



UNIVERSIDAD AUTÓNOMA DE QUERÉTARO

DIVISIÓN DE INVESTIGACIÓN Y POSGRADO

# Computo Evolutivo

## Practica 5

*Algoritmo Genetico Multi-Isas*

Profesor: Dr. Marco Antonio Aceves Fernández

Presenta:

Ing. Osmar Antonio Espinosa Bernal

20 de octubre de 2021

## 1. Objetivo

Desarrollar un algoritmo genetico con varias instancias que se ejecuten de manera independiente para llevar a cabo un aplicacion especifica, esto con el fin de dividir tareas para facilitar la busqueda de una optimizacion dependiendo el caso.

La aplicacion sera llevada a cabo para encontrar el minimo global de la Funcion Rastrigin. La vista en bidimensional de la Funcion Rastrigin se muestra en la figura 1.

## 2. Marco Teorico

La función Rastrigin es una función no convexa y tiene varios mínimos locales. Esta función es un ejemplo típico de función no lineal altamente multimodal, pero las ubicaciones de los mínimos se distribuyen regularmente. Tiene un mínimo global en  $f(x)$  en  $x = (0, \dots, 0)$ . Encontrar el mínimo de esta función es un problema bastante difícil debido a su gran espacio de búsqueda y su gran número de mínimos locales.

En la figura 1 se muestra la representacion bidimensional de la Funcion Rastrigin.

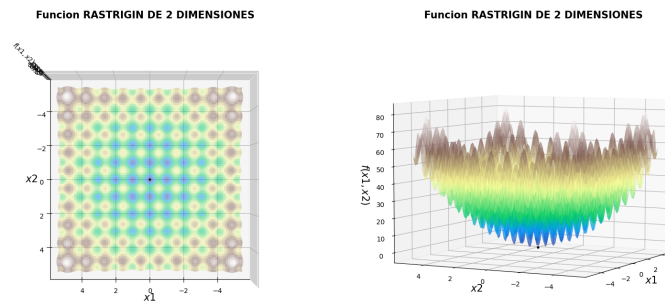


Figura 1: Representacion bidimensional de funcion Rastrigin, se muestra el minimo global localizado con una estrella negra.

### Algoritmos genéticos

Los algoritmos genéticos parten de la premisa de emplear la evolución natural

como un procedimiento de optimización que se caracteriza por tener operaciones básicas que son:

- Selección de cromosomas
- Cruzamiento de cromosomas
- Mutación de cromosomas

Por lo tanto, el algoritmo es una búsqueda empleando dichas operaciones; por ejemplo, se puede plantear una familia de individuos los cuales se seleccionan y después se considera a los mas óptimos para realizar un cruzamiento entre ellos, y la forma de evitar caer en mínimos locales es el empleo de la mutación. Por ello la mutación puede considerarse una operación de fuga de mínimos locales. Esta búsqueda se define como una búsqueda de un espectro amplio.

## Métodos de Selección de cromosomas

- **Selección por torneo:** se eligen subgrupos de individuos de la población, y los miembros de cada subgrupo compiten entre ellos. Sólo se elige a un individuo de cada subgrupo para la reproducción.
- **Selección Rank:** a cada individuo de la población se le asigna un rango numérico basado en su aptitud, y la selección se basa en este ranking, en lugar de las diferencias absolutas en aptitud.
- **Selección Random Monogamico:** Cada individuo de la población es seleccionada de manera aleatoria solo una vez y se marca o elimina para que no sea nuevamente utilizado para la cruce.

Estos son los tipos de seleccion de cromosomas que se eligen para la presente practica.

*Vectores de números (reales o enteros):* Codificación habitual cuando las variables de nuestro problema son numéricas (codificación directa, aunque habrá que definir operadores genéticos adecuados).

- Los operadores de cruce (en n puntos y uniforme) se pueden seguir utilizando.
- El operador de mutación se modifica en función de la situación:
  - Valores ordinales: Se hace que sea más probable cambiar a valores cercanos en la escala.
  - Valores categóricos: Selección aleatoria uniforme.

- Los operadores de cruce y mutación habituales dan lugar a soluciones inadmisibles, por lo que los operadores de mutación deben cambiar al menos dos valores y los de cruce han de diseñarse específicamente para problemas de este tipo.

La probabilidad de mutación ahora se referirá a cromosomas completos, no a genes individuales

## Cruzamiento de cromosomas

Operadores de cruce en 2 puntos: Se seleccionan dos puntos para cortar un cromosoma y después se juntan los fragmentos alternando los cromosomas 1 y 2.

$(1 | 2345 | 6789) (1 | 8765 | 6789)$

$(9 | 8765 | 4321) (9 | 2345 | 4321)$

Operadores de cruce en N puntos: Se seleccionan N puntos para cortar un cromosoma y después se juntan los fragmentos alternando los cromosomas 1 y 2.

$(1 | 23 | 45 | 6789) (1 | 87 | 45 | 4321)$

$(9 | 87 | 65 | 4321) (9 | 23 | 65 | 6789)$

## Mutación

### Operadores de mutación

- **Inserción:** Elegir dos alelos aleatoriamente y colocar el segundo justo después del primero.

$(1 | 2 | 34 | 5 | 6789) \longrightarrow (1 | 25 | 346789)$

- **Intercambio:** Seleccionar dos alelos aleatoriamente e intercambiarlos.

$(1 | 2 | 34 | 5 | 6789) \longrightarrow (1 | 5 | 34 | 2 | 6789)$

- **Inversión:** Seleccionar dos alelos aleatoriamente e invertir la cadena entre ellos.

$(1 | 2345 | 6789) \longrightarrow (1 | 5432 | 6789)$

- **Revuelto[Scramble]** Seleccionar un subconjunto de genes y reordenar aleatoriamente los alelos (el subconjunto no tiene por qué ser contiguo).

( 1 | 2 3 4 5 | 6 7 8 9 ) --- > ( 1 | 3 5 4 2 | 6 7 8 9 )

Se observa como cada operador de mutación afecta de forma diferente al orden relativo y a las relaciones de adyacencia.

### 3. Materiales

Computadora personal portátil MSI, procesador Intel(R)Core(TM) i7-750H CPU @ 2.60GHz, 16GB de memoria RAM, Sistema operativo Windows 10 de 64 bits, Software: Jupyter Notebook y entorno Anaconda, lenguaje Python.

### 4. Metodología

El algoritmo desarrollado es como sigue:

```

Definicion de parametros de islas
    Num generaciones
    Cantidad de isleños
    Tipo Seleccion, cruza y mutacion para cada isla
    Definicion de ejecucion de migraciones
Creacion de isleños
    Evaluación cromosomas
    Ordenación por criterio elitista
for epsilon < 90 % or iteración > a Generaciones
    Llamada a isla1
    Llamada a isla2
    Llamada a isla3
    if generaciones % seasons1 == 0
        Eleccion de isla mejor y peor
        Intercambio de isleños entre ellos
    if generaciones % seasons2 == 0
        Intercambio circular de isleño entre islas
    if generaciones % seasons3 == 0
        Eleccion de isleño para visitar isla
        Cruza con individuo de isla visitada
        Retorno de isleño
    Verificacion de isleños por isla
    Ordenamiento por aptitud
    Se repite for
Graficación de resultados
Fin

```

La ejecución del algoritmo se propone de manera secuencial, primero se ejecuta una isla se guardan sus resultados y se continua con la siguiente. Cuando se termina de ejecutar las 3 islas se procede a realizar las migraciones en caso de que sea necesario dado las generaciones que lleva el contador *for*, al final se reordena los datos de cada isla para manejar los cambios que se realizaron debido a las migraciones.

La definición de generaciones se definen al mismo tiempo que el tipo de cruce, seleccion y mutación. Posteriormente se crean las poblaciones para cada isla, como se muestra en la sección de código de la figura 2.

```
Gener= 200
sel1,Cruza1,Muta1 = 1,0,0 # TIPO DE SELECCION: sel1= TORNEO, sel2= RANK, sel3= MONOGAMICO, sel4= RULETA,
sel2,Cruza2,Muta2 = 2,1,0 # TIPO DE CRUZA: Cruza1= 0 --> cruza 2 puntos; Cruza1= 1 --> cruza N puntos;,
sel3,Cruza3,Muta3 = 3,0,1 # TIPO DE MUTACION: Muta1= 0 --> SCRAMBLE; Muta1= 1 --> INVERSION DE ALELOS
sel4,Cruza4,Muta4 = 4,0,0
numPoblacion= 100

ini1= time.time()
population1,fit= pobisla(numPoblacion)
```

Figura 2: Sección de código de definición de parámetros.

La implementación de multi-islas se realiza con el fin de compartir información entre las islas, esto para poder resolver una tarea "dividiendo el trabajo" con el fin de alcanzar un óptimo de forma más eficiente.

Para ello se proponen tres tipos de migración:

- Migración Mejor/Peor: Se migran varios isleños de la isla con mejor promedio a la de peor promedio, para poder mejorar su promedio
- Migración Circular: Se migran varios pobladores entre las islas de manera circular, esto para que compartan información entre todas las islas. La isla 1 migra *n* isleños a la isla 2 y a su vez recibe de otra isla para complementar los que migraron. Se reacondiciona en caso de más islas.
- Visitante Marinero: Un isleño de cierta isla "visita" otra isla en la que se cruza con un isleño de la isla visitada eliminando *n* isleños para balancear la cantidad de isleños. Después el isleño visitante regresa a su isla.

Las migraciones ocurren cada 6 generaciones para migrar de mejor isla a peor, 3 para migrar de manera circular y 7 generaciones para visita de isleño. También se define la tolerancia DELTA de paro. Como se muestra en la sección de código de la figura 3.

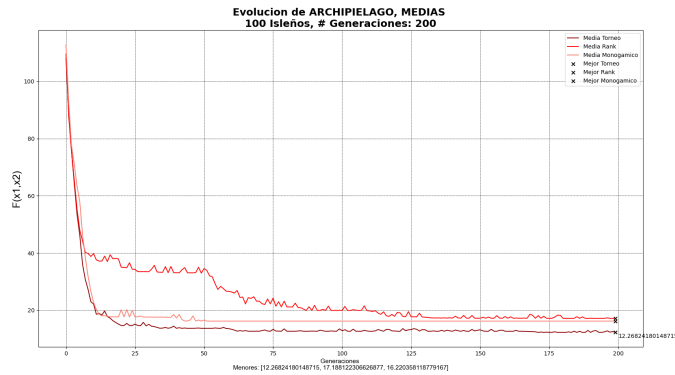
```
17 toleMin= 0.0001
18 seasons,moons,winters= 6,3,7
19 for g in range(Gener):
20     print('Generacion #: ',g+1, 'Distancia Minima: ', '{:.2f}'.format(population1[0][6]))
```

Figura 3: Seccion de codigo de definicion de tiempo de migraciones.

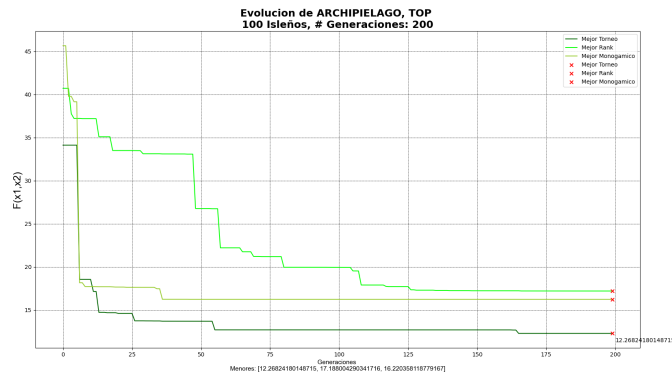
## 5. Resultados y discusión

Se realizaron 4 pruebas variando los parametros como numero de isleños, tipo de cruce y tipo de mutacion para cada tipo de Seleccion, se establecio un 5% de individuos para mutacion de cada poblacion, una tolerancia Delta de 0.0001 para asegurar que se llega al mínimo global con 12 cifras y las generaciones se mantuvieron en 200.

Primero se realizaron las pruebas a las isla sin interaccion entre ellas, esto es, que no hubo migracion entre ellas, para poder hacer comparaciones cuando posteriormente se llevaron a cabo las mismas pruebas pero con migracion entre islas. Las graficas de la figura 4 muestran los resultados obtenidos.



(a) Selección Torneo



(b) Evolucion de las medias de las islas

Figura 4: Evolucion del mejor de las islas



Parametros establecidos para prueba 1 son:

**Isla 1:** Numero de Isleños: 100

Seleccion: Torneo

Tipo de cruza: cruza 2 puntos

Tipo de mutacion: Scramble

**Isla 2:** Numero de Isleños: 100

Seleccion: Torneo

Tipo de cruza: cruza N puntos

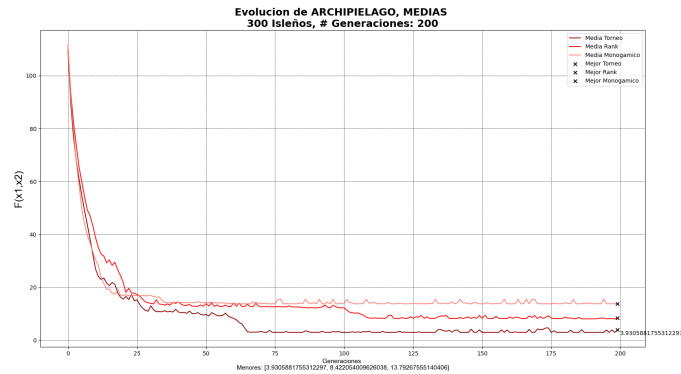
Tipo de mutacion: Scramble

**Isla 3:** Numero de Isleños: 100

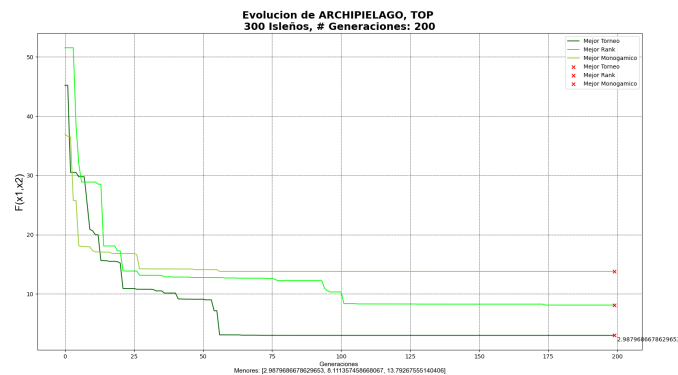
Seleccion: Torneo

Tipo de cruza: cruza 2 puntos

Tipo de mutacion: Inversion de alelos



(a) Seleccion Torneo



(b) Evolucion de las medias de las islas

Figura 5: Evolucion del mejor de las islas

Parametros establecidos para prueba 2 son:

**Isla 1:** Numero de Isleños: 300

Seleccion: Torneo

Tipo de cruza: cruza 2 puntos

Tipo de mutacion: Scramble

**Isla 2:** Numero de Isleños: 300

Seleccion: Torneo

Tipo de cruza: cruza N puntos

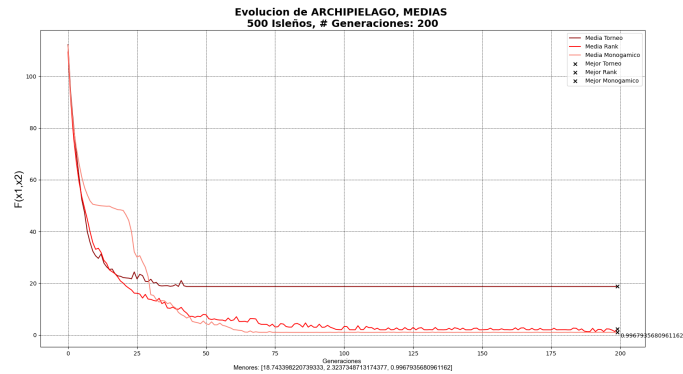
Tipo de mutacion: Scramble

**Isla 3:** Numero de Isleños: 300

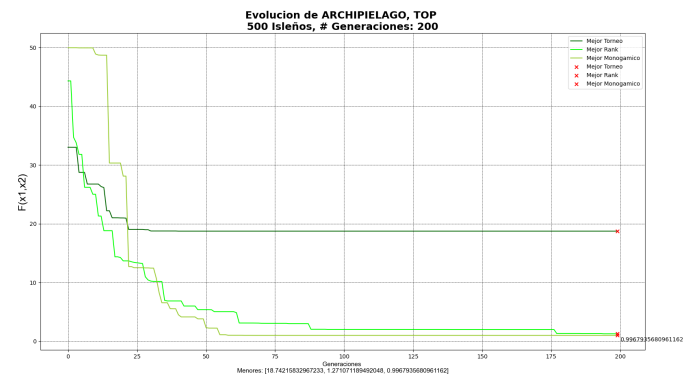
Seleccion: Torneo

Tipo de cruza: cruza 2 puntos

Tipo de mutacion: Inversion de alelos



(a) Seleccion Torneo



(b) Evolucion de las medias de las islas

Figura 6: Evolucion del mejor de las islas

Parametros establecidos para prueba 3 son:

**Isla 1:** Numero de Isleños: 500

Seleccion: Torneo

Tipo de cruza: cruza 2 puntos

Tipo de mutacion: Inversion de Alelos

**Isla 2:** Numero de Isleños: 500

Seleccion: Torneo

Tipo de cruza: cruza N puntos

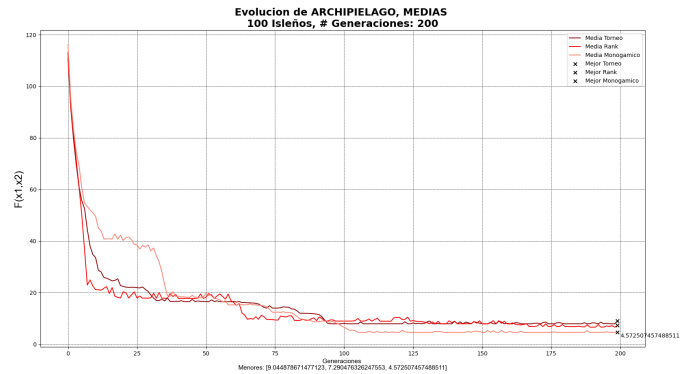
Tipo de mutacion: Scramble

**Isla 3:** Numero de Isleños: 500

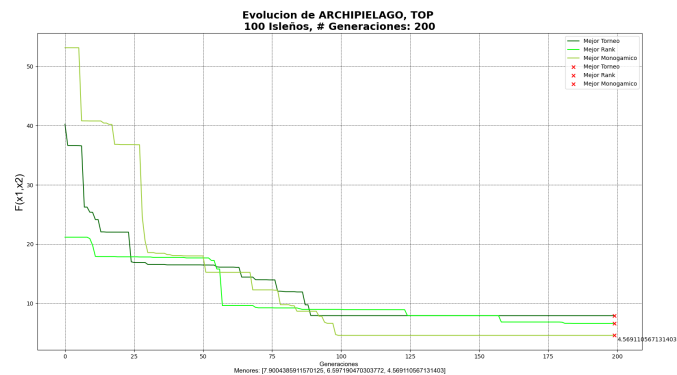
Seleccion: Torneo

Tipo de cruza: cruza 2 puntos

Tipo de mutacion: Scramble



(a) Seleccion Torneo



(b) Evolucion de las medias de las islas

Figura 7: Evolucion del mejor de las islas

Parametros establecidos para prueba 4 son:

**Isla 1:** Numero de Isleños: 100

Seleccion: Torneo

Tipo de cruza: cruza 2 puntos

Tipo de mutacion: Scramble

**Isla 2:** Numero de Isleños: 100

Seleccion: Torneo

Tipo de cruza: cruza 2 puntos

Tipo de mutacion: Scramble

**Isla 3:** Numero de Isleños: 100

Seleccion: Torneo

Tipo de cruza: cruza 2 puntos

Tipo de mutacion: Scramble

### **Pruebas realizadas con migracion entre islas**

Parametros establecidos para prueba 1 son:

**Isla 1:** Numero de Isleños: 100

Seleccion: Torneo

Tipo de cruza: cruza 2 puntos

Tipo de mutacion: Scramble

**Isla 2:** Numero de Isleños: 100

Seleccion: Torneo

Tipo de cruza: cruza N puntos

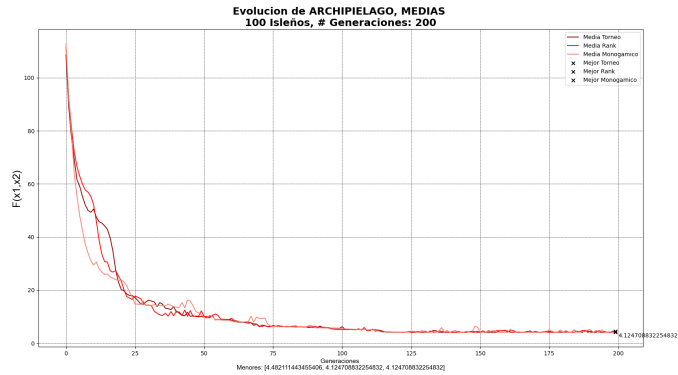
Tipo de mutacion: Scramble

**Isla 3:** Numero de Isleños: 100

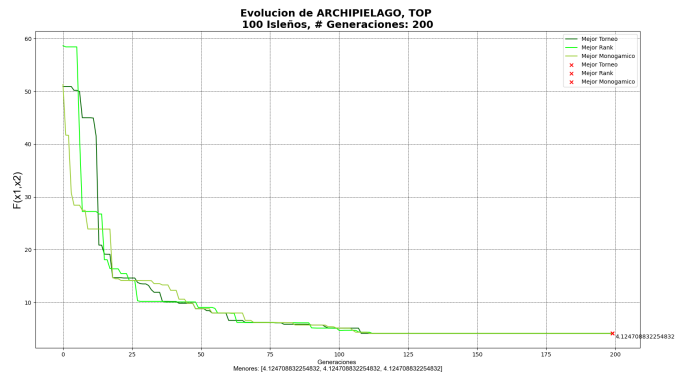
Seleccion: Torneo

Tipo de cruza: cruza 2 puntos

Tipo de mutacion: Inversion de alelos



(a) Seleccion Torneo



(b) Evolucion de las medias de las islas

Figura 8: Evolucion del mejor de las islas

Parametros establecidos para prueba 2 son:

**Isla 1:** Numero de Isleños: 300

Seleccion: Torneo

Tipo de cruza: cruza 2 puntos

Tipo de mutacion: Scramble

**Isla 2:** Numero de Isleños: 300

Seleccion: Torneo

Tipo de cruza: cruza N puntos

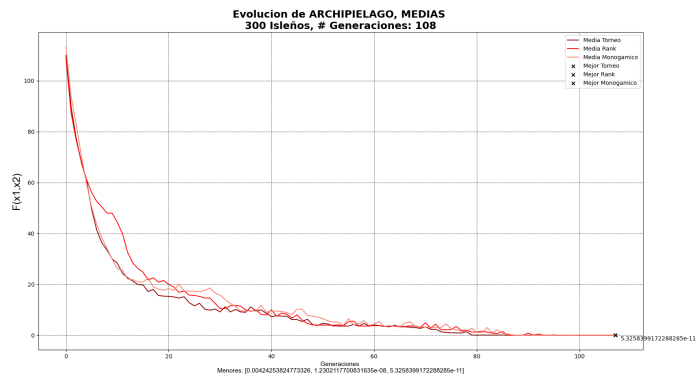
Tipo de mutacion: Scramble

**Isla 3:** Numero de Isleños: 300

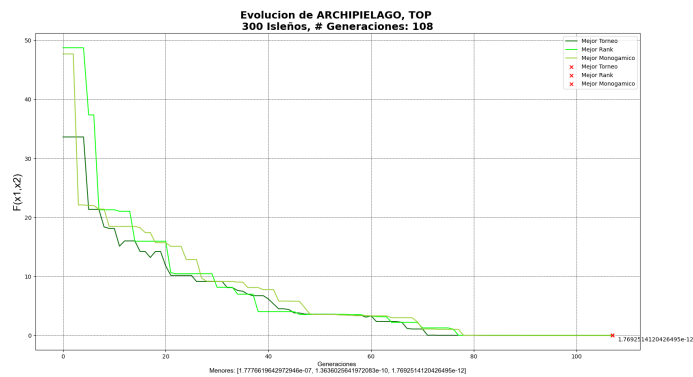
Seleccion: Torneo

Tipo de cruza: cruza 2 puntos

Tipo de mutacion: Inversion de alelos



(a) Seleccion Torneo



(b) Evolucion de las medias de las islas

Figura 9: Evolucion del mejor de las islas

Parametros establecidos para prueba 3 son:

**Isla 1:** Numero de Isleños: 500

Seleccion: Torneo

Tipo de cruza: cruza 2 puntos

Tipo de mutacion: Inversion de Alelos

**Isla 2:** Numero de Isleños: 500

Seleccion: Torneo

Tipo de cruza: cruza N puntos

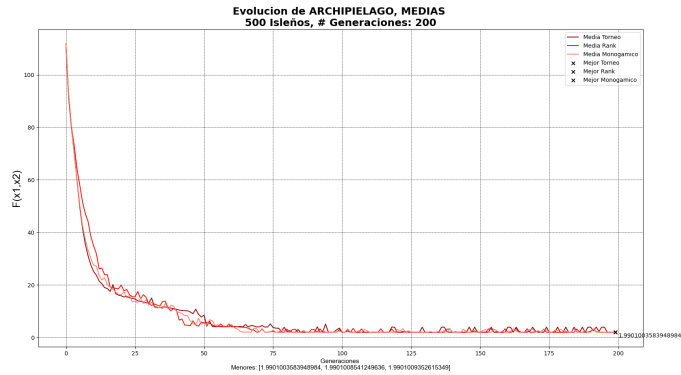
Tipo de mutacion: Scramble

**Isla 3:** Numero de Isleños: 500

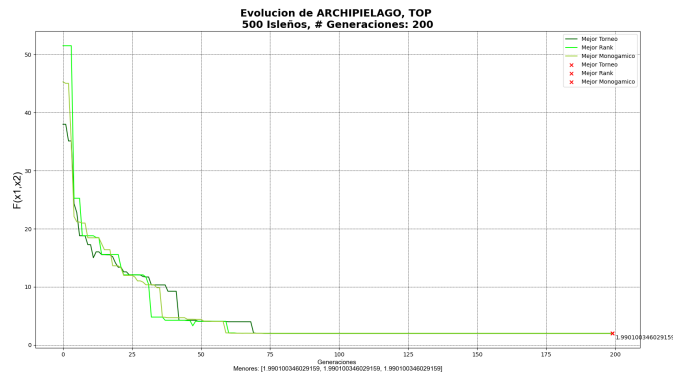
Seleccion: Torneo

Tipo de cruza: cruza 2 puntos

Tipo de mutacion: Scramble



(a) Selección Torneo



(b) Evolucion de las medias de las islas

Figura 10: Evolucion del mejor de las islas

Parametros establecidos para prueba 4 son:

**Isla 1:** Numero de Isleños: 100

Seleccion: Torneo

Tipo de cruza: cruza 2 puntos

Tipo de mutacion: Scramble

**Isla 2:** Numero de Isleños: 100

Seleccion: Torneo

Tipo de cruza: cruza 2 puntos

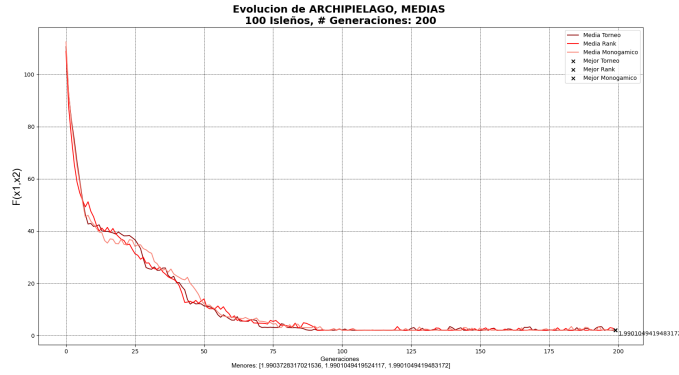
Tipo de mutacion: Scramble

**Isla 3:** Numero de Isleños: 100

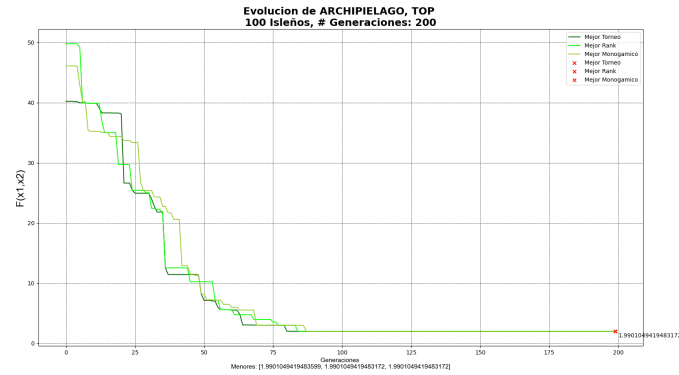
Seleccion: Torneo

Tipo de cruza: cruza 2 puntos

Tipo de mutacion: Scramble



(a) Seleccion Torneo



(b) Evolucion de las medias de las islas

Figura 11: Evolucion del mejor de las islas

## 6. Conclusiones

A partir de las graficas realizadas sin migracion entre islas se observa que dificilmente alcanzan o se acercan al minimo global de  $F(X) = 0$ . Se observa que la mutacion de inversion de Alelos es el que peor desempeño tienen al igual que la seleccion Random Monogamica, sin embargo, la Seleccion Random Monogamico se desempeña bien cuando se selecciona con la mutacion Scramble.

Para todas las pruebas sin interaccion entre islas se observa que el incremento de isleños no muestra mejoría alguna para alcanzar el optimo, esto se demuestra con la grafica que tiene seleccionados los mejores tipos de cruza y mutacion



y se observa que tiene el mismo desempeño que cualquier otra seleccion de parametros. Esto indica que los isleños dentro de cada isla convergen dentro de minimos locales y ya no pueden salir.

Por otro lado, cuando se realizan las pruebas bajo las mismas condiciones pero con migracion entre islas se observa que incluso en una prueba se alcanza el minimo de tolerancia lo que claramente indica que alcanzo el minimo global que es lo mas cerca de 0 posible. Ademas se observa una evolucion mucho mas rapida, suave y casi igual entre cada isla, por lo que compartir informacion entre ellas es un factor a tomar, debido a que alguna isla por mas peor que este su promedio puede contener alguna informacion que se acerque por otro lado al minimo global.

Ademas se verifica que las 3 islas alcanzan un minimo global similar, que si bien no es el mas bajo no distan mucho uno de la otra consiguiendo resultados en generaciones menos prolongadas.

## Referencias

- [1] Anna Demidova, Global Optimization Software and Evolutionary Algorithms, Lomonosov Moscow State University, 119991, Moscow, Russia
- [2] M. Yousaf., (2017), Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator., Computational Intelligence and Neuroscience., Research Article., 7430125., 7 Pages.

## 7. Anexos

# Multi\_Islands\_GAs

October 21, 2021

```
[ ]: from numpy import arange,exp,sqrt,cos,e,pi,meshgrid
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import random
from random import sample
import statistics
import numpy as np
import math
import time

%matplotlib
plt.ioff() # Activa Grafica en ventana nueva
```

## 1 Funcion FUNCION RASTRIGIN

```
[ ]: def Function_Rastrigin(x1,x2,x3,x4,x5,x6):
    return (x1**2 - 10 * np.cos(2 * np.pi * x1)) + \
           (x2**2 - 10 * np.cos(2 * np.pi * x2)) + \
           (x3**2 - 10 * np.cos(2 * np.pi * x3)) + \
           (x4**2 - 10 * np.cos(2 * np.pi * x4)) + \
           (x5**2 - 10 * np.cos(2 * np.pi * x5)) + \
           (x6**2 - 10 * np.cos(2 * np.pi * x6)) + 60
```

## 2 Funcion Convertir de cadena de genes a Decimal

```
[ ]: def lis2decimal(num):
    if num[0] == 1:
        signo=1
    else:
        signo=-1
    decimal=0
    for i in range(1,len(num)):
        decimal+=num[i]*10**(-i+2)

    return decimal*signo
```

### 3 Funcion Convertir x1,x2,x3,x4,x5,x6 a cadena de genes valC1 Y valC2

```
[ ]: def conv2cromos(pob):
    poblacion=[]
    community= pob.copy()
    # x1= 2.312345678910 cambia a x1= [1 2, 3, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1,0]
    # se convertiran 6 cromosomas debido a que la funcion se define con 6 dimensiones,
    →1 por cada dimensión
    for i in community:
        #print(i)
        crom1 = []
        crom2 = []
        crom3 = []
        crom4 = []
        crom5 = []
        crom6 = []
        for j,k,l,m,n,o in zip(i[0],i[1],i[2],i[3],i[4],i[5]):
            # print(j)
            # print(k)
            crom1 += [j]
            crom2 += [k]
            crom3 += [l]
            crom4 += [m]
            crom5 += [n]
            crom6 += [o]
        poblacion +=[(crom1,crom2,crom3,crom4,crom5,crom6,i[6])] # <----- composicion
    →de cada isleño, i[6] es su aptitud,
    nuevos= [] # no se procesa
    valC1= []
    valC2= []
    valC3= []
    valC4= []
    valC5= []
    valC6= []
    #valC2=[]
    for k in range(6): # <--- numero de cromosoma a procesar
        for cro in poblacion:
            # PRIMER caso: si != '-' son valores POSITIVOS, despues se revisa si es de
            →una o dos cifras como 13 o 7,
            predef=[0,0,0,0,0,0,0,0,0,0,0,0,0] # para CRO1
            if cro[k][0] != '-':
                if cro[k][1] == '.': # Condicion de 1 cifra entera positivo
                    predef[0]=1
                    predef[1]=0
                    predef[2]= int(cro[k][0])
                    predef[3:]= cro[k][2:12]
                    predef=[int(i) for i in predef] # Se guarda la cadena convertida
            →de acuerdo al numero de cromosoma
                if k ==0: # cromosomas: x1 o x2 o x3 o x4 o
            →x5 o x6
                    valC1 +=[predef]
                if k ==1:
                    valC2 +=[predef]
```

```

        if k ==2:
            valC3 +=[predef]
        if k ==3:
            valC4 +=[predef]
        if k ==4:
            valC5 +=[predef]
        if k ==5:
            valC6 +=[predef]
        nuevos +=[predef]

    #print(cro[0])
    if cro[k][2] == '.': # Condicion de 2 cifras enteras
        #print(cro[k])
        predef[0]=1
        predef[1]= int(cro[k][0])
        predef[2]= int(cro[k][1])
        predef[3:]= cro[k][3:13]
        predef=[int(i) for i in predef] # Se guarda la cadena convertida
→de acuerdo al numero de cromosoma
        if k ==0: # cromosomas: x1 o x2 o x3 o x4 o
→x5 o x6
            valC1 +=[predef]
        if k ==1:
            valC2 +=[predef]
        if k ==2:
            valC3 +=[predef]
        if k ==3:
            valC4 +=[predef]
        if k ==4:
            valC5 +=[predef]
        if k ==5:
            valC6 +=[predef]
        nuevos +=[predef]

    # x1= 2.312345678910 cambia a x1= [0 2, 3, 1, 2, 3, 4, 5, 6, 7, 8, 9,
→1,0]
    # SEGUNDO caso si == '-' son valores NEGATIVOS, despues se revisa si es de
→una o dos cifras como -13 o -7, para CRO1
    if cro[k][0] == '-':
        if cro[k][2] == '.': # Condicion de 2 cifras enteras negativo
            #print(cro[k])
            predef[0]=0
            predef[1]=0
            predef[2]= int(cro[k][1])
            predef[3:]= cro[k][3:13]
            predef=[int(i) for i in predef] # Se guarda la cadena convertida
→de acuerdo al numero de cromosoma
            if k ==0: # cromosomas: x1 o x2 o x3 o x4 o
→x5 o x6
                valC1 +=[predef]
            if k ==1:
                valC2 +=[predef]
            if k ==2:

```

```

        valC3 +=[predef]
    if k ==3:
        valC4 +=[predef]
    if k ==4:
        valC5 +=[predef]
    if k ==5:
        valC6 +=[predef]
    nuevos +=[predef]

    if cro[k][3] == '.': # Condicion de 1 cifra entera negativo
        predef[0]=0
        predef[1]=int(cro[k][1])
        predef[2]= int(cro[k][2])
        predef[3:] = cro[k][4:14]
        predef=[int(i) for i in predef] # Se guarda la cadena convertida
→de acuerdo al numero de cromosoma
        if k ==0: # cromosomas: x1 o x2 o x3 o x4 o
→x5 o x6

        valC1 +=[predef]
    if k ==1:
        valC2 +=[predef]
    if k ==2:
        valC3 +=[predef]
    if k ==3:
        valC4 +=[predef]
    if k ==4:
        valC5 +=[predef]
    if k ==5:
        valC6 +=[predef]
    nuevos +=[predef]

    return valC1,valC2,valC3,valC4,valC5,valC6

```

## 4 Funcion CRUZA N PUNTOS ALEATORIOS

```

[ ]: def crossNpoints(hx1,hx2,indice):
    # print('Funcion CRUZA N PUNTOS, seleccionado')
    cc= hx1.copy()
    ss= hx2.copy()
    watdog=0
    desc1=cc
    desc2=ss
    valido= False
    while valido == False:
        if watdog == 1000:
            #print('Se alcanzo limite y no se encontro cromos validos: regresan los
→que entraron para evitar cambios')
            return desc1,desc2
            ##### Aqui se seccionan de acuerdo a los indice obtenidos aleatorio
→que determinaran el tamaño de seccion
        cad11= cc[indice[0]].copy() # se seleccionan 4 puntos de corte para este caso
        cad12= cc[indice[0]:indice[1]].copy()

```

```

cad13= cc[indice[1]:indice[2]].copy()
cad14= cc[indice[2]:].copy()

cad21= ss[:indice[0]].copy()
cad22= ss[indice[0]:indice[1]].copy()
cad23= ss[indice[1]:indice[2]].copy()
cad24= ss[indice[2]:].copy()

##### SECCION CRUZAMIENTO N PUNTOS <Aqui se juntan las secciones
→ cortadas en diferentes puntos
desc1= cad11 + cad22 + cad13 + cad24 # con diferentes tamaños>, por ejemplo:
desc2= cad21 + cad12 + cad23 + cad14 # [1,2,3] + [7,6,3,4,5] + [2,3] + [2,5,6]
→ --> desc1

## Seccion verifica que cadenas convertidas a decimal esten en rango de la
→ funcion RASTRIGIN [-5.12 5.12]
valorfx1,valorfx2= lis2decimal(desc1),lis2decimal(desc2)
if valorfx1 < 5.12 and valorfx1 > -5.12 and valorfx2 < 5.12 and valorfx2 > -5.
→ 12:
    valido= True
    return desc1,desc2

indice= sample(range(1,12),3)
indice= sorted(indice)
watdog += 1

return desc1,desc2

```

## 5 Funcion CRUZA 2 PUNTOS ALEATORIOS

```

[ ]: def cross2points(hx1,hx2,indices):
#     print('Funcion CRUZA 2 PUNTOS, seleccionado')
c,cc= hx1.copy(),hx1.copy()
s,ss= hx2.copy(),hx2.copy()
watdog=0
desc1=cc
desc2=ss
valido= False
while valido == False:
    if watdog == 1000:
        #print('Se alcanzo limite y no se encontro cromos validos: Regresan los
→ que entraron para evitar cambios')
        return desc1,desc2
        ##### Se corta en 2 indice dados aleatorios, y se secciona de acuerdo al
→ tamaño dado por los puntos de corte
secc_cc= cc[indices[0]:indices[1]]
secc_ss= ss[indices[0]:indices[1]]
##### SECCION CRUZAMIENTO 2 PUNTOS: se insertan los nuevos genes a
→ los descendientes
desc1= cc[:indices[0]] + secc_ss + cc[indices[1]:] ## [324] + [548235] + [245]
→ --> desc1

```

```

desc2= ss[:indices[0]] + secc_cc + ss[indices[1]:] ## [164] + [245657] + [543]
→--> desc2

## Seccion verifica que cadenas convertidas a decimal esten en rango de la
→funcion RASTRIGIN
valorfx1,valorfx2= lis2decimal(desc1),lis2decimal(desc2)
if valorfx1 < 5.12 and valorfx1 > -5.12 and valorfx2 < 5.12 and valorfx2 > -5.
→12:
    valido= True
    return desc1,desc2

indices= sample(range(1,12),2)
indices= sorted(indices)
watdog += 1 # Contador de control

return c,s # SI NO ENCUENTRA DESCENDEINTES VALIDOS RETORNAN LOS QUE ENTRARON PARA
→EVITAR CAMBIOS

```

## 6 Mutacion SCRAMBLE

6.0.1 Se selecciona un subconjunto de genes y se reordena aleatoriamente los alelos (el subconjunto no tiene por qué ser contiguo).

```

[ ]: def mutacion_Scramble(mut1,mut2,mut3,mut4,mut5,mut6,size,cutin,cut_ind):
#     print('Funcion MUTACION SCRAMBLE, seleccionado')
    c1,chrom1= mut1.copy(),mut1.copy()
    c2,chrom2= mut2.copy(),mut2.copy()
    c3,chrom3= mut3.copy(),mut3.copy()
    c4,chrom4= mut4.copy(),mut4.copy()
    c5,chrom5= mut5.copy(),mut5.copy()
    c6,chrom6= mut6.copy(),mut6.copy()
    size_cut,cutinrange,cut_index= size,cutin,cut_ind
    watdog= 0
    descmut1= chrom1
    descmut2= chrom2
    descmut3= chrom3
    descmut4= chrom4
    descmut5= chrom5
    descmut6= chrom6
    valido= False
    while valido==False:
        if watdog == 1000:
            #print('Se alcanzo limite y no se encontro cromos validos: Retornan los
→que entraron para evitar cambios')
            return descmut1,descmut2,descmut3,descmut4,descmut5,descmut6
        secc_c1= chrom1[cut_index:cut_index+size_cut]
        secc_c2= chrom2[cut_index:cut_index+size_cut]
        secc_c3= chrom3[cut_index:cut_index+size_cut]
        secc_c4= chrom4[cut_index:cut_index+size_cut]
        secc_c5= chrom5[cut_index:cut_index+size_cut]
        secc_c6= chrom6[cut_index:cut_index+size_cut]

```

```

        random.shuffle(secc_c1),random.shuffle(secc_c2),random.shuffle(secc_c3), #
→Aqui se reordenan los genes selectos
        random.shuffle(secc_c4),random.shuffle(secc_c5),random.shuffle(secc_c6), #
→para su reordenamiento

        ##### Se reinserian las seccion reordenada para crear los descendientes
→mutados
        descmut1= chrom1[:cut_index] + secc_c1 + chrom1[cut_index+size_cut:] ## NO SE
→INTERCAMBIAN ENTRE CROMOSOMAS,
        descmut2= chrom2[:cut_index] + secc_c2 + chrom2[cut_index+size_cut:] ## SON
→DOS PORQUE SE MANDAN X1 Y X2,
        descmut3= chrom3[:cut_index] + secc_c3 + chrom3[cut_index+size_cut:] ## SOLO
→SE REORDENAN INTERNAMENTE
        descmut4= chrom4[:cut_index] + secc_c4 + chrom4[cut_index+size_cut:] ## [4821]
→+ [3657841] + [1954] ---> descx
        descmut5= chrom5[:cut_index] + secc_c5 + chrom5[cut_index+size_cut:]
        descmut6= chrom6[:cut_index] + secc_c6 + chrom6[cut_index+size_cut:]

        ##### Se convierten a flotante para su validación
        v1,v2,v3= lis2decimal(descmut1),lis2decimal(descmut2),lis2decimal(descmut3)
        v4,v5,v6= lis2decimal(descmut4),lis2decimal(descmut5),lis2decimal(descmut6)
        if v1 < 5.12 and v1 > -5.12 and v2 < 5.12 and v2 > -5.12 and v3 < 5.12 and v3
→> -5.12 and v4 < 5.12 and v4 > -5.12 and v5 < 5.12 and v5 > -5.12 and v6 < 5.12
→and v6 > -5.12 :
            valido= True
            return descmut1,descmut2,descmut3,descmut4,descmut5,descmut6 # Retorna
→los descendientes nuevos validos

        elif watdog > 30: # En caso de que no sean validos y son mas de 30
→iteraciones busca un nuevo indice de corte
            size_cut=4
            cutinrange= len(chrom1)-1-size_cut
            cut_index= np.random.randint(1,cutinrange)
            watdog = watdog+1 # control de busqueda de cromosomas mutados
            # Retornan los que entraron para evitar cambios en caso de NO encontrar validos
            return c1,c2,c3,c4,c5,c6

```

## 7 Funcion MUTACION INVERSIÓN

```

[ ]: def mutacion_Inversion(mut1,mu2,mu3,mu4,mu5,mu6,size,cutin,cut_ind):
#     print('Funcion MUTACION INVERSION, seleccionado')
    ch1,chrom1= mut1.copy(),mut1.copy()
    ch2,chrom2= mu2.copy(),mu2.copy()
    ch3,chrom3= mu3.copy(),mu3.copy()
    ch4,chrom4= mu4.copy(),mu4.copy()
    ch5,chrom5= mu5.copy(),mu5.copy()
    ch6,chrom6= mu6.copy(),mu6.copy()
    size_cut,cutinrange,cut_index= size,cutin,cut_ind
    watdog= 0
    descmut1= chrom1
    descmut2= chrom2

```



```

descmut3= chrom3
descmut4= chrom4
descmut5= chrom5
descmut6= chrom6
valido= False
while valido==False:
    #print('Mutacion',watdog)
    if watdog == 1000:
        #print('Se alcanzo limite y no se encontro cromos validos: Retornan los
→que entraron para evitar cambios')
        return mut1,mut2,mut3,mut4,mut5,mut6
        secc_s1= chrom1[cut_index:cut_index+size_cut]
        secc_s2= chrom2[cut_index:cut_index+size_cut]
        secc_s3= chrom3[cut_index:cut_index+size_cut]
        secc_s4= chrom4[cut_index:cut_index+size_cut]
        secc_s5= chrom5[cut_index:cut_index+size_cut]
        secc_s6= chrom6[cut_index:cut_index+size_cut]

        ## Se secciona cada cromosoma dado el tamaño de corte para su mutacion y se
→invierten sus posiciones
        secc_s1.reverse(),secc_s2.reverse(),secc_s3.reverse(),secc_s4.
→reverse(),secc_s5.reverse(),secc_s6.reverse()

        descmut1= chrom1[:cut_index] + secc_s1 + chrom1[cut_index+size_cut:] ## Se
→forma la nueva cadena con la seccion
        descmut2= chrom2[:cut_index] + secc_s2 + chrom2[cut_index+size_cut:] ##
→invertida para realizar mutacion
        descmut3= chrom3[:cut_index] + secc_s3 + chrom3[cut_index+size_cut:] ##
→[75821] + [125] + [75978] --> descmutx
        descmut4= chrom4[:cut_index] + secc_s4 + chrom4[cut_index+size_cut:] ##
→[75821] + [125] + [75978] --> descmutx
        descmut5= chrom5[:cut_index] + secc_s5 + chrom5[cut_index+size_cut:]
        descmut6= chrom6[:cut_index] + secc_s6 + chrom6[cut_index+size_cut:]

        v1,v2,v3= lis2decimal(descmut1),lis2decimal(descmut2),lis2decimal(descmut3)
        v4,v5,v6= lis2decimal(descmut4),lis2decimal(descmut5),lis2decimal(descmut6)
        if v1 < 5.12 and v1 > -5.12 and v2 < 5.12 and v2 > -5.12 and v3 < 5.12 and v3
→> -5.12 and v4 < 5.12 and v4 > -5.12 and v5 < 5.12 and v5 > -5.12 and v6 < 5.12
→and v6 > -5.12 :
            valido= True
            return descmut1,descmut2,descmut3,descmut4,descmut5,descmut6 # <----
→RETORNA CROMOSOMAS MUTADOS VALIDOS

        size_cut=4
        cutinrange= len(chrom1)-1-size_cut
        cut_index= np.random.randint(1,cutinrange)
        watdog = watdog+1

    return ch1,ch2,ch3,ch4,ch5,ch6 # Retornan los que entraron para evitar cambios

```

## 8 Funcion SELECCION TORNEO

```
[ ]: def selTorneo(cromosomas, tipoC, indices, indice):
#     print('Funcion SELECCION TORNEO, seleccionado')
    ch1= cromosomas.copy()
    newpob1= []
    indiv= 0
    for i in range(len(ch1)//2):
        gx1,gx2,gx3= ch1[indiv][0],ch1[indiv][1],ch1[indiv][2]
        gx4,gx5,gx6= ch1[indiv][3],ch1[indiv][4],ch1[indiv][5]

        gxx1,gxx2,gxx3=_
        →ch1[len(ch1)-indiv-1][0],ch1[len(ch1)-indiv-1][1],ch1[len(ch1)-indiv-1][2]
        gxx4,gxx5,gxx6=_
        →ch1[len(ch1)-indiv-1][3],ch1[len(ch1)-indiv-1][4],ch1[len(ch1)-indiv-1][5]

        if tipoC==0: # <----- Tipo cruza 2 puntos
            offx1,offxx1= cross2points(gx1,gxx1,indices) # 1.- x1 y 2.- xx1
            offx2,offxx2= cross2points(gx2,gxx2,indices) # 1.- x2 y 2.- xx2
            offx3,offxx3= cross2points(gx3,gxx3,indices) # 1.- x3 y 2.- xx3
            offx4,offxx4= cross2points(gx4,gxx4,indices) # 1.- x4 y 2.- xx4
            offx5,offxx5= cross2points(gx5,gxx5,indices) # 1.- x5 y 2.- xx5
            offx6,offxx6= cross2points(gx6,gxx6,indices) # 1.- x6 y 2.- xx6

            if tipoC==1: # <----- Tipo cruza n puntos
                offx1,offxx1= crossNpoints(gx1,gxx1,indice)
                offx2,offxx2= crossNpoints(gx2,gxx2,indice)
                offx3,offxx3= crossNpoints(gx3,gxx3,indice)
                offx4,offxx4= crossNpoints(gx4,gxx4,indice)
                offx5,offxx5= crossNpoints(gx5,gxx5,indice)
                offx6,offxx6= crossNpoints(gx6,gxx6,indice)

            newpob1 += [(offx1,offx2,offx3,offx4,offx5,offx6)]
            newpob1 += [(offxx1,offxx2,offxx3,offxx4,offxx5,offxx6)]
            indiv += 1

    return newpob1
```

## 9 Funcion SELECCION RANK

```
[ ]: def selRank(cromosomas, tipoC, indices, indice):
#     print('Funcion SELECCION RANK, seleccionado')
    ch2= cromosomas.copy()
    newpob2= []
    indiv=0
    for i in range(len(ch2)//2):
        gx1,gx2,gx3= ch2[2*indiv][0],ch2[2*indiv][1],ch2[2*indiv][2] # Seleccion de_
        →primer sexteto de cromosomas a
        gx4,gx5,gx6= ch2[2*indiv][3],ch2[2*indiv][4],ch2[2*indiv][5] # cruzar

        gxx1,gxx2,gxx3= ch2[2*indiv+1][0],ch2[2*indiv+1][1],ch2[2*indiv+1][2] #_
        →Seleccion de Segundo sexteto de
```

```

gxx4,gxx5,gxx6= ch2[2*indiv+1][3],ch2[2*indiv+1][4],ch2[2*indiv+1][5] #
→ cromosomas a cruzar

if tipoC==0: # <----- Tipo cruza 2 puntos
    offx1,offxx1= cross2points(gx1,gxx1,indices) # 1.- x1 y 2.- xx1
    offx2,offxx2= cross2points(gx2,gxx2,indices) # 1.- x2 y 2.- xx2
    offx3,offxx3= cross2points(gx3,gxx3,indices) # 1.- x3 y 2.- xx3
    offx4,offxx4= cross2points(gx4,gxx4,indices) # 1.- x4 y 2.- xx4
    offx5,offxx5= cross2points(gx5,gxx5,indices) # 1.- x5 y 2.- xx5
    offx6,offxx6= cross2points(gx6,gxx6,indices) # 1.- x6 y 2.- xx6

if tipoC==1: # <----- Tipo cruza n puntos
    offx1,offxx1= crossNpoints(gx1,gxx1,indice)
    offx2,offxx2= crossNpoints(gx2,gxx2,indice)
    offx3,offxx3= crossNpoints(gx3,gxx3,indice)
    offx4,offxx4= crossNpoints(gx4,gxx4,indice)
    offx5,offxx5= crossNpoints(gx5,gxx5,indice)
    offx6,offxx6= crossNpoints(gx6,gxx6,indice)

newpob2 += [(offx1,offx2,offx3,offx4,offx5,offx6)]
newpob2 += [(offxx1,offxx2,offxx3,offxx4,offxx5,offxx6)]
indiv += 1

return newpob2

```

## 10 Funcion SELECCION RANDOM MONOGAMICO

```

[ ]: def selMonogamico(cromosomas,tipoC,indices,indice):
#     print('Funcion SELECCION MONOGAMICO, seleccionado')
    ch3= cromosomas.copy()
    ch3randomizados= cromosomas.copy() # Se hace una copia de la lista de cromosomas
→ para reacomodarlos aleatoriamente
    random.shuffle(ch3randomizados) # y se extraen 2 a la vez para realizar la
→ cruzar
    newpob3= []
    indiv=0
    for i in range(len(ch3)//2):
        hab1= ch3randomizados.pop(0)
        hab2= ch3randomizados.pop(0)

        gx1,gx2,gx3= hab1[0],hab1[1],hab1[2] # Seleccion de primer sexteto de
→ cromosomas a
        gx4,gx5,gx6= hab1[3],hab1[4],hab1[5] # cruzar

        gxx1,gxx2,gxx3= hab2[0],hab2[1],hab2[2] # Seleccion de Segundo sexteto de
        gxx4,gxx5,gxx6= hab2[3],hab2[4],hab2[5] # cromosomas a cruzar

    if tipoC==0: # <----- Tipo cruza 2 puntos
        offx1,offxx1= cross2points(gx1,gxx1,indices) # 1.- x1 y 2.- xx1
        offx2,offxx2= cross2points(gx2,gxx2,indices) # 1.- x2 y 2.- xx2
        offx3,offxx3= cross2points(gx3,gxx3,indices) # 1.- x3 y 2.- xx3
        offx4,offxx4= cross2points(gx4,gxx4,indices) # 1.- x4 y 2.- xx4

```

```

        offx5,offxx5= cross2points(gx5,gxx5,indices) # 1.- x5 y 2.- xx5
        offx6,offxx6= cross2points(gx6,gxx6,indices) # 1.- x6 y 2.- xx6

    if tipoC==1: # <----- Tipo cruza n puntos
        offx1,offxx1= crossNpoints(gx1,gxx1,indice)
        offx2,offxx2= crossNpoints(gx2,gxx2,indice)
        offx3,offxx3= crossNpoints(gx3,gxx3,indice)
        offx4,offxx4= crossNpoints(gx4,gxx4,indice)
        offx5,offxx5= crossNpoints(gx5,gxx5,indice)
        offx6,offxx6= crossNpoints(gx6,gxx6,indice)

    newpob3 += [(offx1,offx2,offx3,offx4,offx5,offx6)]
    newpob3 += [(offxx1,offxx2,offxx3,offxx4,offxx5,offxx6)]
    indiv += 1

return newpob3

```

## 11 Funcion SELECCION RULETA

```

[ ]: def selRuleta(cromosomas,tipoC,indices,indice,odds):
    #     print('Funcion SELECCION RULETA, seleccionado')
    ch4= cromosomas.copy()
    newpob4= []
    indiv=0
    for i in range(len(ch4)//2):
        gen=np.random.choice(len(ch4), 2, p=odds) # Selecciona 2 indices de todos los
        → cromosomas de acuerdo a su aptitud
        a,b= ch4[gen[0]],ch4[gen[1]] # se separan en a y b y se obtienen
        → los cromosomas individuales a[x] y b[x]

        gx1,gx2,gx3= a[0],a[1],a[2] # Seleccion de primer sexteto de cromosomas a
        gx4,gx5,gx6= a[3],a[4],a[5] # cruzar

        gxx1,gxx2,gxx3= b[3],b[4],b[5] # Seleccion de Segundo sexteto de
        gxx4,gxx5,gxx6= b[3],b[4],b[5] # cromosomas a cruzar

    if tipoC==0: # <----- Tipo cruza 2 puntos
        offx1,offxx1= cross2points(gx1,gxx1,indices) # 1.- x1 y 2.- xx1
        offx2,offxx2= cross2points(gx2,gxx2,indices) # 1.- x2 y 2.- xx2
        offx3,offxx3= cross2points(gx3,gxx3,indices) # 1.- x3 y 2.- xx3
        offx4,offxx4= cross2points(gx4,gxx4,indices) # 1.- x4 y 2.- xx4
        offx5,offxx5= cross2points(gx5,gxx5,indices) # 1.- x5 y 2.- xx5
        offx6,offxx6= cross2points(gx6,gxx6,indices) # 1.- x6 y 2.- xx6

    if tipoC==1: # <----- Tipo cruza n puntos
        offx1,offxx1= crossNpoints(gx1,gxx1,indice)
        offx2,offxx2= crossNpoints(gx2,gxx2,indice)
        offx3,offxx3= crossNpoints(gx3,gxx3,indice)
        offx4,offxx4= crossNpoints(gx4,gxx4,indice)
        offx5,offxx5= crossNpoints(gx5,gxx5,indice)
        offx6,offxx6= crossNpoints(gx6,gxx6,indice)

```

```

newpob4 += [(offx1,offx2,offx3,offx4,offx5,offx6)]
newpob4 += [(offxx1,offxx2,offxx3,offxx4,offxx5,offxx6)]
indiv += 1

return newpob4

```

## 12 Creacion de Isleños de comunidades iniciales

```

[ ]: def pobisla(numCromos):
    valMin= -5.12      # rango minimo para x1, x2, x3, x4, x5 y x6
    valMax=  5.12      # rango maximo para x1, x2, x3, x4, x5 y x6
    #numCromos= 100    # Numero de pobladores

    com_1= [ random.uniform(valMin,valMax) for x1 in range(numCromos)] # Alelos 1
    com_2= [ random.uniform(valMin,valMax) for x1 in range(numCromos)] # Alelos 2
    com_3= [ random.uniform(valMin,valMax) for x1 in range(numCromos)] # Alelos 3
    com_4= [ random.uniform(valMin,valMax) for x1 in range(numCromos)] # Alelos 4
    com_5= [ random.uniform(valMin,valMax) for x1 in range(numCromos)] # Alelos 5
    com_6= [ random.uniform(valMin,valMax) for x1 in range(numCromos)] # Alelos 6

    archi=[]
    positions=[]
    odds= []
    for c1,c2,c3,c4,c5,c6 in zip(com_1,com_2,com_3,com_4,com_5,com_6):
        fx= Function_Rastrigin(c1,c2,c3,c4,c5,c6)
        archi+= [(str(c1),str(c2),str(c3),str(c4),str(c5),str(c6),fx)]
        odds+= [fx]
        fx1= Function_Rastrigin(c1,c2,0,0,0,0)
        positions +=[fx1] # <-- datos para grafica, solo 2 dimensiones

    odds= np.array(odds)
    odds= odds/odds.sum()

    va1,va2,va3,va4,va5,va6= conv2cromos(archi)

    ComInicial=[]
    for isle1,isle2,isle3,isle4,isle5,isle6 in zip(va1,va2,va3,va4,va5,va6):
        i1,i2,i3= lis2decimal(isle1),lis2decimal(isle2),lis2decimal(isle3)
        i4,i5,i6= lis2decimal(isle4),lis2decimal(isle5),lis2decimal(isle6)
        Fx= Function_Rastrigin(i1,i2,i3,i4,i5,i6)
        ComInicial +=[(isle1,isle2,isle3,isle4,isle5,isle6,Fx)]

    ComInicial= sorted(ComInicial, key=lambda ComInicial: ComInicial[6], reverse=False)

    return ComInicial,odds

```

## 13 Funcion EVOLUCION DE ARCHIPIELAGO

```
[ ]: def evolucion(pob, tipoS, tipoC, tipoM, odds): # tipoC: Tipo de Cruza: PMX OR OX.  tipoM:
    → Tipo de Mutacion: Scramble or Heuristica
    oldG=pob.copy()
    #listgeneraciones=[]
    toleMin= 0.001
    toleCambio= 0.1
    ## Seccion que elige un indice para cortar en cruza 2 y n puntos por generacion
    size_cut=4
    cutinrange = len(oldG[0][0])-1-size_cut
    cut_index = np.random.randint(1,cutinrange)

    ##### SECCION QUE ELIGE PUNTOS DE CORTE PARA CRUZA 2 y N-PUNTOS
    indices= sample(range(1,12),2) # <--- se eligen 2 indices de corte y se ordenan
    → para iniciar de izquierda a derecha
    indices= sorted(indices)
    indice= sample(range(1,12),3) # <--- se eligen 3 indices de corte y se ordenan
    → para iniciar de izquierda a derecha
    indice= sorted(indice)
    ##### lo hace aleatorio cada nueva generacion
    cromos= oldG.copy()
    if tipoS ==1:
        newGen= selTorneo(cromos,tipoC,indices,indice)
    if tipoS ==2:
        newGen= selRank(cromos,tipoC,indices,indice)
    if tipoS ==3:
        newGen= selMonogamico(cromos,tipoC,indices,indice)
    if tipoS ==4:
        newGen= selRuleta(cromos,tipoC,indices,indice,odds)

    ##### SECCION EVALUACION Y ORDENACION DE NUEVA GENERACION 'newGen'
    pobNueva=[]
    for A in newGen:
        A1,A2,A3= lis2decimal(A[0]),lis2decimal(A[1]),lis2decimal(A[2])
        A4,A5,A6= lis2decimal(A[3]),lis2decimal(A[4]),lis2decimal(A[5])
        Fx= Function_Rastrigin(A1,A2,A3,A4,A5,A6)
        pobNueva +=[(A[0],A[1],A[2],A[3],A[4],A[5],Fx)]
    pobNueva= sorted(pobNueva, key=lambda pobNueva: pobNueva[6], reverse=False)

    ##### SECCION COMBINACION GENERACION ANTERIOR Y NUEVA
    newPoblacion=[] # Se producen 100 individuos
    for indiv1, indiv2 in zip(pobNueva, oldG): # Aqui se mezclan las primeras
        newPoblacion += [indiv1,indiv2] # 2 generaciones y se ordenan
    → los mas aptos para su cruza
    newPoblacion=sorted(newPoblacion, key=lambda newPoblacion: newPoblacion[6],
    → reverse=False)
    pob_Mejor=newPoblacion[:len(newPoblacion)//2] # Seleccion de los mejores
    → individuos igual a n/2

    ##### SECCION MUTACION
    toMolt= (5*len(pob_Mejor))//100 # Seleccion de porcentaje de individuos a mutar
    mut=[]
```

```

for i in range(toMolt):
    cr= random.choice(pob_Mejor)
    mut+= [(cr[0],cr[1],cr[2],cr[3],cr[4],cr[5])]
    pob_Mejor.remove(cr)

molts=[]
for m in mut:
    if tipoM==0: # <----- TIPO DE MUTACION SCRAMBLE
        molt1,molt2,molt3,molt4,molt5,molt6=
→mutacion_Scramble(m[0],m[1],m[2],m[3],m[4],m[5],size_cut,cutinrange,cut_index)
    if tipoM==1: # <----- TIPO DE MUTACION INVERSION
        molt1,molt2,molt3,molt4,molt5,molt6=
→mutacion_Inversion(m[0],m[1],m[2],m[3],m[4],m[5],size_cut,cutinrange,cut_index)

    fx1,fx2,fx3= lis2decimal(molt1),lis2decimal(molt2),lis2decimal(molt3)
    fx4,fx5,fx6= lis2decimal(molt4),lis2decimal(molt5),lis2decimal(molt6)
    Fx1x2= Function_Rastrigin(fx1,fx2,fx3,fx4,fx5,fx6)
    molts+= [(molt1,molt2,molt3,molt4,molt5,molt6,Fx1x2)]

##### REINSECCION DE CROMOSOMAS MUTADOS AL POBLACION NUEVA MEJORADA
for sdr in molts:
    indice=np.random.randint(0,len(pob_Mejor))
    pob_Mejor.insert(indice,sdr)
    pob_Mejor=sorted(pob_Mejor, key=lambda pob_Mejor: pob_Mejor[6], reverse=False)

##### GUARDADO DE LISTA DE LISTAS DE valores flotantes
xn= []
→(lis2decimal(h[0]),lis2decimal(h[1]),lis2decimal(h[2]),lis2decimal(h[3]),lis2decimal(h[4]),lis2decimal(h[5]),lis2decimal(h[6]))
→for h in oldG]

oldless= oldG[0][6]
newless= pob_Mejor[0][6]
if newless <= oldless:
    oldG = pob_Mejor.copy()
if newless > oldless:
    oldG = oldG

return oldG,xn,odds

```

## 14 Funcion CRUZA MARINERO E ISLEÑO

```

[ ]: def cruza_Marinero(marino,isleño,indice):
    hab1= marino
    hab2= isleño

    gx1,gx2,gx3= hab1[0],hab1[1],hab1[2] # Seleccion de primer sexteto de cromosomas a
    gx4,gx5,gx6= hab1[3],hab1[4],hab1[5] # cruzar

    gxx1,gxx2,gxx3= hab2[0],hab2[1],hab2[2] # Seleccion de Segundo sexteto de
    gxx4,gxx5,gxx6= hab2[3],hab2[4],hab2[5] # cromosomas a cruzar

    offx1,offxx1= crossNpoints(gx1,gxx1,indice)

```

```

offx2,offxx2= crossNpoints(gx2,gxx2,indice)
offx3,offxx3= crossNpoints(gx3,gxx3,indice)
offx4,offxx4= crossNpoints(gx4,gxx4,indice)
offx5,offxx5= crossNpoints(gx5,gxx5,indice)
offx6,offxx6= crossNpoints(gx6,gxx6,indice)

A1,A2,A3= lis2decimal(offx1),lis2decimal(offx2),lis2decimal(offx3)
A4,A5,A6= lis2decimal(offx4),lis2decimal(offx5),lis2decimal(offx6)
Fx= Function_Rastrigin(A1,A2,A3,A4,A5,A6)
descendiente1= (offx1,offx2,offx3,offx4,offx5,offx6,Fx)

A1,A2,A3= lis2decimal(offxx1),lis2decimal(offxx2),lis2decimal(offxx5)
A4,A5,A6= lis2decimal(offxx3),lis2decimal(offxx4),lis2decimal(offxx6)
Fx= Function_Rastrigin(A1,A2,A3,A4,A5,A6)
descendiente2= (offxx1,offxx2,offxx3,offxx4,offxx5,offxx6,Fx)

return descendiente1,descendiente2

```

## 15 Funcion MAIN SIN INTERACCION ENTRE ISLAS

```

[ ]: Gener= 200
sel1,Cruza1,Muta1 = 1,0,0 # TIPO DE SELECCION: sel1= TORNEO, sel2= RANK, sel3=
    ↳ MONOGAMICO, sel4= RULETA,
sel2,Cruza2,Muta2 = 2,0,0 # TIPO DE CRUZA: Cruza1= 0 --> cruza 2 puntos; Cruza1= 1
    ↳ --> cruza N puntos;,
sel3,Cruza3,Muta3 = 3,0,0 # TIPO DE MUTACION: Muta1= 0 --> SCRAMBLE; Muta1= 1
    ↳ --> INVERSION DE ALELOS
#sel4,Cruza4,Muta4 = 4,0,0
numPoblacion= 100

ini1= time.time()
population1,fit= pobisla(numPoblacion)
population2,fit= pobisla(numPoblacion)
population3,fit= pobisla(numPoblacion)
#population4,fit4= pobisla(numPoblacion)
lisgraph1= []
lisgraph2= []
lisgraph3= []
# lisgraph4= []
toleMin= 0.0001
seasons,moons,winters= 6,3,7
for g in range(Gener):
    print('Generacion #: ',g+1, 'Distancia Minima: ', '{:.2f}'.
        ↳ format(population1[0][6]))
    # print('Generacion #: ',g+1, 'Distancia Minima: ', '{:.2f}'.
        ↳ format(population2[0][6]))
    # print('Generacion #: ',g+1, 'Distancia Minima: ', '{:.2f}'.
        ↳ format(population3[0][6]))
    # print('Generacion #: ',g+1, 'Distancia Minima: ', '{:.2f}'.
        ↳ format(population4[0][6]))
    ini2= time.time()
    m_Torneo,data1,fit = evolucion(population1,sel1,Cruza1,Muta1,fit)

```



```

fin2= time.time()

ini3= time.time()
m_Rank,data2,fit = evolucion(population2,sel2,Cruza2,Muta2,fit)
fin3= time.time()

ini4= time.time()
m_RanMon,data3,fit = evolucion(population3,sel3,Cruza3,Muta3,fit)
fin4= time.time()

##### RECOPIACION DE DATOS POR GENERACION POR CADA ISLA PARA
→GRAFICAS
lisgraph1+= [data1]
lisgraph2+= [data2]
lisgraph3+= [data3]
#     lisgraph4+= [data4]
##### CRITERIO DE PARO DELTA
Total1= [ p[6] for p in m_Torneo]
Total2= [ p[6] for p in m_Rank]
Total3= [ p[6] for p in m_RanMon]
if sum(Total1) < toleMin and sum(Total2) < toleMin and sum(Total3) < toleMin:
    print('Tolerancia minima permitida alcanzada, paro en generacion #: ',g+1)
    break

population1= m_Torneo
population2= m_Rank
population3= m_RanMon

fin1= time.time()
#     population4,fit4= mejores_Rul,fit4
print(fin1-ini1)
print(fin2-ini2)
print(fin3-ini3)
print(fin4-ini4)

print('lo que se imprime cuando retorna de llamada a funcion... se muestran 10
→primeros isleños')
print()
solo10= [1,2,3,4,5,6,7,8,9,0]
for xj,solo in zip(population1,solo10):
    print(xj[0],xj[1],xj[2],'\n',xj[3],xj[4],xj[5],'\n',xj[6])

```

## 16 Funcion MAIN CON INTERACCION ENTRE ISLAS

```

[ ]: Gener= 200
sel1,Cruza1,Muta1 = 1,0,0 # TIPO DE SELECCION: sel1= TORNEO, sel2= RANK, sel3=
→MONOGAMICO, sel4= RULETA,
sel2,Cruza2,Muta2 = 2,0,0 # TIPO DE CRUZA: Cruza1= 0 --> cruza 2 puntos; Cruza1= 1
→--> cruza N puntos;;
sel3,Cruza3,Muta3 = 3,0,0 # TIPO DE MUTACION: Muta1= 0 --> SCRAMBLE; Muta1= 1
→--> INVERSION DE ALELOS
#sel4,Cruza4,Muta4 = 4,0,0

```

```

numPoblacion= 100

ini1= time.time()
population1,fit= pobisla(numPoblacion)
population2,fit= pobisla(numPoblacion)
population3,fit= pobisla(numPoblacion)
#population4,fit4= pobisla(numPoblacion)
lisgraph1= []
lisgraph2= []
lisgraph3= []
# lisgraph4= []
toleMin= 0.0001
seasons,moons,winters= 6,3,7
for g in range(Gener):
    print('Generacion #: ',g+1, 'Distancia Minima: ', '{:.2f}'.
    →format(population1[0][6]))
#     print('Generacion #: ',g+1, 'Distancia Minima: ', '{:.2f}'.
    →format(population2[0][6]))
#     print('Generacion #: ',g+1, 'Distancia Minima: ', '{:.2f}'.
    →format(population3[0][6]))
#     print('Generacion #: ',g+1, 'Distancia Minima: ', '{:.2f}'.
    →format(population4[0][6]))
    ini2= time.time()
    m_Torneo,data1,fit = evolucion(population1,sel1,Cruza1,Muta1,fit)
    fin2= time.time()

    ini3= time.time()
    m_Rank,data2,fit = evolucion(population2,sel2,Cruza2,Muta2,fit)
    fin3= time.time()

    ini4= time.time()
    m_RanMon,data3,fit = evolucion(population3,sel3,Cruza3,Muta3,fit)
    fin4= time.time()

    ##### RECOPIACION DE DATOS POR GENERACION POR CADA ISLA PARA
    →GRAFICAS
    lisgraph1+= [data1]
    lisgraph2+= [data2]
    lisgraph3+= [data3]
#     lisgraph4+= [data4]
    ### ***** MIGRACION ENTRE MEJOR A PEOR ISLA DE ACUERDO AL
    →PROMEDIO DE CADA ISLA CADA 5 GENERACIONES
    if (g+1)%seasons ==0:
        print('MIGRACION ENTRE MEJOR A PEOR ISLA DE ACUERDO AL PROMEDIO DE CADA ISLA')
        ### Obtencion de medias para conocer a la peor y mejor isla ###
        datum1= [Function_Rastrigin(q[0],q[1],q[2],q[3],q[4],q[5]) for q in data1]
        datum2= [Function_Rastrigin(q[0],q[1],q[2],q[3],q[4],q[5]) for q in data2]
        datum3= [Function_Rastrigin(q[0],q[1],q[2],q[3],q[4],q[5]) for q in data3]
        d1,d2,d3= statistics.mean(datum1),statistics.mean(datum2),statistics.
        →mean(datum3) # Obtencion de promedios
        w_isla=[d1,d2,d3]

        ## PEOR ISLA PROMEDIO

```

```

        if max(w_isla)== w_isla[0]:
            migraw1,migraw2= m_Torneo.pop(0),m_Torneo.pop(0) # EXTRAE EL PRIMER
→ELEMENTO Y DESPUES EL SEGUNDO, SON IGUALES
            if max(w_isla)== w_isla[1]: # YA QUE SE ELIMINA EL
→ELEMENTO EN ESE INDICE Y SE DESPLAZA
                migraw1,migraw2= m_Rank.pop(0), m_Rank.pop(0)
            if max(w_isla)== w_isla[2]:
                migraw1,migraw2= m_RanMon.pop(0), m_RanMon.pop(0)

        ## MEJOR ISLA PROMEDIO
        if min(w_isla)== w_isla[0]: # len(m_seleccion) indica el indice donde se
→insertan los migrantes en esta caso al final
            m_Torneo.insert(len(m_Torneo),migraw1), m_Torneo.
→insert(len(m_Torneo),migraw2) # INSERCIÓN DE MIGRANTE AL FINAL DE
            migrab1,migrab2= m_Torneo.pop(0), m_Torneo.pop(0)
→ # ISLA CON MEJOR PROMEDIO PARA INTER-
            if min(w_isla)== w_isla[1]: #
→CAMBIAR ENTRE PEOR Y MEJOR ISLA
                m_Rank.insert(len(m_Rank),migraw1), m_Rank.insert(len(m_Rank),migraw2)
                migrab1,migrab2= m_Rank.pop(0), m_Rank.pop(0)
            if min(w_isla)== w_isla[2]:
                m_RanMon.insert(len(m_RanMon),migraw1), m_RanMon.
→insert(len(m_RanMon),migraw2)
                migrab1,migrab2= m_RanMon.pop(0), m_RanMon.pop(0)

        ## INSERCIÓN DE MIGRANTES DE MEJOR ISLA A LA PEOR ISLA PROMEDIO
        if max(w_isla)== w_isla[0]:
            m_Torneo.insert(len(m_Torneo),migraw1),m_Torneo.
→insert(len(m_Torneo),migraw2)
            if max(w_isla)== w_isla[1]:
                m_Rank.insert(len(m_Rank),migraw1),m_Rank.insert(len(m_Rank),migraw2)
            if max(w_isla)== w_isla[2]:
                m_RanMon.insert(len(m_RanMon),migraw1),m_RanMon.
→insert(len(m_RanMon),migraw2)

        ### ===== MIGRACION CIRCULAR ALEATORIO ENTRE ISLAS CADA 3
→GENERACIONES
        if (g+1)%moons== 0:
            ci= sample(range(1,numPoblacion),3) # SE ELIGEN UN ISLEÑO ALEATORIO DE CADA
→ISLA
            ci= sorted(ci)
            print('MIGRACION Circular entre islas cada 3 generaciones')

            # Se extrae un cromosoma aleatorio de cada isla.
            migra_cir1,migra_cir2,migra_cir3= m_Torneo.pop(ci[0]), m_Rank.pop(ci[1]),
→m_RanMon.pop(ci[2])

            # se reinserta cromosoma a cada isla-lista, con 'insert(index,elemento)'
            m_RanMon.insert(len(m_RanMon),migra_cir1),m_Torneo.
→insert(len(m_Torneo),migra_cir2),m_Rank.insert(len(m_Rank),migra_cir3)

        ### ----- VISITANTE MARINERO ENTRE ISLAS DE PEOR A MEJOR

```

```

if (g+1)%winters== 0:
    indice= sample(range(1,12),3) # Se obtienen los indices para realizar CRUZA N
→PUNTOS
    indice= sorted(indice)
    print('MIGRACION VISITANTE MARINERO entre islas cada 5 generaciones')
    ### SE OBTIENEN LAS MEDIAS PARA CONOCER LA MEJOR Y PEOR ISLA ###
    datum1= [Function_Rastrigin(q[0],q[1],q[2],q[3],q[4],q[5]) for q in data1]
    datum2= [Function_Rastrigin(q[0],q[1],q[2],q[3],q[4],q[5]) for q in data2]
    datum3= [Function_Rastrigin(q[0],q[1],q[2],q[3],q[4],q[5]) for q in data3]
    d1,d2,d3= statistics.mean(datum1),statistics.mean(datum2),statistics.
→mean(datum3) # Obtencion de promedios
    w_isla=[d1,d2,d3]

    ## MEJOR ISLA PROMEDIO
    if min(w_isla)== w_isla[0]:
        Marino= m_Torneo[0] # ISLEÑO CON MEJOR PROMEDIO PARA INTERCAMBIAR
→GENES
    if min(w_isla)== w_isla[1]:
        Marino= m_Rank[0] # ISLEÑO CON MEJOR PROMEDIO PARA INTERCAMBIAR
→GENES
    if min(w_isla)== w_isla[2]:
        Marino= m_RanMon[0] # ISLEÑO CON MEJOR PROMEDIO PARA INTERCAMBIAR
→GENES

    ## PEOR ISLA PROMEDIO
    if max(w_isla)== w_isla[0]:
        Isleño,h= m_Torneo.pop(0), m_Torneo.pop(-1) # ISLEÑO CON MEJOR
→PROMEDIO PARA INTERCAMBIAR GENES
    if max(w_isla)== w_isla[1]:
        Isleño,h= m_Rank.pop(0), m_Rank.pop(-1) # ISLEÑO CON MEJOR
→PROMEDIO PARA INTERCAMBIAR GENES
    if max(w_isla)== w_isla[2]:
        Isleño,h= m_RanMon.pop(0), m_RanMon.pop(-1) # ISLEÑO CON MEJOR
→PROMEDIO PARA INTERCAMBIAR GENES

    ## CRUZA N PUNTOS
    offx1,offxx1= cruza_Marinero(Marino,Isleño,indice) # Cruzamiento por n
→puntos para obtener nuevos y rellenar peor ISLA

    ## REINSERCIÓN DE DESCENDIENTE NUEVOS A PEOR ISLA PROMEDIO
    if max(w_isla)== w_isla[0]:
        m_Torneo.insert(len(m_Torneo),offx1),m_Torneo.insert(len(m_Torneo),offxx1)
    if max(w_isla)== w_isla[1]:
        m_Rank.insert(len(m_Rank),offx1),m_Rank.insert(len(m_Rank),offxx1)
    if max(w_isla)== w_isla[2]:
        m_RanMon.insert(len(m_RanMon),offx1),m_RanMon.insert(len(m_RanMon),offxx1)

    # Ordenamiento de poblacion antes de entrar a evolucionar nuevamente
    m_Torneo= sorted(m_Torneo, key=lambda m_Torneo: m_Torneo[6], reverse=False)
    m_Rank= sorted(m_Rank, key=lambda m_Rank: m_Rank[6], reverse=False)
    m_RanMon= sorted(m_RanMon, key=lambda m_RanMon: m_RanMon[6], reverse=False)

```

```

##### CRITERIO DE PARO DELTA
Total1= [ p[6] for p in m_Torneo]
Total2= [ p[6] for p in m_Rank]
Total3= [ p[6] for p in m_RanMon]
if sum(Total1) < toleMin and sum(Total2) < toleMin and sum(Total3) < toleMin:
    print('Tolerancia minima permitida alcanzada, paro en generacion #: ',g+1)
    break

population1= m_Torneo
population2= m_Rank
population3= m_RanMon
fin1= time.time()
# population4,fit4= mejores_Rul,fit4
print(fin1-ini1)
print(fin2-ini2)
print(fin3-ini3)
print(fin4-ini4)

print('lo que se imprime cuando retorna de llamada a funcion... se muestran 10_
→primeros isleños')
print()
solo10= [1,2,3,4,5,6,7,8,9,0]
for xj,solo in zip(population1,solo10):
    print(xj[0],xj[1],xj[2], '\n',xj[3],xj[4],xj[5], '\n',xj[6])

```

## 17 Graficas de evolucion con tres Tipos de seleccion

```

[ ]: topGeneration1= []
topGeneration2= []
topGeneration3= []
topGeneration4= []
topG1_fx1x2=[]
topG2_fx1x2=[]
topG3_fx1x2=[]
topG4_fx1x2=[]
G=0
axisx= []
#for b,e,s,t in zip(lisgraph1,lisgraph2,lisgraph3,lisgraph4):
for b,e,s in zip(lisgraph1,lisgraph2,lisgraph3):
    topG1_fx1x2+= [Function_Rastrigin(b[0][0],b[0][1],b[0][2],b[0][3],b[0][4],b[0][5])]
    topG2_fx1x2+= [Function_Rastrigin(e[0][0],e[0][1],e[0][2],e[0][3],e[0][4],e[0][5])]
    topG3_fx1x2+= [Function_Rastrigin(s[0][0],s[0][1],s[0][2],s[0][3],s[0][4],s[0][5])]
    #topG4_fx1x2+=
    →[Function_Rastrigin(t[0][0],t[0][1],t[0][2],t[0][3],t[0][4],t[0][5])]
    axisx+= [G]
    G+=1

g=0
#for r,f,v,w in zip(lisgraph1,lisgraph2,lisgraph3,lisgraph4):
top1=[]
for r,f,v in zip(lisgraph1,lisgraph2,lisgraph3):
    mediaGen1=[]

```

```

mediaGen2=[]
mediaGen3=[]
mean1= [Function_Rastrigin(q[0],q[1],q[2],q[3],q[4],q[5]) for q in r]
mean2= [Function_Rastrigin(q[0],q[1],q[2],q[3],q[4],q[5]) for q in f]
mean3= [Function_Rastrigin(q[0],q[1],q[2],q[3],q[4],q[5]) for q in v]
#mean1= [Function_Rastrigin(q[0],q[1],q[2],q[3],q[4],q[5]) for q in r]
topGeneration1+= [statistics.mean(mean1)]
topGeneration2+= [statistics.mean(mean2)]
topGeneration3+= [statistics.mean(mean3)]
#topGeneration4+= [statistics.mean(mean4)]
g+=1

minimo= [topG1_fx1x2[-1],topG2_fx1x2[-1],topG3_fx1x2[-1]]
minimo1= [topGeneration1[-1],topGeneration2[-1],topGeneration3[-1]]
less= min(minimo)
less1= min(minimo1)
## GRAFICA DE TOP POR ISLAS
fig, ax = plt.subplots(figsize=(12,6))
plt.grid(color = 'k', linestyle = '--', linewidth = 0.5)
ax.plot(axisx,topG1_fx1x2, 'darkgreen',label='Mejor Torneo',zorder=1)
ax.plot(axisx,topG2_fx1x2, 'lime',label='Mejor Rank',zorder=2)
ax.plot(axisx,topG3_fx1x2, 'yellowgreen',label='Mejor Monogamico',zorder=3)
ax.scatter(axisx[-1],topG1_fx1x2[-1],color='r',marker='x',label='Mejor_
↳Torneo',zorder=7)
ax.scatter(axisx[-1],topG2_fx1x2[-1],color='r',marker='x',label='Mejor Rank',zorder=8)
ax.scatter(axisx[-1],topG3_fx1x2[-1],color='r',marker='x',label='Mejor_
↳Monogamico',zorder=9)
plt.title('Evolucion de ARCHIPIELAGO, TOP \n%d Isleños, # Generaciones: {0}'.
↳format(len(axisx)) %(numPoblacion),fontweight="bold",fontsize=18)
ax.annotate(less,(axisx[-1],less),xytext=(axisx[-1]+1,less-1))
ax.legend()
ax.set_xlabel('Generaciones \nMenores: ' + str(minimo),font="Arial",fontsize=11)
ax.set_ylabel('F(x1,x2)',font="Arial",fontsize=18)
plt.show()
## GRAFICA DE MEDIAS
fig, ax = plt.subplots(figsize=(12,6))
plt.grid(color = 'k', linestyle = '--', linewidth = 0.5)
ax.plot(axisx,topGeneration1, 'darkred',label='Media Torneo',zorder=1)
ax.plot(axisx,topGeneration2, 'red',label='Media Rank',zorder=2)
ax.plot(axisx,topGeneration3, 'salmon',label='Media Monogamico',zorder=3)
ax.scatter(axisx[-1],topGeneration1[-1],color='k',marker='x',label='Mejor_
↳Torneo',zorder=7)
ax.scatter(axisx[-1],topGeneration2[-1],color='k',marker='x',label='Mejor_
↳Rank',zorder=8)
ax.scatter(axisx[-1],topGeneration3[-1],color='k',marker='x',label='Mejor_
↳Monogamico',zorder=9)
plt.title('Evolucion de ARCHIPIELAGO, MEDIAS \n%d Isleños, # Generaciones: {0}'.
↳format(len(axisx)) %(numPoblacion),fontweight="bold",fontsize=18)
ax.annotate(less1,(axisx[-1],less1),xytext=(axisx[-1]+1,less1-2))
ax.legend()
ax.set_xlabel('Generaciones \nMenores: ' + str(minimo1),font="Arial",fontsize=11)
ax.set_ylabel('F(x1,x2)',font="Arial",fontsize=18)
plt.show()

```

## 18 GRAFICA 3D DE FUNCION RASTRIGIN

```
[ ]: x1 = np.linspace(-5.12, 5.12, 100)
x2 = np.linspace(-5.12, 5.12, 100)
X, Y = np.meshgrid(x1, x2)
Z = (X**2 - 10 * np.cos(2 * np.pi * X)) + \
    (Y**2 - 10 * np.cos(2 * np.pi * Y)) + 20
a,b=0,0
optimo= Function_Rastrigin(a,b,0,0,0,0)
figure = plt.figure(figsize=(12,8))
##### FIRST PLOT ##### se rota con klik izquierdo; se aumenta con click derecho
axis = figure.add_subplot(121, projection='3d')
axis.plot_surface(X, Y, Z, rstride=1,cstride=1,cmap='terrain', shade= "false", alpha=0.
    ↪5,zorder=1)
#plt.contour(x,y,results,cmap='Spectral',offset=-1,zorder=2)
axis.set_title('Funcion RASTRIGIN DE 2 DIMENSIONES',fontweight='bold',fontsize=15)
axis.scatter3D(a,b,optimo,s=10,color='black',marker='p',zorder=3)
axis.set_xlabel('$x1$',fontweight='bold',fontsize=15)
axis.set_ylabel('$x2$',fontweight='bold',fontsize=15)
axis.set_zlabel('$f(x1,x2)$ ',fontweight='bold',fontsize=10)
axis.view_init(90,90) # Ajusta como aparece inicialmente grafica en grados
##### SECOND PLOT ##### se rota con klik izquierdo; se aumenta con click derecho
axis = figure.add_subplot(122, projection='3d')
axis.plot_surface(X, Y, Z, rstride=1,cstride=1,cmap='terrain', shade= "false", alpha=0.
    ↪5,zorder=1)
#plt.contour(x,y,results,cmap='Spectral',offset=-1,zorder=2)
axis.set_title('Funcion RASTRIGIN DE 2 DIMENSIONES',fontweight='bold',fontsize=15)
axis.scatter3D(a,b,optimo,s=10,color='black',marker='p',zorder=3)
axis.set_xlabel('$x1$',fontweight='bold',fontsize=15)
axis.set_ylabel('$x2$',fontweight='bold',fontsize=15)
axis.set_zlabel('$f(x1,x2)$ ',fontweight='bold',fontsize=15)
axis.view_init(10,210) # Ajusta como aparece inicialmente grafica en grados
ax.set_xlim(-6,6)
ax.set_xlim(-6,6)
plt.show()
```