SOFT-COMPUTING

Algoritmos evolutivos

Juan José Córdoba Zamora
Juan Casado Ballesteros

Universidad de Alcalá

ÍNDICE

Introducción

Hemos programado un total de 6 algoritmos de optimización:

- evolutivo (1+1)
- evolutivo (n+m)
- harmónico
- temple simulado
- evolutivo con búsqueda local
- genético

Para validar los algoritmos hemos utilizado las tres funciones proporcionadas evaluadas en sus respectivos dominios.
Los parámetros de cada algoritmo han sido ajustados mediante múltiples iteraciones con múltiples valores para cada una de las funciones, por lo que los resultados que salen de haberlo ejecutado son los mejores que hemos conseguido modificando sigma, ancho de banda, etc….
Mostraremos los resultados del mejor fenotipo (se entiende por mejor fenotipo el individuo que mejor fitness ha obtenido) en 30 ejecuciones de cada algoritmo para cada función junto a su fitness, así como la fitness media de cada mejor fenotipo en las 30 ejecuciones.
Adicionalmente explicaremos tanto la representación elegida para fenotipos y poblaciones, así como la arquitectura del código creado incluyendo capturas de código como apoyo para las explicaciones.

Arquitectura del código

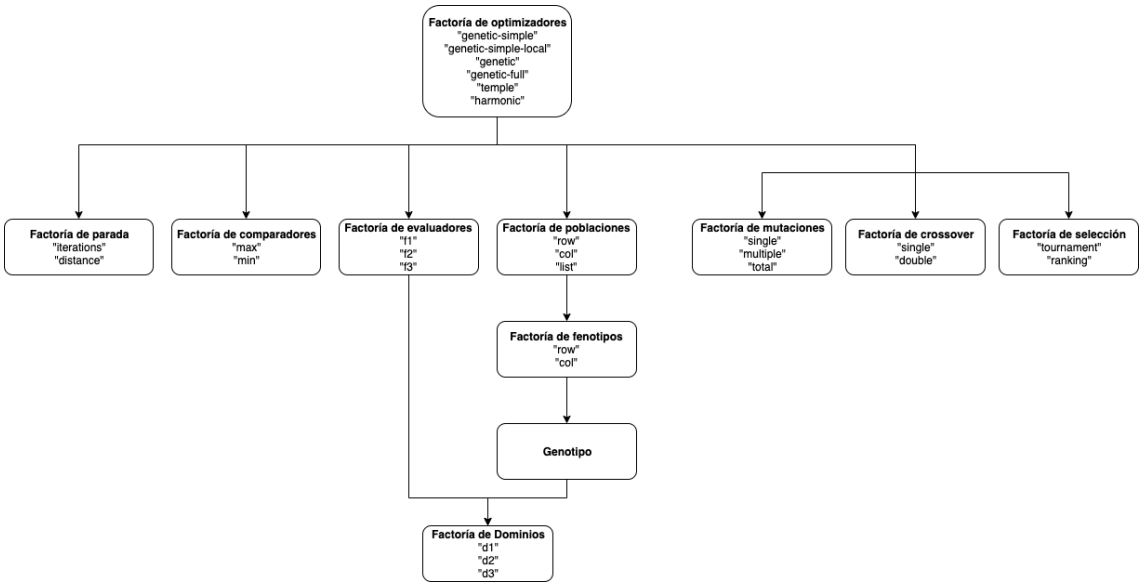
Hemos intentado hacer que el código fuera lo más modular y reutilizable posible.

Para ello hemos realizado un estudio previo de los algoritmos que deseábamos implementar para detectar las acciones comunes a realizar en varios de ellos de modo que las hemos extraído en funciones para poderlas reutilizar.

Tras hacer esto decidimos unir las acciones relacionadas mediante factorías de modo que pudiéramos seleccionar y configurar aquella función que deseábamos utilizar de una forma cómoda. Las factorías creadas nos devuelven como resultado funciones lambda parcialmente configuradas y listas para ser utilizadas en los algoritmos. De este modo podemos posponer la ejecución de cada función hasta que sea necesario de modo que podemos crear la estructura de múltiples sin ejecutarlos y luego ejecutarlos todos juntos cuando sea necesario. Adicionalmente los parámetros invariables a lo largo de una optimización de cada función pueden quedar rellenados sin tener que especificarlos cada vez.

Finalmente, cuando hemos construido nuestros algoritmos se los podemos pasar a un ejecutor para que los ejecute y procese los resultados.

Como aclaración cuando indicamos fenotipo se refiere a los individuos de una población, es decir, a las filas y cuando decimos genotipo se refiere a los valores que tiene el individuo, es decir, a las columnas.



Modelado del problema

Para modelar el problema hemos decidido representar las poblaciones como matrices en las que cada fila sea un fenotipo distinto y cada elemento de la fila un genotipo.
La matriz de la población tendrá tantas columnas como genes los fenotipos y tantas filas como fenotipos la población.
Para evaluar un fenotipo solo tendremos que extraer de la matriz la fila correspondiente y pasarla por la función de evaluación para obtener su fitness. Posteriormente podremos saber qué fitness es mejor utilizando un comparador que implementará la función máximo o mínimo según lo hayamos configurado.
En el caso de temple o de los algoritmos evolutivos más simples la población es de solo un fenotipo por lo que nuestra matriz de población será de una única fila.
En el caso de la optimización harmónica los individuos serán filas, la matriz de población la de harmonías y cada armonía estará en una fila de la matriz.

Para cada algoritmo ha sido necesario encontrar los parámetros de ajuste que mejor se adaptaban a él. Esto lo hemos hecho mirando las gráficas de evolución del fitness para cada punto de ajuste hasta encontrar un balance entre obtener mejores resultados y que el tiempo de ejecución fuera menor. Pararemos cada algoritmo estableciendo el número de iteraciones cuando este haya convergido ya y deje de mejorar sus resultados. En algún caso lo haremos algo antes de modo que los resultados sean ligeramente peores pero el tiempo de ejecución se reduzca.

Conclusiones

Compararemos los algoritmos en la calidad de los resultados proporcionados para cada función y en el tiempo que tardan en optimizar. En tiempo depende por su puesto de las iteraciones, pero ya que hemos ajustado cada algoritmo con un número de iteraciones tal que más no logran mejorar el resultado que ha proporcionado creemos que es justo comprarlos por este parámetro también.

Compararemos también las diferencias entre el mejor resultado y la media. No es basta proporcionar un buen resultado a veces, es necesario hacerlo con frecuencia suficiente.

- **evolutivo (1+1):** llega a soluciones suficientemente buenas en un tiempo alto para la función 1. En la función 2 no logra tan buenos resultados como otras opciones. En la función 3 logra mejores resultados que otras opciones, pero estos no son demasiado buenos.
- **evolutivo (n+m):** tarda más que la opción anterior, pero logra mejores resultados que el algoritmo anterior para las funciones 1 y 2, especialmente para la función 2. En la función 3 logra peores resultados.
- **harmónico:** tarda más que los dos algoritmos anteriores logrando peores resultados en las funciones 1 y 3 pero logrando muy buenos resultados en la función 2. Tiene problemas para salir de los mínimos locales cuando la matriz de harmonías se llena con la misma harmonía.
- **temple simulado:** Logra muy buenos resultados en todas las funciones en un tiempo reducido. Cabe destacar la gran varianza en la media para la función 3. Es necesario establecer una sigma mayor de lo normal pues esta va cayendo con el tiempo.
- **evolutivo con búsqueda local:** Logra los segundos mejores resultados en la función 1 y los segundos mejores en la 3. No obstante en la función 2 tiene muy malos resultados pues no es capaz de salir de los mínimos locales. Tarda mucho en optimizar debido que para obtener esos buenos resultados son necesarias muchas iteraciones.
- **genético:** logra optimizar en un número muy reducido de iteraciones logrando los mejores resultados en las funciones 1 y 2. Los resultados que proporciona son muy consistentes a diferencia de en otros algoritmos donde hay mayor diferencia entre el mejor resultado y le media de ellos. No hemos encontrado diferencias entre la selección por ranking y torneo, aunque puede que eso cambie y se utilizaran otras funciones.

Por lo generar para la función 1 la mejor sigma encontrada está entorno a 0.5 lo que nos indica que la función no es excesivamente irregular. En el caso de la función 2 la sigma debe ser mucho mayor, entorno a 4, esto se debe a la presencia de mínimos locales de los que debemos escapar. Por el contrario, en la función 3 sigmas menores, cerca de 0.01 funcionan mejor, es necesario explorar esta función dando pequeños pasos.

Hemos podido comprobar que el cruce es un método de exploración mejor que otros. Permite reducir en gran medida los tiempos de ejecución llegando a buenas soluciones. Adicionalmente se comprueba que es bueno explorar los mínimos locales pues proporciona buenos resultados, aunque para lograr una mayor velocidad de ejecución en mejor hacerlo al final de la ejecución dejando caer la sigma como en el temple simulado.

Evolutivo (1+1)

Este es el algoritmo más simple implementado.
En él tendremos como población un único fenotipo el cual mutaremos y compararemos con el fenotipo anterior. En caso de que tras la mutación haya mejorado nos quedaremos con este nuevo fenotipo siendo descartado en caso contrario. Repetiremos el proceso hasta que la función de parada nos lo indique.

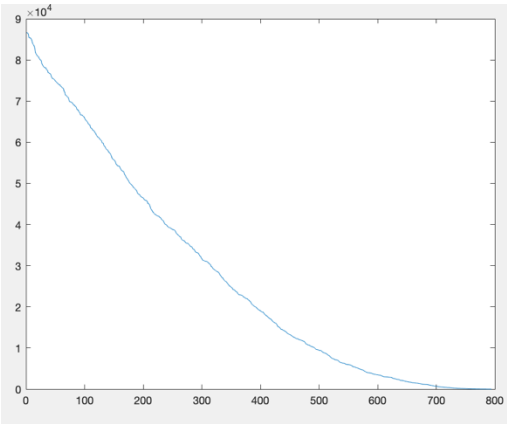
```
function [best_fenotype, best_fitness] = GeneticSimple (evaluator, comparator, stopper, newFenotype,
sigma)
    best_fenotype = newFenotype();
    best_fitness = [evaluator(best_fenotype)];
    while ~stopper()
        fenotype = totalMutation(sigma, best_fenotype);
        fitness = evaluator(fenotype);
        if comparator(fitness, best_fitness)
            best_fitness = [best_fitness fitness];
            best_fenotype = fenotype;
        end
    end
end
```

Función 1

population_len: 0
sigma: 0.5000
fenotype_len: 30
max_iterations: 100000
percentage: 0
domain: "d1"
comparator: "min"
evaluator: "f1"
stopper: "iterations"
mutation: ""
crossover: ""
selection: ""
algorithm: "genetic-simple"
lambda: 0

Resultados:

Best fitness: 1.8211
Mean fitness: 2.6972
Time: 15.9048
Best fenotype:
[0.21095, -0.030979, -0.27793, -0.19943, -0.43942, 0.25256, -0.077149, -0.13863, -0.24671, 0.67183, 0.22206, 0.058031, -0.16859, 0.055771, 0.1763, -0.099959, 0.19135, -0.072371, 0.46583, -0.1888, 0.024478, -0.16482, 0.057407, 0.14316, 0.22424, 0.31938, -0.27646, -0.28063, 0.12137, -0.26614]

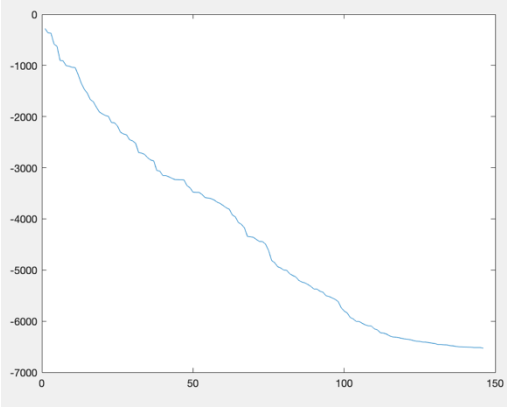


Función 2

population_len: 0
sigma: 4
fenotype_len: 30
max_iterations: 100000
percentage: 0
domain: "d2"
comparator: "min"
evaluator: "f2"
stopper: "iterations"
mutation: ""
crossover: ""
selection: ""
algorithm: "genetic-simple"
lambda: 0

Resultados:

Best fitness: -9308.6466
Mean fitness: -7483.1483
Time: 18.5609
Best fenotype:
[-557.6337, 416.2545, -303.7474, 422.2108, 421.4748, 201.8215, 200.0209, -300.9751, -299.31, -560.091, 420.6743, 5.8656, -559.3981, 204.3195, -564.9936, -301.3546, 200.9901, -303.7655, 204.1654, 64.0198, -305.599, -559.3673, -128.1384, -307.2912, 202.2739, 420.4788, 206.3063, 420.6493, -306.7615, -24.573]

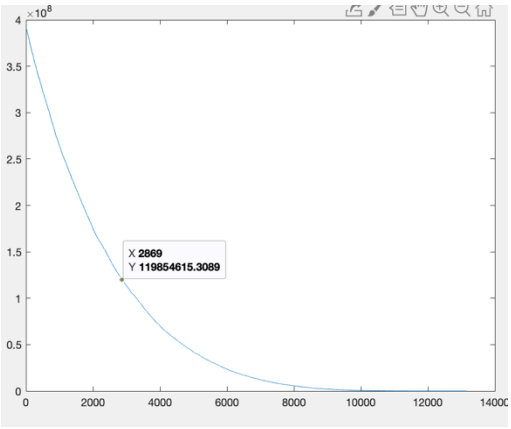


Función 3

population_len: 0
sigma: 0.0090
fenotype_len: 30
max_iterations: 100000
percentage: 0
domain: "d3"
comparator: "min"
evaluator: "f3"
stopper: "iterations"
mutation: ""
crossover: ""
selection: ""
algorithm: "genetic-simple"
lambda: 0

Resultados:

Best fitness: 19.8502
Mean fitness: 57.4743
Time: 61.4096
Best fenotype:
[0.99477, 1.0025, 0.99564, 0.98804, 0.96903, 0.93192, 0.8547, 0.7179, 0.51909, 0.27159, 0.065076, -0.006892, 0.016311, 0.0024455, 0.004865, 0.022483, -0.0046623, 0.014039, 0.018119, -0.0064954, 0.001468, 0.0012809, 0.010598, -0.006102, 0.01824, 0.017241, 0.011041, -0.005504, 0.0053004, 0.0076534]



En las gráficas de las 3 funciones se aprecia como el número de iteraciones necesario para poder llegar a una solución próxima a la óptima no son relativamente muchas y el fitness decae de forma muy rápida con este algoritmo.

Evolutivo (n+m)

En este algoritmo tendremos n padres que generan m hijos. Para cada padre generaremos la cantidad de hijos indicada por mutación del padre. Para cada padre almacenaremos el mejor hijo que se genere y finalmente sustituiremos al padre por dicho hijo.
Para saber el mejor fenotipo de la población deberemos evaluar toda la población a cada ciclo del algoritmo.

```
function [best_fenotype, best_fitness] = Genetic (evaluator, stopper, comparator, newPopulation, sigma,
lambda)
    population = newPopulation();
    fenotype_length = size(population,2);
    getBest = @(population) bestInPopulation (evaluator, comparator, population);
    [best_fenotype, best_fitness] = getBest(population);
    while ~stopper()
        for i = 1:size(population,1)
            fenotype_padre = population(i,:);
            fitness_padre = evaluator(fenotype_padre);
            best_hijo = fenotype_padre;
            for j = 1:lambda
                fenotype_hijo = fenotype_padre + sigma*randn(1, fenotype_length);
                fitness_hijo = evaluator(fenotype_hijo);
                if comparator(fitness_hijo, fitness_padre)
                    fitness_padre = fitness_hijo;
                    best_hijo = fenotype_hijo;
                end
            end
            population(i,:) = best_hijo;
        end
        [best_fenotype_pop, best_fitness_pop] = getBest(population);
        if comparator(best_fitness_pop, best_fitness)
            best_fenotype = best_fenotype_pop;
            best_fitness = [best_fitness best_fitness_pop];
        end
    end
end
```

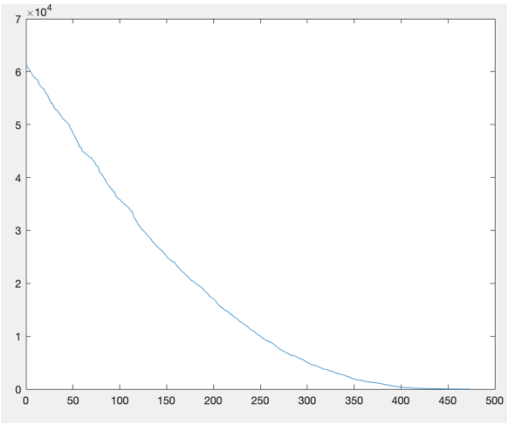
Función 1

population_len: 100
sigma: 0.5000
lambda: 5
fenotype_len: 30
max_iterations: 20000
domain: "d1"
comparator: "min"
evaluator: "f1"
stopper: "iterations"
algorithm: "genetic"
percentage: 0
mutation: ""
crossover: ""

selection: ""

Resultados:

Best fitness: 1.038
Mean fitness: 1.8595
Time: 757.5122
Best fenotype:
[0.44325, -0.14328, 0.031304, 0.25895, -0.018969, 0.20059, -0.021948, 0.025066, -0.070968, 0.037584, 0.25074, -0.28986, 0.018204, -0.13059, 0.17109, 0.57348, -0.04804, 0.019944, 0.17505, -0.019049, -0.14253, -0.12521, 0.14884, 0.2441, -0.040265, -0.059, 0.12635, -0.067281, -0.072627, -0.0050729]

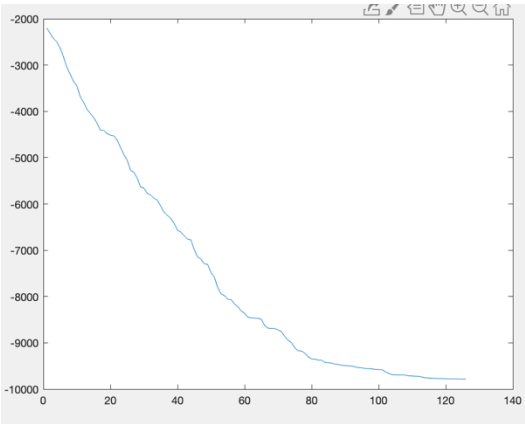


Función 2

population_len: 100
sigma: 0.5000
lambda: 5
fenotype_len: 30
max_iterations: 20000
domain: "d2"
comparator: "min"
evaluator: "f2"
stopper: "iterations"
algorithm: "genetic"
percentage: 0
mutation: ""
crossover: ""
selection: ""

Resultados:

Best fitness: -11689.2311
Mean fitness: -10441.4946
Time: 1036.5367
Best fenotype:
[-558.1312, -299.4165, -298.9196, -124.8529, 202.0345, -304.8899, 421.9652, -304.4277, -306.8699, -301.2579, 422.6503, -304.7943, -300.3963, 420.6102, 424.8559, -560.6275, -557.9801, 203.3426, -559.9566, -302.3158, -557.6032, -302.9884, -561.3357, 206.7744, 204.2220, -560.0982, -301.8514, 201.6522, 420.2780, -29.0523]

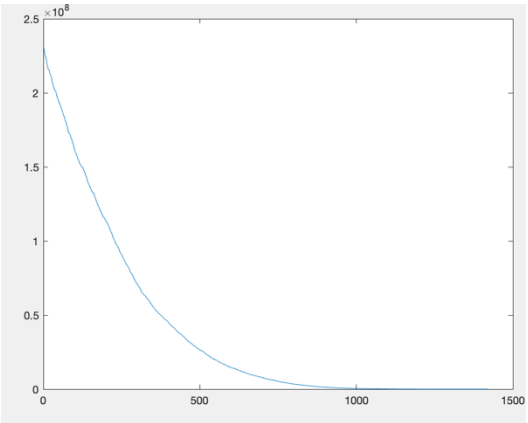


Función 3

population_len: 100
sigma: 0.5000
lambda: 5
fenotype_len: 30
max_iterations: 20000
domain: "d3"
comparator: "min"
evaluator: "f3"
stopper: "iterations"
algorithm: "genetic"
percentage: 0
mutation: ""
crossover: ""
selection: ""

Resultados:

Best fitness: 22.5142
Mean fitness: 23.1196
Time: 1234.2115
Best fenotype:
[0.9561, 1.0100, 1.0094, 1.0163, 0.9652, 0.9258, 0.9305, 0.8348, 0.7210, 0.6080, 0.3369, 0.1212, 0.0182, 0.0725, 0.0222, -0.0454, -0.0140, 0.0002, 0.0185, -0.0087, -0.0000, 0.0172, 0.0015, 0.0695, 0.0575, -0.0369, 0.0445, 0.0008, -0.0165, 0.0216]



En las gráficas de este algoritmo vemos como ocurre lo mismo que en el algoritmo anterior, aunque se aprecia como el número de iteraciones necesarias para llegar al valor óptimo es bastante menor que en el algoritmo anterior y las soluciones dadas también son mejor que en el anterior.

Harmónico

Tendremos una matriz de armonía a partir de la que en cada iteración del algoritmo generaremos una nueva que solo añadiremos cuando esta sea mejor que la peor que había anteriormente.

Para generar la nueva armonía tomaremos con una probabilidad alta un elemento de alguna de las armonías de la matriz. En caso contrario generaremos una armonía completamente nueva (HMCR). Adicionalmente cuando se termina de aplicar HCMR, aplicamos PAR, con una probabilidad del 30% sobre el resultado obtenido en HMCR.

```
function [best_fenotype, best_fitness] = Harmonic (evaluator, comparator, stopper, newGenotype,
newPopulation, mutator)
    population = newPopulation();
    population = sortPopulation(evaluator, population);
    worst_fitness = evaluator(population(end,:));
    getBest = @(population) bestInPopulation (evaluator, comparator, population);
    [best_fenotype, best_fitness] = getBest(population);
    while ~stopper()
        new_fenotype = zeros(1, size(population,2));
        for i = 1:size(population, 2)
            if randi(100, 1) > 20
                row_index = randi(size(population, 1),1);
                new_fenotype(i) = population(row_index, i);
            else
                new_fenotype(i) = newGenotype();
            end
        end
        if randi(100,1) < 30
            new_fenotype = mutator(new_fenotype);
        end
        new_fitness = evaluator(new_fenotype);
        if comparator(new_fitness, worst_fitness)
            population = sortPopulation(evaluator, population);
            population(end,:) = new_fenotype;
            worst_fitness = evaluator(population(end-1,:));
            population = population(randperm(size(population,1)),:);
        end
        [best_fenotype, best_fitness_pop] = getBest(population);
        best_fitness = [best_fitness best_fitness_pop];
    end
end
```

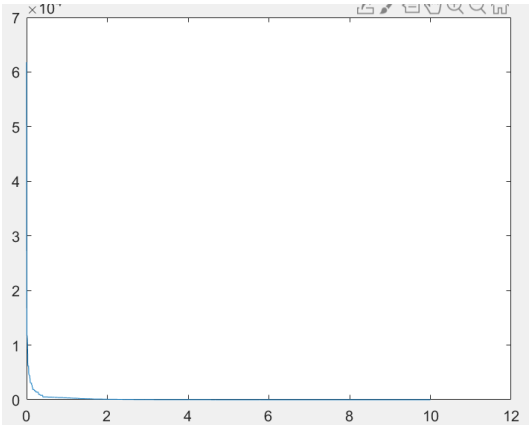
Función 1

population_len = 15
sigma = 0.5
fenotype_len = 30
max_iterations = 1000000
domain = "d1"
comparator = "min"
evaluator = "f1"

stopper = "iterations"
mutation = "multiple"
algorithm = "harmonic"
lambda = 0
percentage = 0
crossover = ""
selection = ""

Resultados:

Best fitness: 1.9203
Mean fitness: 3.5867
Time: 5367.8386
Best fenotype:
[0.2265, 0.0252, 0.2770, 0.3930, 0.0209, -0.4104, -0.1435, 0.0296, -0.1516, 0.2458, 0.2179, 0.1600, 0.1732, 0.1779, 0.0204, -0.1553, -0.0287, 0.3907, 0.0489, -0.1357, -0.6946, 0.1799, 0.0697, -0.0729, 0.1858, 0.0602, -0.6350, -0.0333, -0.0435, 0.2461]



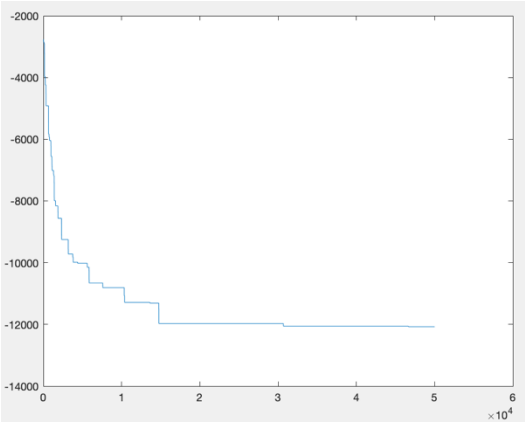
Función 2

population_len: 100
sigma: 0.5000
fenotype_len: 30
max_iterations: 50000
domain: "d2"
comparator: "min"
evaluator: "f2"
stopper: "iterations"
algorithm: "harmonic"
lambda: 0
percentage: 0
mutation: ""
crossover: ""
selection: ""

Resultados:

Best fitness: -12661.8457
Mean fitness: -10512.0907
Time: 332.4372

Best fenotype:
[422.1920, 420.4013, 425.6812, 425.8126, 422.4990, 419.4164, 423.4983 ,422.6571, 421.8814, 429.0519, 422.9469, 418.6026, 416.8365, 421.8598, 414.3248, 423.0877, 418.7131, 418.2425, 411.9589, 423.0504, 423.5328, 419.8906, 418.0196, 419.9327, 416.0102, 417.6549, 424.7221, 421.9550, 420.7995, 430.2426]

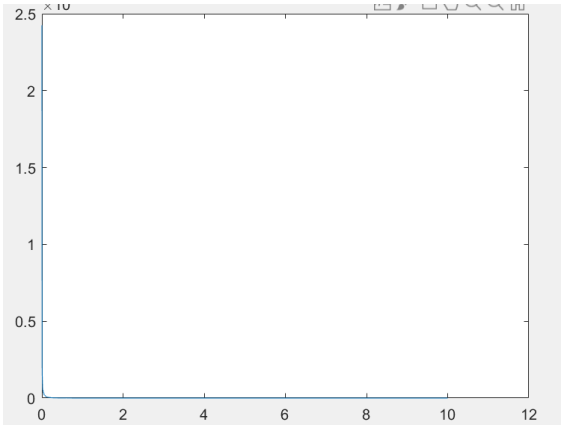


Función 3

population_len: 15
sigma: 0.5
fenotype_len: 30
max_iterations: 1000000
domain: "d3"
comparator: "min"
evaluator: "f3"
stopper: "iterations"
algorithm: "harmonic"
lambda: 0
percentage: 0
mutation: ""
crossover: ""
selection: ""

Resultados:

Best fitness: 339.8433
Mean fitness: 468.4421
Time: 6934.6746
Best fenotype:
[0.0379, -0.0140, 0.2756, 0.0830, -0.2068, -0.2994, 0.1475, -0.3222, 0.7741, 0.9107, 0.9082, 1.0458, 0.8920, 0.9024, 0.9170, 0.9680, 1.0258, 0.9200, 0.9174, 0.8539, 0.8516, 0.8031, 0.8270, 0.7910, 0.3719, 0.0472, -0.9984, 1.9695, 3.8055, 14.5356]



Al utilizar este algoritmo se aprecia como existe bastante error entre el valor real y el valor que el algoritmo ofrece para las fórmulas 1 y 3, en cambio, para la 2 el valor se aproxima muy buena.

Temple simulado

Este algoritmo se basa en explotar la solución actual mediante mutaciones que al principio serán elevadas y poco a poco se irán reduciendo. Adicionalmente existe una probabilidad que también se va reduciendo con el tiempo de tomar una solución nueva que sea peor que la actual.
Utilizando la nomenclatura de los algoritmos evolutivos la población del temple es un único fenotipo.

```
function [best_fenotype, fitness] = Temple (evaluator,comparator, stopper, newFenotype, sigma, lambda)
   best_fenotype = newFenotype();
   best_fitness = evaluator(best_fenotype);
   fitness = best_fitness;
   while ~stopper()
      for i = 1:lambda
         new_fenotype = totalMutation(sigma, best_fenotype);
         new_fitness = evaluator(new_fenotype);
         if comparator(new_fitness, best_fitness)
            best_fenotype = new_fenotype;
            best_fitness = new_fitness;
            fitness = [fitness best_fitness];
         else
            if rand(1,1) < exp(-(abs(best_fitness - new_fitness))/sigma)
               best_fenotype = new_fenotype;
               best_fitness = new_fitness;
               fitness = [fitness best_fitness];
            end
         end
      end
      sigma = sigma*0.95;
   end
end
```

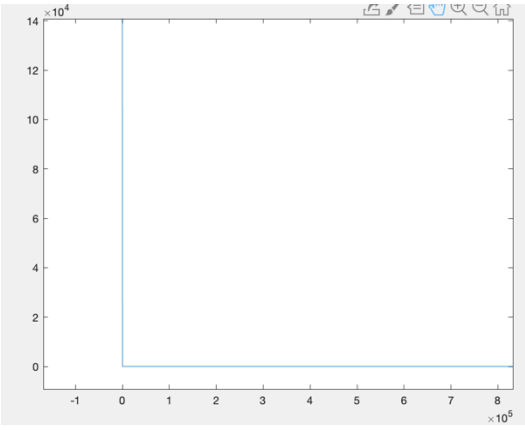
Función 1

sigma: 10
lambda: 200
fenotype_len: 30
max_iterations: 5000
domain: "d1"
comparator: "min"
evaluator: "f1"
stopper: "iterations"
algorithm: "temple"
population_len: 0
percentage: 0
mutation: ""
crossover: ""
selection: ""

Resultados:

Best fitness: 0.0011344
Mean fitness: 0.0020065

Time: 120.3244
Best fenotype:
[0.0040069, 0.0092013, -0.0068205, -0.010699, -0.00030072, 0.0083245, 0.0022638, 0.0020951, -0.0023259, -0.0044461, 0.0019081, -0.0069426, 0.0055672, 0.0055552, -3.8654e-05, -0.0001298, -0.015935, -0.0042696, -0.0067076, -0.0028467, 0.0020405, -0.0028652, 0.0017797, -0.0051757, -0.0047024, 0.011859, -0.0035173, 0.0078277, -0.0067247, -0.0023819]

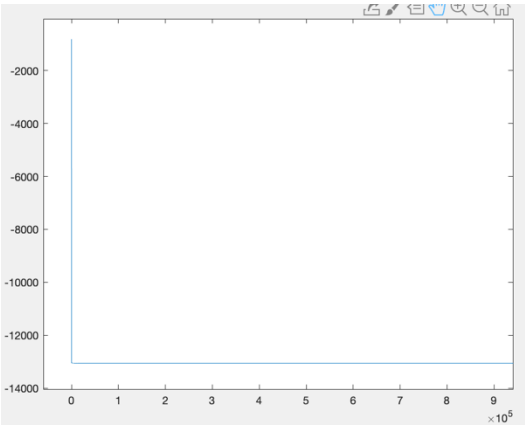


Función 2

sigma: 100
lambda: 200
fenotype_len: 30
max_iterations: 5000
domain: "d2"
comparator: "min"
evaluator: "f2"
stopper: "iterations"
algorithm: "temple"
population_len: 0
percentage: 0
mutation: ""
crossover: ""
selection: ""

Resultados:

Best fitness: -11416.3623
Mean fitness: -10913.2774
Time: 147.9809
Best fenotype:
[717.131, -559.2238, 420.9496, -894.7924, -1309.1309, -894.7129, 717.1042, 203.9108, -894.6083, 65.5375, -894.7091, 65.5645, -894.7301, -302.4004, -25.9164, 203.721, -1309.2418, 1545.9769, 203.8831, 65.5609, 203.8302, 717.1673, -124.8019, 717.0552, -302.5168, -302.401, -559.161, -302.6018, 65.6091, 420.8279]

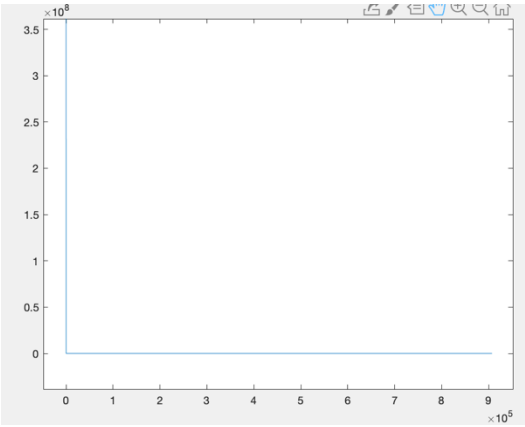


Función 3

sigma: 10
lambda: 200
fenotype_len: 30
max_iterations: 5000
domain: "d3"
comparator: "min"
evaluator: "f3"
stopper: "iterations"
algorithm: "temple"
population_len: 0
percentage: 0
mutation: ""
crossover: ""
selection: ""

Resultados:

Best fitness: 20.4452
Mean fitness: 660.5903
Time: 129.8356
Best fenotype:
[0.99486, 0.98993, 0.98024, 0.96099, 0.92372, 0.85381, 0.73013, 0.53516, 0.29062, 0.09193, 0.017587, 0.010158, 0.0098418, 0.010594, 0.010524, 0.010176, 0.010305, 0.01064, 0.010088, 0.010119, 0.010542, 0.010352, 0.010358, 0.010211, 0.010392, 0.01008, 0.010119, 0.010094, 0.0099563, 0.00054545]



En las gráficas de este algoritmo se aprecia como la disminución del fitness, ósea el tiempo que tarda en encontrar el fitness óptimo es muy poca, ocurre en muy pocas iteraciones y encima el valor que ofrece es relativamente bueno para lo que se ha tardado en programar este algoritmo.

Evolutivo con búsqueda local

Esta es una modificación sobre el algoritmo evolutivo (1, 1). Del mismo modo que en él solo habrá un fenotipo en la población. La diferencia entre ambos reside en que el nuevo fenotipo es resultado de la aceptación de mutaciones por cada uno de sus genotipos si mejoraban la solución local y no exclusivamente de mutaciones sobre todo el fenotipo.

Se incluye pues nos ha proporcionado buenos resultados para las funciones en las que la exploración local es favorable.

```
function [best_fenotype, best_fitness] = GeneticSimpleLocal (evaluator, comparator, stopper,
newFenotype, sigma)
    best_fenotype = newFenotype();
    best_fitness = [evaluator(best_fenotype)];
    while ~stopper()
        for i = 1:length(best_fenotype)
            fenotype = best_fenotype;
            fenotype(i) = elementMutation(sigma, fenotype(i));
            fitness = evaluator(fenotype);
            if comparator(fitness, best_fitness)
                best_fitness = [best_fitness fitness];
                best_fenotype = fenotype;
            end
        end
    end
end
```

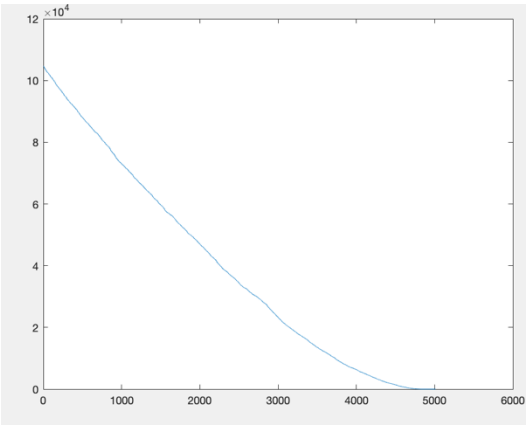
Función 1

population_len: 0
sigma: 0.4000
fenotype_len: 30
max_iterations: 100000
percentage: 0
domain: "d1"
comparator: "min"
evaluator: "f1"
stopper: "iterations"
mutation: ""
crossover: ""
selection: ""

Resultados:

algorithm: "genetic-simple-local"
lambda: 0
Best fitness: 5.8539e-10
Mean fitness: 1.4364e-09
Time: 811.1708
Best fenotype:
[1.649e-06, 6.2205e-07, -3.6025e-06, 1.2402e-06, -3.0185e-06, 1.3406e-05, 2.5593e-06, -3.533e-06, 2.4262e-06, -2.9201e-06, 4.7075e-07, 9.6967e-06, 2.8543e-06,
-1.0593e-06, -8.17e-06, 4.1032e-06, -4.3371e-06, -2.5425e-06, -4.7938e-06,

-3.5188e-07, 4.6525e-07, 1.9217e-06, 4.2056e-07, -5.7565e-06, 2.8907e-06, -8.9249e-07, 2.6966e-06, -1.4416e-06, -5.3273e-06, 5.0906e-06]

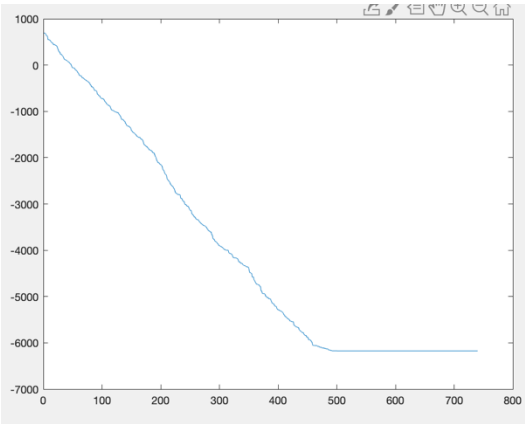


Función 2

population_len: 0
sigma: 4
fenotype_len: 30
max_iterations: 100000
percentage: 0
domain: "d2"
comparator: "min"
evaluator: "f2"
stopper: "iterations"
mutation: ""
crossover: ""
selection: ""

Resultados:

algorithm: "genetic-simple-local"
lambda: 0
Best fitness: -9174.0814
Mean fitness: -7720.8389
Time: 409.3978
Best fenotype:
[-302.525, -559.1486, 420.9688, 203.8144, 203.8141, -559.1486, 203.8143, 203.8143, -124.8294, -302.5249, 420.9687, 203.8143, -302.525, -124.8293, -302.5249, -302.5248, 420.9688, 420.9687, 420.9687, 203.8143, -559.1487, -124.8294, -124.8294, 420.9688, -302.5248, 420.9687, 420.9688, 203.8142, 420.9686, -25.8775]

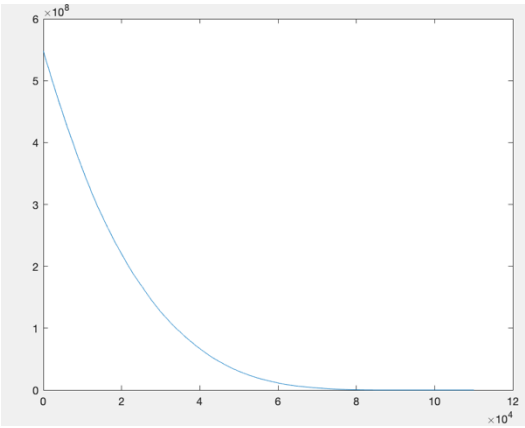


Función 3

population_len: 0
sigma: 0.0090
fenotype_len: 30
max_iterations: 100000
percentage: 0
domain: "d3"
comparator: "min"
evaluator: "f3"
stopper: "iterations"
mutation: ""
crossover: ""
selection: ""

Resultados:

algorithm: "genetic-simple-local"
lambda: 0
Best fitness: 0.072451
Mean fitness: 0.013906
Time: 9371.2604
Best fenotype:
[1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0001, 1.0001, 1.0002, 1.0004, 1.0008, 1.0015, 1.0031, 1.0062, 1.0123, 1.0247, 1.0501, 1.1028, 1.2162]



Este algoritmo ofrece muy buenos resultados especialmente para la función 1 y la 3, dada la estructura de la función, aunque tarda bastante más en llegar al fitness óptimo.

Genético

En este algoritmo se deja de utilizar la mutación como principal fuente de exploración siendo ahora sustituida por el cruce entre fenotipos. Los fenotipos de la población se cruzarán creando hijos, adicionalmente los padres podrán mutar con una pequeña probabilidad. Finalmente, padres e hijos competirán en base a una función de selección, que en nuestro caso será o ranking o torneo para dar lugar a la nueva población.

```
function [best_fenotype, best_fitness] = FullGenetic (evaluator, comparator, stopper, mutator, crosser,
selector, newPopulation)
    population = newPopulation();
    getBest = @(population) bestInPopulation (evaluator, comparator, population);
    [best_fenotype, best_fitness] = getBest(population);
    while ~stopper()
        childs = zeros(size(population,1)*2,size(population,2));
        for i = 1:(size(population,1))
            if mod(i,2)==0
                index1 = randi(size(population,1), 1);
                index2 = randi(size(population,1), 1);
                [child1, child2] = crosser(population(index1,:), population(index2,:));
                childs(i,:) = child1;
                childs(i+1,:) = child2;
            end
        end
        mutations = [];
        for i = 1:size(population,1)
            if randi(100, 1) <= 5
                mutations = [mutations; mutator(population(i,:))];
            end
        end
        population = selector ([population; childs; mutations]);
        [best_fenotype, new_fitness] = getBest(population);
        best_fitness = [best_fitness new_fitness];
    end
end
```

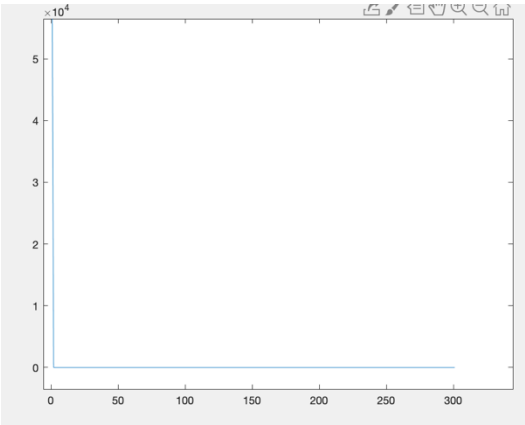
Función 1 – Selección Ranking

population_len: 200
sigma: 0.1000
fenotype_len: 30
max_iterations: 300
percentage: 0.3000
domain: "d1"
comparator: "min"
evaluator: "f1"
stopper: "iterations"
mutation: "multiple"
crossover: "double"
selection: "ranking"

algorithm: "genetic-full"
lambda: 0

Resultados:

Best fitness: 0
Mean fitness: 0
Time: 21.0094
Best fenotype:
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

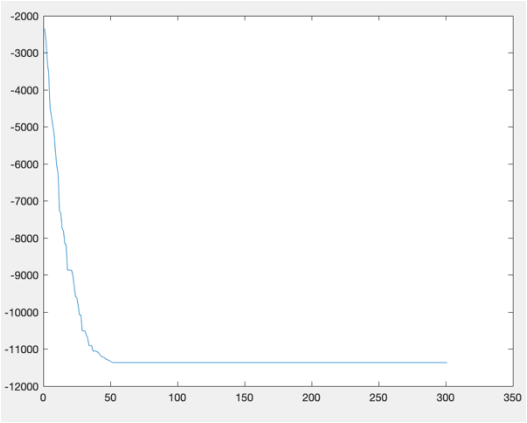


Función 2 - Selección Ranking

population_len: 200
sigma: 0.0100
fenotype_len: 30
max_iterations: 300
percentage: 0.3000
domain: "d2"
comparator: "min"
evaluator: "f2"
stopper: "iterations"
mutation: "multiple"
crossover: "double"
selection: "ranking"
algorithm: "genetic-full"
lambda: 0

Resultados:

Best fitness: -11625.4307
Mean fitness: -11195.3086
Time: 24.7252
Best fenotype:
[421.5812, 414.5187, 449.9187, 419.1525, 424.513, 415.4136, 401.7014, 412.4287, 433.2081, 427.2333, 409.484, 414.519, -282.9614, 423.1587, 415.9346, 438.4508, 418.6093, 410.2908, 409.0206, 410.0536, 414.9278, 405.4059, 445.8699, 447.9868, 419.4712, 421.425, 405.9713, 427.5829, 400.084, -326.1283]

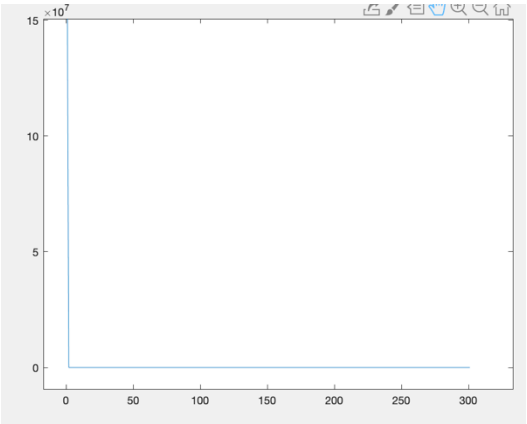


Función 3 - Selección Ranking

population_len: 200
sigma: 0.0100
fenotype_len: 30
max_iterations: 300
percentage: 0.3000
domain: "d3"
comparator: "min"
evaluator: "f3"
stopper: "iterations"
mutation: "multiple"
crossover: "double"
selection: "ranking"
algorithm: "genetic-full"
lambda: 0

Resultados:

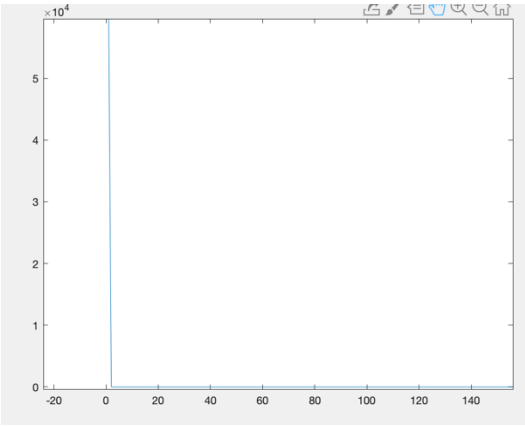
Best fitness: 28.6811
Mean fitness: 28.715
Time: 20.7367
Best fenotype:
[0.026238, 0.0027715, 0.011969, 0.0091962, 0.0087458, 0.0067806, 0.012023, 0.0134, 0.0072406, 0.0091149, 0.010131, 0.0096703, 0.0095833, 0.0093841, 0.010492, 0.0146, 0.011024, 0.0066364, 0.010862, 0.0096137, 0.014762, 0.0090187, 0.0095691, 0.0088862, 0.010466, 0.0096235, 0.01235, 0.010562, 0.010116, 0]



Este algoritmo con estas características ofrece muy buenos resultados en muy pocas iteraciones, aunque tampoco esta muy lejos del genetico n + m, pero si que require de más tiempo para programar y luego los resultados son parecidos aunque muy buenos.

Función 1 - Selección Torneo

population_len: 200
sigma: 0.1000
fenotype_len: 30
max_iterations: 300
percentage: 0.3000
domain: "d1"
comparator: "min"
evaluator: "f1"
stopper: "iterations"
mutation: "multiple"
crossover: "double"
selection: "tournament"
algorithm: "genetic-full"
lambda: 0
Best fitness: 0
Mean fitness: 0
Time: 457.5112
Best fenotype:
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0]



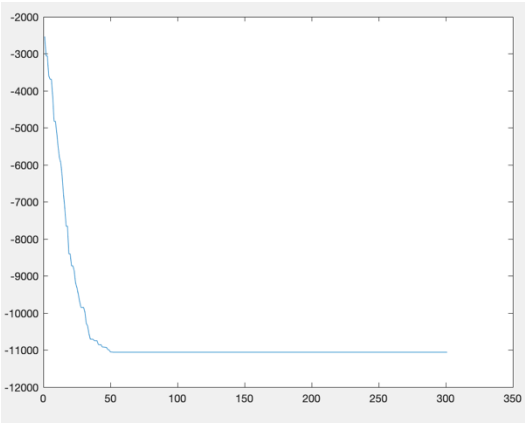
Función 2 - Selección Torneo

population_len: 200
sigma: 0.0100
fenotype_len: 30
max_iterations: 300
percentage: 0.3000
domain: "d2"
comparator: "min"

evaluator: "f2"
stopper: "iterations"
mutation: "multiple"
crossover: "double"
selection: "tournament"
algorithm: "genetic-full"
lambda: 0

Resultados:

Best fitness: -11778.6246
Mean fitness: -11055.6666
Time: 560.9411
Best fenotype:
[421.0487, 417.5194, 435.5103, 426.4395, 426.0726, 396.5675, 425.8914, 436.8135, 426.537, 418.7481, 434.7861, 449.2662, 422.5856, 400.2978, 426.7905, 432.7142, 406.042, -294.9587, 416.7914, 410.7734, 420.2406, -288.8643, 413.3323, 419.9058, 419.334, 431.764, 414.0634, 422.6959, 406.4652, 446.907]



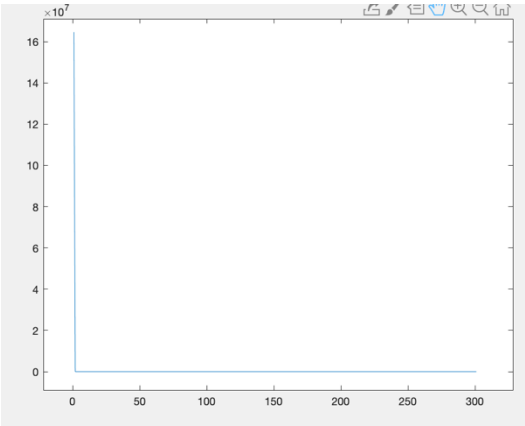
Función 3 - Selección Torneo

population_len: 200
sigma: 0.0100
fenotype_len: 30
max_iterations: 300
percentage: 0.3000
domain: "d3"
comparator: "min"
evaluator: "f3"
stopper: "iterations"
mutation: "multiple"
crossover: "double"
selection: "tournament"
algorithm: "genetic-full"
lambda: 0

Resultados:

Best fitness: 28.6789
Mean fitness: 28.7163

Time: 459.9416
Best fenotype:
[0.044872, 0.0094314, 0.010584, 0.0093229, 0.0075577, 0.0057395, 0.0094895, 0.010701, 0.0062252, 0.00041029, 0.0038849, 0.010182, 0.010212, 0.0019599, 0.0065727, 0.0091327, 0.011667, 0.010031, 0.008805, 0.0057274, 0.0074588, 0.0073731, 0.005808, 0, 0.0077768, 0.0020045, 0.0037879, 0.010403, 0.0092765, 0]



Este algoritmo con estas configuraciones ofrece también muy buenos resultados, y el número de iteraciones que tarda en conseguirla son bajas, aunque al igual que para el anterior require tiempo extra para programarlon aunque los resultados y el número de iteraciones en el que se llega al mínimo merece la pena.