Tarea 7 - Optimización estocástica

Erick Salvador Alvarez Valencia

CIMAT A.C.,
erick.alvarez@cimat.mx

Resumen En el presente reporte se hablará sobre la implementación de un algoritmo genético estándar el cual fue usado para minimizar 7 diferentes tipos de funciones tanto unimodales como multimodales.
Se describirá la implementación de dicho algoritmo, así como los resultados obtenidos al realizar varias pruebas con cada función y con diferentes parámetros. Finalmente se darán algunas conclusiones sobre la funcionalidad del algoritmo genético.

Keywords: Algoritmo genético, cruce, mutación, mapeo de fenotipos.

1. Implementación del algoritmo genético

Para esta tarea el enfoque principal fue el de implementar la versión estándar o más básica del algoritmo genético, así como las versiones más simples de los operadores de cruce y mutación.
Primeramente, el objetivo de implementar y ejecutar este algoritmo fue el de minimizar 7 funciones que son tanto unimodales como multimodales. Las funciones son:

1. **Sphere** : $\sum_{i=1}^{d} x_i^2$
2. **Ellipsoid** : $\sum_{i=1}^{d} 10^{6\frac{i-1}{d-1}} x_i^2$
3. **Zakharov** : $\sum_{i=1}^{d} x_i^2 + (\sum_{i=1}^{d} 0{,}5ix_i^2)^2 + (\sum_{i=1}^{d} 0{,}5ix_i^2)^4$
4. **Rosenbrock** : $\sum_{i=1}^{d-1}[(1-x_i)^2 + 100(x_{i+1} - x_i^2)^2]$
5. **Ackley** : $-20\exp(-0{,}2\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}) - \exp(\frac{1}{d}\sum_{i=1}^{d} cos(2\pi x_i)) + 20 + e$
6. **Griewangk** : $\sum_{i=1}^{d} \frac{x_i^2}{4000} - \prod_{i=1}^{d} cos(\frac{x_i}{\sqrt{i}}) + 1$
7. **Rastrigin** : $10d + \sum_{i=1}^{d}[x_i^2 - 10cos(2\pi x_i)]$

Todas las funciones cuentan con un óptimo local equivalente a cero, y en casi todas el óptimo se encuentra en el vector de ceros. Más adelante se dará más información de cada función.

La idea es trabajar todas las funciones con un dominio en R^{10} y con un rango cerrado. Para representar los vectores de entrada a las funciones se hizo mediante vectores binarios que trabajan con una discretización uniforme de la función dentro de su dominio.

La idea es que si tenemos un dominio de la función en $[a, b]$ y queremos representar cada valor real mediante un valor binario de 15 bits entonces se crearán particiones de $2^{15} - 1$ valores y de esta forma con la representación binaria podremos hacer un mapeo a los valores reales que se encuentran dentro del dominio de la función.
A continuación se mostrará el pseudocódigo del algoritmo que fue implementado para hacer el mapeo de binario a real.

Algorithm 1 Mapeo de binarios a reales.

```
1: procedure BinToReal(bin, a, b)
2:     d ← size(bin).
3:     res ← 0.
4:     part ← (b−a)/(2^d − 1).
5:     dec ← 1.
6:     for i = d - 1 → 0 do
7:         res ← dec * bin[i].
8:         dec ← dec * 2.
9:     return a + res ∗ part
```

El algoritmo anterior convierte el vector binario a si representación entera y finalmente se retorna el valor $a+res*part$ donde a y b son los límites del dominio de la función. De esta forma el mapeo nos asegura que cualquier representación binaria de 15 bits estará dentro del dominio de la función. Finalmente cada individuo de la población será un vector de tamaño 10x15 de números binarios el cual se mapeará a un vector de tamaño 10 de números reales.
Una vez teniendo bien definida la representación de los individuos de la población y el mapeo de dichos individuos a números reales se procedió a realizar la implementación del algoritmo genético para minimizar las funciones anteriormente descritas.
La forma general en como trabaja el algoritmo es la siguiente:

Algorithm 2 Algoritmo de evolución de la población.

```
1: procedure EvolvePop(popAct)
2:     newPop ← {}.
3:     eval ← evaluatePop(popAct).
4:     for i = 1 → sameIndNo do
5:         newPop ← newPop ⋃ eval[i].binaryVector.
6:     for i = sameIndNo + 1 → popSize do
7:         p1 ← tournametSelection(popAct, eval).
8:         p2 ← tournametSelection(popAct, eval).
9:         if random(0, 1) ≤ crossoverRatio then
10:            s1, s2 ← crossover(p1, p2).
11:            newPop ← newPop ⋃ mutation(s1).
12:            newPop ← newPop ⋃ mutation(s2).
13:        else
14:            newPop ← newPop ⋃ mutation(s1).
15:            newPop ← newPop ⋃ mutation(s2).
16:    return newPop
```

Algorithm 3 Algoritmo genético.

```
1: procedure GeneticAlgorithm
2:     pop ← generatePop().
3:     bestFitness ← ∞.
4:     iter ← 0.
5:     while bestFitness > tol and iter < iterMax do
6:         pop ← EvolvePop(pop).
7:         betsFitness ← getBestFitness(pop).
8:         iter ← iter + 1.
9:     return newPop
```

En los Algoritmos 2 y 3 podemos apreciar la composición general del algoritmo genético usado. La función principal se describe en el algoritmo 3 donde podemos ver que simplemente se genera una población aleatoria de N individuos donde N es una valor arbitrario y posteriormente esta población se va evolucionando dentro de un ciclo hasta que se cumplan los parámetros de convergencia los cuales son que el valor de la función encontrado hasta el momento sea muy cercano al óptimo o que se cumpla el número límite de iteraciones, para la implementación real se debe cumplir que se llegue al número límite de evaluaciones de la función, la cual es de 300000.
En el algoritmo 2 podemos apreciar la función que se encarga de evolucionar a la población para generar una nueva. Se puede apreciar que se manda llamar a la función *evaluatePop* la cual evalua cada individuo de la población y al final regresa una lista ordenada que contiene a cada individuo asociado al fitness que

produjo. Después, como los individuos están ordenados con respecto al fitness que produjeron es fácil identificar a los mejores de la generación pasada, por lo cual se asegura que sobrevivan un subconjunto de dichos individuos, donde el tamaño de ese subconjunto es arbitrario.
Posteriormente para el resto que individuos que faltan se tienen que generar con respecto a los individuos de la generación anterior y para ello se recurre a los operadores de cruza y mutación. El proceso consiste en elegir a dos padres de la población y la elección se hace mediante un torneo, el cual se describirá más adelante. Una vez que se eligieron los padres y con cierta probabilidad la caul denotaremos como crossover rate se realiza un proceso de cruza con ellos y se generan dos nuevos individuos que contienen características de ambos padres y finalmente se realiza un proceso de mutación a estos nuevos hijos y se añade a la nueva población. En caso de no entrar a la probabilidad de cruce simplemente se añaden los padres seleccionados por el torneo a la nueva población, aquí se les puede realizar o no la mutación, para este caso si se mutaron los padres.
Para el proceso de cruza y mutación se usaron los operadores de cruza por un punto y mutación uniforme, los cuales son descritos a continuación:

Algorithm 4 Cruza de individuos.

```
1: procedure CROSSOVER(Parent1, Parent2)
2:     sz ← Parent1.size().
3:     point ← random(0, 1).
4:     son1 ← [].
5:     son2 ← [].
6:     p ← random(1, sz).
7:     if p < 0.5 then
8:         son1 ← son ⋃ Parent1[1:point] ⋃ Parent2[point:sz].
9:         son2 ← son ⋃ Parent2[1:point] ⋃ Parent1[point:sz].
10:    else
11:        son1 ← son ⋃ Parent2[1:point] ⋃ Parent1[point:sz].
12:        son2 ← son ⋃ Parent1[1:point] ⋃ Parent2[point:sz].
13:    return son1, son2.
```

Algorithm 5 Mutación de individuos.

```
1: procedure MUTATION(Individual, mutationRate)
2:     res ← Individual.
3:     sz ← Individual.size().
4:     for i = 1 → sz do
5:         if random(0, 1) ≤ mutationRate then
6:             flipBit(res, i).
7:     return res
```

En los dos algoritmos anteriores se muestra el proceso de cruza y mutación realizado en la implementación del algoritmo genético. Primeramente tenemos el operador de cruza por un punto el cual toma dos individuos de mismas dimensiones y aleatoriamente decide un punto de corte y los genes de las particiones se distribuyen uniformemente a los dos nuevos hijos.
Para el proceso de mutación se itera sobre todos los genes del individuo y con cierta probabilidad se hace un cambio en el i-ésimo gen, como estamos trabajando en un dominio binario lo que se hace es cambiar el cero por el uno o viceversa. Hay que mencionar que la probabilidad para hacer estos cambios debe ser baja ya que el proceso de mutación promueve la diversidad en la población y el tener mucha diversidad evitará que lleguemos a buenos resultados a lo largo del tiempo, aunque por otro lado el hecho de tener poca diversidad provocará que el algoritmo se estanque muy fácil en óptimos locales lo cual también es malo.
Finalmente, tenemos la parte de la selección por torneo y a la hora de elegir a los padres de un subconjunto de individuos para crear hijos hay que tener cuidado de seguir teniendo buenos resultados ya que si elegimos a cualquier par de vectores esto provocará que los resultados que generen los hijos no sean buenos. A continuación se muestra el algoritmo para hacer el torneo de selección de padres.

Algorithm 6 Torneo de selección.

```
1: procedure TournamentSelection(Pop, noCandidates)
2:     randomShuffle(Pop).
3:     candidates ← [].
4:     for i = 1 → noCandidates do
5:         candidates ← candidates ⋃ Pop[i].
6:     return getBest(candidates)
```

En el algoritmo anterior muestra el procedimiento de torneo por selección en el cual se pasa la población actual y un entero indicando el número de candidatos a tratar, posteriormente se realiza una permutación aleatoria a la población y se toma a los primeros N individuos de la misma como candidatos. Finalmente se regresa el mejor de ellos, es decir, el que mejor fitness dió. En esta parte podemos tener calculado el fitness de cada individuo para evitar hacer esto de nuevo y no evaluar muchas más veces la función. Para el caso de la presente tarea se decidió hacer un torneo binario, es decir, se tomó $N = 2$.

2. Ejecuciones y resultados

A continuación se mostrarán los resultados obtenidos al aplicar el algoritmo genético a cada una de las funciones. La información de las ejecuciones es la siguiente:

1. Por cada función se hicieron 30 ejecuciones del algoritmo usando distintas semillas aleatorias.

2. Para la selección por torneo se utilizaron 40 candidatos a ser posibles padres.
3. En la generación de la nueva población se conservaron los 20 mejores individuos de la anterior.
4. La probabilidad de mutación por cada bit del individuo fue de 1 %.
5. La probabilidad para hacer cruza fue de 80 %.
6. La tolerancia para detener el algoritmo al encontrar el óptimo era de 10^{-3}.

Ahora se mostrarán los resultados por cada función, en donde se mostrará: El mejor valor de fitness encontrado, el vector real que generó dicho valor, la media y desviación estándar del número de evaluación de funciones, la media y desviación estándar del error resultante. De la misma forma se hizo un análisis de la diversidad de la población así como uno de la evolución del fitness promedio en donde se capturaba la información cada 5 generaciones y se iba almacenando en un archivo, esto por cada corrida del algoritmo. Finalmente se crearon gráficas que explican lo antes mencionado.

2.1. Función 1 - Sphere

Para esta función el dominio era [-600, 600], su óptimo global era un vector de ceros y el valor para el óptimo global era cero. Los valores que mejor resultados dieron son:

1. **Mejor fitness encontrado:** 0.00235297.

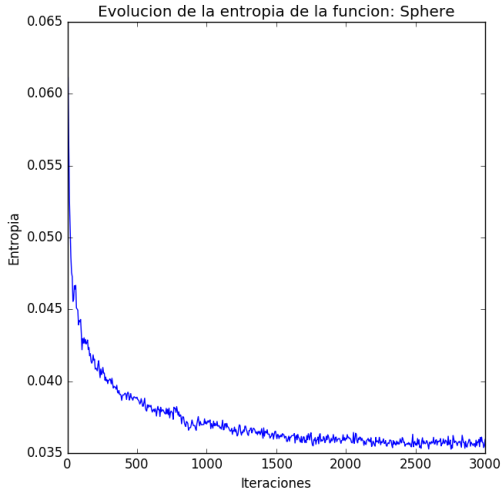
2. **Óptimo local encontrado:** [0.0183111 -0.0183111 0.0183111 -0.0183111 -0.0183111 0.0183111 0.0183111 0.0183111 0.0183111 -0.0183111 0.0183111].

Se puede apreciar que el algoritmo encontró un fitness bastante cercano a cero así como el vector que genera dicho fitness está muy cercano al óptimo local.
A continuación se muestra una tabla con los resultados obtenidos.

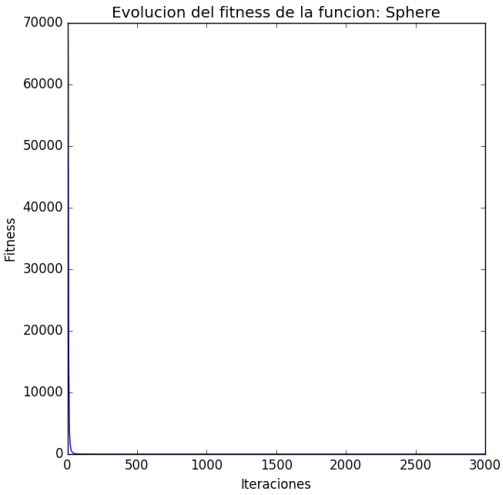
Cuadro 1: Resultados de las 30 ejecuciones para la función *Sphere*

	Función Sphere
Número promedio de evaluaciones a la función	300000
Desviación estándar del número de evaluaciones	0
Error promedio	0.00335297
Desviación estándar del error	2.1684e-18

Podemos ver en la Tabla 1. que para todas las ejecuciones el algoritmo realizó las 300000 evaluaciones de la función que tenía permitidas, esto nos lo confirma la desviación estándar de las evaluaciones. Y de la misma forma, en promedio se encontraron errores de magnitud 10^{-2}. Ahora se mostrarán las gráficas de la evolución de la diversidad poblacional y el fitness.



(a) Figura 1. Evolución de la diversidad poblacional para la función Sphere.



(b) Figura 2. Evolución del fitness de la función Sphere.

Podemos ver en las Figuras anteriores la evolución de la diversidad y del fitness a lo largo de las ejecuciones, lo primero que podemos notar es que el

fitness decreció de manera desmesuradamente rápida en las primeras iteraciones, esto concuerda con la evolución de su diversidad la cual sigue un descenso a lo largo de la ejecución.

2.2. Función 2 - Ellipsoid

Para esta función el dominio era [-20, 20], su óptimo global era un vector de ceros y el valor para el óptimo global era cero. Los valores que mejor resultados dieron son:

1. **Mejor fitness encontrado:** 9.71227e-05.

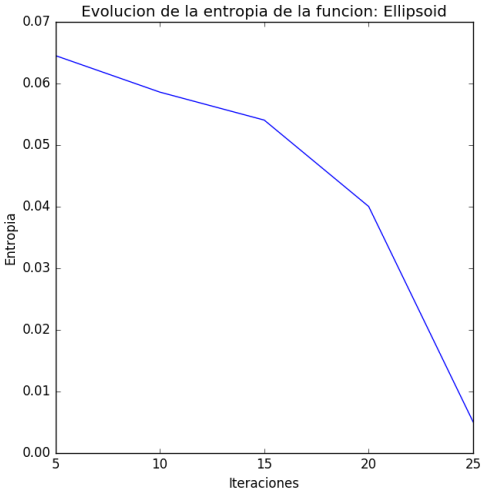
2. **Óptimo local encontrado:** [0.968657 -0.0299081 1.0773 -0.0872829 -0.493789 -0.0103763 -0.0152593 0.021363 -0.00183111 -0.00549333].

Podemos ver que el algoritmo encontró un fitness bastante cercano a cero así como el vector que genera dicho fitness está muy cercano al óptimo local.
A continuación se muestra una tabla con los resultados obtenidos.

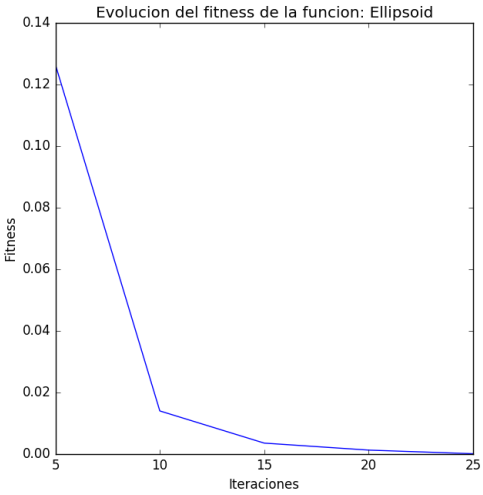
Cuadro 2: Resultados de las 30 ejecuciones para la función *Ellipsoid*

	Función Ellipsoid
Número promedio de evaluaciones a la función	2170
Desviación estándar del número de evaluaciones	268.514
Error promedio	0.000819132
Desviación estándar del error	0.000137163

Podemos apreciar que en promedio de las 30 ejecuciones se obtuvieron buenos resultados, tanto de errores pequeños como el número de evaluaciones de la función fueron muy pocos. Ahora se mostrarán las gráficas de la evolución de la diversidad poblacional y el fitness.



(c) Figura 3. Evolución de la diversidad poblacional para la función Ellipsoid.



(d) Figura 4. Evolución del fitness de la función Ellipsoid.

Podemos apreciar en las Figuras anteriores que el fitness fue descendiendo bastante rápido en las primeras iteraciones y como esta función es unimodal

se puede ver que tras el descenso se iba a llegar al valor del óptimo global. De la misma forma se puede apreciar que la entropía poblacional es bastante baja desde un comienzo (menor a 0.1) y posteriormente va descendiendo aún más hasta llegar casi al cero, esto indica que la población se iba tornando más uniforme ya que en esta parte el algoritmo iba haciendo más intensificación que exploración.

2.3. Función 3 - Zakharov

Para esta función el dominio era [-20, 20], su óptimo global era un vector de ceros y el valor para el óptimo global era cero. Los valores que mejor resultados dieron son:

1. **Mejor fitness encontrado:** 2.7662e-04.

2. **Óptimo local encontrado:** [-0.00061037 -0.00061037 -0.00305185 -0.00183111 0.00061037 0.00061037 -0.00183111 0.00305185 -0.00061037 0.00061037].

Para esta función podemos notar que el algoritmo encontró un muy buen resultado con un error de magnitud 10^{-4} así como el vector que genera dicho error contiene valores muy cercanos a cero.
A continuación se muestra una tabla con los resultados obtenidos.

Cuadro 3: Resultados de las 30 ejecuciones para la función *Zakharov*

	Función Zakharov
Número promedio de evaluaciones a la función	242773
Desviación estándar del número de evaluaciones	76029.4
Error promedio	0.0186893
Desviación estándar del error	0.137973

Para esta función vemos que en promedio no se llegaba al límite de evaluaciones de la función aunque por lo visto por el error promedio el cual es más grande que la tolerancia general, el algoritmo en ocasiones si tenía que detenerse porque llegaba al límite de evaluaciones. Podemos notar que en general el error promedio fue pequeño, llegando a una magnitud de 10^{-1}. Ahora se mostrarán las gráficas de la evolución de la diversidad poblacional y el fitness.

(e) Figura 5. Evolución de la diversidad poblacional para la función Zakharov.

(f) Figura 6. Evolución del fitness de la función Zakharov.

Podemos ver en las gráficas anteriores que primeramente la diversidad poblacional fue decrececiendo a través de la ejecución, vemos que los valores de la

diversidad en las primeras ejecuciones no eran tan altos por lo cual el algoritmo desde el inicio pudo encontrar el óptimo local sin tanta exploración, y esto se ve reflejado en la Figura siguiente, la cual muestra lo anterior mencionado.

2.4. Función 4 - Rosenbrock

Para esta función el dominio era [-20, 20], su óptimo global era un vector de unos y el valor para el óptimo global era cero. Los valores que mejor resultados dieron son:

1. **Mejor fitness encontrado:** 7.28474.

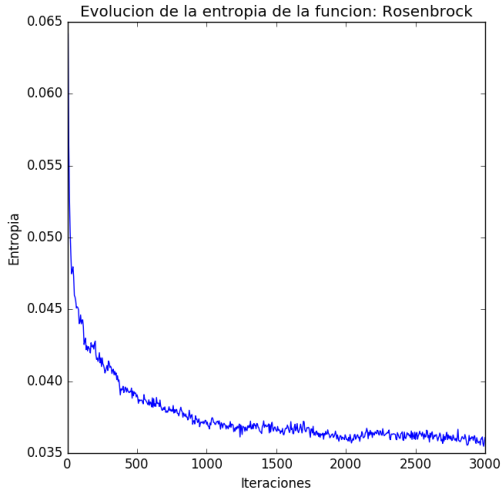
2. **Óptimo local encontrado:** [0.624409 0.390027 0.153203 -0.00061037 -0.00061037 0.0103763 -0.00061037 -0.00061037 -0.00061037 -0.00061037].

Podemos apreciar que para esta función no se obtuvieron tan buenos resultados ya que el mejor fitness encontrado está relativamente lejos del óptimo global el cual corresponde a cero, de igual manera el vector que generó dicho fitness tienes valores muy cercanos a cero, en cambio debería estar con valores muy cercanos a uno.
A continuación se muestra una tabla con los resultados obtenidos.

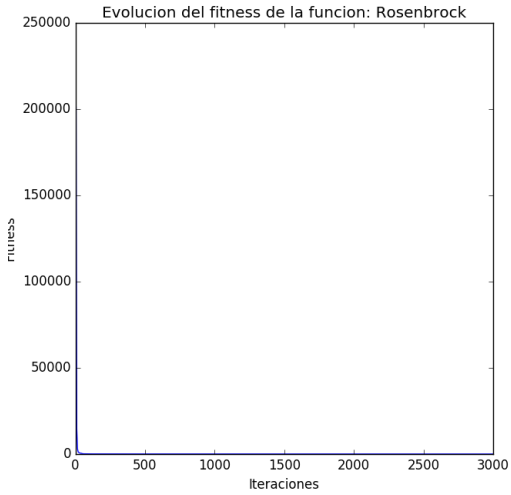
Cuadro 4: Resultados de las 30 ejecuciones para la función *Rosenbrock*

	Función Rosenbrock
Número promedio de evaluaciones a la función	300000
Desviación estándar del número de evaluaciones	0
Error promedio	35.3015
Desviación estándar del error	74.4078

Se puede apreciar en la tabla anterior que el algoritmo siempre llegó al límite de evaluaciones de la función el cual es 300000, esto lo confirmamos con la desviación estándar de las evaluaciones. Por otro lado tenemos que el promedio del error es bastante alto, sabiendo que el óptimo global es cero. Ahora se mostrarán las gráficas de la evolución de la diversidad poblacional y el fitness.



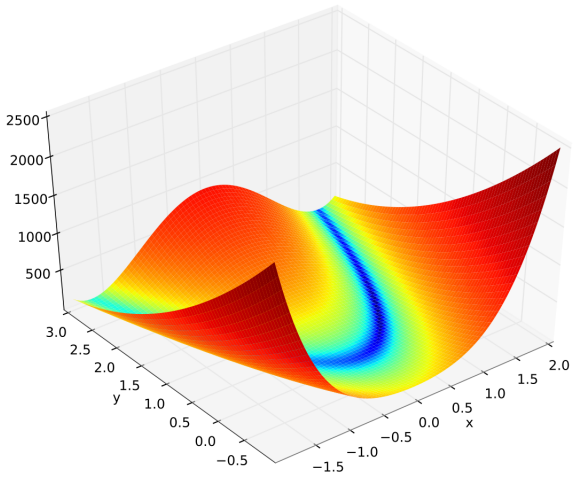
(g) Figura 7. Evolución de la diversidad poblacional para la función Rosenbrock.



(h) Figura 8. Evolución del fitness de la función Rosenbrock.

Para las gráficas anteriores podemos ver la evolución del fitness como de la diversidad de la función Rosenbrock, en esta parte vemos que la gráfica que

muestra la evolución del fitness coincide con la de la primer función donde el fitness decrece muy rápido aunque como se pudo apreciar en la tabla anterior no se llegó al óptimo global, lo que se podría decirse que el algoritmo se estancó en un cierto óptimo local y de ahí no pudo minimizar más, hay que recordar que el óptimo global de la función se encuentra en un valle el cual es dificil de minimizar tal como se muestra en la Figura 9.



(i) Figura 9. Gráfica de la función de Rosenbrock en un vecindad cerca del óptimo global.

2.5. Función 5 - Ackley

Para esta función el dominio era [-20, 20], su óptimo global era un vector de ceros y el valor para el óptimo global era cero. Los valores que mejor resultados dieron son:

1. **Mejor fitness encontrado:** 0.00246132.

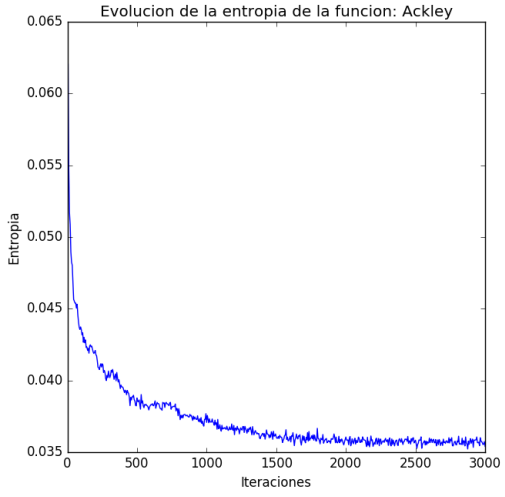
2. **Óptimo local encontrado:** [0.00061037 0.00061037 0.00061037 0.00061037 0.00061037 0.00061037 -0.00061037 -0.00061037 0.00061037 0.00061037].

Para esta función podemos ver que se obtuvieron buenos resultados, ya que se logró encontrar un valor muy cerca del óptimo global así como el vector que generó dicho fitness contiene valores muy cercanos a cero.
A continuación se muestra una tabla con los resultados obtenidos.

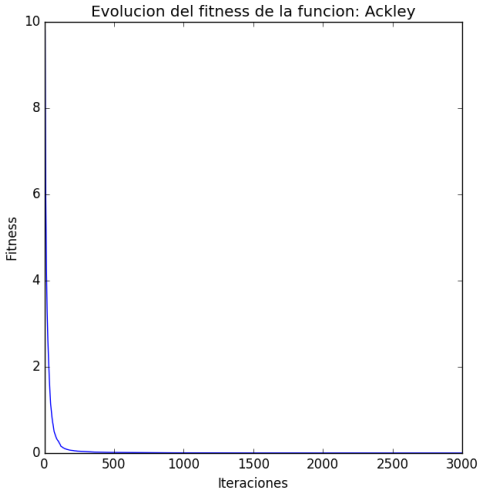
Cuadro 5: Resultados de las 30 ejecuciones para la función *Ackley*

	Función Ackley
Número promedio de evaluaciones a la función	300000
Desviación estándar del número de evaluaciones	0
Error promedio	0.00251799
Desviación estándar del error	0.000212023

Para esta función podemos ver que el algoritmo llegó al límite del número de evaluaciones de la misma en las 30 corridas, y esto lo confirmamos en la desviación estándar. Por otra parte vemos que en promedio el algoritmo encontró buenos resultados de la función, teniendo un error con magnitud de 10^{-2}, esto también se confirma en la desviación estándar del error con un valor bastante bajo. Ahora se mostrarán las gráficas de la evolución de la diversidad poblacional y el fitness.



(j) Figura 10. Evolución de la diversidad poblacional para la función Ackley.



(k) Figura 11. Evolución del fitness de la función Ackley.

En las dos gráficas anteriores se muestra la evolución del fitness y de la diversidad de la función de Ackley en las cuales vemos que ambas descienden a través de las evaluaciones, se aprecia que el fitness de la función descendió muy rápido y como se pudo ver en la Tabla 5 el algoritmo encontró el óptimo global de la función pese a que la misma es munltimodal por lo tanto tiene más de un óptimo local. Por otra parte vemos que la diversidad fue descendiendo rápidamente y por lo tanto vemos en el algoritmo hizo una búsqueda más intensiva ya que encontró la zona del óptimo global.

2.6. Función 6 - Griewangk

Para esta función el dominio era [-600, 600], su óptimo global era un vector de ceros y el valor para el óptimo global era cero. Los valores que mejor resultados dieron son:

1. **Mejor fitness encontrado:** 0.0106771.

2. **Óptimo local encontrado:** [-3.16782 -0.0183111 5.40178 0.0183111 0.0183111 -0.0183111 -0.0183111 0.0183111 0.0183111 0.0183111].

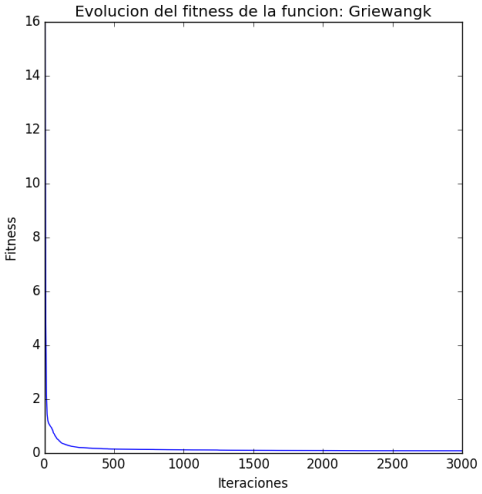
Para esta función podemos ver que los resultados obtenidos fueron bastante buenos ya que se encontró un fitness muy cercano al óptimo, así como el vector que generó dicho fitness contiene valores muy ceranos a cero.
A continuación se muestra una tabla con los resultados obtenidos.

Cuadro 6: Resultados de las 30 ejecuciones para la función *Griewangk*

	Función Griewangk
Número promedio de evaluaciones a la función	300000
Desviación estándar del número de evaluaciones	0
Error promedio	0.0867839
Desviación estándar del error	0.0322637

Para esta función podemos ver que el algoritmo llegó al límite del número de evaluaciones de la misma en las 30 corridas, y esto lo confirmamos en la desviación estándar. Por otra parte vemos que en promedio el algoritmo encontró buenos resultados de la función, teniendo un error con magnitud de 10^{-1}, esto también se confirma en la desviación estándar del error con un valor bastante bajo. Ahora se mostrarán las gráficas de la evolución de la diversidad poblacional y el fitness.

(l) Figura 12. Evolución de la diversidad poblacional para la función Griewangk.



(m) Figura 13. Evolución del fitness de la función Griewangk.

En las dos gráficas anteriores se muestra la evolución del fitness y de la diversidad del algoritmo al trabajar con la función de Griewangk. Algo que podemos notar en la Figura 13 es que desde las poblaciones iniciales se tenían valores muy bajos de la función y posterioremente al avanzar algunas iteraciones estos valores fueron minimizándose hasta llegar muy cerca del óptimo. Y por otra parte tenemos que la diversidad poblacional fue descendiendo muy rápido conforme a las iteraciones del algoritmo, esto concuerda con lo anterior mencionado.

2.7. Función 7 - Rastrigin

Para esta función el dominio era [-20, 20], su óptimo global era un vector de ceros y el valor para el óptimo global era cero. Los valores que mejor resultados dieron son:

1. **Mejor fitness encontrado:** 0.000739112.

2. **Óptimo local encontrado:** [-0.00061037 0.00061037 -0.00061037 -0.00061037 0.00061037 0.00061037 -0.00061037 0.00061037 -0.00061037 -0.00061037].

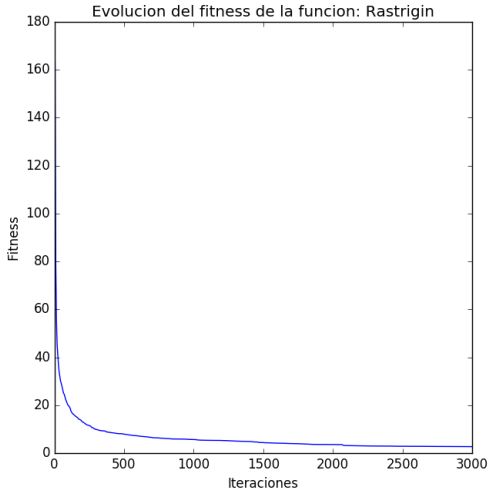
Para esta función podemos ver que los resultados obtenidos fueron bastante buenos ya que se encontró un fitness muy cercano al óptimo, así como el vector que generó dicho fitness contiene valores muy ceranos a cero.
A continuación se muestra una tabla con los resultados obtenidos.

Cuadro 7: Resultados de las 30 ejecuciones para la función *Rastrigin*

	Función Rastrigin
Número promedio de evaluaciones a la función	292957
Desviación estándar del número de evaluaciones	19621
Error promedio	2.6844
Desviación estándar del error	2.60471

Para esta función vemos que en promedio no se llegaba al límite de evaluaciones de la función aunque por lo visto por el error promedio el cual es más grande que la tolerancia general, el algoritmo en ocasiones si tenía que detenerse porque llegaba al límite de evaluaciones. Podemos notar que en general el error promedio no fue tan bueno ya que se llegó a valores de dos. Ahora se mostrarán las gráficas de la evolución de la diversidad poblacional y el fitness.

(n) Figura 14. Evolución de la diversidad poblacional para la función Rastrigin.



(ñ) Figura 15. Evolución del fitness de la función Rastrigin.

Por último tenemos las gráficas de la evolución del fitness y la diversidad de la poblacional que se generaron al minimizar la función de Rastrigin. Podemos ver que para esta función la evolución poblacional generó un descenso escalonado y es que al ver la Figura 15 notamos que en este caso no se tenía un descenso tan rápido como se tuvo con otras funciones.

Notas y comentarios finales. Previamente a dar conclusiones finales se responderán dos preguntas planteadas en la descripción de la tarea.

1. Si usamos representación binaria para una variable que en los reales vive en el dominio (-20, 20) ,¿cuántos bits necesitamos usar para que la variable represente soluciones con precisión de 0.001? Es decir, que la codificación del intervalo nos de una delta menor o igual a 0.001.

   Solución : Tenemos que nuestro intervalo está definido como $a = -20$ y $b = 20$. Ahora como estamos trabajando con n nuestra partición del dominio estará formada por $\frac{b-a}{2^n-1}$. Finalmente queremos que lo anterior de una delta menor a $\frac{1}{1000}$ con lo que tenemos.

$$\frac{b-a}{2^n-1} \leq \frac{1}{1000}$$

$$40000 \leq 2^n - 1$$

$$n \geq 15{,}28 \rightarrow n \geq 15$$

Tenemos que nuestra representación binaria debe tener al menos 15 bits para representar soluciones con precisión de $\frac{1}{1000}$.

2. Si usamos representación binaria para una variable que en los reales vive en el dominio (-20, 20) ,¿cuántos bits necesitamos usar para que la variable represente soluciones con precisión de 0.00001? Es decir, que la codificación del intervalo nos de una delta menor o igual a 0.00001.

   Solución : Realizamos el mis procedimiento que en el ejercicio anterior pero esta vez queremos que la partición sea menor o igual que $\frac{1}{100000}$

   $$\frac{b-a}{2^n-1} \leq \frac{1}{100000}$$
   $$4000000 \leq 2^n - 1$$
   $$n \geq 21{,}93 \rightarrow n \geq 21$$

   Tenemos que nuestra representación binaria debe tener al menos 21 bits para representar soluciones con precisión de $\frac{1}{100000}$.

3. ¿Qué es la presión de selección? (del operador de selección, digamos el torneo o la selección proporcional. Inclusive la selección natural tiene presión de selección).

   Respuesta : La medida en la que los individuos con mayor aptitud tienen una mayor probabilidad de formar parte de la siguiente población, se conoce como presión de selección. En general, cuanto mayor sea el tamaño del torneo k mayor presión de selección se obtendrá, dado que cada individuo seleccionado habrá tenido que superar en aptitud a un número mayor de soluciones candidatas. Una presión de selección alta provoca la aceleración de la convergencia del algoritmo, pero aumenta la posibilidad de que este converja hacia un óptimo local, lo cual se conoce como "convergencia prematura".

2.8. Comentarios finales

En el presente reporte se describió la implementación de un algoritmo genético básico que fue usado para minimizar funciones con dominio en los reales mediante una representación la cual se hacía mediante una discretización y un mapeo con vectores de números binarios. Se pudo mostrar que en general el algoritmo ofreció muy buenos resultados para la mayoría de las funciones trabajadas, en donde se vio que el error de los mejores resultados producidos por el algoritmo fue bastante bajo, en algunos casos se logró llegar a resultados más bajos que la tolerancia establecida, en algunos otros se logró llegar a resultados muy parecidos a la tolerancia y en el caso de la función de Rosenbrock se mostró que los resultados no fueron tan buenos ya que el error de la mejor solución era relativamente alto.
De la misma forma se mostró para cada función la evaluación de la diversidad

y del fitness promedio obtenidos al realizar las 30 ejecuciones en el cluster de CIMAT. Se pudo apreciar que en la mayoría de funciones el fitness descendía muy rápido al llevar muy pocas iteraciones el algoritmo. Para las funciones que se trabajó, esto anterior resultó muy bien ya que se pudo llegar al óptimo global de las mismas, pero en otros casos esto no es tan bueno ya que se tiene una convergencia prematura y esto puede provocar óptimos locales que no den resultados tan buenos.