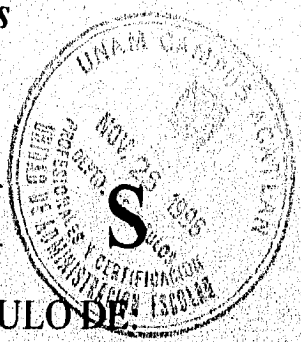
33
2y

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES
A C A T L Á N

*RELAJAMIENTO DE AGREGADOS ATÓMICOS USANDO ALGORITMOS GENÉTICOS*



# T E S I S

PARA OBTENER EL TITULO DE:

# LICENCIADO EN MATEMÁTICAS APLICADAS Y COMPUTACIÓN

P R E S E N T A :

MARIO PÉREZ ÁLVAREZ



TESIS CON
FALLA DE ORIGEN

# TESIS

# COMPLETA

A mis Padres, por su gran apoyo, cariño y comprensión.

A mis abuelos, por la gran herencia de amistad y amor que me han dado.

A mis hermanas, tíos(as) y primos(as), por su apoyo

**Agradecimientos**

A los Doctores David Romeu, Humberto Terrones, y Alfredo Gómez, por su apoyo, múltiples comentarios y sugerencias para elaborar esta tesis.

A los Doctores Luis Beltrán, Samuel Tehuacanero y Cristina Zorrilla, por apoyo técnico y sugerencias para la elaboración de esta tesis.

Al Sr. Alfredo Sánchez, por apoyo fotográfico.

Agradezco también el apoyo económico otorgado por CONACyT a través del proyecto 0088P-E.

Vaya un agradecimiento especial al Dr. José Luis Aragón por su trato, su tiempo, su amistad y sus consejos, sin los cuales no hubiese sido posible la elaboración de esta tesis.

A todos ellos, **muchas gracias**.

# Contenido

# Parte I

# INTRODUCCIÓN

Desde sus orígenes el área de la computación ha tenido un amplio espectro de aplicaciones. A últimas fechas la computadora se ha vuelto un instrumento imprescindible para el quehacer de la ciencia, cuyo desarrollo se ha incrementado conforme se incrementa también el desarrollo de programas de cómputo (software).

Gracias al desarrollo de la computación se ha podido estudiar la formación y propiedades de *agregados atómicos pequeños*, por medio de la simulación numérica. El estudio de estos agregados es relevante para diversas áreas de la física, algunas de las cuales se enlistan enseguida.

- **En nucleación y crecimiento de cristales.** Al hablar de nucleación nos referimos a un proceso mediante el cual, a partir de un agregado primordial de átomos, llamado *núcleo*, se agregan otros para formar un "cristal". Un cristal convencional es un arreglo bien ordenado, de átomos o moléculas, que llena el espacio por medio de traslaciones de sus celdas unitarias. En cualquier estructura cristalina hay ciertas simetrías. Por ejemplo: Un cristal se dice que tiene simetría rotacional 3, si el cristal se ve exactamente igual después de que es rotado 120 grados (como sucede en un triángulo equilátero)[13]. Un resultado clásico en cristalografía, nos dice que los cristales pueden tener únicamente simetría rotacional tres, cuatro o seis, pero un cristal no puede tener simetría cinco, por la misma razón que es imposible construir un plano cubierto completamente por pentágonos. Se ha estudiado la formación y propiedades de *agregados atómicos pequeños*, debido a que los resultados obtenidos, han provisto de un entendimiento básico sobre las fuerzas responsables de la cohesión de materiales.
- **En catálisis.** El término, catálisis, se refiere a cierto tipo de trans-

formación química que para efectuarse exige la presencia de una sustancia determinada llamada catalizador, la que al concluir la reacción permanece inalterada. Son múltiples las sustancias que sirven como catalizadores. Una de las más comunes es el agua. El enmohecimiento del hierro, es decir, la combinación del hierro con el oxígeno de la atmósfera es un buen ejemplo de esta forma de catálisis.

En años recientes ha habido un renovado interés para tratar el efecto del tamaño de agregados atómicos pequeños en relación con la actividad catalítica. La conclusión principal de algunos experimentos es la conexión entre la actividad catalítica y el específico arreglo de los átomos en la superficie de *partículas metálicas pequeñas*. Por lo tanto la caracterización de la forma de las partículas es un paso básico en el entendimiento de la actividad de los catalizadores.

- **En cuasicristales.** Un cristal presenta orden orientacional y traslacional de largo alcance, Los *cuasicristales* también lo presentan, pero con características particulares. Como ya vimos, un cristal convencional puede tener simetría rotacional 3, 4 ó 6 pero no puede tener simetría 5. Sin embargo los cuasicristales presentan ejes con simetría 5, que son incompatibles con las reglas de cristalografía convencional.

  Las investigaciones en la microestructura de los cuasicristales revelaron una clase de orden orientacional no cristalino ni tampoco amorfo. Entonces se les nombró cuasicristales.

  Los cuasicristales guardan una relación muy estrecha con la estructura de *las partículas pequeñas*, en lo que se refiere a la simetría cinco.

La simulación de partículas pequeñas es de gran ayuda para su estudio, ya que ayuda a entender su estructura y formación. La simulación se basa

en encontrar la configuración de mínima energía de las partículas. Para esto se utilizan diversos potenciales que relacionan la energía de atracción de los átomos de las partículas con la distancia que hay entre ellos. Si minimizamos la energía de una partícula con un número dado de átomos obtenemos la configuración buscada.

Los métodos de minimización tradicionales han servido para la simulación de partículas pequeñas, pero algunos de estos métodos suelen llegar sólo a mínimos locales cuando lo que se busca es un mínimo absoluto, mientras que otros llegan al mínimo absoluto con gran dificultad y suelen consumir gran tiempo de cómputo.

En esta tesis se estudiará un nuevo método: *Algoritmos Genéticos*. Estos garantizan encontrar el mínimo de cualquier función además de ser muy eficiente. Se implementará y se hará una comparación de resultados con los obtenidos con otros métodos.

En el capítulo I se revisan brevemente algunos algoritmos de optimización que son particularmente útiles para funciones no lineales. En el capítulo II revisamos la técnica de los Algoritmos Genéticos, sus estructuras, sus operaciones y el proceso que siguen para encontrar el mínimo de una función. En el capítulo III se estudia el problema del mínimo de la energía de partículas pequeñas usando potenciales, con uno o más mínimos, y se implementa un algoritmo genético para resolver el problema. En el capítulo IV se analizan los resultados de aplicar el algoritmo genético al sistema de agregados de oro, utilizando un potencial del tipo Lennard-Jones con dos mínimos.

# Parte II

# CAPITULADO

# CAPITULO 1

# MÉTODOS DE OPTIMIZACIÓN DE FUNCIONES.

## 1.1 Introducción

El hallar el mínimo (o máximo) de una función es un problema central en física, ingeniería y muchas otras disciplinas. Nos referiremos a éste como el problema de optimización de una función. Más precisamente, este problema consiste en encontrar el punto $x = (x_1, x_2, x_3, ...x_n)$ que evaluado en una función $f(x_1, x_2, x_3, ...x_n)$, nos de el mejor valor posible para ésta (mínimo o máximo), este punto será entonces el punto óptimo $\mathbf{x^*}$ de la función $f(\mathbf{x})$.

La optimización de una función es un problema que afortunadamente tiene diversas formas de solucionarse, sin embargo, la elección de un método es a menudo otro problema. Se requiere de un estudio muy detallado debido a que la función puede tener distintas características. Como por ejemplo, puede tratarse de una función lineal, entera, o no lineal; de una o varias variables; continua o discontinua; con o sin restricciones; con o sin mínimos o máximos locales; etc.

Este capítulo trata de métodos que optimizan funciones no lineales. Existe una gran variedad de métodos de optimización de funciones, que los podemos dividir en dos categorías, **Determinísticos y Estocásticos** [10].

La lógica de los métodos determinísticos sigue invariablemente los siguientes pasos:

1. Partir de un punto inicial, preferentemente cercano al punto óptimo.

2. Calcular la dirección hacia la cual debe avanzar para llegar al punto óptimo.

3. Calcular la longitud del paso que va a aplicar. Es decir, que tanto debe dirigirse en la dirección calculada, antes de calcular una nueva dirección.

Los métodos determinísticos se distinguen por que generalmente encuentran el mínimo (máximo) local o relativo más cercano al punto inicial, pero este puede no ser el óptimo global o absoluto. Además de que la mayoría de estos métodos requieren del uso de la primera derivada de la función objetivo (gradiente) para calcular las direcciones, y algunas veces de la segunda derivada (Hessiano). El calculo de estas derivadas implican consumo de tiempo y limitan los problemas que pueden resolver. Otro problema de este tipo de métodos es que pueden divergir si el punto inicial es lejano a algún mínimo (máximo).

Los métodos estocásticos se basan en el azar. Utilizan la generación aleatoria de posibles soluciones para acercarse al óptimo de una función. Estos métodos pueden encontrar el óptimo global de una función, pero por sus características aleatorias no se puede saber a ciencia cierta en cuanto tiempo.

De los métodos que a continuación describiremos sólo *el método de simulación de recocido* es estocástico, mientras que los demás, *el método de pasos descendientes*, *el método del gradiente conjugado* y *el método de Newton-Raphson*, son determinísticos.

Estos métodos fueron seleccionados por ser los que se han utilizado para resolver el problema de relajamiento de agregados atómicos [10].

## 1.2 El método de pasos descendientes.

Uno de los métodos de optimización más comunes que han sido utilizados para diversos problemas que contienen funciones no lineales es el llamado *método*

*de pasos descendientes.* Este método es determinístico, y como la mayoría de estos métodos emplea al gradiente como instrumento en la búsqueda de los óptimos de una función. Recurre al gradiente (pendiente de la función) a modo de indicador de la dirección sobre la cual se encuentra el óptimo, y requiere de la determinación arbitraria de un primer punto de partida.

El gradiente de una función de $n$ variables independientes $f(\mathbf{x}) = f(x_1, x_2, .., x_n)$ se puede escribir como sigue:

$$\nabla f(\mathbf{x}) = \left( \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \cdots, \frac{\partial f}{\partial x_n} \right). \tag{1.1}$$

Todos los métodos que utilizan el gradiente de la función para encontrar su mínimo [8], se valen de la ecuación general:

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \alpha \mathbf{H} \nabla f|_{x=x^k} \tag{1.2}$$

en donde $\mathbf{x}^k$ es el resultado de la anterior iteración, $\mathbf{x}^{k+1}$ es el nuevo resultado, $\nabla f$ el gradiente de $f$ nos da la dirección sobre la cual se encontrará la nueva $\mathbf{x}^{k+1}$. $\mathbf{H}$ es una matriz de $n \times n$ positiva definida (condición necesaria para que la función llegue a un mínimo) que nos ayuda a encontrar un mínimo y su forma depende de cada método, y $\alpha$ es un número real que nos da la longitud del paso. Los métodos que utilizan el gradiente sólo difieren en la forma en que $\mathbf{H}$ y $\alpha$ son seleccionados.

En cada iteración, se calculará una nueva longitud $\alpha$ y una nueva dirección $\nabla f$, para alcanzar algún mínimo.

Para encontrar el mínimo de una función cualquiera $f(\mathbf{x}) = f(x_1, x_2, ..., x_n)$, por *métodos que utilizan el gradiente*, se requiere que $f(\mathbf{x})$ sea diferenciable y continua.

La lógica que sigue el método de pasos descendientes, es simple. Toma direcciones sucesivas perpendiculares en cada iteración para llegar al mínimo (máximo) local más cercano al punto de partida como se ilustra en la figura 1.1,

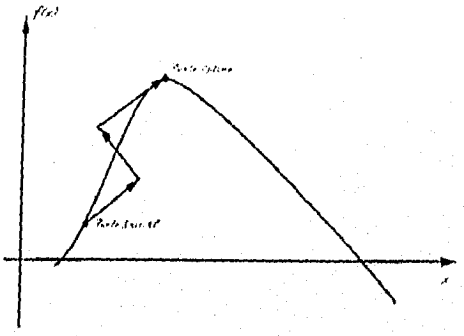


Figura 1.1: Representación de la forma de encontrar un óptimo

Como ya vimos, los métodos que utilizan el gradiente sólo difieren en la forma en que H y $\alpha$ son seleccionados. El *método de pasos descendientes* toma a la *matriz identidad* como H (que por definición es positiva definida), y para calcular $\alpha_i$ maximiza la longitud del paso que puede dar en una dirección.

Dado un punto de la función $f(\mathbf{x}^i)$, $\mathbf{x}^i = (x_1^i, x_2^i, ..., x_n^i)$, la dirección hacia el mínimo de la función es [9]

$$-\nabla f(\mathbf{x}^i) \tag{1.3}$$

La longitud del recorrido en esta dirección debe ser la máxima posible, de manera que para encontrar el parámetro $\alpha_i$ debemos maximizar primero la función:

$$h(\alpha_i) = f(\mathbf{x}^i - \alpha_i \nabla f(\mathbf{x}^i)) \tag{1.4}$$

Observemos que $h(\alpha_i)$ es una función de una sola variable, ya que $\mathbf{x}^i$ es conocida, por lo tanto, si igualamos a 0 la derivada de $h(\alpha_i)$ encontraremos el parámetro $\alpha_i$ que buscamos.

La ecuación recursiva del método de pasos descendientes para encontrar un mínimo de la función $f(\mathbf{x}^i)$ es la siguiente (véase ecuación 1.2):

$$\mathbf{x}^{i+1} = \mathbf{x}^i - \alpha_i \nabla f(\mathbf{x}^i),\ i = 0, 1, 2... \tag{1.5}$$

El algoritmo se detiene cuando la diferencia de el valor de la función en dos iteraciones sucesivas sea menor que un error mínimo dado $\varepsilon > 0$, es decir cuando

$$\left| f(\mathbf{x}^{k+1}) - f(\mathbf{x}^k) \right| < \varepsilon \tag{1.6}$$

El problema de este método es que asegura llegar a un mínimo (o máximo) dependiendo del punto de partida $\mathbf{x}^0$, pero no se sabe que clase de mínimo (o máximo): local o absoluto. Además, necesita de un método diferente para calcular, en cada iteración, la longitud del paso $\alpha$, y necesita calcular también el gradiente de la función. Además su convergencia al mínimo es lenta debido a que las direcciones son perpendiculares.

## 1.3 El método del gradiente conjugado

Otro de los métodos determinísticos que utilizan el gradiente para llegar al mínimo de la función pero que converge más rápido a la solución es el del *gradiente conjugado*, o también llamado de direcciones conjugadas. Aunque la lógica del método es similar a la de todos los métodos que utilizan el gradiente, (también se vale de la ecuación recursiva 1.2 para llegar al mínimo) el método del gradiente conjugado tiene la propiedad que en vez de tomar *direcciones sucesivas perpendiculares* (como el caso de del método de pasos descendientes) se calcula para cada iteración una nueva dirección. Es decir, converge cambiando la dirección en cada iteración, además de la longitud del paso para esa dirección. Ésto hace que el método converja en menos pasos.

Este método además de un punto inicial $\mathbf{x}^0 = (x_1^0, x_2^0, ..., x_n^0)$ también requiere de una dirección inicial $\mathbf{s}^0 = (s_1^0, s_2^0, ..., s_n^0)$ como punto de partida. La ecuación recursiva de este método es la siguiente:

$$\mathbf{x}^{i+1} = \mathbf{x}^i + \alpha_i \mathbf{s}^i \tag{1.7}$$

Donde $\alpha$ la longitud del paso de cada iteración, se obtiene de la optimización

de la función de una sola variable:

$$g(\alpha_i) = f(\mathbf{x}^i + \alpha_i \mathbf{s}^i) \tag{1.8}$$

El vector de direcciones $\mathbf{s}^{i+1}$ debe ser conjugado al $\mathbf{s}^i$, es decir, debe satisfacer la condición

$$(\mathbf{s}^{i+1})^T \mathbf{H}(\mathbf{x}^{i+1})\mathbf{s}^i = 0 \tag{1.9}$$

donde $\mathbf{H}(\mathbf{x}^{i+1})$ es el Hessiano de $f(\mathbf{x})$ (que se supone positivo definido) evaluado en $\mathbf{x}^{i+1}$ que se calculó anteriormente.

El método del gradiente conjugado, es uno de los más eficientes, puesto que por el cambio constante de dirección, puede converger rápidamente al mínimo de la función. Sin embargo, este método aparte de depender del punto de partida $\mathbf{x}^0$, también depende de una dirección inicial $\mathbf{s}^0$ para que pueda converger al mínimo local.

## 1.4 El método de Newton-Raphson

El método de Newton-Raphson es muy parecido al de pasos descendientes. Es también uno de los métodos que usan el gradiente y es más utilizado que los anteriores métodos, pero no por ello es más eficiente. Utiliza la ecuación recursiva

$$\mathbf{x}^{i+1} = \mathbf{x}^i - \mathbf{H}^{-1}(\mathbf{x}^i)\nabla f(\mathbf{x}^i) \tag{1.10}$$

para encontrar un mínimo local de la función $f(\mathbf{x})$. $\mathbf{H}^{-1}(\mathbf{x}^i)$ es la matriz inversa del Hessiano (que se supone positiva definida) de la función, evaluado en el punto $\mathbf{x}^k$, y $\nabla f(\mathbf{x}^k)$ es el gradiente de la función, también evaluado en el punto $\mathbf{x}^k$.

Se supone que la función $f(\mathbf{x})$ es continua y diferenciable, de tal suerte que cuando $\nabla f(\mathbf{x}^*) = 0$ habremos encontrado el mínimo de la función.

Este método, no necesita de un método auxiliar que calcule el óptimo de una función de una sola variable, como en los métodos anteriores. Una limitación muy fuerte que presenta es que para algunas funciones, puede divergir a

pesar de que el punto de partida este cercano a un mínimo. Además es necesario utilizar un método auxiliar para calcular el inverso de la matriz Hessiana de la función, y calcularla en cada iteración es regularmente muy costoso en tiempo de cómputo.

## 1.5 El método de simulación de recocido (Simmulated Annealing)

A diferencia de los métodos mostrados anteriormente, éste no es un método que necesite del gradiente de la función. Este método es estocástico, no analítico, y permite resolver problemas mediante la simulación de variables aleatorias que toman los valores de las posibles soluciones del problema.

El método en realidad es una variante del conocido *método de Montecarlo* [10], pero introduce un nuevo criterio de aceptación o rechazo de una nueva solución. A este criterio se le conoce como *Montecarlo-Metropoli*.

El método de Montecarlo se usó inicialmente en los trabajos de John Von Newmann y Stanislaw Ulam, en los últimos años de la década de los cuarentas, relacionados con problemas de barreras nucleares de protección. Con el advenimiento de las computadoras, en los primeros años de la década de los cincuenta, el método experimentó un avance substancial.

La base del método de Montecarlo es la generación de números aleatorios para la simulación de un proceso cuya marcha dependa de factores aleatorios, y con ello poder resolver diversos problemas matemáticos.

El método de simulación de recocido difiere del método de Montecarlo, en que en la simulación introduce una variable de temperatura, que determinará la evolución del método.

Los pasos del método de Montecarlo son los siguientes:

1. Se inicializa el algoritmo con un punto inicial (puede ser generado aleatoriamente).

2. Se hace un cambio en la configuración con un generador aleatorio.

3. Se evalúan la configuración inicial y la que tiene el cambio en la función objetivo.

4. Se introduce un parámetro de control T (temperatura) muy alto, que descenderá cada cierto número de pasos (normalmente 100-1000) [10].

El algoritmo inicia con un punto inicial de partida que puede ser obtenido aleatoriamente, pero dentro del dominio de la función. El punto inicial, así como todos los demás puntos que se obtengan, deben ser posibles soluciones al problema. En seguida, algunn de las variables $\mathbf{x}_i^0 = (x_1^0, x_2^0, ...x_n^0)$, escogida aleatoriamente, se mueve en una dirección escogida al azar una distancia también aleatoria $d$ no mayor que una constante predeterminada al inicio. Entonces tendremos una nueva solución

$$\mathbf{x}_i^1 = (x_1^1, x_2^1, ..., x_j^1 + d, ..., x_n^1)$$

Si resulta que $f(\mathbf{x}_i^1) < f(\mathbf{x}_i^0)$, es decir si es mejor que la anterior la nueva solución $\mathbf{x}_i^1$ se acepta. En caso contrario se pasa al criterio de *Montecarlo-Metropali* para aceptarla o rechazarla.

En el criterio Montecarlo-Metropoli, si $f(\mathbf{x}_i^1) \geq f(\mathbf{x}_i^0)$, entonces se utiliza una distribución de probabilidad de Maxwell-Boltzmann [13] ($f(u) = \exp(\frac{\Delta U}{kT})$) como criterio para aceptar o rechazar la nueva solución. Esto se realiza de la siguiente manera. Se escoge un número aleatorio entre cero y uno y se compara con el valor de $\exp(\frac{\Delta U}{kT})$ donde $\Delta U = |f(\mathbf{x}_i^1) - f(\mathbf{x}_i^0)|$, $T$ es la temperatura (que al principio del algoritmo es grande y va desrendiendo cada cierto número constante de iteraciones, según la complejidad de la función a optimizar) y $k$ es la constante de Boltzman: $8.62 * 10^{-5} \frac{eu}{oK}$. Si el número aleatorio escogido es menor o igual que el valor que toma la función, se acepta la solución $\mathbf{x}_i^1$, si no, se rechaza y la anterior $\mathbf{x}_i^0$se mantiene como posible solución.

El método se detiene cuando

$$|f(\mathbf{x}_i^{k+1}) - f(\mathbf{x}_i^k)| < \epsilon.$$

o bien, cuando el número de iteraciones es mayor que un número fijado al principio del programa.

La principal desventaja de este método es que no sabemos en cuantos pasos llegará al mínimo. Por sus características aleatorias, puede llegar al mínimo de la función en un número muy alto de iteraciones o en un número muy corto.

# CAPÍTULO 2

# ALGORITMOS GENÉTICOS

## 2.1 Introducción

Darwin sugirió que cada generación de seres vivos se compone de una población de entidades con características que varían ligeramente de la media. Esta diversidad se manifiesta como una variación en los cromosomas de los entes que forman la población. Algunos individuos pueden ser ligeramente más grandes, otros pueden poseer órganos de formas ligeramente diferente y algunos pueden poseer habilidades un poco por arriba, o por abajo, de lo normal. Las diferencias pueden ser muy pequeñas, pero aquellos cuyas características estén mejor adaptadas al medio ambiente, tenderán a vivir más y a reproducirse a velocidad más alta. Eventualmente, una acumulación de características favorables podría dar lugar a una incapacidad de procrear con el tipo original, dando lugar a una nueva especie. A este proceso de supervivencia del más apto, Darwin lo llamó *selección natural*.

El proceso evolutivo se basa en operaciones genéticas en los cromosomas, como resultado de la recombinación sexual. Los cromosomas son cadenas de moléculas de ADN que pueden verse como cadenas de caracteres en un alfabeto de base 4 (los cuatro nucleótidos: Adenina, Timina, Guanina y Citocina). Las generaciones posteriores pueden poseer cadenas de nucleótidos de sus progenitores, de tal suerte que las responsables de una mejor adaptación son trans-

mitidas a futuras generaciones. Eventualmente, pueden producirse errores en la transmisión de código genético, lo que origina alteraciones de las cadenas que se denominan mutaciones.

Un **Algoritmo Genético** es un algoritmo matemático que transforma una "población" de objetos matemáticos, con un *parámetro* de "aptitud" asociado a cada uno de ellos, en una nueva población. Para establecer el paradigma con la evolución, las principales operaciones involucradas para pasar de una generación a otra son: *reproducción, combinación y mutación.* Un algoritmo genético se basa en el proceso evolutivo: las posibles soluciones a un problema particular se someten a un "ambiente" artificial que se diseña para motivar la supervivencia del más apto, esto es, de los individuos que se aproximen más a la solución buscada.

Debido a su naturaleza, los algoritmos genéticos son básicamente mecanismos de búsqueda. Han gozado de particular interés debido a su habilidad para hallar soluciones a una gran diversidad de problemas, como optimización de funciones, modelado de células biológicas, estrategias en teoría de juegos, robótica, modelos de spin, etc. [7].

Los ingredientes básicos para resolver un problema específico utilizando un algoritmo genético son [7]

1. Definir un esquema de representación. Esto es, posibles soluciones al problema (**fenotipo**) deben codificarse en cadenas (**cromosomas o genotipo**) utilizando un alfabeto de un número finito de letras. Usualmente se recurre al código binario.

2. Definir la medida de aptitud. Si, por ejemplo, el problema es hallar una ecuación que ajuste datos experimentales, la aptitud de una cadena (que corresponde a una posible ecuación) se medirá por la suma de las diferencias al cuadrado entre esta función y los datos experimentales.

3. Definir los parámetros y las variables que controlarán el algoritmo: máximo número de elementos de la población, número de generaciones, frecuencias de reproducción, de combinación y de mutación, etc.

Una vez que el problema se ha preparado siguiendo estas tres reglas, el algoritmo empieza generando, al azar, una población inicial de individuos (cadenas). A partir de este punto, se procede iterativamente evaluando la aptitud de cada individuo y generando una nueva población, aplicando las operaciones básicas descritas anteriormente:

1. Copiando una cadena ya existente (reproducción)

2. Creando dos nuevas cadenas combinando subcadenas, escogidas al azar, de dos cadenas ya existentes (combinación)

3. Creando una nueva cadena partiendo de otra ya existente cambiando un carácter de la cadena en una posición que se elige al azar (mutación).

Estas operaciones se aplican individualmente a las cadenas de cada generación, de acuerdo con un criterio probabilístico o elitista; tenderán más a reproducirse, combinarse o mutarse aquellas cadenas que tengan mayor aptitud, para heredar a la nueva generación sus genes. Finalmente, el resultado del algoritmo genético es la cadena más apta de cualquier generación. Este resultado es la solución (exacta o aproximada) del problema.

Después de esta rápida revisión de la esencia del método, podemos discutir ahora en más detalle cada uno de los pasos. La generación de una población inicial de individuos, es el primer y más importante paso de un algoritmo genético. En primer lugar, se debe lograr que el código genético que se utilice para esto pueda abarcar todas y cada una de las posibles soluciones al problema que se esté tratando. Es decir, que si el problema es encontrar los tres primeros

dígitos significativos del número $\pi$, la representación del código genético, debe poder abarcar todos los números de tres dígitos del sistema decimal (del 000 al 999). Para representar genéticamente todas las posibles soluciones a un problema, deben utilizarse cadenas de caracteres con un alfabeto finito, usualmente se utiliza para esto el código binario. Así pues, tendríamos que la representación de los primeros tres dígitos del número $\pi$ podría ser 011001100 (314 en octal).

Después de haber definido un esquema de representación de las posibles soluciones al problema, se debe saber qué elementos de la población son las mejores soluciones al problema. Si por ejemplo, el problema es minimizar una función $f(x)$, aquellos valores de $x$ que evaluados en la función sean los menores, serán los elementos de la población más aptos. Y de acuerdo a esto, se determinará que elementos son los que se reproducirán, combinarán o mutarán y cuales se eliminarán para la siguiente generación.

El algoritmo empieza generando una población de individuos al azar, y de acuerdo a su aptitud, se seleccionarán a los individuos más aptos para que pasen sin cambios a la siguiente generación (*Reproducción*), estos también deben ser los más propensos para la operación de *Combinación* que se hace como sigue. Consideramos los siguientes dos individuos de una población

$$\begin{matrix} 11100 \\ 01010 \end{matrix} \tag{2.1}$$

Seleccionamos al azar una posición de las cadenas:

$$\begin{matrix} 11|100 \\ 01|010 \end{matrix} \tag{2.2}$$

La combinación consiste en intercambiar el código genético así:

$$\begin{matrix} 11010 \\ 01100 \end{matrix} \tag{2.3}$$

que da lugar a dos hijos 2.3 de la pareja 2.1 y que serán parte de la siguiente generación de individuos.

La operación de mutación es más simple. Dada una cadena 2.4 también se le selecciona al azar una posición

$$\hat{1}1100 \tag{2.4}$$

El carácter en la posición seleccionada se muta, en este caso, de 1 a 0, con lo que queda:

$$01100 \tag{2.5}$$

que es un nuevo individuo 2.5 de la siguiente generación.

Es importante observar que los individuos que se elgen para reproducción también pueden ser elegidos para la combinación o la mutación; y los individuos que no son elegidos, simplemente se eliminan, ya que obviamente son estos los menos aptos de la población, y lo que queremos es conservar los genes que nos dan buenos resultados para obtener en cada generación mejores individuos o mejores posibles soluciones al problema.

También es importante definir desde el inicio del algoritmo genético el tamaño de la población, los elementos de la población que se reproducirán o que probabilidad hay de reproducción, así como de combinación y de mutación, y cuantos deben descecharse en cada generación, así como cuantas generaciones se consideran que son necesarias para llegar a una solución.

Un ejemplo de como funciona un algoritmo genético es en este momento lo más adecuado para su comprensión. Consideremos el siguiente problema:

$$\begin{aligned} &\textit{Maximizar } f(x) = x^2 - 4x + 8 \\ &\text{en el intervalo } [0, 6] \end{aligned}$$

Primero definimos un esquema de representación. Si tenemos que encontrar $x$ en un intervalo de 0 a 6, podemos codificar las posibles soluciones como cadenas

de números binarios de 6 dígitos para tener un total de $2^6$ posibles soluciones al problema, utilizando un factor de conversión $\frac{6-0}{2^6-1}+0=(0.0952)$. Por ejemplo, para el genotipo $011110=14$ tenemos el fenotipo $(14)(0.0952)=1.333$.

Ahora definimos la medida de aptitud como el fenotipo de $x$ evaluado en $f(x)$, por ejemplo, para $x=1.333$ tenemos $f(1.333)=4.444$.

El siguiente paso es definir los parámetros y las variables que controlarán el algoritmo. Para este problema definiremos los siguientes:

número de elementos de población = 5

número de generaciones = 4

probabilidad de reproducción = 0.3

probabilidad de combinación = 0.6

probabilidad de mutación = 0.1

Ahora podemos comenzar el proceso recursivo, partiendo de una población inicial generada al azar. En el ejemplo de la tabla 2.6 las parejas seleccionadas para combinación se marcaron con * y +, y los genes que se mutarán aparecen con un ^. Las cadenas eliminadas no tienen ninguna marca y las cadenas que pasaran sin ningún cambio (reproducción) tendrán un ·. Recordemos que las cadenas que son seleccionadas para combinación también pueden ser seleccionadas para reproducción y/o mutación al mismo tiempo.

| | *generación* 1 | | *generación* 2 | | *generación* 3 | | *generación* 4 | |
|---|---|---|---|---|---|---|---|---|
| *No.* | $f(x)$ | *genotipo* | $f(x)$ | *genotipo* | $f(x)$ | *genotipo* | $f(x)$ | *genotipo* |
| 1 | 4.32 | ·*011011 | 4.32 | ·*011011 | 4.32 | *011011 | 4.00 | 010101 |
| 2 | 8.39 | +101011 | 4.58 | 01$\hat{1}$101 | 4.32 | 011011 | 4.04 | 010011 |
| 3 | 13.87 | 110110 | 5.78 | *100011 | 4.00 | +*01$\hat{0}$101 | 4.00 | 010101 |
| 4 | 6.32 | +000101 | 7.62 | 101001 | 4.04 | +010011 | 4.32 | 011011 |
| 5 | 6.32 | *100101 | 5.78 | 000111 | 8.39 | 101011 | 4.00 | 010101 |

(2.6)

El resultado obtenido fue $f(x = 10111 = 2) = 4$

Al criterio que se utilizó para escoger a los individuos y aplicarles operaciones genéticas se le llama *Regla de la Ruleta*. Este nombre viene del hecho de que cada individuo tiene la probabilidad de ser escogido de acuerdo con su parámetro de aptitud, que en este caso es $f(x)$, es decir, el azar determina a los elementos escogidos como si se tratara de una ruleta donde es más fácil que los ganadores sean determinados números. Existe otro criterio para escoger individuos que se le llama *Elitista*, en este se elige un número constante de elementos que se operan, es decir, el programador elige a los individuos que se operarán de acuerdo con su parámetro de aptitud, esto es, que los mejores individuos siempre serán escogidos [11].

Las operaciones genéticas descritas anteriormente, son las más utilizadas en los algoritmos genéticos pero existen muchas otras como son la Inversión y la Combinación Múltiple, que se utilizan para problemas especiales que son de difícil convergencia [5]. La Inversión consiste en tomar dos posiciones al azar de una cadena e invertir los genes que quedaron dentro de estas posiciones, esto es

$$0110110 \rightarrow 01|1011|0 \rightarrow 01|1101|0 \rightarrow 0111010.$$

La Combinación Múltiple de dos cadenas consiste en tomar varias posiciones al azar y combinar las cadenas de la misma manera que en la combinación simple, es decir

$$\begin{array}{ccccc} 1110110 & & |111|01|10| & & |011|01|11| & & 0110111 \\ 0111011 & \rightarrow & |011|10|11| & \rightarrow & |111|10|10| & \rightarrow & 1111010 \end{array}.$$

Las posibilidades que se tienen con los algoritmos genéticos son muchas y muy bastas, pero en realidad se trata básicamente de un método de búsqueda de soluciones óptimas. La principal ventaja que tiene este nuevo método sobre los demás es que puede optimizar una función, por más compleja que sea, es decir,

*garantiza llegar al mínimo absoluto de una función sin necesidad de un punto de partida cercano a éste.* Esto es una consecuencia de que se analizan resultados diferentes en cada iteración y la gran posibilidad de operaciones genéticas que se tienen.

La principal desventaja que tienen los algoritmos genéticos es que no existe un criterio realmente eficaz para determinar en que momento hay que parar el proceso, es decir, cuantas generaciones son necesarias. Otra desventaja es que el tiempo de convergencia de los algoritmos genéticos (y que tienen todos los métodos de optimización) crece exponencialmente conforme crece el número de variables que contenga la función. Los algoritmos genéticos, también dependen mucho de la habilidad del programador para adecuar el algoritmo al problema. Aparte de diseñar un código genético que contenga todas las posibles soluciones al problema, también se deben ajustar parámetros como el tamaño de la población y el criterio para escoger los elementos de la población que se van a operar genéticamente con la combinación, reproducción y mutación.

## 2,2 Implementación del algoritmo

La implementación en computadora de un algoritmo para resolver un problema dado requiere de:

1. Seleccionar un lenguaje.
2. Implementar las estructuras de datos necesarias.
3. Hacer el programa lo más eficiente posible.

Un algoritmo genético puede ser implementado en cualquier lenguaje estructurado de computación (pascal, C, fortran, etc.), sin embargo, para hacer la programación más fácil, podemos buscar un lenguaje con herramientas que faciliten la programación. El lenguaje de programación C, nos proporciona

herramientas de programación muy útiles para los algoritmos genéticos, como son la facilidad para manejar estructuras y apuntadores, además de operadores de bits, que nos pueden evitar el manejo de cadenas de caracteres y sustituirlas por cadenas de bits que ocupan menos memoria y su manejo es mucho más eficiente.

Para poder empezar con las estructuras de datos que contendrá el programa, es necesario definir primero algunas constantes necesarias para el programa:

```
#define POPSIZE
#define LCHROM
#define NUMGEN
#define PCROSS
#define PMUT
```

Donde POPSIZE es el tamaño de la población, LCHROM es el tamaño máximo de la cadena de bits, NUMGEN es el número de generaciones o iteraciones que realizara el programa, PREP, PCROSS y PMUT son respectivamente la probabilidad de reproducción, de combinación y de mutación.

Los *algoritmos genéticos* procesan poblaciones de cadenas. Por ende, no resulta sorprendente que la estructura de datos primaria para un algoritmo genético sea una población de cadenas. Hay varias formas de implementar poblaciones en un programa, por ejemplo, podemos construir una población

como una *estructura o arreglo de individuos* como el siguiente:

| *Número de Individuo* | *genotipo cadena* | *fenotipo* $X$ | *parámetro* $F(X)$ |
|---|---|---|---|
| 1 | 00001111 | 15 | 225 |
| 2 | 00001001 | 9 | 81 |
| 3 | 00001101 | 13 | 169 |
| . | . | . | . |
| . | . | . | . |
| . | . | . | . |
| *POPSIZE* | 00000111 | 7 | 49 |

(2.7)

Donde cada individuo contiene el **fenotipo** (el o los parámetros decodificados), el **genotipo** (el cromosoma artificial o cadena de bits), el **parámetro de aptitud** (función objetivo), y **POPZISE** que es número de individuos o tamaño de la población.

El código en lenguaje C de la tabla 2.8 declara una estructura que puede contener a la población correspondiente a este modelo. Declaramos entonces a la población de el Algoritmo Genético, como un arreglo de la estructura **struct population**, en el arreglo **pop**[POPSIZE].

```
struct population {
float x;
float fitness;
char individual[LCHROM + 1];
} * pop[POPSIZE];
```

(2.8)

Refiriéndonos al modelo , tenemos las constantes: el tamaño de población POPSIZE, y la longitud de cadena LCHROM. El arreglo *individual[LCHROM+1]*, contiene la cadena de unos y ceros que representan el cromosoma artificial o

*genotipo* (éste arreglo esta indexado entre 0 y *[LCHROM+1]* porque una cadena de caracteres requiere de un elemento más para asignarle el fin de la cadena: NULL). La variable real llamada *fitness* es el parámetro de *aptitud* o valor de ajuste y una variable real llamada *x* es el valor decodificado de la cadena o *fenotipo*. Explorando un poco más, vemos que el arreglo *individual* es una cadena de caracteres que sólo contendrá unos y ceros, y que por lo tanto, utilizando más herramientas de el lenguaje C, ésta se podría cambiar por una cadena de bits de tamaño LCHROM, pero para los fines de este ejemplo es mejor dejarlo como esta, ya que es más comprensible aunque menos óptimo.

En el Algoritmo Genético, aplicamos operadores genéticos a una población entera en cada generación. Para implementar esta operación, podemos utilizar dos poblaciones que no se traslapen, simplificando así la generación de los descendientes y el reemplazo de los progenitores. Las declaraciones de las dos poblaciones *oldpop* y *newpop* están mostradas en la estructura 2.9 junto con la declaración de otras variables globales del programa. Con estas dos poblaciones, es una labor fácil crear nuevos descendientes de los miembros de *oldpop* usando los operadores genéticos; colocando esos nuevos individuos en *newpop*, y poniendo finalmente *newpop* en *oldpop*. Existen otros métodos de mayor eficiencia de almacenamiento para manejar poblaciones; podríamos mantener a una sola población traslapada (pop[POPSIZE]) y prestar atención más cuidadosa de quien reemplaza a quien en poblaciones sucesivas. Tampoco hay una razón en particular para que mantener el tamaño de la población constante, las poblaciones naturales seguramente cambian de tamaño y puede haber motivación para permitir variación del tamaño de la población de generación en generación durante la búsqueda genética artificial. Sin embargo, para nuestro ejemplo preferimos mantener las cosas sencillas, y esto a guiado la elección de

poblaciones no traslapadas de tamaño constante.

$$\begin{aligned} &struct\ population\ *oldpop[POPSIZE], *newpop[POPSIZE],\ *optsize;\\ &float\ li,\ ls,\ sumfitness\\ &double\ factor \end{aligned} \tag{2.9}$$

Las constantes y otras variables globales de nuestro ejemplo afectan la operación de todo el código. En la estructura *optsize* guardamos el mejor resultado que encuentre el algoritmo genético. Observando la estructura 2.9, vemos algunas constantes de tipo entero; entre ellas está *POPSIZE* que ya utilizamos en la estructura 2.8. Esta importante variable corresponde a lo que hemos estado llamando tamaño de población. Adicionalmente, las constantes *LCHROM* es la longitud de la cadena, *NUMGEN* es el número de generaciones, y PCROSS, PMUT son las probabilidades de combinación y mutación respectivamente. La variable real *sumfitness* es la suma de los valores de la variable *fitness* de la población. Esta variable es importante durante la *selección de ruleta*. Las demás variables *li*, *ls* y *factor*, son exclusivas de nuestro ejemplo, ya que son los límites inferior y superior que puede tomar la variable $x$, y el factor de conversión de entero a real.

Una vez definidas las estructuras necesarias, el siguiente paso en la implementación de un algoritmo genético es generar una población inicial de individuos. La población inicial, normalmente se genera aleatoriamente, ésto es que para cada individuo de la población, el genotipo tomará un valor aleatorio que después se decodificará en el fenotipo para encontrar después su valor de aptitud. La función *generatepop()* se encarga de generar aleatoriamente la cadena de unos y ceros y además llama a la función *decode(i)* que decodifica cada elemento de la población que va generando, la función *decode(i)* a su vez llama a la función *function(x)* que se encarga de encontrar el parámetro de aptitud de cada elemento de la población que se va generando. Las funciones

*generatepop()*, *decode(i)* y *function(x)* se muestran enseguida.

```
void generatepop(){
int i, j;
    for(i = 0; i < POPSIZE; i++){
        oldpop[i] = malloc(sizeof(structpopulation));
        newpop[i] = malloc(sizeof(structpopulation));
        for(j = 0; j < LCHROM; j++)
            oldpop[i]- > individual[j] = myrandom(2) + 48;
        oldpop[i]- > individual[LCHROM] = NULL;
        newpop[i]- > individual[LCHROM] = NULL;
        decode(i);
    }
}
```

```
void decode(int i){
int j;
float sum = 0.0, fac = 1.0;
    for(j = LCHROM - 1; j >= 0; j-){
        sum+ = (oldpop[i]- > individual[j] - 48) * fac;
        fac* = 2.0;
    }
    oldpop[i]- > x = sum * factor + li;
    oldpop[i]- > fitness = function(oldpop[i]- > x);
}
```

```
float function(float r){
int i;
float sum = 0.0;
    for(i = 1; i <= 5; i++)
        sum+ = (float)(i * cos((i + 1) * r + i));
    return − sum;
}
```

Con nuestra población inicializada, necesitamos entender los tres operadores *reproducción, combinación y mutación* esenciales para la operación de un Algoritmo Genético. Los tres operadores del algoritmo genético simple pueden ser implementados cada uno en funciones. Antes de que veamos cada rutina, debemos recordar el hilo común que atraviesa a los tres operadores: cada uno depende de una selección de operandos aleatoria (*ruleta*) o *elitista* en algunos casos. El método de la ruleta puede ilustrarse así: tirar un dardo cargado sobre un blanco que contiene a todos los elementos de la población, el dardo tenderá a caer sobre los elementos con mejor parámetro de aptitud y seleccionar el elemento sobre el que caiga el dardo.

La función 2.10 se encarga de implementar la descripción anterior:

```
int select(){
float random1, partsum = 0.0;
int j = 0;
    do{
        random1 = myrandom(32767)/32767.0 * sumfitness;
        dopartsum+ = oldpop[j]- > fitness;
        j + +;
    }while(partsum < random1 && j <= POPSIZE);
    return j - 1;
}
```
(2.10)

Esta es quizás la manera más sencilla para implementar la selección en un algoritmo genético, pero la *selección elitista* es en muchos casos más eficiente, aunque implementarla resulte más complejo, ya que se deben de hacer muchas pruebas para saber cuantos y cuales elementos de la población serán operados genéticamente y cuales morirán. Estos parámetros sólo se pueden encontrar implementando el algoritmo y haciendo pruebas con diferentes valores para medir su eficiencia en cada caso y hacer un examen estadístico para poder elegir los mejores valores de los parámetros.

De nuestras descripciones previas, sabemos que nuestro siguiente paso es la operación *cambinación (crossover)*. El operador combinación es implementado en una función que nosotros hemos llamado crossover. La rutina crossover toma dos cadenas progenitoras llamadas *oldpop[i]->individual* y *oldpop[j]->individual* y genera dos cadenas hijas llamadas newpop[i1]->*individual* y newpop[j1]->*individual*. Las posiciones de las cadenas (i,j,i1,j1) se pasan a la función crossover. Dentro de crossover las operaciones reflejan nuestra descripción anterior. En la función principal del programa determinamos si vamos a realizar el crossover con el actual par de cromosomas progenitores. Específicamente, echamos un volado con

una moneda cargada, si sale cara (verdadero) con la probabilidad PCROSS. El volado es simulado en la función *myrandom()*, donde esta a su vez llama a la rutina de número pseudoaleatorio *rand*. Dentro de la función, el lugar del cruce es seleccionado también con la función *myrandom*, el cual regresa un entero pseudoaleatorio entre cero y LCHROM límites inferior y superior especificados. El intercambio parcial de crossover es llevado a cabo en los dos anexos *for* en el código de la rutina crossover, ilustrada enseguida.

```
voidcrossover(inti, intj, inti1, intj1){
int r1, k;
    r1 = myrandom(LCHROM);
    for(k = 0; k < r1; k++){
        newpop[i1]- > individual[k] = oldpop[i]- > individual[k];
        newpop[j1]- > individual[k] = oldpop[j]- > individual[k];
    }
    for(k = r1; k < LCHROM; k++){
        newpop[i1]- > individual[k] = oldpop[j]- > individual[k];
        newpop[j1]- > individual[k] = oldpop[i]- > individual[k];
    }
}
```

La función *mutation* simplemente cambia algunos de los genes con una probabilidad de PMUT del elemento de la población elegido, su código se muestra enseguida:

```
void mutation(int i, int j){
int k;
  reproduction(i, j);
  for(k = 0; k < LCHROM; k + +)
   if(mygrandom(100) < PMUT)
    cwpop[j]- > individual[k] = (newpop[j]- > individual[k] == 48)?49 : 48;
}
```

*mutation* llama a su vez a la función *reproduction*, que no hace más que copiar tal cual la cadena de *oldpop*[i] a *newpop*[j]; esta función se encuentra enseguida:

```
void reproduction(int i, int j){
    strcpy(newopop[j]- > individual, oldpop[i]- > individual);
}
```

El código completo del programa de ejemplo puede consultarse en el apéndice A. Este programa, minimiza una función con varios mínimos locales, dentro de un intervalo dado. Las variables globales **li** y **ls** son los límites inferior y superior del intervalo.

# CAPITULO 3

# APLICACIÓN AL PROBLEMA DEL MÍNIMO DE ENERGÍA POTENCIAL DE PARTÍCULAS PEQUEÑAS.

## 3.1 Partículas pequeñas

Existe una energía de atracción entre dos átomos (o interacción potencial), cuando éstos se encuentran a una distancia relativamente corta entre ellos. Cuando dos o más átomos se unen por una interacción se forma un agregado de átomos o molécula. Una partícula pequeña es un agregado de átomos con un rango corto de interacción (aproximadamente de hasta 200 Å [Angstroms] ).

El estudio de partículas pequeñas tiene una gran importancia debido a que tienen características que no se encuentran en partículas grandes. Por ejemplo, una partícula pequeña puede tener simetrías que son prohibidas en una estructura cristalina como es el caso de la simetría 5.

Nosotros podemos simular la formación de partículas pequeñas a partir de un modelo bien definido y controlable. El comportamiento de un sistema real puede ser muy próximo a los hechos observables en condiciones experimentales específicas en la computadora. De esta forma, podemos evaluar cantidades que ayudan a entender el comportamiento de la materia en diversas condiciones físicas. Así, la computadora actúa como aparato de medición y en función de los datos de entrada y de las condiciones que definen el problema es posible realizar experimentos numéricos y de esta manera probar teorías analíticas,

hipótesis de simplificación, formas explícitas de los potenciales de interacción entre las partículas que componen a un sistema físico y, en especial, confrontar los resultados del experimento numérico con los datos de un experimento real.

Enseguida discutiremos la simulación de la configuración de mínima energía de partículas pequeñas utilizando potenciales oscilantes.

## 3.2 Cálculos energéticos

La energía de una estructura atómica ésta dada por la interacción potencial entre cada par de átomos. La interacción potencial de cada par de átomos a su vez esta dada por la distancia entre éstos. Así, podemos definir entonces a la energía de cada par de átomos como una función $V(r)$ donde $r$ es la distancia entre dos átomos.

La energía potencial $V(r)$ que existe entre dos átomos es muy difícil de encontrar. La forma precisa del potencial interatómico de un material se determina haciendo suposiciones y gran cantidad de simplificaciones. Hasta ahora, los potenciales interatómicos que se han calculado, son empíricos.

Actualmente existen numerosos trabajos de simulación que utilizan potenciales con un sólo mínimo. Tal es el caso del potencial de Lennard-Jones.6-12

$$V(r) = \frac{A}{r^{12}} - \frac{B}{r^6}$$

donde $A$ y $B$ son parámetros empíricos determinados a partir de medidas independientes en la fase gaseosa. Este potencial también se puede expresar en función de la profundidad del pozo ($E_0$) y de la separación en equilibrio para dos átomos ($r0$), de la siguiente forma

$$U(r) = E_0 \left[ \left( \frac{r0}{r} \right)^6 \left( 2 - \left( \frac{r0}{r} \right)^6 \right) \right]$$

donde r es la distancia entre dos átomos. Este potencial, se ha utilizado en el estudio de propiedades termodinámicas de gases y de algunos metales, también

se ha utilizado para estudiar partículas pequeñas. En la figura 3.1 podemos ver la forma del potencial.

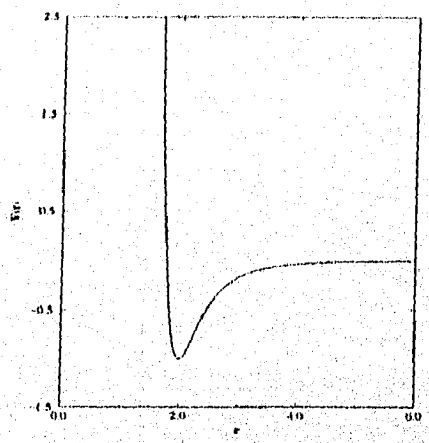



Figura 3.1: Representación de un potencial empírico con un mínimo

Actualmente se ha dirigido la atención a potenciales oscilantes, esto es, que contienen varios mínimos, como el que encontramos en la figura 3.2. Se ha comprobado mediante experimentos que la forma que contienen estos potenciales es más parecida a la que existe en la naturaleza. La figura 3.2, nos muestra la forma que puede tener uno de estos potenciales.

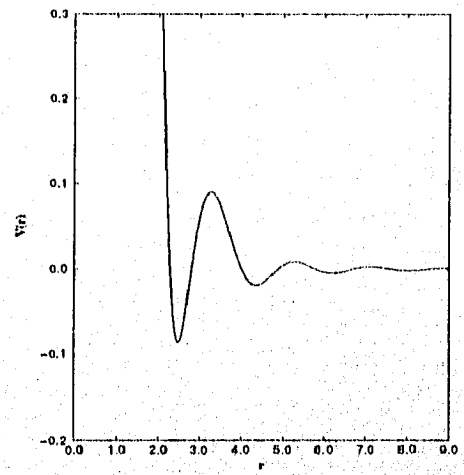



Figura 3.2: Representación de un potencial empírico con varios mínimos

Encontrar estructuras atómicas de mínima energía utilizando estos potenciales no es fácil, debido a que tienen uno o varios mínimos locales, y por lo tanto existen una gran variedad de configuraciones que pueden encontrar las partículas y que son tan estables que parecen ser estructuras de mínima energía, pero que en realidad no lo son, como la que podemos ver en la figura 3.3.

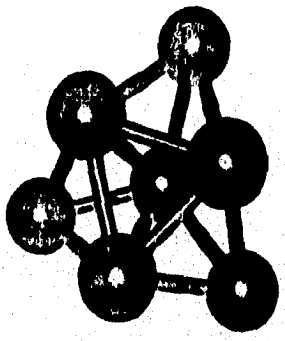


Figura 3.3: Configuración metaestable con N = 7

Para encontrar las estructuras de mínima energía, se han utilizado varios métodos como los revisados en el capítulo I, debido a que estos son algoritmos que pueden minimizar funciones no lineales de varias variables. Hasta ahora el método más utilizado para este problema ha sido el *método de simulación de recocido* (*Simmulated Annealing*), que ha probado ser el más eficiente, debido ha que puede encontrar mínimos absolutos partiendo de posiciones de los átomos totalmente aleatorias. Pero esto no lo exchuye de poder caer eventualmente en mínimos locales.

Uno de los objetivos de este trabajo es utilizar los algoritmos genéticos para minimizar la energía de partículas pequeñas, y medir la eficiencia de este nuevo método de simulación.

## 3.3 Cálculo del mínimo

Como hemos visto, el problema es encontrar estructuras atómicas pequeñas de mínima energía. La energía de una estructura atómica esta dada por la

interacción potencial entre cada par de átomos. La interacción potencial de cada par de átomos a su vez esta dada por la distancia entre estos. Así, se ha definido a la energía de cada par de átomos como una función $V(r)$ donde $r$ es la distancia entre dos átomos. $V(r)$ contiene la energía potencial entre dos átomos de distancia $r$, pero la energía total $E$ de una partícula pequeña es la suma de la energía que hay entre cada par de átomos

$$E = \frac{1}{2}\sum_{i,j=1}^{n} V(r_{ij}),\ i \neq j \tag{3.1}$$

donde $n$ es el número total de átomos en la partícula, y $r_{ij}$ es la distancia del átomo $i$ al átomo $j$ (se toma un medio de la suma porque la interacción del átomo $i$ con el átomo $j$ es la misma que del átomo $j$ con el átomo $i$). Esta función determina la energía potencial de una molécula o partícula pequeña.

La distancia entre dos átomos está en función de la posición en el espacio de cada átomo. Es decir, que si queremos medir la distancia entre dos átomos, primero necesitamos tener la posición de éstos. Tenemos dos formas de representar esta posición numéricamente; una de ellas es por medio del sistema de coordenadas rectangulares, es decir, que cada átomo tiene una posición $(x, y, z)$ en el espacio. La otra forma de representar la posición de los átomos es mediante el sistema de coordenadas esféricas , es decir, que cada átomo tiene una posición $(r, \theta, \phi)$ en el espacio.

Para calcular la distancia entre dos átomos cuando su posición se representa por medio de coordenadas rectangulares es simplemente:

$$r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$$

donde $(x_i, y_i, z_i)$ son las coordenadas cartesianas del i-ésimo átomo.

Para calcular la distancia entre dos átomos cuando su posición se representa por medio de coordenadas esféricas :

$$r_{ij} = \sqrt{r_i^2 + r_j^2 - 2 * r_i * r_j * (sin(\theta_i) * sin(\theta_j) * cos(\phi_i - \phi_j) + cos(\theta_i) * cos(\theta_j))}$$

donde $(r_i, \theta_i, \phi_i)$ son las coordenadas esféricas del i-ésimo átomo. Esta última es sin duda más complicada, sin embargo, como veremos la representación en forma esférica resultará muy conveniente, debido a que los ángulos $\theta$ y $\phi$ sólo pueden tomar valores entre 0 y $2\pi$ y podemos entonces codificarlos en forma binaria más fácilmente, además $r$, por ser la distancia del origen al átomo, toma valores únicamente positivos y también podemos darle a $r$ un valor máximo, que será equivalente a obligar a la partícula a moverse sólo dentro de una esfera de radio $r$. Esto último es muy conveniente pues podemos simular el efecto de una presión sobre la partícula.

De esta manera, el problema a resolver es el siguiente:

$$Minimizar\ E = \frac{1}{2}\sum_{i,j=1}^{n} V(r_{ij}),\ i \neq j \tag{3.2}$$

con lo que encontraremos la posición en el espacio que deben tener los átomos de la partícula, es decir, las coordenadas $(x, y, z)$ o bien $(r, \theta, \phi)$ de cada uno de los $n$ átomos que la componen y así poder obtener su configuración geométrica.

Finalmente conviene hacer un comentario sobre el tiempo de cómputo que hay que invertir como función del tamaño de la partícula. El tipo de problema al que pertenece minimizar una función con un número de variables que puede aumentar se le denomina *NP (non-deterministic polynomial-time)*[14], esto quiere decir que el tiempo de cómputo crece a medida que la partícula tiene más átomos. A su vez los problemas NP se dividen en NP-completo y NP-duro, que se refieren a la forma en que crece el tiempo de cómputo. El problema al que nos enfrentamos, se ha comprobado que es del tipo NP-duro, esto quiere decir que el tiempo de cómputo para resolverlo *crece de manera exponencial* [14] conforme aumenta el número de átomos de la partícula que queremos minimizar. Se trata entonces, de un problema que está lejos de ser trivial.

## 3.4 Implementación de un Algoritmo Genético

Para aplicar un algoritmo genético al problema de la ecuación 3.2, primero definiremos el esquema de representación de las coordenadas cartesianas y después para coordenadas esféricas .

Las coordenadas de una partícula de $n$ átomos serían:

$$(x_1, y_1, z_1), (x_2, y_2, z_2), \ldots, (x_n, y_n, z_n)$$

donde el rango de valores que pueden tomar es:

$$(-\infty, -\infty, -\infty) < (x_i, y_i, z_i) < (+\infty, +\infty, +\infty)$$

Como se puede observar, la cadena de números binarios que realmente necesitaríamos para representar las soluciones al problema, sería infinitamente grande. Sin embargo, no necesitamos realmente un espacio tan grande para poder formar una partícula de $n$ átomos en el espacio. Nosotros podemos definir un espacio específico en donde queramos formar la partícula, dejando que los átomos se acomoden libremente en ese espacio. Así que podemos definir la formación de la partícula dentro de un poliedro rectangular de lados $L_1$, $L_2$, $L_3$ y definir el rango de las coordenadas de la siguiente forma:

$$(0, 0, 0) < (x_i, y_i, z_i) < (L_1, L_2, L_3)$$

Ahora podemos definir una cadena finita de unos y ceros. Podemos aprovechar que en lenguaje C al declarar un entero corto positivo (unsigned short Int) reservamos dos bytes de memoria que pueden representar un número entero entre 0 y 65535 pero en lenguaje máquina tenemos una cadena de unos y ceros de longitud 16. Un ejemplo de esto son las siguientes cadenas:

| *cadena* | *número entero* |
|---|---|
| 0000000000001111 | 15 |
| 1010101000001001 | 43529 |
| 1111111111111111 | 65535 |

De manera que si queremos representar un número real entre 0 y $L_1$ basta con multiplicar el número entero que ya tenemos representado implícitamente en la cadena de unos y ceros por un factor $\frac{L_1}{65535}$ y tendremos la representación de una de la coordenada $x$ (La precisión del resultado dependerá de que tan pequeño sea $L_1$, entre más grande sea el resultado será menos preciso). Un ejemplo de la representación de la coordenada $x$ para $L_1 = 10.0$ sería:

| *cadena* | *número entero* | $x$ |
|---|---|---|
| 0000000000001111 | 15 | 0.002288853284 |
| 1010101000001001 | 43529 | 6.642009641 |
| 1111111111111111 | 65535 | 10.0 |

Ahora tenemos ya una representación en una cadena binaria para las coordenadas cartesianas $(x, y, z)$.

El siguiente paso ahora es definir la medida de aptitud, ésta será obviamente la energía de la partícula que definimos en la ecuación 3.1.

Los parámetros y las variables que controlarán el algoritmo, así como el código completo del programa que encuentra el mínimo de energía de partículas pequeñas se puede consultar en el apéndice B.

# CAPITULO 4

# RESULTADOS

## 4.1 Introducción

En esta sección se presentan los resultados de la aplicación del Algoritmo Genético al problema de minimizar la energía de partículas pequeñas de oro usando potenciales semiempíricos con dos mínimos. Se escogió el oro ya que es un metal que se presta para ser estudiado y además se cuenta con información de otras fuentes que puede servir para hacer una comparación con el presente trabajo [13].

Se analizarán partículas pequeñas de 4 hasta 26 átomos. El objetivo es comparar con las que resultan de usar un potencial estándar de Lennard-Jones con un sólo mínimo.

## 4.2 Descripción del Potencial

Los resultados que a continuación se presentan fueron obtenidos usando un potencial simiempírico que es una variación del potencial de Lennard-Jones 6-12. La variación consiste en que favorece la formación de configuraciones pentagonales en una partícula, debido a que tiene dos mínimos, el primero de ellos se encuentra en el punto de equilibrio $r_0$ y el segundo mínimo se encuentra en $r_0 + \tau$. $\tau$ es la llamada razón áurea (o razón dorada), que tiene que ver mucho con la simetría 5, como se muestra en la figura 4.1.

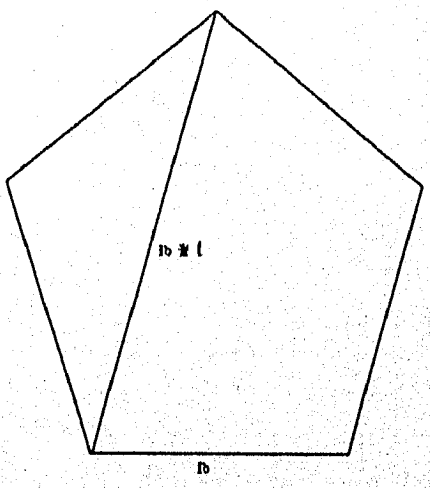


Figura 4.1: Relación de $\tau$ con la simetría 5

En la figura 4.1 podemos ver como $\tau$ esta directamente relacionada con cualquier pentágono.

La expresión del potencial en función de la profundidad del pozo ($E_0$) y de la separación en equilibrio para dos átomos ($r_0$) es

$$V(r) = -E_0\left[\frac{1}{3}\left(\frac{r_0}{r}\right)^4\left[7 - 4\left(\frac{r_0}{r}\right)^3 - Ce^{-\alpha(\frac{r}{r_0}-\beta)^2} + De^{-\alpha(\frac{r}{r_0}-r)^2}\right]\right]$$

donde $C = 0.73$, $\beta = 1.35$, $\alpha = 50$, $D = 0.09$ y $\tau = \frac{1+\sqrt{5}}{2}$. Su gráfica se muestra en la figura 4.2.

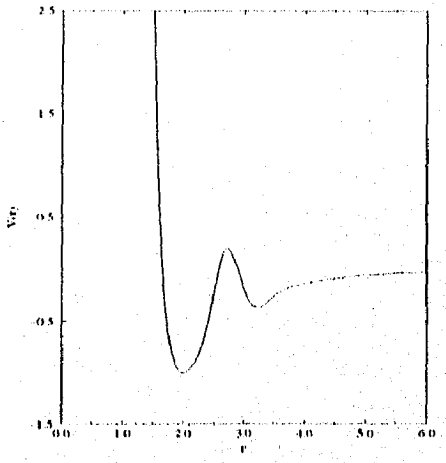



Figura 4.2: Gráfica de un potencial empírico con dos mínimos

Actualmente existen numerosos trabajos de simulación que utilizan potenciales empíricos, pero éstos no dicen por qué es mejor utilizar un potencial y no otro. El potencial de Lennard-Jones ha sido utilizado para el problema de minimización de energía de partículas metálicas pequeñas, y para cálculos de estructura y energía de frontera de grano en metales [13]. Su expresión es la siguiente:

$$V(r) = A/r^{12} - B/r^{6}$$

Donde A y B son parámetros empíricos determinados a partir de medidas independientes en la fase gaseosa. Este potencial también se puede expresar en función de la profundidad del pozo ($E_0$) y de la separación en equilibrio para dos átomos ($r_0$), de la siguiente forma:

$$U(r) = E_0 \left[ \left( \frac{r_0}{r} \right)^{6} \left( 2 - \frac{r_0}{r} \right)^{6} \right]$$

Y su gráfica es la de la figura 4.3.

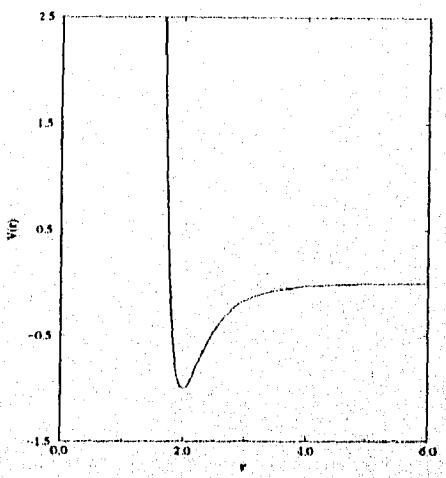



Figura 4.3: Gráfica de un potencial simple de Lennard-Jones

## 4.3 Configuraciones de mínima energía potencial

Se aplicaron las técnicas de algoritmos genéticos descritas a lo largo de este trabajo para encontrar la configuración de mínima energía de partículas de oro de N = 4 a N = 26 átomos. Los resultados se muestran en la tabla 4.1. La primer columna es el tamaño de la partícula, la segunda columna indica la energía (en unidades arbitrarias), la tercer columna muestra el número de generaciones del algoritmo genético con las que se llego a la configuración de mínima energía, partiendo de una configuración aleatoria de átomos, y la cuarta columna indica el número de mutaciones por generación con las que se comenzó el algoritmo genético (recordemos que el número de mutaciones disminuye conforme avanza el programa). La última columna indica el tiempo de CPU medido en una

Silicon-Graphics INDY (R5000, 150MHz) bajo IRIX 5.3.

| N | Energía | generaciones | mutaciones | tiempo en seg. |
|---|---|---|---|---|
| 4 | −5.99 | 3000 | 10 | 2 |
| 5 | −9.35 | 3000 | 20 | 4 |
| 6 | −13.07 | 5000 | 30 | 8 |
| 7 | −17.79 | 8000 | 50 | 19 |
| 8 | −27.20 | 8000 | 50 | 24 |
| 9 | −37.72 | 12000 | 60 | 47 |
| 10 | −38.59 | 12000 | 80 | 58 |
| 11 | −38.59 | 15000 | 90 | 88 |
| 12 | −45.49 | 20000 | 100 | 133 |
| 13 | −53.41 | 30000 | 130 | 238 |
| 14 | −58.35 | 30000 | 140 | 278 |
| 15 | −64.33 | 35000 | 150 | 365 |
| 16 | −70.66 | 50000 | 160 | 609 |
| 17 | −70.66 | 50000 | 160 | 822 |
| 18 | −84.70 | 80000 | 200 | 1200 |
| 19 | −92.88 | 100000 | 200 | 1692 |
| 20 | −99.51 | 130000 | 200 | 2448 |
| 21 | −106.27 | 150000 | 200 | 3055 |
| 22 | −114.25 | 180000 | 200 | 3996 |
| 23 | −122.88 | 210000 | 220 | 5113 |
| 24 | −129.55 | 250000 | 230 | 6718 |
| 25 | −136.14 | 290000 | 240 | 8514 |
| 26 | −142.56 | 350000 | 250 | 10922 |

(Tabla 5.1)

donde la energía esta dada en unidades de $E_0$, y la separación en equi-

librio de los átomos es de $r_0 = 2$ Å.

La configuración para N = 4 es un tetraedro [Figura 1]. Esta configuración es sumamente estable, ya que tiene una longitud de $r_0$ (distancia en equilibrio para 2 átomos) en cada lado. Es además la primera configuración atómica que presenta volumen, y se presenta con mucha frecuencia en sólidos amorfos y otras estructuras cristalinas [13].

La configuración de N = 5 es una bipirámide triangular [Figura 1]. Esta formada por dos tetraedros opuestos, que comparten la misma base, también resulta ser una configuración estable y simétrica.

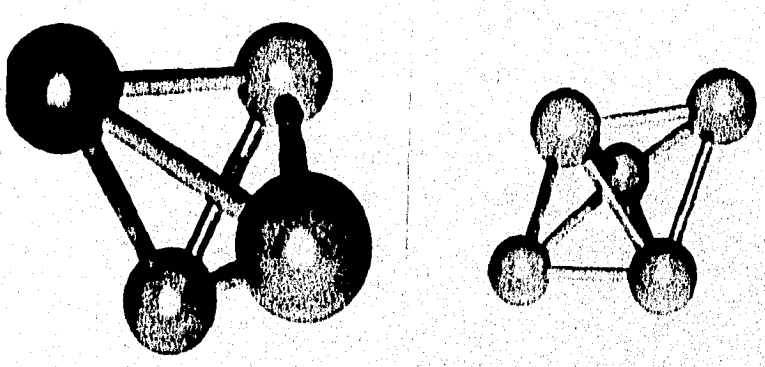


Figura 1: Configuraciones con N = 4 y N = 5

La configuración lograda con N = 6 átomos [Figura 2], es la primera que muestra influencia del potencial que favorece a la geometría cinco, ya que con un potencial común de Lennard-Jones, la configuración de mínima energía que se forma es el octaedro (energía de -11.55) [13], mientras que la encontrada con este nuevo potencial empírico es una configuración que forman tres tetraedros. Esta configuración es parte de un decaedro o bipirámide pentagonal, ya que se puede ver que sólo le hace falta un átomo para poder formarlo.

Para N = 7 se forma el decaedro o bipiramide pentagonal [Figura 2]. Esta configuración es sumamente estable y es la primera configuración con simetría cinco.

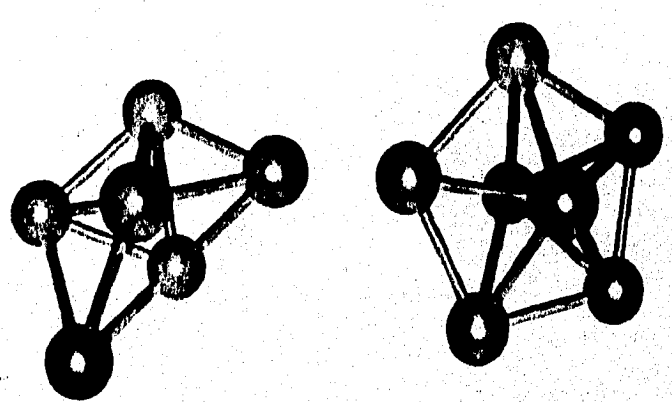


Figura 2: Configuraciones con N = 6 y N = 7

La configuración de N = 8 es un decaedro con un átomo añadido formando un tetraedro más [Figura 3]. No es una configuración muy estable, pero sigue estando formada por tetraedros como las demás configuraciones.

En la configuración de N = 9 vemos un decaedro con dos átomos añadidos formando tetraedros [Figura 3]. Consiste en realidad de dos decaedros trasladados.

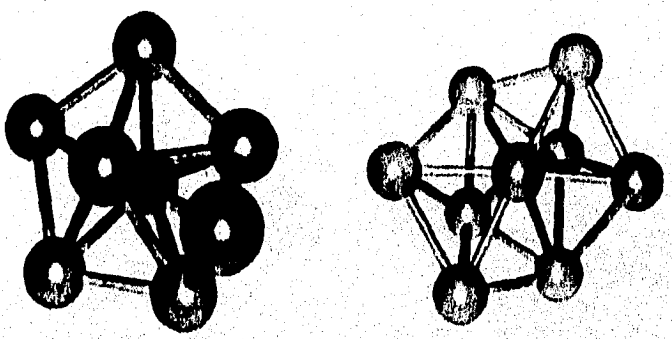


Figura 3: Configuraciones con N = 8 y N = 9

Con N = 10 [Figura 4] tenemos que simplemente se añade el átomo a la configuración con N = 9 [Figura 3], formando un tetraedro más, y puede observarse que se forma de tres decaedros traslapados.

Para N = 11 [Figura 4] se tiene la configuración anterior con otro decaedro añadido.

También para la configuración de N = 12 [Figura 5] ocurre lo mismo, pero en ésta vemos que sólo le falta un átomo para formar un icosaedro, con lo que podemos ver que efectivamente el potencial empírico que se está probando tiende a formar configuraciones con volumen y simetría cinco.

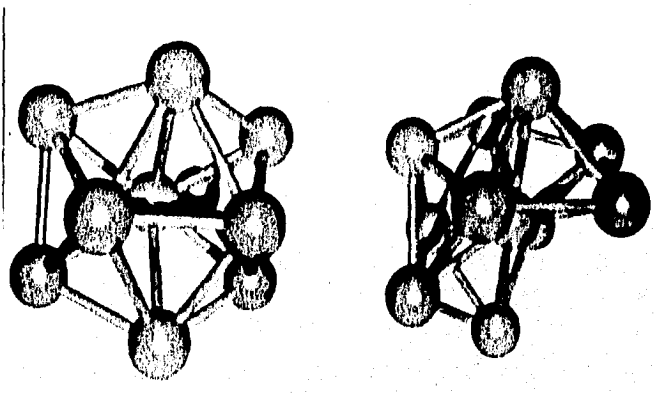


Figura 4: Configuraciones con N = 10 y N = 11

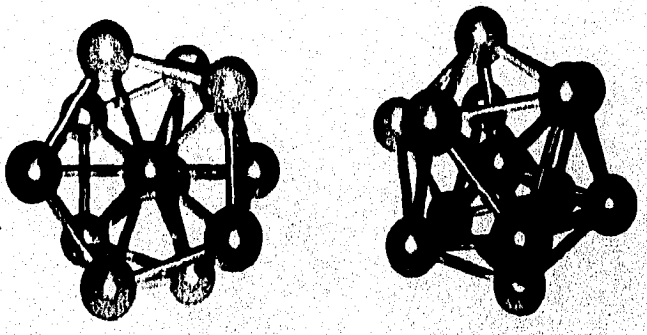


Figura 5: Configuraciones con N = 12 y N = 13

La configuración que se forma con N = 13 es efectivamente un icosaedro (Figura 5). Esta configuración es muy especial, ya que además de poseer simetría

cinco, es la configuración regular con mayor simetría después de la esfera. Tiene doce ejes de rotación con simetría cinco, veinte ejes de simetría tres y treinta ejes de simetría dos. La simetría del icosaedro lo convierte en el sólido con menor razón superficie-volumen.

Para N = 14 [Figura 6], la configuración que se forma es un icosaedro con un átomo añadido completando un tetraedro más. Cabe mencionar que hasta ahora, todas las configuraciones están formadas de tetraedros y preferentemente estos completan decaedros traslapados. También las configuraciones de N = 15 [Figura 6] hasta N= 18 [Figura 8] son el icosaedro con átomos añadidos de tal manera que completan decaedros traslapados.

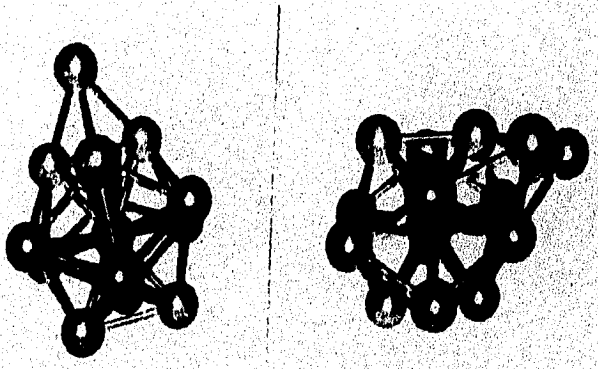


Figura 6: Configuraciones con N = 14 y N = 15

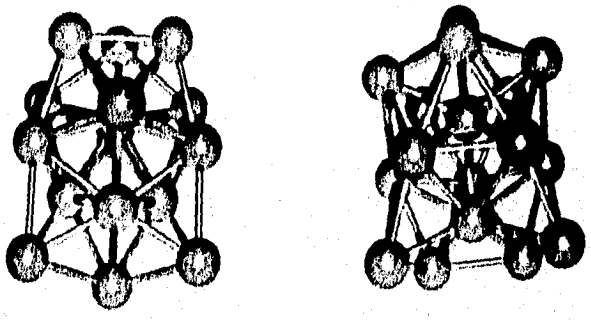


Figura 7: Configuraciones con N = 16 y N = 17

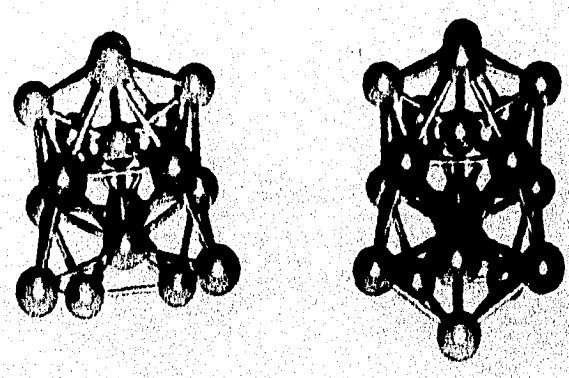


Figura 8: Configuraciones con N = 18 y N = 19

La configuración con N = 19 es tal que pueden verse dos icosaedros traslapados que comparten un decaedro [Figura 8]. Por lo tanto también tiene

simetría cinco y es la partícula más pequeña con dos átomos en el centro, cada uno de estos átomos es también el centro de un icosaedro.

Hasta ahora podemos ver que las configuraciones que se han formado efectivamente tienden a formar configuraciones con volumen y de simetría cinco, y que además están formadas de tetraedros que tienden a completar decaedros y ahora ya han completado dos icosaedros traslapados en la configuración de $N = 19$.

Para las configuraciones correspondientes a N = 20 [Figura 9] hasta $N = 22$ [Figura 10], se observa que sigue siendo la misma configuración que con 19 átomos con más átomos añadidos, que siguen completando decaedros.

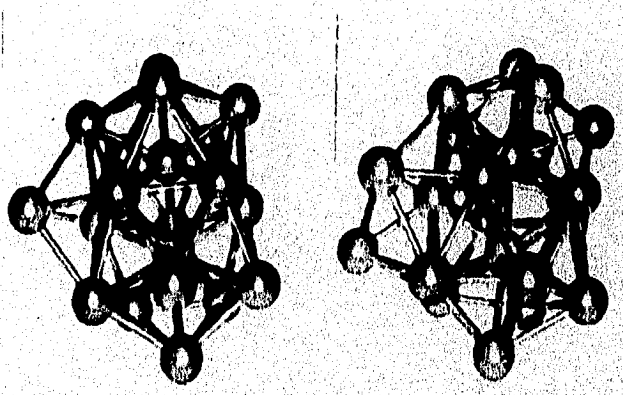


Figura 9: Configuraciones con N = 20 y N = 21

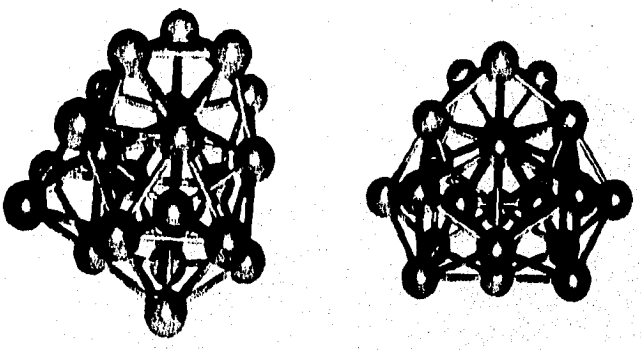


Figura 10: Configuraciones con N = 22 y N = 23

La configuración correspondiente a N = 23 átomos [Figura 10] forma dos configuraciones como la de N = 19 átomos [Figura 8] traslapadas. Esta configuración también es muy estable.

Las configuraciones correspondientes a N = 24 y N = 25 [Figura 11] corresponden a la misma configuración de N = 23 [Figura 10] con uno y dos átomos añadidos respectivamente.

Finalmente la configuración correspondiente a N = 26 [Figura 12] tiene simetría cinco y pueden verse icosaedros traslapados.

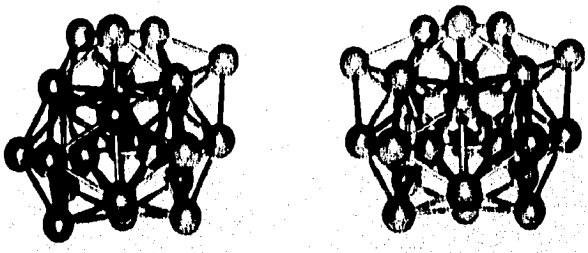


Figura 11: Configuraciones con N = 24 y N = 25

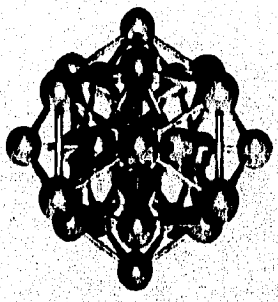


Figura 12: Configuración con N= 26

# Parte III

# CONCLUSIONES

En este trabajo se realizó una aplicación de los Algoritmos Genéticos para la hallar la configuración de mínima energía de partículas pequeñas de oro.

Los Algoritmos Genéticos son un nuevo método de optimización con potencial para reemplazar a los métodos más tradicionales para resolver problemas que involucran hallar el mínimo de una función de múltiples variables.

En este trabajo se implemento un Algoritmo Genético eficiente para encontrar estructuras atómicas de mínima energía utilizando un potencial empírico y obteniendo resultados que se comparan con los ya conocidos pero obtenidos por otros métodos. También se aplicó un potencial empírico con dos pozos (mínimos) que favorece la simetría 5 encontrándose resultados nuevos en los que se verifica que el potencial favorece las estructuras pentagonales.

De acuerdo con los resultados obtenidos se puede decir que los Algoritmos Genéticos son eficientes y confiables en la resolución de este tipo de problemas. Analizando los métodos vistos en el capítulo 1 y comparándolos con un Algoritmo Genético, podemos concluir lo siguiente:

Una de las grandes desventajas de los métodos revisados en el capítulo 1 como el método de pasos descendientes, el método del gradiente conjugado y el método de Newton, es que suponen que la función debe ser diferenciable, con objeto de poder avaluar el gradiente, lo cual es una desventaja al evaluar analíticamente en cada iteración las funciones contenidas en el gradiente. Recordemos que las funciones del gradiente deben ser alimentadas apriori en forma explícita al programa, donde los parámetros de estas funciones cambian de valor en cada iteración. Esta tarea de evaluación analítica, puede ser muy laboriosa y en ocasiones imposible, sobre todo cuando se trata de funciones complejas con muchas variables independientes. Además, la mayor desventaja de estos métodos, es que sólo llegan a un mínimo local, o al más cercano al

punto inicial.

El método de simulación de recocido (Simmulated Annealing), es un método estocástico que sin ser tan eficiente en cuanto al número de iteraciones necesarias para alcanzar un mínimo (máximo) como los métodos basados en el gradiente, tiene la ventaja de que sirve tanto para funciones diferenciables como no diferenciables y además puede llegar al mínimo (máximo) absoluto si el decremento en la temperatura es lo suficientemente pequeño (lo que puede implicar enormes tiempos de cómputo).

Recapitulando, los Algoritmos Genéticos son un mecanismo de búsqueda eficiente y rápido, capaz de ser aplicado a una gran variedad de problemas. El programa desarrollado es eficiente y práctico para partículas pequeñas. En agregados grandes hay que implementar un esquema de codificación diferente. Cabe mencionar que en el caso del problema que aquí se plantea (minimizar la energía de partículas pequeñas), el tiempo de cómputo aumenta exponencialmente conforme aumenta el número de átomos de la partícula (ver capítulo III).

Existen varias formas de implementar un Algoritmo Genético para resolver un problema específico, pero al implementarlo se tiene que ajustar el algoritmo al problema, es decir, es necesario afinar el programa en cuanto al porcentaje de Reproducción, Combinación y Mutación necesarios para el óptimo funcionamiento de un programa en cada problema. Además también es necesario hacer un análisis acerca de cuantas generaciones son necesarias para que el programa llegue al resultado, ya que no hay manera de saber cuando detener un Algoritmo Genético.

# Parte IV

# REFERENCIAS

1. Goldberg, David E., "The Existential Pleasures of Genetic Algorithms", Reporte interno, 1994. Illinois Genetic Algorithms Laboratory (IlliGAL).

2. P. Sutton and S. Boyden, "Genetic Algorithms: A general Search Procedure", Am. J. Phys. 62, June 1994, 549-552.

3. Keith, Grant, "An Introduction to Genetic Algorithms", C/C++ Users Jounal Advanced Solutions for C/C++ Programmers, 13, Nuber 3 , March 1995., pp 45-58.

4. Forrest, Stephanie, "GeneticAlgorithms: Principles of Selection Applied to Computation", SCIENCE 261, 13 August 1993, pp 872-878.

5. Goldberg, David E. "Genetic Algorithms in search, optimization and Machine Learning", Addison-Wesley, (Reading MA,1989).

6. Davis, L. "Handbook of Genetic Algorithms", Van Nostrand (Reinhold NY, 1991).

7. Koza, J. R. "Genetic Programming. On the Programming of Computers by Means of Natural Selection", The MIT Press, (Cambridge, MA 1993).

8. Pierre, Aonald A. "Optimization Theory with Applications", General Publishing Company, Canada, 1986.

9. Prawda, Juan, "Métodos y modelos de Investigación de Operaciones vol. I Modelos Deterministicos", Limusa, México 1990.

10. Xino, Yongliang and Williams, E. Donald, "Genetic algorithm: a new approach to the prediction of the structure of molecular clusters", Chemical Physics Letters, Vol 215, number 1,2,3. 26 November 1993. pp 17-24.

11. Holland, John H. "Genetic Algorithms", Scientific American, July 1992, pp 44-50

12. Mahfoud, Samir W., Goldberg, David E. "Parallel Recombinative Simulated Annealing: A Genetic Algorithm", Illinois Genetic Algorithms Laboratory (IlliGAL) Report No. 93006-July 1993.

13. Terrones Maldonado, Humberto, "Potenciales Empíricos aplicados al problema

de la estructura en partículas pequeñas ", Tesis, Universidad Iberoamericana, Mex. 1996.

14. Wille, L.T. and Vennik, J., "Computational complexity of ground-state determination of atomic clusters", Letter to the Editor, J. Phys. A: Math. Gen. 18 (1995) L429-L422, Printed in Great Britain.

# Parte V

# APÉNDICE A

```
/*****************************************/
/*    Simple Genetic Algorithm - SGA */
/*        Haploid Version                */
/*     Mario Perez Alvarez  1995         */
/*     Instituto de Fisica UNAM          */
/*                                       */
/*                                       */
/*****************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#define FAC 65535.0
#define S 1
#define P_COMB 4
#define P_REP 1
#define P_MUT 1
struct pobla{
      unsigned short  string;
      float  valor,fitnes;
      }*ip[100];
short next;
short myrandom(short);

void valor_f(struct pobla * );

float fun(float r){
int i;
float sum = 0.0;
     for(i=1;i<=5;i++)
          sum += (float)(i*cos( (i+1)*r + i ));
     return sum;
}



void reproduccion(short,short);
void combinacion(short,short);
void mutacion(short,short);
void ordena(void);
void generapobla(void);
void destruyepobla();


float li,ls;
short md,n = 10;
main(){
short r1,j,t=0,i;
```

```
/* determine input and output from program args */

 srand( (unsigned) time(NULL) );
 printf("Limites Inferior y Superior\n");
 scanf("%d %d",&li,&ls);
 printf("Número de dígitos = ");
 scanf("%d",&md);

 generapobla();
 ordena();
printf("generacion  0:    r = %f   V(r) = %f  \n",ip[0]->valor,ip[0]->fitnes );
 while( t < 50 ) {    /*  */
     j=6; i=0;
     do{
          reproduccion(j,i);
           i++;
           j++;
       }while(j<8);
       i=0;
          do{

          combinacion(i,i+1);

           i+=2;

     }while(i<6);
     j=6; i=0;
     do{
          reproduccion(j,i);
           i++;
           j++;
       }while(i<6);
       i=0;
       do{
              mutacion(j,i);
              i++;
       }while(j<n);

       ordena();
       ++t;
     printf("generacion  %d:    r = %f   V(r) = %f  \n",t,ip[0]->valor,ip[0]->fitnes );
 }
 destruyepobla();
 return 0;
}

/*-------------------------------------------------------------------*/
/* - create a new generation of individuals                          */
/*-------------------------------------------------------------------*/
```

```
void generapobla(){
short i;
   for(i=0;i<n;i++){
      ip[i] = malloc(sizeof(struct pobla));
      ip[i]->string=myrandom(FAC);       /* DEFINE UNA POBLACION INICIAL. */
      ip[i]->valor = ls - ip[i]->string*(ls-li)/FAC;
      ip[i]->fitnes = fun(ip[i]->valor);
   }
}

/*-------------------------------------------------------------------------*/
/*  - contains random number generator and related utilities,              */
/* including advance_random, warmup_random, random, randomize, flip, and rnd  */
/*-------------------------------------------------------------------------*/

short myrandom(short r)
{
  return rand()%r;
}
void valor_t(struct pobla *i){

        i->valor = ls - i->string*(ls-li)/FAC;
}

/*-------------------------------------------------------------------------*/
/*  sort for selection individuals.                                        */
/*-------------------------------------------------------------------------*/

void ordena(){
short i,j;
struct pobla *aux;
   for(i=0;i<n;i++)
     for(j=i+1;j<n;j++)
       if( ip[i]->fitnes  >  ip[j]->fitnes ){
         aux = ip[i];
         ip[i] = ip[j];
         ip[j] = aux;
       }
}

/*-------------------------------------------------------------------------*/
/*  - Genetic Operators: Reproduction, Crossover and Mutation              */
/*-------------------------------------------------------------------------*/
void reproduccion(short i,short j){

   ip[j]->string = ip[i]->string;
   ip[j]->valor = ip[i]->valor;
   ip[j]->fitnes = ip[i]->fitnes;
```

```
}

void mutacion(short j, short i){
short r1;
   r1=myrandom(16);
   ip[j]->string = ip[i]->string^(short)(pow(2,r1)+0.5);
   ip[j]->valor = ls - ip[j]->string*(ls-li)/FAC;
   ip[j]->fitnes = fun(ip[j]->valor);

}

void combinacion(short l,short j){
short r1,aux;
   r1=myrandom(16);
   aux =ip[i]->string;
   ip[i]->string = (ip[i]->string&~(short)(pow(2,r1)+0.5))|(ip[j]-
>string&(short)(pow(2,r1)+0.5));
   ip[j]->string = (ip[j]->string&~(short)(pow(2,r1)+0.5))|(aux&(short)(pow(2,r1)+0.5));
   ip[i]->valor = ls - ip[i]->string*(ls-li)/FAC;
   ip[i]->fitnes = fun(ip[i]->valor);
   ip[j]->valor = ls - ip[j]->string*(ls-li)/FAC;
   ip[j]->fitnes = fun(ip[j]->valor);
}
/*-------------------------------------------------------------------------*/
/* - free Memory                                                           */
/*-------------------------------------------------------------------------*/

void destruyepobla(){
short l;
   for(l=0;l<n;l++)
      free(ip[l]);
}
```

# Parte VI

# APÉNDICE B

```
/*

PROGRAMA DE SIMULACION DE RELAJAMIENTO DE PARTICULAS PEQUE&AS.
USANDO ALGORITMOS GENETICOS

*/
/*****************************************/
/*      Genetic Algorithm - SGA          */
/*                                       */
/*      Mario Perez Alvarez  1995        */
/*      Instituto de Fisica UNAM         */
/*                                       */
/*                                       */
/*****************************************/
# define Iris

# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <math.h>
# include <time.h>
#if defined(Iris)
# include "/usr/local/include/forms.h"      /* Forms */
#endif
# define PI 3.141592654
# define METAESTABLE 5000



struct element              /* Contiene La estructura atomica de entrada del PDB */
       {
        char symbol[2];     /* simbolo atomico */
          float xyz[3];     /* coordenadas del atomo (x,y,z) */
          short tipo;       /* auxiliar de tipo de atomo */
       };

struct poblacion            /* Contiene la poblacion del algoritmo Genético */
     {
         unsigned short xyz[3][60];  /* coordenadas enteras de los atomos (x,y,z) (para
manejo de bits) */
         double energia;             /* energia de la particula peque&a (o del iesimo
elemento de la poblacion) */
        };

struct element *atom[60];       /* Estructura del archivo PDB */
struct poblacion *pobla[100];   /* Poblacion del A G  */
char name[4];                   /* contiene "ATOM" para verificar la entrada del PDB y
ponerlo en el PDB de salida*/
```

```
short no_atoms=0;                /* numero total de atomos de la particula */
float potencial[3][1000];        /* contiene las funciones de los potenciales que se
manejan */
char alal[90],alco[90],coco[90],pdbin[90],pdbout[90];      /* contiene los nombres de
los archivos de entrada */
unsigned long N_I;                                  /* numero de iteraciones */
unsigned int Nom_of_Pop,Max_fit,Nom_Max_of_Mut,Delta_of_Mut;  /* parametros del
A G (tama&o de poblacion, elementos a combinar, mutar,etc)*/
float R1;                                  /* radio de la esfera */
double factor1,factor2;                    /* factor de conversion de las coordenadas de
los atomos, de enteros a reales */
int mask1[15],mask2[15];                   /* mascaras auxiliares para la mutacion y la
combinacion */


void Read_INI(void);          /* lee el archivo genetic.ini que contiene los datos de
entrada del programa */
short ReadPDB(char *);        /* lee un archivo con formato PDB */
void ReadPotential(char *,char *, char *);   /* lee los archivos que contienen las
funciones de los potenciales */
float Fill_vector(short, short, char*);      /* convierte caracteres de entrada de los
archivos a enteros */
double EnergiaIni(void);                   /* calcula la energia del PDB leido */
void Generate_pop(void);                   /* genera la poblacion inicial aleatoriamente */
void Generate_new_pop(void);               /* genera una nueva poblacion inicial
aleatoriamente */
double energ(short);                       /* calcula la energia de la particula iesima de la
poblacion */
unsigned short myrandom(unsigned short);     /* regresa un numero aleatorio entre 0 y
n */
void mySort(void);                         /* ordena la poblacion por grado de aptitud */
void Reproduction(short,short);            /* operador de reproduccion de un elemento de
la poblacion*/
void Crossover(short,short);               /* operador de combinacion de dos elementos */
void Mutation(short,short);                /* operador de mutacion de un elemento de la
poblacion */
void Inversion(short ,short );             /* operador de Inversion de un elemento de la
poblacion */
void MueveElemento(void);                  /* mueve el elemento optimo de la poblacion a
la estructura del formato de salida */
void WritePDB(short, char *,short);        /* escribe el archivo PDB que contiene el
elemento optimo */
unsigned short Gray(unsigned short);       /* convierte el codigo binario de un entero  a
codigo Gray */
unsigned short Degray(unsigned short);     /* convierte el codigo Gray de un entero a
codigo binario */
unsigned short Invert(unsigned short);     /* invierte un numero entero bit a bit */
FILE *logp;
```

```
/* PROCESO PRINCIPAL */


void main()
{
unsigned int cont=0,cont2=0;
unsigned int i,j,mut,swich,l;
double energia1,mul=1.0,a[3];
short no_of_PDB=0;


#if defined(iris)
long dev;         /* Forms */
short val;        /* Forms */
FL_FORM *form;     /* Forms */
FL_OBJECT *obj,*botonE, *botonS;   /* Forms */
#endif
   printf("Leyendo genetic.ini  \n");
   Read_INI();
   printf("Leyendo archivo pdb  \n");
   ReadPDB(pdbin);

   printf("Leyendo los potenciales en archivos .dat  \n\n \n");
   ReadPotential(alal,alco,coco);
   printf("Inicio \n");
   mut = Nom_Max_of_Mut;

   factor1 = R1/32768.0;
   factor2 = (6.283185307)/32768.0;

   for(i=0;i<15;i++){

     mask1[i]=(unsigned short)(mul+0.5);
     mask2[i]=(unsigned short)(mul*2.0-0.5);
     mul *= 2;
   }
   energia1 = EnergiaIni();
   srand( (unsigned) time(NULL) );
   if((logp = fopen( "genetic.log", "w")) == '\0' ){
     printf("Sorry, I cant open the file genetic.log");
     exit(1);
   }
   printf("Energia Inicial: %f   \n",energia1/1000.0);
   fprintf(logp,"Energia Inicial: %f  \n",energia1/1000.0);
   Generate_pop();
   mySort();
   printf("generacion  0: Energia: %f   \n",pobla[0]->energia/1000.0);
```

```
    cont =0;
    swich=Delta_of_Mut;

    Max_fit=(short)(Nom_of_Pop/3 + 0.5);
#if defined(iris)
    /* Define Control Forms */
    fl_init();
      form = fl_bgn_form(FL_UP_BOX,320.0,120.0);
      botonE = fl_add_button(FL_NORMAL_BUTTON,40.0,20.0,100.0,30.0,"Exit");
      botonS = fl_add_button(FL_NORMAL_BUTTON,200.0,20.0,100.0,30.0,"Save");
    fl_end_form();
    fl_show_form(form,FL_PLACE_MOUSE,TRUE,NULL);
/* End Control forms */
#endif
/*

  core

*/
printf("\n\nhola mundo\n\n");
    do {    /* COMIENZA PROCESO RECURSIVO PRINCIPAL */
          j = Max_fit; i = 0; cont++;
        energia1=pobla[0]->energia;
          while(i<Max_fit){
              if(i==0) Reproduction(i,Nom_of_Pop-1);
                  Inversion(i,j);
              if(myrandom(100)<50) Crossover(i,i+1);
              if(myrandom(100)<20) Mutation(i,mut);

                  i++;
                  j++;
          }
/*        while(i<Nom_of_Pop){
                  Mutation(i,mut);
                  i++;
          }*/

          mySort();
          if(swich==cont){
            mut = (mut>1)?mut-1:Nom_Max_of_Mut;
            swich += Delta_of_Mut;
          }

      if(pobla[0]->energia!=energia1){
       fprintf(logp,"generacion  %i: Energia: %f \n",cont,pobla[0]->energia/1000.0 );
       printf("generacion  %i: Energia: %f   %d\n",cont,pobla[0]->energia/1000.0,mut);
       cont2=0;
      }
```

```
        else{
         if(cont2++>=METAESTABLE){
           /*MueveElemento(); */
           for(j=0;j<no_atoms;j++){
               a[0] =  pobla[0]->xyz[0][j]*factor1;
             a[1] =  pobla[0]->xyz[1][j]*factor2;
             a[2] =  pobla[0]->xyz[2][j]*factor2;
             atom[j]->xyz[0] = a[0]*sin(a[1])*cos(a[2]);
             atom[j]->xyz[1] = a[0]*sin(a[1])*sin(a[2]);
             atom[j]->xyz[2] = a[0]*cos(a[1]);
           }

           WritePDB(no_atoms,pdbout,no_of_PDB++);
           srand( (unsigned) time(NULL) );
          /* Generate_new_pop();*/

          for(i=1;i < Nom_of_Pop;i++){
              for(j=1;j<no_atoms;j++){
                 pobla[i]->xyz[0][j] = myrandom(32768);
                 pobla[i]->xyz[1][j] = myrandom(32768);
                 pobla[i]->xyz[2][j] = myrandom(32768);
            }
            pobla[i]->energia = energ(i);
          }
          cont2=0;
         }
        }


#if defined(iris)
/* Check Forms */
      obj = fl_check_forms();
      if(obj == FL_EVENT)
       {
        dev = fl_qread(&val);
       }
      else if (obj != NULL)
           if (obj == botonE) break;
           else if (obj == botonS) {
                MueveElemento();
                WritePDB(no_atoms,pdbout,no_of_PDB++);
              }
/* End of Check Forms */
#endif

   }while(cont<N_I);                    /* TERMINA PROCESO RECURSIVO DEL
ALGORITMO GENETICO*/
```

```
    MueveElemento();


    WritePDB(no_atoms,pdbout,no_of_PDB++);
    fclose(logp);
    for(i=0;i<Nom_of_Pop;i++)
      free(pobla[i]);
    for(i=0;i<no_atoms;i++)
      free(atom[i]);
}

void Read_INI(){
FILE *inip;
char line[90];
char aux[90];
short j,i,n;
   if ((inip = fopen("geneticp.ini","r")) == NULL)
   {
    printf("Sorry, I can't open the file geneticp.ini \n");
    exit(1);
   }
   i=1;
  while (!feof(inip)){

     if(fgets(line,90,inip))
       for(j=0;j<90;j++){
         aux[j]=line[j];
         if(line[j]==' ') break;
       }
      n=j;
     switch(i){
         case 1: for(j=0;j<n;j++)
                   ala[j]=aux[j];
          break;
          case 2: for(j=0;j<n;j++)
                   alco[j]=aux[j];
          break;
          case 3: for(j=0;j<n;j++)
                   coco[j]=aux[j];
          break;
          case 4: for(j=0;j<n;j++)
                   pdbin[j]=aux[j];
          break;
          case 5: for(j=0;j<n;j++)
                   pdbout[j]=aux[j];
          break;
          case 6: N_I=atoi(line);   /*numero de iteraciones*/
          break;
          case 7: Nom_of_Pop=atoi(line);   /*elementos de la poblacion*/
```

```
            break;
            case 8:  R1=atof(line);   /*Radio de la esfera */
            break;
              case 9: Nom_Max_of_Mut=atoi(line); /*numero de mutaciones */
              break;
              case 10: Delta_of_Mut=atoi(line); /*delta de mutaciones */
        }
        i++;
    }
    fclose(inip);
}


short ReadPDB(char *inputfile)
{
 FILE *ffread;
 char line[82];
 char campo[20];
 short i=0;
 if ((ffread = fopen(inputfile,"r")) == NULL)
   {
    printf("Sorry, I can't open the file %s\n",inputfile);
    exit(1);
   }


 while (!feof(ffread))
 {
    if (fgets(line,82,ffread))
    {
      for(i=0; i<4; i++)  campo[i] = line[i];

      if ((campo[0] == 'A') && (campo[1] == 'T') && (campo[2] == 'O') && (campo[3] ==
'M') )
        {
                    atom[no_atoms] = malloc(sizeof(struct element));
                for(i=0; i<=3; i++)  name[i] = line[i];
                name[4]=' ';
                    atom[no_atoms]->symbol[0] = line[12];
                    if(atom[no_atoms]->symbol[0]=='A') atom[no_atoms]->tipo=0;
                    else atom[no_atoms]->tipo=1;
                    atom[no_atoms]->symbol[1] = line[13];
                    atom[no_atoms]->xyz[0] = Fill_vector(8,30,line);
                    atom[no_atoms]->xyz[1] = Fill_vector(8,38,line);
                    atom[no_atoms]->xyz[2] = Fill_vector(8,46,line);
                no_atoms++;

        }
    }
```

```
}


 fclose(ifread);
 return (no_atoms);
}



float Fill_vector(short end, short posicion, char *line_fill)

{
   short j=0;
   char campo_fill[10];
   float campo_real;

   for(j=0; j<end; j++)
   campo_fill[j] = line_fill[j+posicion];

     campo_real = (float) atof(campo_fill);
     return(campo_real);
}




void ReadPotential(char  *alal,char *alco,char *coco)
{
FILE *file;

short counter = 0 ;
short tipo =0;
float dist,poten;

    if( ( file = fopen( alal , "r") ) == NULL){
      printf("Sorry, I can't open the file %s",alal);
      exit(1);
    }

  while(fscanf(file, "%f %f",&dist,&poten ) != EOF ){

     counter = (short)(dist*100+0.005)-100;

     potencial[tipo][counter] = poten*1000.0;
  }
  fclose(file) ;
```

```
    counter = 0;
    tipo = 1;
    if( ( file = fopen( alco , "r") ) == NULL){
        printf("Sorry, I can't open the file %s",alco);
        exit(1);
    }

    while( fscanf(file, "%f %f",&dist,&poten) !=EOF ){
        potencial[tipo][counter] = poten*1000.0;
        counter++ ;
    }
   fclose(file) ;

    counter = 0;
    tipo = 2;
    if( ( file = fopen( coco , "r") ) == NULL){
        printf("Sorry, I can't open the file %s",coco);
        exit(1);
    }

   while( fscanf(file, "%f %f",&dist,&poten) != EOF){
        potencial[tipo][counter] = poten*1000.0;
        counter++ ;
   }

  fclose(file) ;

}

double Energlalni(){
double a[3],b[3];
short j,k,l;
 double sum=0,dist=0;

     for(j=0;j<no_atoms-1;j++)
          for(k=j+1;k<no_atoms;k++){
          for(l=0;l<3;l++){
            a[l] = atom[j]->xyz[l];
            b[l] = atom[k]->xyz[l];
          }

            dist = (double)sqrt( (a[0]-b[0])*(a[0]-b[0])+(a[1]-b[1])*(a[1]-b[1])+(a[2]-
b[2])*(a[2]-b[2]) );
            if(dist<10)
            sum = sum + potencial[atom[j]->tipo+atom[k]->tipo][(short)(dist*100+0.005)-
100];
       }
       return sum;
}
```

```
unsigned short myrandom(unsigned short r)
{
 return (short)rand()%r;
}

void Generate_pop(void)
{
short i, j,l;
double a[3],b[3];
      pobla[0] = malloc(sizeof(struct poblacion));
      for(j=0;j<no_atoms;j++)
      {
        a[0] = atom[j]->xyz[0];
        a[1] = atom[j]->xyz[1]+1.0E-10;
        a[2] = atom[j]->xyz[2]+1.0E-10;
          b[0] = sqrt(a[0]*a[0]+a[1]*a[1]+a[2]*a[2]);    /* r */
        b[1] = atan( sqrt(a[0]*a[0]+a[1]*a[1])/a[2] );  /* phi */
        b[2] = atan(a[1]/a[0]);                         /* theta */

           if(a[2]<0){

            b[1] = 3.141592654+b[1];
           }
           if(a[0]<0){
            b[2] = 3.141592654+b[2];

           }

           if(b[2]<0) b[2]=6.283185307+b[2];

        pobla[0]->xyz[0][j] = (unsigned short)(b[0]/factor1);
        if(pobla[0]->xyz[0][j]>32767)pobla[0]->xyz[0][j]=32767;
        pobla[0]->xyz[1][j] = (unsigned short)(b[1]/factor2);
        if(pobla[0]->xyz[1][j]>32767)pobla[0]->xyz[1][j]=32767;
        pobla[0]->xyz[2][j] = (unsigned short)(b[2]/factor2);
          if(pobla[0]->xyz[2][j]>32787)pobla[0]->xyz[2][j]=32767;
     /*   printf("%f %f %f \n",b[0],b[1],b[2]);
        printf("%f %f %f
\n",b[0]*sin(b[1])*cos(b[2]),b[0]*sin(b[1])*sin(b[2]),b[0]*cos(b[1]));*/
      }
      pobla[0]->energia = energ(0);
  /*  printf(" %f\n",pobla[0]->energia);
      exit(0);*/
   for(i=1;i < Nom_of_Pop;i++){
      pobla[i] = malloc(sizeof(struct poblacion));
      for(j=1;j<no_atoms;j++){

          pobla[i]->xyz[0][j] = myrandom(32768);
```

```
            pobla[i]->xyz[1][j] = myrandom(32768);
            pobla[i]->xyz[2][j] = myrandom(32768);

        }
        pobla[i]->energia = energ(i);
    }

}
void Generate_new_pop(void)
{
short i, j,l;

    for(i=1;i < Nom_of_Pop;i++){
        for(j=1;j<no_atoms;j++){

            pobla[i]->xyz[0][j] = myrandom(32768);
            pobla[i]->xyz[1][j] = myrandom(32768);
            pobla[i]->xyz[2][j] = myrandom(32768);

        }
        pobla[i]->energia = energ(i);
    }

}
double energ(short i)
{
 double a[3],b[3];
 unsigned short j,k;
 double sum=0,dist;
     for(j=0;j<no_atoms-1;j++)
         for(k=j+1;k<no_atoms;k++){
           a[0] = pobla[i]->xyz[0][j]*factor1;
               a[1] = pobla[i]->xyz[1][j]*factor2;
               a[2] = pobla[i]->xyz[2][j]*factor2;
               b[0] = pobla[i]->xyz[0][k]*factor1;
               b[1] = pobla[i]->xyz[1][k]*factor2;
               b[2] = pobla[i]->xyz[2][k]*factor2;
           dist = (double)sqrt( a[0]*a[0]+b[0]*b[0]-2*a[0]*b[0]*(
sin(a[1])*sin(b[1])*cos(a[2]-b[2])+cos(a[1])*cos(b[1]) ) );
           if(dist<1)   return 1.0E+10;

           if(dist< 10 )
              sum += potencial[atom[j]->tipo+atom[k]->tipo][(short)(dist*100+0.005)-100];
         }
     return sum;
}

void mySort(){
short i,j;
```

```
struct poblacion *aux;
    for(i=0;i<Nom_of_Pop-1;i++)
        for(j=i+1;j<Nom_of_Pop;j++){
          if(pobla[j]->energia<pobla[i]->energia){
            aux = pobla[i];
            pobla[i] = pobla[j];
            pobla[j] = aux;
          }
     }
}

void Reproduction(short i,short k){
short j,l;
    for(j=0;j<no_atoms;j++)
      for(l=0;l<3;l++)
          pobla[k]->xyz[l][j]  = pobla[i]->xyz[l][j];
          pobla[k]->energia = pobla[i]->energia;
}

void Mutation(short i, short m){
short r1,r2,j;
   for(j=0;j<m;j++){
     r1=myrandom(3);
     r2=myrandom(no_atoms);
     pobla[i]->xyz[r1][r2] = Degray(Gray(pobla[i]->xyz[r1][r2]) ^ mask1[myrandom(15)]);
   }
   pobla[i]->energia = energ(i);
}

void Crossover(short i,short k){
short l,j,r1,r2,r3;

   r1=myrandom(no_atoms);
   for(j=0;j<=r1;j++)
       for(l=0;l<3;l++)
         pobla[k+Max_fit]->xyz[l][j]  = pobla[i]->xyz[l][j];

    for(j=r1+1;j<no_atoms;j++)
      for(l=0;l<3;l++){
        pobla[k+Max_fit]->xyz[l][j] = pobla[k]->xyz[l][j];
          pobla[k]->xyz[l][j] = pobla[i]->xyz[l][j];
        }

   r3=myrandom(3);
   if(r3<2)
     for(l=r3+1;l<3;l++){
       pobla[k+Max_fit]->xyz[l][r1] = pobla[k]->xyz[l][r1];
         pobla[k]->xyz[l][r1] = pobla[i]->xyz[l][r1];
     }
```

```
    r2=myrandom(15);

    pobla[k+Max_fit]->xyz[r3][r1] = Degray( (Gray(pobla[k+Max_fit]->xyz[r3][r1]) &
 ~mask2[r2]) | (Gray(pobla[k]->xyz[r3][r1]) & mask2[r2]) );
    pobla[k]->xyz[r3][r1] = Degray( (Gray(pobla[k]->xyz[r3][r1]) & ~mask2[r2]) |
 (Gray(pobla[k+Max_fit]->xyz[r3][r1]) & mask2[r2]) );

    pobla[k]->energia = energ(k);
    pobla[k+Max_fit]->energia = energ(k+Max_fit);

  }
  void Inversion(short i,short k){
  short l,j,r1,r2,r3,r4,b,t1,j1,r;
  unsigned x,y;
      r1 = myrandom(no_atoms);
      r2= myrandom(no_atoms);
      if(r1>r2){
          l=r1;
          r1=r2;
          r2=l;
      }

      for(j=0;j<r1;j++)
          for(l=0;l<3;l++)
            pobla[k]->xyz[l][j] = pobla[i]->xyz[l][j];
    for(j=no_atoms-1;j>r2;j--)
          for(l=2;l>=0;l--)
            pobla[k]->xyz[l][j] = pobla[i]->xyz[l][j];

      r3=myrandom(3);
      r4=myrandom(3);

      for(l=0;l<r3;l++)
          pobla[k]->xyz[l][r1] = pobla[i]->xyz[l][r1];
      for(l=2;t>r4;l--)
          pobla[k]->xyz[l][r2] = pobla[i]->xyz[l][r2];

      r= myrandom(15);
      pobla[k]->xyz[r3][r1] = (pobla[i]->xyz[r3][r1] & ~mask2[r]) | Invert(pobla[i]->xyz[r4][r2]
& mask2[r]);
      r= myrandom(15);
      pobla[k]->xyz[r4][r2] = Invert(pobla[i]->xyz[r3][r1] & ~mask2[r]) | (pobla[i]->xyz[r4][r2]
& mask2[r]);

      l = (r3==2)?0:r3+1; l1 = (r4==0)?2:r4-1;
      j = r1+1; j1 = r2-1;
      while(1){
        if(j>=r2&&l>=r4) break;
```

```
            pobla[k]->xyz[l][j]= Invert(pobla[i]->xyz[l1][j1]);

            if(l1--==0){l1=2;j1--;}
            if(l++==2){l=0;j++;}
        }

    Crossover(i,k);

}
void MueveElemento(){

 double a[3];
 short j;
     for(j=0;j<no_atoms;j++){
         a[0] =  pobla[0]->xyz[0][j]*factor1;
       a[1] =  pobla[0]->xyz[1][j]*factor2;
       a[2] =  pobla[0]->xyz[2][j]*factor2;
       atom[j]->xyz[0] = a[0]*sin(a[1])*cos(a[2]);
       atom[j]->xyz[1] = a[0]*sin(a[1])*sin(a[2]);
       atom[j]->xyz[2] = a[0]*cos(a[1]);
     }
}


void WritePDB(short no_atoms,char *outfile,short k)

{
 short lugar=0;
 FILE *lfwrite;
 short i=0;
 char aux[90];
 char *point;
 short k1,k2;
     k1=k/10;
     k2=k%10;
    point = aux;
    strcpy(aux,outfile);
    while(aux[lugar++]!='.'||aux[lugar]==' ');
    aux[--lugar]=(char)(k1+48);
    aux[++lugar]=(char)(k2+48);
    aux[++lugar]='\0';
    strcat(aux,".pdb");

    printf("Escribiendo archivo %s Energia:%f\n",aux,pobla[0]->energia/1000.0);
    fprintf(logp,"Escribiendo archivo %s Energia: %f\n",aux,pobla[0]->energia/1000.0);
    if ((lfwrite = fopen(aux,"w")) == NULL)
 {
 printf("Sorry, I can't open the file %s \n",aux);
 exit(0);
```

```
        }
        for (i=0; i<no_atoms; i++)
            {
             fprintf(ffwrite,"%c%c%c%c",name[0],name[1],name[2],name[3]);
             fprintf(ffwrite,"      %c%c",atom[i]->symbol[0],atom[i]->symbol[1]);
             fprintf(ffwrite,"            %8.3f",atom[i]->xyz[0]);
             fprintf(ffwrite,"%8.3f",atom[i]->xyz[1]);
             fprintf(ffwrite,"%8.3f",atom[i]->xyz[2]);
             fprintf(ffwrite,"%c",'\n');
            }

       fclose(ffwrite);
     /*   printf("Escribiendo archivo %s Energia:%f\n",aux,pobla[0]->energia/1000.0); */
    return;
    }

    /* Translations between fixed point ints and reflected Gray code */

    unsigned short Gray( unsigned short c){
            return c^(c>>1);
    }

    unsigned short Degray( unsigned short c){
    char instring[15], outstring[15],last = 0;
    short i=0,j=1,out=0,length;
            while(c>0){
               instring[i]=c%2;  c /= 2;
               i++;
            }
            length=i;
            for (i=length; i>=0; i--)
            {
               outstring[i] = (instring[i] == 1)?(!last):last;
                    last = outstring[i];
            }
            for (i=0; i<length; i++)
            {
               out += outstring[i] * j;
               j =(int)(j*2 + 0.3);
            }
            return out;
    }

    unsigned short Invert(unsigned short x){
    unsigned short b;
      for(b=0;x!=0;x>>=1)
        b = (b<<1)|(x&01);
      return b;
    }
```