mt + esi * 4],eax

     inc esi
     cmp esi,227
     jl CicloN_M

     ;for (; kk < N - 1; kk++)
CicloN_1:
     ;unsigned long y = (mt[kk] & UPPER_MASK) | (mt[kk + 1] & LOWER_MASK);
     mov eax,[mt + esi * 4]
     mov ebx,[mt + 4 + esi * 4]
     and eax,MASCARA_SUP
     and ebx,MASCARA_INF
     or eax,ebx            ;eax es y
     ;mt[kk] = mt[kk + (M - N)] ^ (y >> 1) ^ MAGIC(y);
     mov ebx,eax
     shr eax,1
     and ebx,1
     neg ebx
     and ebx,09908b0dfh
     xor eax,ebx
     mov ebx,[mt - 908 + esi * 4]    ;908 = (M - N) * 4 =  (397 - 624) * 4
```

```
     xor eax,ebx

     mov [mt + esi * 4],eax

     inc esi
     cmp esi,623
     jl CicloN_1

     ;unsigned long y = (mt[N - 1] & UPPER_MASK) | (mt[0] & LOWER_MASK);
     mov eax,[mt + 2492]
     mov ebx,[mt]
     and eax,MASCARA_SUP
     and ebx,MASCARA_INF
     or eax,ebx            ;eax es y
     ;mt[N - 1] = mt[M - 1] ^ (y >> 1) ^ MAGIC(y);
     mov ebx,eax
     shr eax,1
     and ebx,1
     neg ebx
     and ebx,09908b0dfh
     xor eax,ebx
     mov ebx,[mt + 1584]
     xor eax,ebx
     mov [mt + 2492],eax

     ;state->mti = 0
     mov dword [mti],0

Tempering:
     ;k = mt[state->mti]
     mov ecx,[mti]
     mov ebx,[mt + ecx * 4]
     ;k ^= (k >> 11)
     mov eax,ebx
     shr ebx,11
     xor eax,ebx
     ;k ^= (k << 7) & 0x9d2c5680UL
     mov ebx,eax
     shl ebx,7
     and ebx,09d2c5680h
     xor eax,ebx
     ;k ^= (k << 15) & 0xefc60000UL
     mov ebx,eax
     shl ebx,15
     and ebx,0efc60000h
     xor eax,ebx
     ;k ^= (k >> 18)
     mov ebx,eax
     shr ebx,18
     xor eax,ebx

     ;state->mti++
     inc dword [mti]
     popad
     ret         ;********** gsl_rng_uniform
```

```
;
*****************************************************************************
**** gsl_rng_uniform_pos ***
gsl_rng_uniform_pos:
     pushad
     cmp dword [mti],624
     jl Tempering_pos

     ;Generar 624 words
     ;for (kk = 0; kk < N - M; kk++)
     xor esi,esi
CicloN_M_pos:
     ;unsigned long y = (mt[kk] & UPPER_MASK) | (mt[kk + 1] & LOWER_MASK);
     mov eax,[mt + esi * 4]
     mov ebx,[mt + 4 + esi * 4]
     and eax,MASCARA_SUP
     and ebx,MASCARA_INF
     or eax,ebx            ;eax es y
     ;mt[kk] = mt[kk + M] ^ (y >> 1) ^ MAGIC(y);
     mov ebx,eax
     and ebx,1
     neg ebx
     and ebx,09908b0dfh
     shr eax,1
     xor eax,ebx
     xor eax,[mt + 1588 + esi * 4]   ;1588 = M * 4  (397*4)

     mov [mt + esi * 4],eax

     inc esi
     cmp esi,227
     jl CicloN_M_pos

     ;for (; kk < N - 1; kk++)
CicloN_1_pos:
     ;unsigned long y = (mt[kk] & UPPER_MASK) | (mt[kk + 1] & LOWER_MASK);
     mov eax,[mt + esi * 4]
     mov ebx,[mt + 4 + esi * 4]
     and eax,MASCARA_SUP
     and ebx,MASCARA_INF
     or eax,ebx            ;eax es y
     ;mt[kk] = mt[kk + (M - N)] ^ (y >> 1) ^ MAGIC(y);
     mov ebx,eax
     shr eax,1
     and ebx,1
     neg ebx
     and ebx,09908b0dfh
     xor eax,ebx
     mov ebx,[mt - 908 + esi * 4]    ;908 = (M - N) * 4 =  (397 - 624) * 4
     xor eax,ebx

     mov [mt + esi * 4],eax

     inc esi
     cmp esi,623
     jl CicloN_1_pos
```

```
     ;unsigned long y = (mt[N - 1] & UPPER_MASK) | (mt[0] & LOWER_MASK);
     mov eax,[mt + 2492]
     mov ebx,[mt]
     and eax,MASCARA_SUP
     and ebx,MASCARA_INF
     or eax,ebx            ;eax es y
     ;mt[N - 1] = mt[M - 1] ^ (y >> 1) ^ MAGIC(y);
     mov ebx,eax
     shr eax,1
     and ebx,1
     neg ebx
     and ebx,09908b0dfh
     xor eax,ebx
     mov ebx,[mt + 1584]
     xor eax,ebx
     mov [mt + 2492],eax

     ;state->mti = 0
     mov dword [mti],0

Tempering_pos:
     ;k = mt[state->mti]
     mov ecx,[mti]
     mov ebx,[mt + ecx * 4]
     ;k ^= (k >> 11)
     mov eax,ebx
     shr ebx,11
     xor eax,ebx
     ;k ^= (k << 7) & 0x9d2c5680UL
     mov ebx,eax
     shl ebx,7
     and ebx,09d2c5680h
     xor eax,ebx
     ;k ^= (k << 15) & 0xefc60000UL
     mov ebx,eax
     shl ebx,15
     and ebx,0efc60000h
     xor eax,ebx
     ;k ^= (k >> 18)
     mov ebx,eax
     shr ebx,18
     xor eax,ebx

     ;state->mti++
     inc dword [mti]

     mov [ValorUniform],eax
     fild qword [ValorUniform]
     fld qword [DivisorDouble]
     fdiv
     popad
     ret         ;********** gsl_rng_uniform_pos


;
*****************************************************************************
**** gsl_rng_set ***
```

```
gsl_rng_set:
     push ebp
     mov ebp,esp
     mov edx, dword [ebp+8]       ;EDX = Semilla

     ;if (s == 0) s = 4357;   /* the default seed is 4357 */
     cmp edx,0
     jne SigueSet
     mov edx,4357
SigueSet:
     ;state->mt[0]= s & 0xffffffffUL;
     ;and edx,0ffffffffh     ;**** usado solo en cpu x86-64
     mov dword [mt],edx

     ;for (i = 1; i < N; i++)
     mov esi,1
CicloSet:
     ; state->mt[i] = (1812433253UL * (state->mt[i-1] ^ (state->mt[i-1] >>
30)) + i);
     mov ebx,[mt - 4 + esi * 4]
     mov eax,ebx
     shr ebx,30
     xor eax,ebx
     mov ebx,1812433253
     mul ebx
     add eax,esi

     ; state->mt[i] &= 0xffffffffUL;
     ;and eax,0ffffffffh         ;**** usado solo en cpu x86-64
     mov [mt + esi * 4],eax
     inc esi
     cmp esi,624
     jl CicloSet

     ;state->mti = i
     mov dword [mti],624  ;esi
     leave
     ret         ;********** gsl_rng_set

;Fin de mt19937.asm
```

# Apéndice F

# Resultados de las pruebas a los módulos programados en lenguaje ensamblador

## F.1 Ackley

### Pruebas al operador de Aptitud

Función: **Ackley**

Archivo de informe para C: apAckleyC.txt
Archivo de informe para Ensamblador: apAckleyASM.txt
Llamada a la función en C: p.ap = aptitud(p.x);
Llamada a la función en Ensamblador: p.ap = Aptitud(p.x);

| Datos generados en C | Datos generados en ensamblador | Observaciones |
|---|---|---|
| f(x)=-21.105973035274168 | f(x)=-21.105973035274168 | Datos idénticos |
| 30 Dimensiones: | 30 Dimensiones: | |
| -5.213116013 | -5.213116013 | |
| 30.310434625 | 30.310434625 | |
| 15.519155659 | 15.519155659 | |
| -23.817770980 | -23.817770980 | |
| -23.678495407 | -23.678495407 | |

| | | |
|---|---|---|
| -21.076371662 | -21.076371662 | |
| 25.740896471 | 25.740896471 | |
| -5.394317500 | -5.394317500 | |
| 12.554047808 | 12.554047808 | |
| 6.822094686 | 6.822094686 | |
| 10.673255995 | 10.673255995 | |
| 30.198936984 | 30.198936984 | |
| -28.745755449 | -28.745755449 | |
| -15.955529913 | -15.955529913 | |
| -5.833108656 | -5.833108656 | |
| 3.757085867 | 3.757085867 | |
| 20.740018390 | 20.740018390 | |
| -20.388197064 | -20.388197064 | |
| -3.186439782 | -3.186439782 | |
| -3.449999034 | -3.449999034 | |
| 16.076897234 | 16.076897234 | |
| 9.245129786 | 9.245129786 | |
| 4.099974893 | 4.099974893 | |
| -2.634227186 | -2.634227186 | |
| 28.531959586 | 28.531959586 | |
| 13.819054805 | 13.819054805 | |
| 12.812548317 | 12.812548317 | |
| -5.461469136 | -5.461469136 | |
| 1.450770080 | 1.450770080 | |
| 16.197793648 | 16.197793648 | |

## F.2 Rastrigin

### Pruebas al operador de Aptitud

Función: **Rastrigin**

Archivo de informe para C: apRastriginC.txt
Archivo de informe para Ensamblador: apRastriginASM.txt
Llamada a la función en C: p.ap = aptitud(p.x);
Llamada a la función en Ensamblador: p.ap = Aptitud(p.x);

| Datos generados en C | Datos generados en ensamblador | Observaciones |
|---|---|---|
| f(x)=-559.741941753746573 | f(x)=-559.741941753746460 | Iguales hasta 1E-12 |
| 30 Dimensiones: | 30 Dimensiones: | Genes idénticos |
| -0.814549377 | -0.814549377 | |
| 4.736005410 | 4.736005410 | |
| 2.424868072 | 2.424868072 | |
| -3.721526716 | -3.721526716 | |
| -3.699764907 | -3.699764907 | |
| -3.293183072 | -3.293183072 | |
| 4.022015074 | 4.022015074 | |

| | | |
|---|---|---|
| -0.842862109 | -0.842862109 | |
| 1.961569970 | 1.961569970 | |
| 1.065952295 | 1.065952295 | |
| 1.667696249 | 1.667696249 | |
| 4.718583904 | 4.718583904 | |
| -4.491524289 | -4.491524289 | |
| -2.493051549 | -2.493051549 | |
| -0.911423228 | -0.911423228 | |
| 0.587044667 | 0.587044667 | |
| 3.240627873 | 3.240627873 | |
| -3.185655791 | -3.185655791 | |
| -0.497881216 | -0.497881216 | |
| -0.539062349 | -0.539062349 | |
| 2.512015193 | 2.512015193 | |
| 1.444551529 | 1.444551529 | |
| 0.640621077 | 0.640621077 | |
| -0.411597998 | -0.411597998 | |
| 4.458118685 | 4.458118685 | |
| 2.159227313 | 2.159227313 | |
| 2.001960675 | 2.001960675 | |
| -0.853354553 | -0.853354553 | |
| 0.226682825 | 0.226682825 | |
| 2.530905257 | 2.530905257 | |

## F.3 Schwefel

### Pruebas al operador de Aptitud

Función: **Schwefel**

Archivo de informe para C: apSchwefelC.txt
Archivo de informe para Ensamblador: apSchwefelASM.txt
Llamada a la función en C: p.ap = aptitud(p.x);
Llamada a la función en Ensamblador: p.ap = Aptitud(p.x);

| Datos generados en C | Datos generados en ensamblador | Observaciones |
|---|---|---|
| f(x)=-446358.799470973201096 | f(x)=-446358.799470973201096 | Datos idénticos |
| 30 Dimensiones: | 30 Dimensiones: | |
| -16.290987539 | -16.290987539 | |
| 94.720108202 | 94.720108202 | |
| 48.497361434 | 48.497361434 | |
| -74.430534313 | -74.430534313 | |
| -73.995298147 | -73.995298147 | |
| -65.863661445 | -65.863661445 | |
| 80.440301471 | 80.440301471 | |
| -16.857242188 | -16.857242188 | |
| 39.231399400 | 39.231399400 | |

| | | |
|---|---|---|
| 21.319045895 | 21.319045895 | |
| 33.353924984 | 33.353924984 | |
| 94.371678075 | 94.371678075 | |
| -89.830485778 | -89.830485778 | |
| -49.861030979 | -49.861030979 | |
| -18.228464550 | -18.228464550 | |
| 11.740893335 | 11.740893335 | |
| 64.812557469 | 64.812557469 | |
| -63.713115826 | -63.713115826 | |
| -9.957624320 | -9.957624320 | |
| -10.781246983 | -10.781246983 | |
| 50.240303855 | 50.240303855 | |
| 28.891030583 | 28.891030583 | |
| 12.812421541 | 12.812421541 | |
| -8.231959958 | -8.231959958 | |
| 89.162373706 | 89.162373706 | |
| 43.184546265 | 43.184546265 | |
| 40.039213491 | 40.039213491 | |
| -17.067091051 | -17.067091051 | |
| 4.533656500 | 4.533656500 | |
| 50.618105149 | 50.618105149 | |

## F.4 Weierstrass

### Pruebas al operador de Aptitud

Función: **Weierstrass**

Archivo de informe para C: ApWeierstrassC.txt
Archivo de informe para Ensamblador: apWeierstrassASM.txt
Llamada a la función en C: p.ap = aptitud(p.x);
Llamada a la función en Ensamblador: p.ap = Aptitud(p.x);

| Datos generados en C | Datos generados en ensamblador | Observaciones |
|---|---|---|
| f(x)=-62.523093469049527 | f(x)=-62.523093469043481 | Iguales hasta 1E-11 |
| 30 Dimensiones: | 30 Dimensiones: | Genes idénticos |
| -0.162909875 | -0.162909875 | |
| 0.947201082 | 0.947201082 | |
| 0.484973614 | 0.484973614 | |
| -0.744305343 | -0.744305343 | |
| -0.739952981 | -0.739952981 | |
| -0.658636614 | -0.658636614 | |
| 0.804403015 | 0.804403015 | |
| -0.168572422 | -0.168572422 | |
| 0.392313994 | 0.392313994 | |
| 0.213190459 | 0.213190459 | |
| 0.333539250 | 0.333539250 | |

| | | |
|---|---|---|
| 0.943716781 | 0.943716781 | |
| -0.898304858 | -0.898304858 | |
| -0.498610310 | -0.498610310 | |
| -0.182284646 | -0.182284646 | |
| 0.117408933 | 0.117408933 | |
| 0.648125575 | 0.648125575 | |
| -0.637131158 | -0.637131158 | |
| -0.099576243 | -0.099576243 | |
| -0.107812470 | -0.107812470 | |
| 0.502403039 | 0.502403039 | |
| 0.288910306 | 0.288910306 | |
| 0.128124215 | 0.128124215 | |
| -0.082319600 | -0.082319600 | |
| 0.891623737 | 0.891623737 | |
| 0.431845463 | 0.431845463 | |
| 0.400392135 | 0.400392135 | |
| -0.170670911 | -0.170670911 | |
| 0.045336565 | 0.045336565 | |
| 0.506181051 | 0.506181051 | |

## F.5 Operador Selección()

### Pruebas a los operadores

Función: Rastrigin
Dimensiones: 30
Población: 500 individuos

Discrepancias en los archivos de informe generados.

Operador: **Selección**
**Selecciona individuos de una población.**
**Selecciona los primeros individuos (“Elitistas”) de la población ordenada.**
**El resto los selecciona mediante sorteo (hasta completar la nueva población).**

Archivo de informe para C: opSeleccionC.txt
Archivo de informe para Ensamblador: opSeleccionASM.txt
Llamada a la función en C: seleccion();
Llamada a la función en Ensamblador: Seleccion(p,psig);

| Linea | Dato generado en C | Dato generado en ensamblador | Observaciones |
|---|---|---|---|
| 499 | F(15)=-447.615534078694 | f(15)=-447.615534078693 | * |
| 796 | F(24)=-455.833192807747 | f(24)=-455.833192807746 | * |
| 897 | d(0)=4.07133312 | d(0)=4.07133311 | Para f(27) |
| 963 | d(0)=-3.83997155 | d(0)=-3.83997154 | Para f(29) |
| 996 | d(0)=-4.58173710 | d(0)=-4.58173709 | Para f(30) |
| 1095 | d(0)=-4.48906904 | d(0)=-4.48906903 | Para f(33) |
| 1128 | d(0)=-3.29731961 | d(0)=-3.29731962 | Para f(34) |
| 1225 | F(37)=-447.615534078694 | F(37)=-447.615534078693 | |
| 1392 | D(0)=3.11352715 | D(0)=3.11352714 | Para f(42) |
| 1458 | D(0)=-2.69874811 | D(0)=-2.69874810 | Para f(44) |
| 1722 | D(0)=2.04868136 | D(0)=2.04868137 | Para f(52) |
| 1887 | D(0)=0.59313806 | D(0)=0.59313805 | Para f(57) |

| | | | |
|---|---|---|---|
| 1986 | D(0)=-1.97706140 | D(0)=-1.97706139 | Para f(60) |
| 2019 | D(0)=-2.60346506 | D(0)=-2.60346505 | Para f(61) |
| 2085 | D(0)=-4.75300650 | D(0)=-4.75300649 | Para f(63) |
| 2116 | F(64)=-455.833192807747 | f(64)=-455.833192807746 | |
| 2415 | D(0)=4.48971473 | D(0)=4.48971474 | Para f(73) |
| 2481 | D(0)=-0.95186022 | D(0)=-0.95186021 | Para f(75) |
| 2514 | D(0)=4.37475191 | D(0)=4.37475190 | Para f(76) |
| 2613 | D(0)=-0.82140837 | D(0)=-0.82140838 | Para f(79) |
| 2646 | D(0)=-4.94614279 | D(0)=-4.94614280 | Para f(80) |
| 2908 | F(88)=-555.344372325842 | F(88)=-555.344372325843 | |
| 2976 | D(0)=2.03576911 | D(0)=2.03576912 | Para f(90) |
| 3009 | | | Seguir comparando los archivos opSC.txt y opSA.txt |
| | | | De acuerdo a lo observado, las discrepancias no influyen en los |
| | | | resultados. Estas discrepancias son producidas durante la |
| | | | conversión de double a string en la función fprintf de C. |
| | | | El detalle pendiente a revisar es porqué las diferencias en las |
| | | | dimensiones son sólo en la d(0). |

## F.6 Operador Cruce()

### Pruebas a los operadores

Función: Rastrigin
Dimensiones: 30
Población: 500 individuos

Discrepancias en los archivos de informe generados.

Operador: **Cruce**
**Cruce Aritmético: Combina linealmente los vectores de cromosoma de dos padres para generar dos hijos.**

Archivo de informe para C: opCruceC.txt
Archivo de informe para Ensamblador: opCruceASM.txt
Llamada a la función en C: cruce();
Llamada a la función en Ensamblador: Cruce(psig);

| Linea | Dato generado en C | Dato generado en ensamblador | Observaciones |
|---|---|---|---|
| 4 | f(0)=-435.480962460247 | f(0)=-435.480962473041 | Iguales hasta en 1E-7. Genes idénticos. |
| 169 | f(5)=-416.653055458577 | f(5)=-416.653055550149 | Iguales hasta en 1E-6. D(0) idéntico hasta 1E-7, los demás idénticos. |
| 268 | f(8)=-483.887502325964 | f(8)=-483.887502044287 | Iguales hasta en 1E-6. Genes idénticos. |
| 301 | f(9)=-457.598676225055 | f(9)=-457.598676276916 | Iguales hasta en 1E-7. D(0) idéntico hasta 1E-7, los demás idénticos. |
| 400 | f(12)=-476.408859059358 | f(12)=-476.408859295116 | Iguales hasta en 1E-6. Genes idénticos. |
| 433 | f(13)=-465.002020438925 | f(13)=-465.002020420219 | Iguales hasta en 1E-7. Genes idénticos. |
| 499 | f(15)=-518.815113077439 | f(15)=-518.815113218325 | Iguales hasta en 1E-6. Genes idénticos. |
| 565 | f(17)=-492.835969094826 | f(17)=-492.835969095564 | Iguales hasta en 1E-8. Genes idénticos. |
| 598 | f(18)=-488.926474316877 | f(18)=-488.926474268567 | Iguales hasta en 1E-6. Genes idénticos. |
| 763 | f(23)=-403.192398566697 | f(23)=-403.192398547085 | Iguales hasta en 1E-7. Genes idénticos. |
| 829 | f(25)=-515.086114029090 | f(25)=-515.086114058286 | Iguales hasta en 1E-7. Genes idénticos. |

| | | | |
|---|---|---|---|
| 862 | f(26)=-422.520496435696 | f(26)=-422.520496552304 | Iguales hasta en 1E-6. Genes idénticos. |
| 895 | f(27)=-363.178303123036 | f(27)=-363.178303220476 | Iguales hasta en 1E-6. Genes idénticos. |
| 928 | f(28)=-556.110391688575 | f(28)=-556.110391681793 | Iguales hasta en 1E-8. Genes idénticos. |
| 961 | f(29)=-447.949503472220 | f(29)=-447.949503636298 | Iguales hasta en 1E-6. Genes idénticos. |
| 996 | d(0)=-4.58173710 | d(0)=-4.58173709 | Para f(30) |
| 1095 | d(0)=-4.48906904 | d(0)=-4.48906903 | Para f(33) |
| | | | Continuar el análisis con los archivos opCC.txt y opCA.txt |

## F.7 Operador Mutacion()

## Pruebas a los operadores

Función: Rastrigin
Dimensiones: 30
Población: 500 individuos

Discrepancias en los archivos de informe generados.

Operador: **Mutación**
**Modifica un gen seleccionado aleatoriamente para cada individuo de los no elitistas.**

Archivo de informe para C: opMutacionC.txt
Archivo de informe para Ensamblador: opMutacionASM.txt
Llamada a la función en C: mutacion(g+(PeriodoMigratorio*k));
Llamada a la función en Ensamblador: Mutacion(psig,g+(PeriodoMigratorio*k));

| Linea | Dato generado en C | Dato generado en ensamblador | Observaciones |
|---|---|---|---|
| 4 | f(0)=-435.480962460247 | f(0)=-435.480962473041 | Iguales hasta en 1E-7. Genes idénticos. |
| 169 | f(5)=-416.653055458577 | f(5)=-416.653055550149 | Iguales hasta en 1E-6. D(0) idéntico hasta 1E-7, los demás idénticos. |
| 268 | f(8)=-483.887502325964 | f(8)=-483.887502044287 | Iguales hasta en 1E-6. Genes idénticos. |
| 301 | f(9)=-457.598676225055 | f(9)=-457.598676276916 | Iguales hasta en 1E-7. D(0) idéntico hasta 1E-7, los demás idénticos. |
| 400 | f(12)=-476.408859059358 | f(12)=-476.408859295116 | Iguales hasta en 1E-6. Genes idénticos. |
| 433 | f(13)=-465.002020438925 | f(13)=-465.002020420219 | Iguales hasta en 1E-7. Genes idénticos. |
| 499 | f(15)=-518.815113077439 | f(15)=-518.815113218325 | Iguales hasta en 1E-6. Genes idénticos. |
| 565 | f(17)=-492.835969094826 | f(17)=-492.835969095564 | Iguales hasta en 1E-8. Genes idénticos. |
| 598 | f(18)=-488.926474316877 | f(18)=-488.926474268567 | Iguales hasta en 1E-6. Genes idénticos. |
| 763 | f(23)=-403.192398566697 | f(23)=-403.192398547085 | Iguales hasta en 1E-7. Genes idénticos. |
| 829 | f(25)=-515.086114029090 | f(25)=-515.086114058286 | Iguales hasta en 1E-7. Genes idénticos. |
| 862 | f(26)=-402.109958702006 | f(26)=-402.789116921228 | ***MUTACION |
| 868 | D(4)=0.99735942 | D(4)=1.05374969 | Este es el gen mutado. Los demás quedan idénticos. |

| | | | |
|---|---|---|---|
| 895 | f(27)=-363.178303123036 | f(27)=-363.178303220476 | Iguales hasta en 1E-6. Genes idénticos. |
| 928 | f(28)=-556.110391688575 | f(28)=-556.110391681793 | Iguales hasta en 1E-8. Genes idénticos. |
| 961 | f(29)=-447.949503472220 | f(29)=-447.949503636298 | Iguales hasta en 1E-6. Genes idénticos. |
| 994 | f(30)=-519.379672079107 | f(30)=-522.210293450711 | ***MUTACION |
| 996 | D(0)=-4.58173710 | D(0)=-4.58173709 | Iguales hasta en 1E-8. |
| 997 | D(1)=-4.86767327 | D(0)=-4.80326605 | Este es el gen mutado. Los demás quedan idénticos. |
| 1027 | f(31)=-546.187857803996 | f(31)=-554.570396179652 | ***MUTACION |
| 1032 | D(3)=0.23793930 | D(3)=0.38577586 | Este es el gen mutado. Los demás quedan idénticos. |
| 1095 | D(0)=-4.48906904 | D(0)=-4.48906903 | Para f(33) |
| 1128 | d(0)=-3.29731961 | D(0)=-3.29731962 | Para f(34) |
| 1159 | f(35)=-555.908000243538 | f(35)=-555.073285307993 | ***MUTACION |
| 1163 | D(2)=1.52209622 | D(2)=1.44055877 | Este es el gen mutado. Los demás quedan idénticos. |
| 1225 | f(37)=-426.612576398651 | f(37)=-426.612576398586 | Iguales hasta en 1E-9. Genes idénticos. |
| 1258 | f(38)=-589.693904576365 | f(38)=-589.797311553212 | ***MUTACION |
| 1278 | D(18)=1.03104819 | D(18)=1.03770508 | Este es el gen mutado. Los demás quedan idénticos. |
| 1324 | f(40)=-587.133019087015 | f(40)=-581.384956038992 | ***MUTACION |
| 1352 | D(26)=4.17768365 | D(26)=4.0640664 | Este es el gen mutado. Los demás quedan idénticos. |
| 1357 | f(41)=-433.931014010910 | f(41)=-433.931014022670 | Iguales hasta en 1E-7. Genes idénticos. |
| 1390 | f(42)=-441.973563140332 | f(42)=-441.973563097931 | Iguales hasta en 1E-6. Genes idénticos. |
| 1423 | f(43)=-485.042883661944 | f(43)=-488.023576428486 | ***MUTACION |
| 1448 | D(23)=-4.87632665 | D(23)=-4.80644012 | Este es el gen mutado. Los demás quedan idénticos. |
| 1456 | f(44)=-391.973048764551 | f(44)=-391.973048869493 | Iguales hasta en 1E-7. Genes idénticos. |
| 1489 | f(45)=-492.987108502750 | f(45)=-488.402438166902 | ***MUTACION |
| 1511 | D(20)=-3.16857416 | D(20)=-3.07539668 | Este es el gen mutado. Los demás quedan idénticos. |
| 1522 | f(46)=-410.229444179562 | f(46)=-410.229444176465 | Iguales hasta en 1E-8. Genes idénticos. |
| 1588 | f(48)=-529.906370356839 | f(48)=-529.906370510478 | Iguales hasta en 1E-6. Genes idénticos. |
| 1654 | f(50)=-377.712593951840 | f(50)=-378.014185931371 | ***MUTACION |
| 1675 | D(19)=-0.99157336 | D(19)=-1.0341523 | Este es el gen mutado. Los demás quedan idénticos. |
| 1687 | f(51)=-505.530865451250 | f(51)=-505.135709840192 | ***MUTACION |
| 1697 | D(8)=-1.05349299 | D(8)=-1.03252726 | Este es el gen mutado. Los demás quedan idénticos. |
| 1722 | D(0)=2.04868136 | D(0)=2.04868137 | Para f(52) |
| | | | Seguir revisando discrepancias con los archivos |
| | | | opMC.txt contra opMA.txt (Mar-13-2012) |

## F.8 Función GeneraPoblacion()

### Pruebas a los operadores

Función: Rastrigin
Dimensiones: 30
Población: 500 individuos

Discrepancias en los archivos de informe generados.

Operador: **Generación aleatoria de la primera población**
Archivo de informe para C: opGenPobC.txt
Archivo de informe para Ensamblador: opGenPobASM.txt
Llamada a la función en C: Sin identificación, es un ciclo en la función main()
Llamada a la función en Ensamblador: GeneraPoblacion

| Linea | Dato generado en C | Dato generado en ensamblador | Observaciones | |
|---|---|---|---|---|
| 1027 | f(31)=-612.576076497104 | f(31)=-612.576076497105 | Total de lineas en el informe: | 16502 |
| 1258 | f(38)=-486.338457173144 | f(38)=-486.338457173145 | Total de lineas discrepantes: | 34 |
| 1984 | f(60)=-595.732953346453 | f(60)=-595.732953346452 | Razón de lineas discrepantes: | 0.20% |
| 2512 | f(76)=-629.272998597265 | f(76)=-629.272998597266 | Impacto de las discrepancias: | No significativo |
| 4063 | f(123)=-502.338450425941 | f(123)=-502.338450425940 | Justificación: | Todas las discrepancias son del orden (+ -) 1E-12 |
| 4327 | f(131)=-542.344012271911 | f(131)=-542.344012271912 | | |
| 4558 | f(138)=-447.615534078694 | f(138)=-447.615534078693 | | |
| 5119 | f(155)=-545.993388697975 | f(155)=-545.993388697974 | | |
| 5185 | f(157)=-505.028020113617 | f(157)=-505.028020113616 | | |
| 5218 | f(158)=-494.121225713901 | f(158)=-494.121225713900 | | |

| | | | | |
|---|---|---|---|---|
| 5779 | f(175)=-562.026803929708 | f(175)=-562.026803929707 | | |
| 6439 | f(195)=-455.833192807747 | f(195)=-455.833192807746 | | |
| 6802 | f(206)=-695.300231883509 | f(206)=-695.300231883508 | | |
| 7066 | f(214)=-559.848585381647 | f(214)=-559.848585381648 | | |
| 7264 | f(220)=-534.096440745555 | f(220)=-534.096440745554 | | |
| 8683 | f(263)=-570.731410512002 | f(263)=-570.731410512001 | | |
| 8848 | f(268)=-645.428649994492 | f(268)=-645.428649994491 | | |
| 8914 | f(270)=-521.317971712207 | f(270)=-521.317971712206 | | |
| 8947 | f(271)=-555.344372325842 | f(271)=-555.344372325843 | | |
| 10267 | f(311)=-564.952673122400 | f(311)=-564.952673122399 | | |
| 10795 | f(327)=-553.703526333911 | f(327)=-553.703526333912 | | |
| 11521 | f(349)=-550.377757712831 | f(349)=-550.377757712832 | | |
| 12082 | f(366)=-602.040791555482 | f(366)=-602.040791555481 | | |
| 12544 | f(380)=-619.933333691320 | f(380)=-619.933333691319 | | |
| 13270 | f(402)=-484.643293855773 | f(402)=-484.643293855774 | | |
| 13402 | f(406)=-476.513623142310 | f(406)=-476.513623142309 | | |
| 13699 | f(415)=-593.308376994163 | f(415)=-593.308376994162 | | |
| 13765 | f(417)=-616.291427703116 | f(417)=-616.291427703115 | | |
| 14161 | f(429)=-489.227464599895 | f(429)=-489.227464599896 | | |
| 15382 | f(466)=-481.564599691721 | f(466)=-481.564599691722 | | |
| 15514 | f(470)=-524.095030295238 | f(470)=-524.095030295237 | | |
| 15547 | f(471)=-515.177898879923 | f(471)=-515.177898879922 | | |
| 15712 | f(476)=-566.472755958180 | f(476)=-566.472755958179 | | |
| 15844 | f(480)=-558.378350674512 | f(480)=-558.378350674513 | | |

## F.9 Función OrdenaPoblacion()

### Pruebas a los operadores

Función: Rastrigin

Dimensiones: 30
Población: 500 individuos

Discrepancias en los archivos de informe generados.

Operador: **Ordenación de la población**
**Ordena sólo los primeros "Migrables", el resto los deja desordenados.**
Archivo de informe para C: opOrdenaC.txt
Archivo de informe para Ensamblador: opOrdenaASM.txt
Llamada a la función en C: OrdenaPoblacion();
Llamada a la función en Ensamblador: OrdenaPoblacion(p,&Aux);

| Linea | Dato generado en C | Dato generado en ensamblador | Observaciones |
|---|---|---|---|
| 499 | f(15)=-447.615534078694 | f(15)=-447.615534078693 | Era f(138) antes de la ordenación |
| 796 | f(24)=-455.833192807747 | f(24)=-455.833192807746 | Era f(195) antes de la ordenación |
| 1027 | f(31)=-612.576076497104 | f(31)=-612.576076497105 | La ordenación se hace idéntica con el operador en |
| 1258 | f(38)=-486.338457173144 | f(38)=-486.338457173145 | ensamblador contra el operador en C. |
| 1984 | f(60)=-595.732953346453 | f(60)=-595.732953346452 | Las discrepancias en el informe son las producidas |
| 2512 | f(76)=-629.272998597265 | f(76)=-629.272998597266 | desde la generación de la primera población. |
| 4063 | f(123)=-502.338450425941 | f(123)=-502.338450425940 | |
| 4327 | f(131)=-542.344012271911 | f(131)=-542.344012271912 | |
| 5119 | f(155)=-545.993388697975 | f(155)=-545.993388697974 | |
| 5185 | f(157)=-505.028020113617 | f(157)=-505.028020113616 | |
| 5218 | f(158)=-494.121225713901 | f(158)=-494.121225713900 | |
| 5779 | f(175)=-562.026803929708 | f(175)=-562.026803929707 | |
| 6802 | f(206)=-695.300231883509 | f(206)=-695.300231883508 | |
| 7066 | f(214)=-559.848585381647 | f(214)=-559.848585381648 | |
| 7264 | f(220)=-534.096440745555 | f(220)=-534.096440745554 | |
| 8683 | f(263)=-570.731410512002 | f(263)=-570.731410512001 | |
| 8848 | f(268)=-645.428649994492 | f(268)=-645.428649994491 | |
| 8914 | f(270)=-521.317971712207 | f(270)=-521.317971712206 | |
| 8947 | f(271)=-555.344372325842 | f(271)=-555.344372325843 | |

| | | | |
|---|---|---|---|
| 10267 | f(311)=-564.952673122400 | f(311)=-564.952673122399 | |
| 10795 | f(327)=-553.703526333911 | f(327)=-553.703526333912 | |
| 11521 | f(349)=-550.377757712831 | f(349)=-550.377757712832 | |
| 12082 | f(366)=-602.040791555482 | f(366)=-602.040791555481 | |
| 12544 | f(380)=-619.933333691320 | f(380)=-619.933333691319 | |
| 13270 | f(402)=-484.643293855773 | f(402)=-484.643293855774 | |
| 13402 | f(406)=-476.513623142310 | f(406)=-476.513623142309 | |
| 13699 | f(415)=-593.308376994163 | f(415)=-593.308376994162 | |
| 13765 | f(417)=-616.291427703116 | f(417)=-616.291427703115 | |
| 14161 | f(429)=-489.227464599895 | f(429)=-489.227464599896 | |
| 15382 | f(466)=-481.564599691721 | f(466)=-481.564599691722 | |
| 15514 | f(470)=-524.095030295238 | f(470)=-524.095030295237 | |
| 15547 | f(471)=-515.177898879923 | f(471)=-515.177898879922 | |
| 15712 | f(476)=-566.472755958180 | f(476)=-566.472755958179 | |
| 15844 | f(480)=-558.378350674512 | f(480)=-558.378350674513 | |

# Referencias

[1] L. Araujo and C. Carlos, Algoritmos Evolutivos, Un enfoque práctico, 1st ed. Alfaomega Ra-Ma, 2009, p. 330.

[2] S. Luke and Et.al., "ECJ 21 A Java-based Evolutionary Computation Research System." [En línea]. Disponible en: http://cs.gmu.edu/~eclab/projects/ecj/.

[3] B. Punch and D. Zongker, "lil-gp 1.1Beta." [En línea]. Disponible en: http://garage.cse.msu.edu/software/lil-gp/.

[4] B. L. Browning, "BEAGLE genetic analysis software." [En línea]. Disponible en: http://faculty.washington.edu/browning/beagle/beagle.html.

[5] M. Keijzerm, J. J. Merelo, G. Romero, and M. Schoenauer, "Evolving Objects (EO): Evolutionary Computation Framework." [En línea]. Disponible en: http://eodev.sourceforge.net/.

[6] T. J. Rothwell, "The GNU C Reference Manual." [En línea]. Disponible en: http://www.gnu.org/software/gnu-c-manual/.

[7] "GCC optimization - Gentoo Wiki." [En línea]. Disponible en: https://wiki.gentoo.org/wiki/GCC_optimization.

[8] I. Castro Liera, "Paralelización de Algoritmos de Optimización Basados en Poblaciones Usando GPGPU," Instituto Tecnológico de La Paz, 2011.

[9] A. Geist and Et.al., "PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing." [En línea]. Disponible en: http://www.netlib.org/pvm3/book/pvm-book.html.

[10] M. A. Castro Liera and F. Herrera Fernández, "Optimización paramétrica de sistemas de identificación difusa, mediante algoritmos genéticos distribuidos con período migratorio dinámico," in Convención Internacional de Ingeniería Mecánica, Eléctrica e Industrial, 2006, p. 10.

[11] R. Hyde, Write Great Code, Volume 2: Thinking Low-Level, Writing High-Level, 1st ed. No Starch Press, 2012, p. 640.

[12] R. U. Joshi, "Code Optimization Using the GNU C Compiler LG #71," LInux Gaette, 2001. [En línea]. Disponible en: http://linuxgazette.net/issue71/joshi.html.

[13] D. E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley Professional, 1989, p. 432.

[14] D. Floreano and C. Mattiussi, Bio-Inspired Artificial Intelligence: Theories, Methods, and Technologies (Intelligent Robotics and Autonomous Agents series). The MIT Press, 2008, p. 659.

[15] J. H. Holland, Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence (Complex Adaptive Systems). The MIT Press, 1975, p. 211.

[16] M. A. Castro Liera, "Un algoritmo genético distribuido con aplicación en la identificación difusa de un proceso fermentativo," Universidad Central, "Marta Abreu" de Las Villas, 2009.

[17] J. A. Morales Viscaya, "Un algoritmo híbrido para la optimización en paralelo de problemas contínuos," Instituto Tecnológico de La Paz, 2012.

[18] P. N. Suganthan, N. Hansen, J. J. Liang, K. Deb, A. Auger, S. Tiwari, and T. Report, "Problem Definitions and Evaluation Criteria for the CEC 2005 Special Session on Real-Parameter Optimization," Singapore, 2005.

[19] R. Flickenger, Linux Server Hacks: 100 Industrial-Strength Tips and Tools. O'Reilly Media, 2009, p. 242.

[20] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," ACM Trans. Model. Comput. Simul., vol. 8, no. 1, pp. 3–30, Jan. 1998.

[21] M. Matsumoto and T. Nishimura, "Mersenne Twister: A random number generator (since 1997/10)," 1997. [En línea]. Disponible en: http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html.

[22] "GNU Scientific Library – Reference Manual: Top." [En línea]. Disponible en: http://www.gnu.org/software/gsl/manual/html_node/.