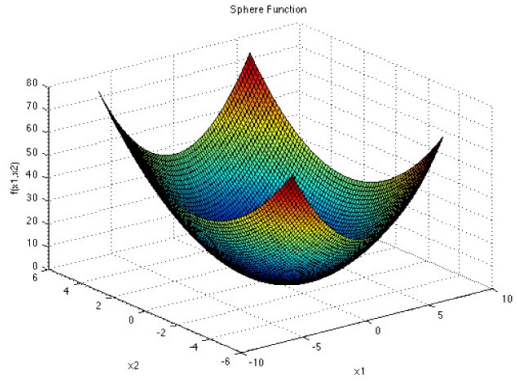
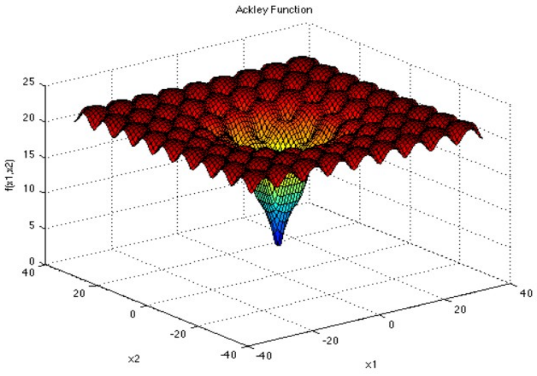
Tarea 1 Algoritmos Evolutivos. Algoritmo Genético.
Juan Gerardo Fuentes Almeida, MCC

Implementación.

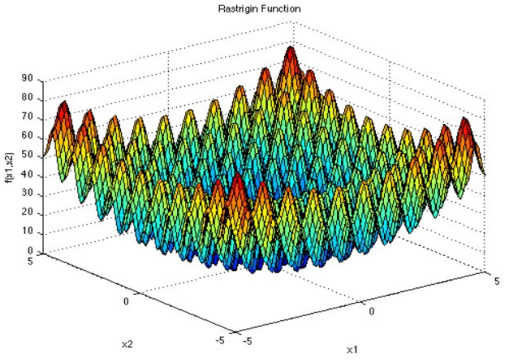
Estas gráficas muestran las funciones con las que trabajamos en esta práctica:



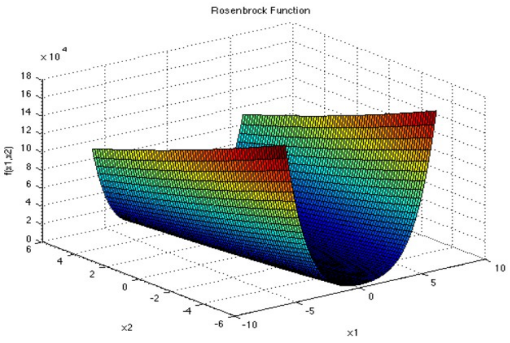
$$f(\mathbf{x}) = \sum_{i=1}^{d} x_i^2$$



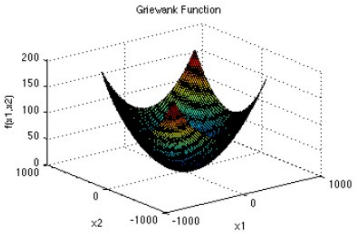
$$f(\mathbf{x}) = -a\exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right) + a + \exp(1)$$

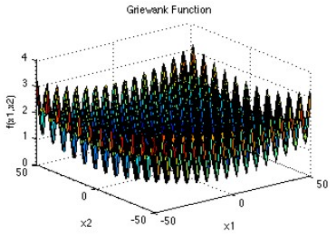


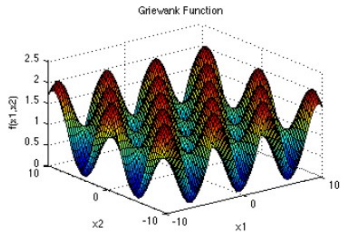
$$f(\mathbf{x}) = 10d + \sum_{i=1}^{d}\left[x_i^2 - 10\cos(2\pi x_i)\right]$$

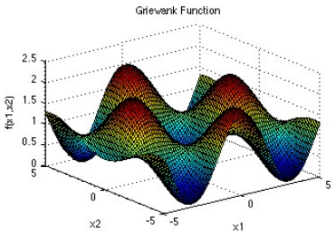


$$f(\mathbf{x}) = \sum_{i=1}^{d-1}\left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right]$$









$$f(\mathbf{x}) = \sum_{i=1}^{d}\frac{x_i^2}{4000} - \prod_{i=1}^{d}\cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

A continuación se muestra un pseudocódigo general del Algoritmo Genético:

```
Input:
N←PopulationSize, maxIte, t←0
pc ← crossover probability

Pt← InitializePopulation(N)
while( t < maxIte )
{
        Fitness ← EvaluateFitness( Pt , Elit );
        Pt'  ← Selection( Pt, Fitness );
        Pt'' ← Reproduction(Pt');
        Pt''' ← Mutation ( Pt'', pc );
        Pt ← Replacement( Pt''', Elit);
        t=t+1;
}
Return BestResult;
```

Parámetros del algoritmo:

- N = tamaño de la población, establecida en 50 porque se pensó que se tendrían mejores resultados con un mayor número de iteraciones, ya que el número de iteraciones es inversamente proporcional a este parámetro.
- dim = dimensión de las variables de la función objetivo, establecida en 10 y en 30.
- Número de iteraciones: determinado por 10000* dim/N, siendo N el tamaño de la población.
- Pc = probabilidad de cruza, establecida en 0.85 para tener un buen grado de cruza en nuestro algoritmo.
- Pm = probabilidad de mutación, establecido implícitamente en 1/L, donde L es la longitud en bits de la cadena binara, la cual fue establecida en 20, esto quiere decir que por cada variable de cada individuo alteramos un bit.

Codificación:

Para implementar este algoritmo utilizamos el recurso <bitset> de C++ el cual consiste en un arreglo binario cuyo constructor recibe un entero como parámetro, la cadena binaria generada puede ser accesada con el operador [], puede ser modificada con el operador set() y con el operador flip().

De manera que cuando generamos la población inicial, generamos N*d cadenas binarias para cada una de las variables de la población, la aleatorización se da por un “volado” al poner a 1 o a cero cada bit de todas las cadenas.

El segundo paso de nuestro algoritmo es la evaluación del Fitness, para esto tenemos que convertir nuestras cadenas binarias a números reales utilizando la siguiente expresión de conversión:

$$x_{10} = x_2 * \Delta + x_{min}$$

donde Δ es el tamaño de paso con el cual discretizamos el dominio de las variables de la función objetivo, y el cual esta determinado por

$$\Delta = \frac{x_{max} - x_{min}}{2^L - 1}$$

Dentro de la función que evalúa el Fitness para cada individuo, se determina el individuo con el mejor valor de Fitness (Elit) el cual conservamos en un vector aparte para no perderlo durante las demás fases del algoritmo.

Selección.

Para esta fase del algoritmo genético, hemos elegido el torneo binario como criterio de selección, básicamente seleccionamos N veces aleatoriamente un par de individuos de la población y comparamos sus valores de fitness, el individuo mas apto (en este caso el que tengo menor valor de fitness) es el seleccionado para la reproducción.

Presión de Selección. Son aquellas condiciones a las que se somete una población con el fin de que los individuos mas aptos sobrevivan. Variando el número de individuos que participan en cada torneo se puede modificar la presión de selección. Cuando participan muchos individuos en cada torneo, la presión de selección es elevada y los peores individuos apenas tienen oportunidades de reproducción.

Selection():

```
for 1 to N
{
        x1,x2 ← randomSelection( Pt )
        if  Fitness( x1 ) < Fitness( x2 )
                Pt' ← x1
        else
                Pt' ← x2
        end if
}
```

Reproducción.

Para la reproducción, seleccionamos N/2 veces aleatoriamente un par de individuos de la población, y mediante el valor de probabilidad de cruza que ya definimos, determinamos de forma aleatoria si estos individuos se cruzaran o no, si no se cruza, se copian tal como están en la siguiente población:

Reproduction():

```
for 1 to N/2
{
        x1,x2 ← randomSelection( Pt' )
        if  random[0,1] < pc
                Pt'' ← CrossOver( x1,x2 )
        else
                Pt'' ←  x1,x2
        end if
}
```

Cruza.

Para la Cruza, utilizamos el método uniforme, para los dos individuos seleccionados, recorremos cada uno de sus alelos decidiendo aleatoriamente con probabilidad 1/2 si los alelos se cruzaran o no, después de esto los copiamos a la siguiente población:

```
Crossover ( x1,x2 ):
for i=1 to #bits
{
        if  random[0,1] < 0.5
                swap ( x1[i], x2[i] )
        end if
}
```

Reemplazo.

Para este algoritmo, el 100% de los individuos es copiado a la siguiente generación, con excepción de una posición aleatoria en donde se agrega el individuo elit. Esto es con el propósito de que el Fitness promedio de la población mejore poco a poco.

Resultados.

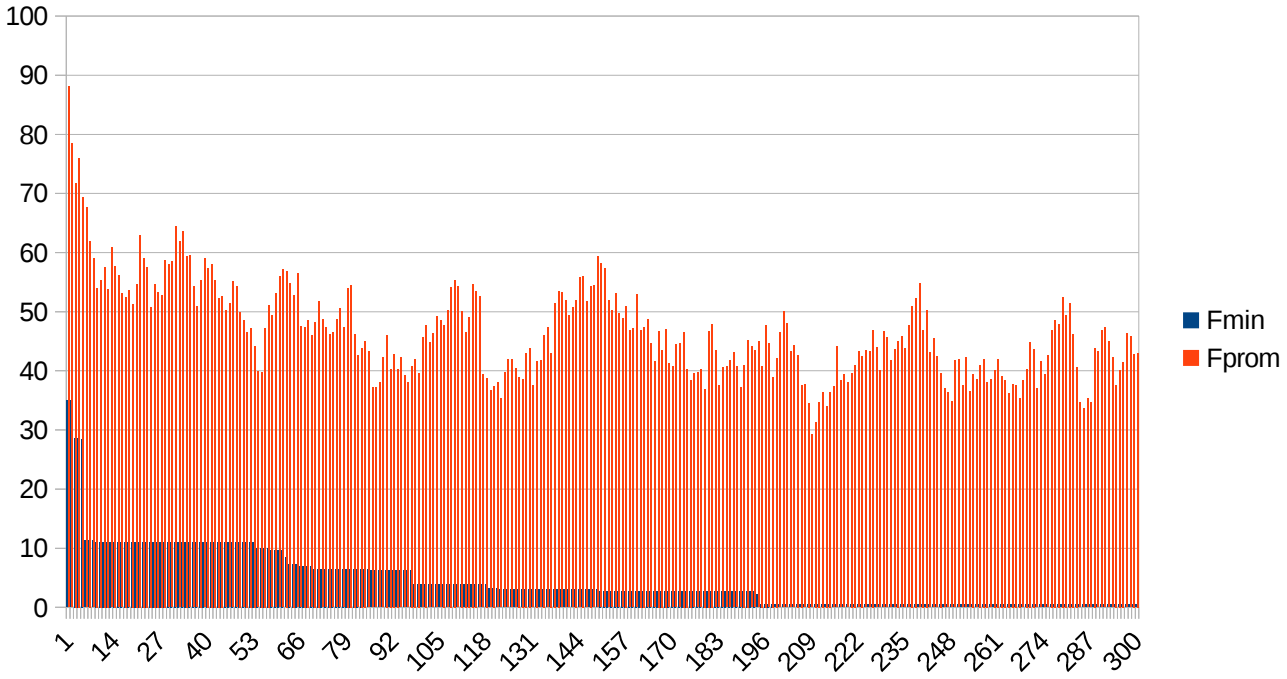
A continuación se muestra una tabla de los distintos resultados que obtuvimos durante 50 corridas para cada instancia, de cada corrida se obtiene el mejor valor de fitness, y con los 50 valor calculamos las métricas que se indican:

Function	dim	min	max	mean	median	std. Dev.
Sphere	10	0.0177056	0.398713	0.101672	0.0714511	0.086872
Sphere	30	8.11206	23.3211	14.3905	14.2244	3.12026
Ackley	10	2.21424	5.93446	3.74477	3.70586	0.71985
Ackley	30	11.316	15.1256	13.4357	13.6003	0.868166
Griewank	10	1.02478	2.41537	1.37839	1.28183	0.272585
Griewank	30	33.5083	76.1179	54.9895	55.3956	11.0535
Rastrigin	10	9.75747	31.1054	20.4981	20.711	4.92186
Rastrigin	30	139.682	205.578	172.53	174.74	16.3675
Rosenbrock	10	3.81045	21.652	10.2728	10.2436	2.67429
Rosenbrock	30	198.238	635.771	361.163	367.651	91.7279

En general se observa que el algoritmo es bastante malo para 30 dimensiones, especialmente para las funciones de Rastrigin y Rosenbrock, ya que la primera tiene demasiados mínimos locales y la segunda consta de un valle casi plano donde parecería difícil encontrar el mínimo global.

Gráfico de Fitness.

A continuación se muestra un gráfico donde se aprecia la diferencia entre el Fitness promedio y el mínimo, sólo se han mostrado 300 generaciones de una corrida, utilizando la función Esfera con dimensión igual a 10, aunque se observa un ligero descenso en el Fitness promedio, este es en general muy variado a lo largo de toda la gráfica, esto puede ser debido a que utilizamos cruza uniforme, ya que esta genera mayor diversidad que los demás tipos de cruzas, se observa también la evolución del mejor Fitness el cual si se comporta como se esperaba, descendiendo paulatinamente hasta alcanzar un mínimo.



Compilación/Ejecución.

El programa implementado incluye un makefile para compilarse, que soporta los comandos *make*, *make run* y *make clean*, así como un script de ejecución en el cluster para correr todas las instancias, los resultados se generan en un archivo para cada programa.

El programa recibe tres parámetros:

1. Tamaño de la población, número entero (se utilizó 50 en esta implementación).
2. Dimensión de las variables, número entero (se utilizó 10 y 30)
3. Referencia a la función objetivo a evaluar:
   - Sphere (0) [-5.12, 5.12]
   - Ackley (1) [-32.768, 32.768]
   - Griewank (2) [-600, 600]
   - Rastrigin (3) [-5.12, 5.12]
   - Rosenbrock (4) [-2.048, 2.048]
4. Limite inferior de la variable (x_{min})
5. Limite superior de la variable (x_{max})

```
almeida@almeida-X501A1: ~
user_demo@el-insurgente:~$ cd G_Fuentes/Genetic
user_demo@el-insurgente:~/G_Fuentes/Genetic$ make
g++               -std=c++11 -c   src/functions.cpp -o obj/functions.o
g++               -std=c++11 -c   src/genetic.cpp -o obj/genetic.o
g++               -std=c++11 -c   src/main.cpp -o obj/main.o
g++               -std=c++11 -c   src/memo.cpp -o obj/memo.o
g++               -std=c++11      -o bin/ejecutable obj/functions.o obj/genetic.o
 obj/main.o obj/memo.o -lgomp
user_demo@el-insurgente:~/G_Fuentes/Genetic$ cd
user_demo@el-insurgente:~$ cd G_Fuentes/Genetic
user_demo@el-insurgente:~/G_Fuentes/Genetic$ g++ -Wall -o script script.cpp
user_demo@el-insurgente:~/G_Fuentes/Genetic$ cd
user_demo@el-insurgente:~$ ./G_Fuentes/Genetic/script
```