METAHEURÍSTICAS

Informe Práctica 2

Algoritmos implementados

- Algoritmo evolutivo
- Algoritmo diferencial

Grupo 5

Pablo Morillas Plaza - 26526616A
Salvador Perfectti Martin - 75943965N

Indice

Enlace a Drive con todos los logs

Detalles del problema

Se nos otorgan una serie de funciones matemáticas con el objetivo de **poner a prueba la eficacia y eficiencia de los algoritmos implementados**. El objetivo en este caso es el **valor más cercano al óptimo global** que devuelva la función *f(x)* en un dominio en concreto.

- Entendemos como óptimo global al menor valor posible dentro de un intervalo [a , b]

Las funciones empleadas han sido extraídas de la siguiente librería de funciones:

https://www.sfu.ca/~ssurjano/optimization.html

En esta ocasión, hemos implementado dos algoritmos genéticos, uno basado en evolución diferencial y otro evolutivo estándar.
Vamos a detallar los **detalles y restricciones** del problema:

Los algoritmos usarán un vector de **d** dimensiones (en este caso 10) donde cada x_i representa un valor entre el intervalo a evaluar:
$\mathbf{(x_1, x_2, \dots, x_n), x_n \in [a, b]}$ (donde a y b representan el intervalo entre los que están acotados)

Para evaluar cada solución, usaremos el resultado que nos devuelve dicha función matemática y calcularemos la diferencia entre el valor a buscar, el óptimo y el valor de f(x):
$\mathbf{Valor = |\Omega - f(x_1, x_2, \dots, x_n)|}$ (donde Ω representa el óptimo global de dicha función)

Consideraciones

Cada algoritmo implementado cuenta con una serie de restricciones a tener en cuenta

- **Algoritmo evolutivo**: se han implementado dos operadores de cruce distintos; cruce media y cruce BLX-Alfa.
- **Algoritmo diferencial**: en este caso solo hay un operador de cruce, se aplica para cada alelo:

$$\mathbf{Individuo_{nuevo} = Individuo_{padre} + F^*(Individuo_{aleatorio} + Individuo_{aleatorio}) \ \ si\ aleatorio < 0,5}$$
$$\mathbf{Individuo_{nuevo} = Individuo_{objetivo}\ en\ otro\ caso}$$

Operadores comunes

Parametrización

Dentro del programa hay un **archivo de configuración** *config.txt* en el que se parametrizan los parámetros usados para los algoritmos, he aquí una lista de estos con las estructura de datos usada para guardarlos:

- **ArrayList<String>** archivos: Aquí se guardan los archivos con la información de las funciones matemáticas, además indica que objeto de tipo función crear. Los archivos de las funciones guardan el intervalo y el óptimo global.
- **ArrayList<String>** algoritmos: Aquí se guardan los algoritmos a usar, en nuestro caso BLK o MA.
- **ArrayList<long>** semillas: Array con semillas para la generación de los aleatorios. Las semillas usadas son {75943965, 76943595, 95943765, 95953764, 45935769}
- **Integer** dimensión: Este valor representa el tamaño del vector de la solución.
- **Integer** población: Este valor guarda el número de individuos por población.
- **Integer** evaluaciones: Este valor guarda el número de evaluaciones máximas por ejecución.
- **Double** probabilidad_cruce: Aquí se guarda la probabilidad que se usará para cruzar los individuos.
- **Double** probabilidad_mutación: Ídem para la mutación de un individuo.
- **Integer** k: Este valor guarda el número de participantes por torneo.
- **Integer** k_reemplazo: Este valor representa el número de participantes para el torneo usado para conservar el élite.
- **Double** alfa: Valor alfa para el cruce BLX-Alfa.

Métodos comunes

- **void ejecutar()**: ejecuta el algoritmo entero.
- **void evaluar()**: halla al mejor individuo de la población.
- **Individuo hallar_ganador(Individuo[] participantes)**: ordena los participantes y elige al que menor fitness tenga.
- **void ordenar()**: ordena de menor a mayor un array de individuos usando un *quicksort*.

Pseudocódigo

Algoritmo evolutivo

```
INICIO algoritmo
        t = 0
        número de evaluaciones = 0
        inicializar población
        evaluar población

        MIENTRAS no se cumpla condición parada HACER
                t++
                Siguiente_Población = torneo(poblacion)
                recombinar Siguiente_Población
                SI probabilidad de mutación
                        mutar Siguiente_Población
                reemplazar Población por Siguiente_Población
        FIN MIENTRAS
FIN algoritmo
```

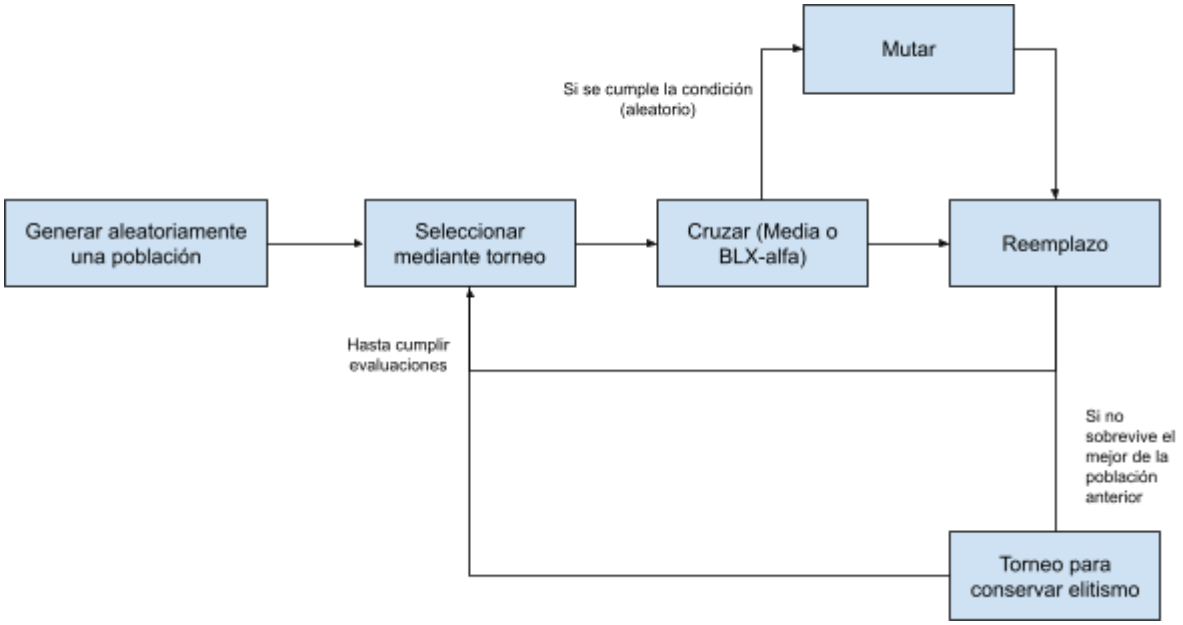
```
INICIO seleccionar
        candidatos[];

        PARA i = 0 HASTA k
                candidatos[i] = población[aleatorio]
        FIN PARA

        ordenar(candidatos) //Quicksort para ordenar de mayor a menor fitness

        DEVOLVER candidatos[0]
FIN seleccionar
```

```
INICIO cruceMedia
        Individuo hijo
        suma

        PARA i = 0 HASTA dimension
        suma = 0
                PARA j = 0 HASTA número_padres
                        suma = suma + padres[j].gen[i]
                FIN PARA
                hijo.gen[i] = suma / número_padres
        FIN PARA

        DEVOLVER hijo
FIN cruceMedia
```

```
INICIO cruceBLX
        Individuo hijo
        mínimo
        máximo

        PARA i = 0 HASTA dimension
                PARA j = 0 HASTA número_padres

                        SI padres[j].gen[i] < mínimo
                                mínimo = padres[j].gen[i]
                        FIN SI

                        SI padres[j].gen[i] > máximo
                                máximo = padres[j].gen[i]
                        FIN SI

                FIN PARA

                rango = máximo - mínimo
                intervalo_mínimo = mínimo - rango * alfa
                intervalo_máximo = máximo - rango * alfa

                hijo.gen[i] =  aleatorio(intervalo_mínimo, intervalo_máximo)

        FIN PARA


        DEVOLVER hijo
FIN cruceMedia
```

Este algoritmo sigue el **esquema básico de un algoritmo evolutivo**, generamos una población aleatoriamente e iniciamos la ejecución del algoritmo. Seleccionamos dos padres mediante un torneo y los cruzamos obteniendo un hijo. Una vez cruzados, elegimos aleatoriamente si mutar o no el individuo generado. Repetimos hasta completar la población y hacemos un proceso de reemplazo, en el cuál se cambia la anterior población por la nueva y si el mejor individuo de la anterior población no se ha mantenido, hacemos un torneo donde seleccionamos a 4 individuos de la nueva población aleatoriamente y sustituimos al peor por el mejor de la anterior población.

El **esquema** de funcionamiento del algoritmo evolutivos adaptado a nuestro problema sería el siguiente:



Selección (torneo) y cruce

Para el torneo, seleccionamos aleatoriamente k individuos y de ellos escogemos el ganador, que será el individuo con **mejor fitness**. Cuando repitamos el proceso 2 veces (número de padres) haremos el cruce, en el que podemos usar dos técnicas

- ### Cruce de la media
  El alelo **i** del nuevo individuo será la media de todos los alelos **i** de los padres.

- ### Cruce BLX-alfa
  Cada alelo **i** del nuevo individuo será un aleatorio creado en el intervalo:

  [min - rango * alfa, max + rango * alfa]

  Donde:
  - **min** es el menor valor de alelo **i** de los padres
  - **max** es el mayor valor de alelo **i** de los padres
  - **rango** es la cantidad de valores entre min y max
  - **alfa** es un parámetro entre [0,1]

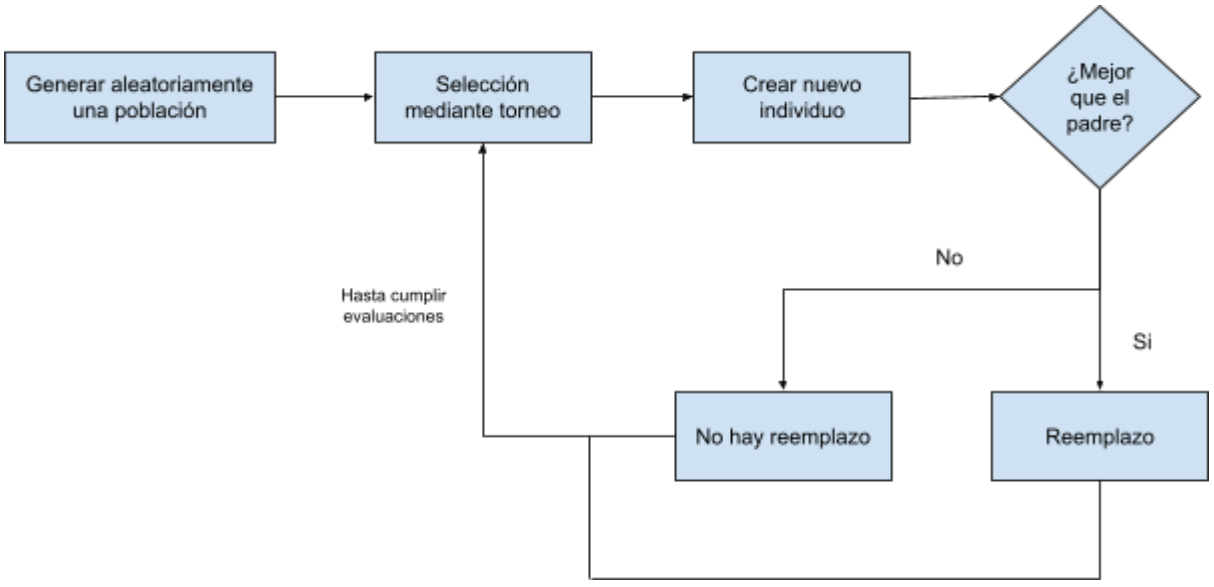
Mutación

Una vez generado el nuevo individuo, éste tiene una probabilidad completamente aleatoria de mutar. En la mutación escogemos un alelo aleatorio y cambiamos su valor por otro completamente aleatorio también.

Reemplazo

Una vez la nueva población, esta se reemplaza por la anterior para seguir el proceso evolutivo. Puede ser que el mejor individuo de la anterior población no sobreviva en la siguiente, por lo que para solucionar esto haremos un torneo escogiendo a 4 individuos aleatoriamente de la nueva población y sustituyendo al que tenga peor fitness por el mejor individuo de la anterior población. Con esto conseguimos mantener el elitismo.

Algoritmo diferencial



```
INICIO algoritmo
        t = 0
        número de evaluaciones = 0
        inicializar población inicial()
        evaluar población()

        MIENTRAS no se cumpla condición de parada:
                PARA i = 1 HASTA tamaño población:
                        hallar inidividuo objetivo()
                        hallar 2 individuos aleatorios()
                        calcular factor mutacion()

                        PARA j = 1 HASTA dimensión:
                                SI aleatorio < p:
                                        nuevo individuo[j] = vector 1 diferencia()
                                SI NO:
                                        nuevo individuo[j] = individuo objetivo[j]
                        FIN PARA

                        SI nuevo individuo es peor que padre:
                                nuevo = padre
                        FIN SI

                        añadir nuevo inidividuo a nueva poblacion
                        número de evaluaciones++
                FIN PARA

                poblacion actual = nueva poblacion
                t++
        FIN MIENTRAS
FIN algoritmo
```

El código de este algoritmo está basado en el visto en las transparencias de clase. Al ser un algoritmo breve y parcialmente sencillo no nos hemos visto necesitados de hacer funciones adicionales.

Al comienzo generamos una población aleatoria, con el objetivo de tener un espacio de búsqueda **amplio**, la principal característica de este algoritmo es que a medida que avanzan las poblaciones generadas, las soluciones van convergiendo todas entre sí, esto se debe a que una de las características principales de la evolución diferencial es que **un nuevo individuo sustituye al padre sí y solo sí es mejor que este último**.

Para la recombinación del nuevo individuo a generar se usa el operador de **vector 1 diferencia**. Básicamente cada gen del nuevo individuo es calculado de esta manera si se cumple cierta probabilidad *p*; en caso contrario se copia el gen del individuo objetivo (hallada aleatoriamente)

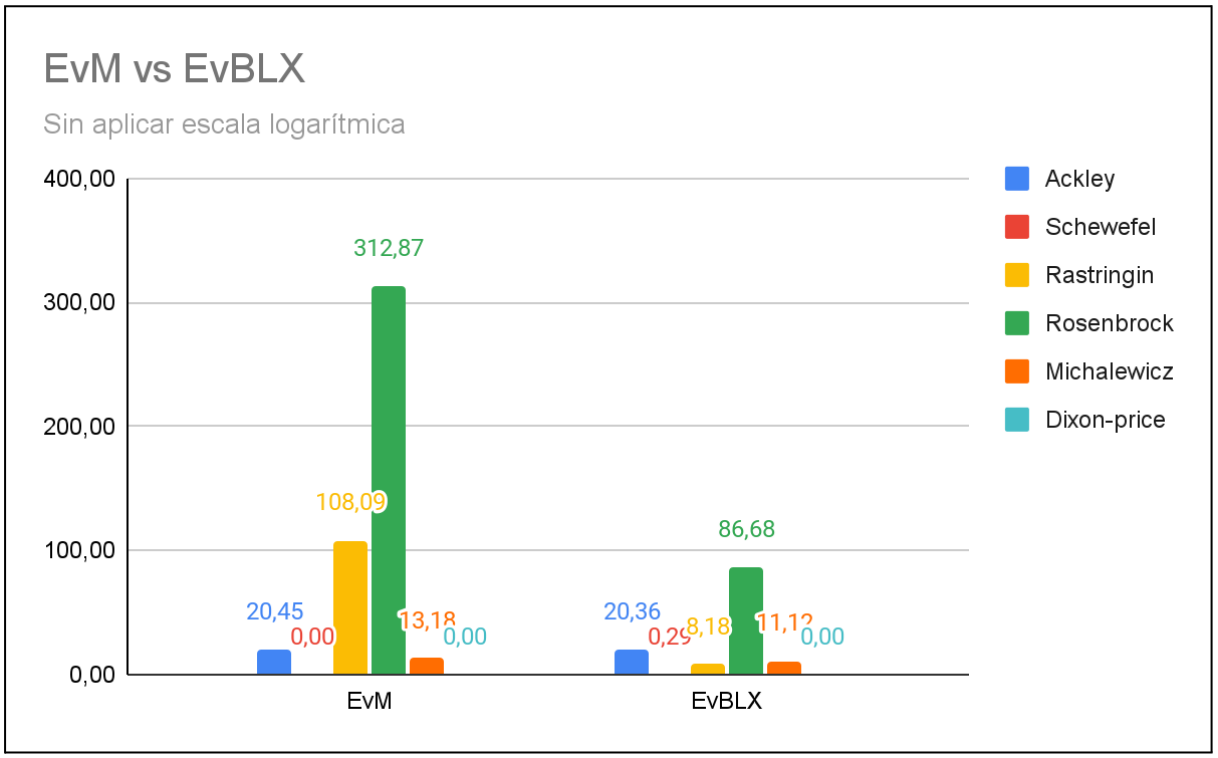
El operador vector 1 diferencia usa al individuo padre y dos individuos seleccionados aleatoria y aplica la siguiente operación:

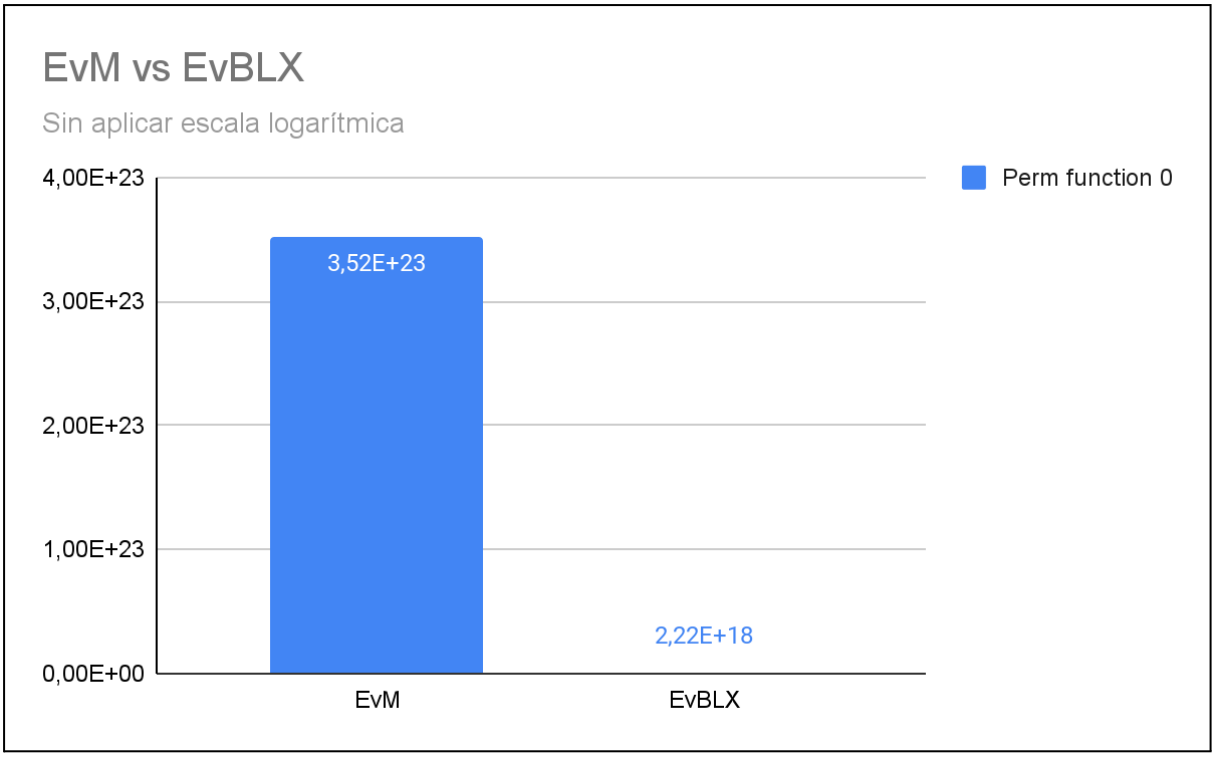
$$nuevo[i] = padre[i] + F(a_1[i] + a_2[i]),\ i \in [1, \text{número de alelos}],\ F \in [0, 1]$$

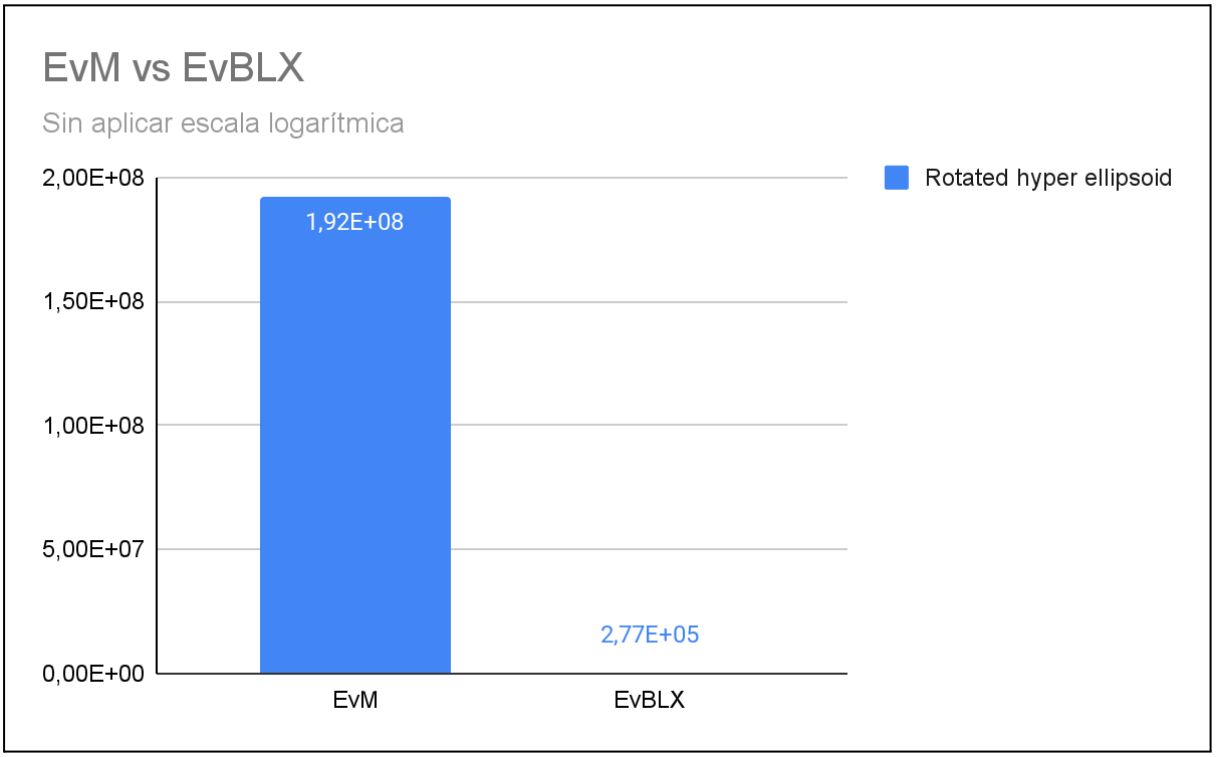
F es un número real generado aleatoriamente en cada generación de nuevo individuo.

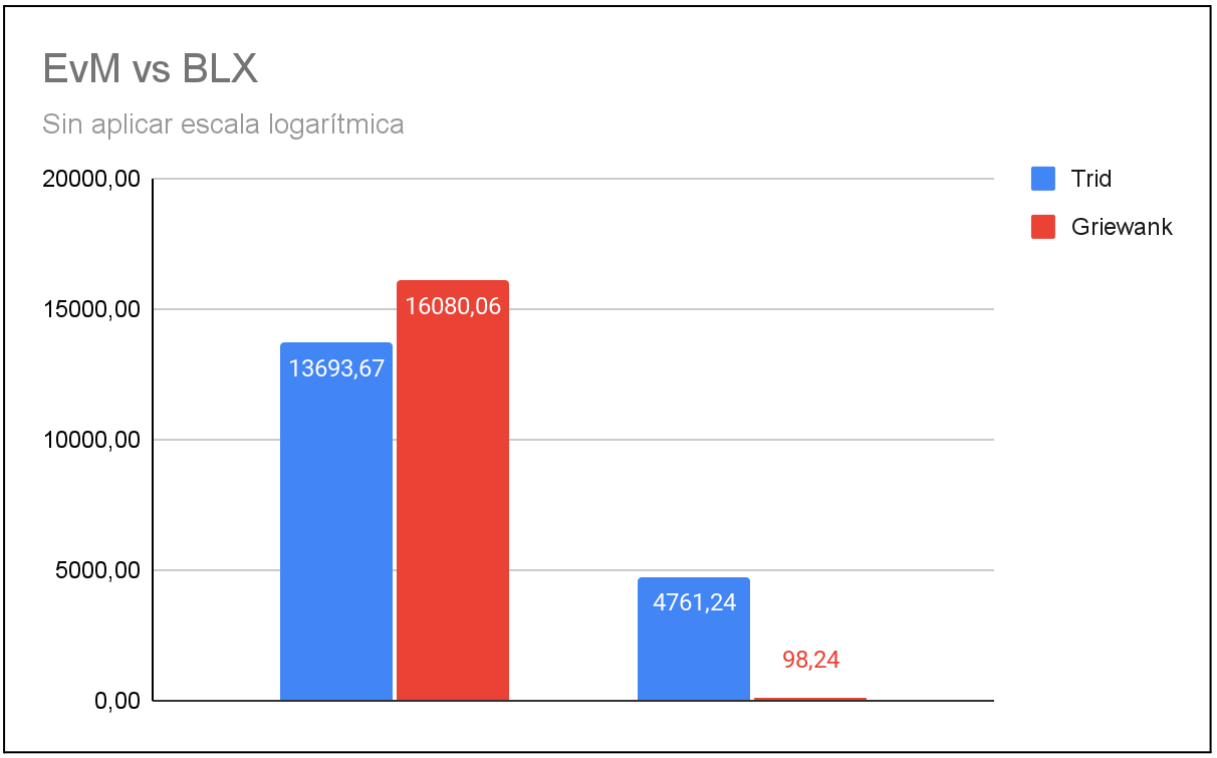
Análisis de resultados

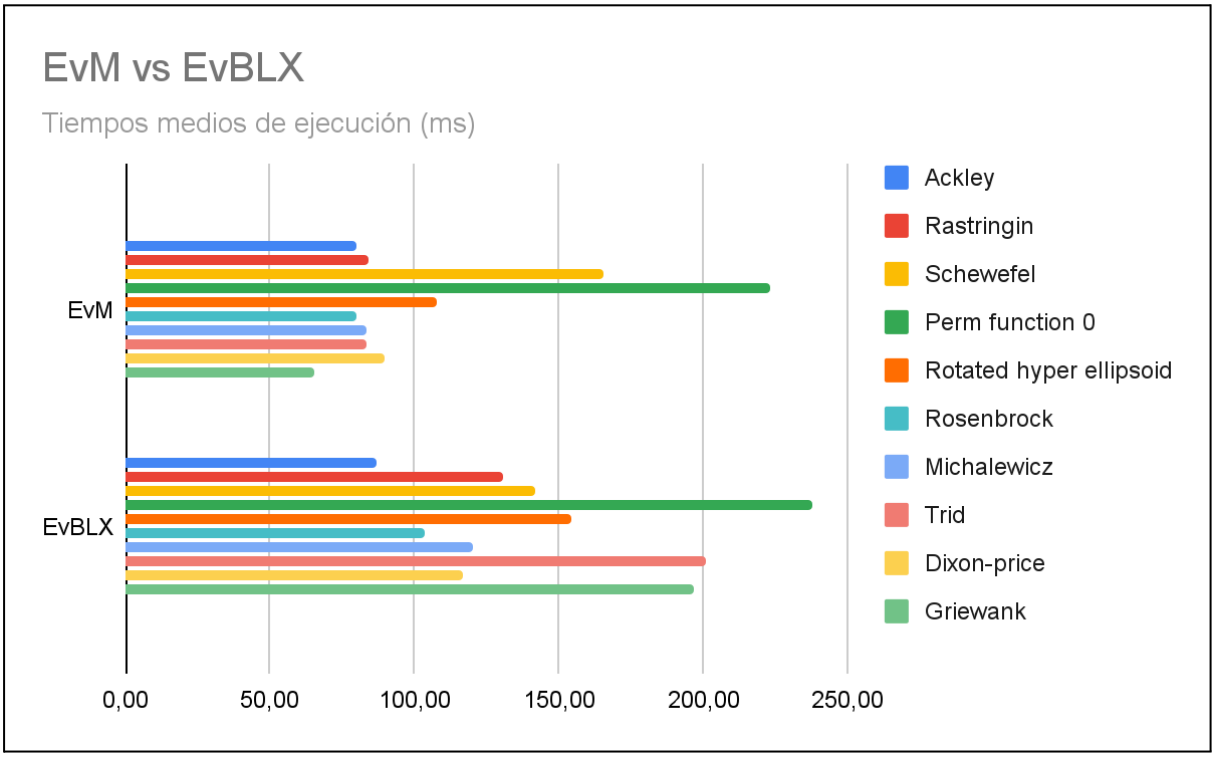
EvM vs EvBLX




EvM vs EvBLX
Sin aplicar escala logarítmica
4,00E+23
3,00E+23
2,00E+23
1,00E+23
0,00E+00
3,52E+23
2,22E+18
EvM
EvBLX
Perm function 0


EvM vs EvBLX
Sin aplicar escala logarítmica
2,00E+08
1,50E+08
1,00E+08
5,00E+07
0,00E+00
1,92E+08
2,77E+05
EvM
EvBLX
Rotated hyper ellipsoid





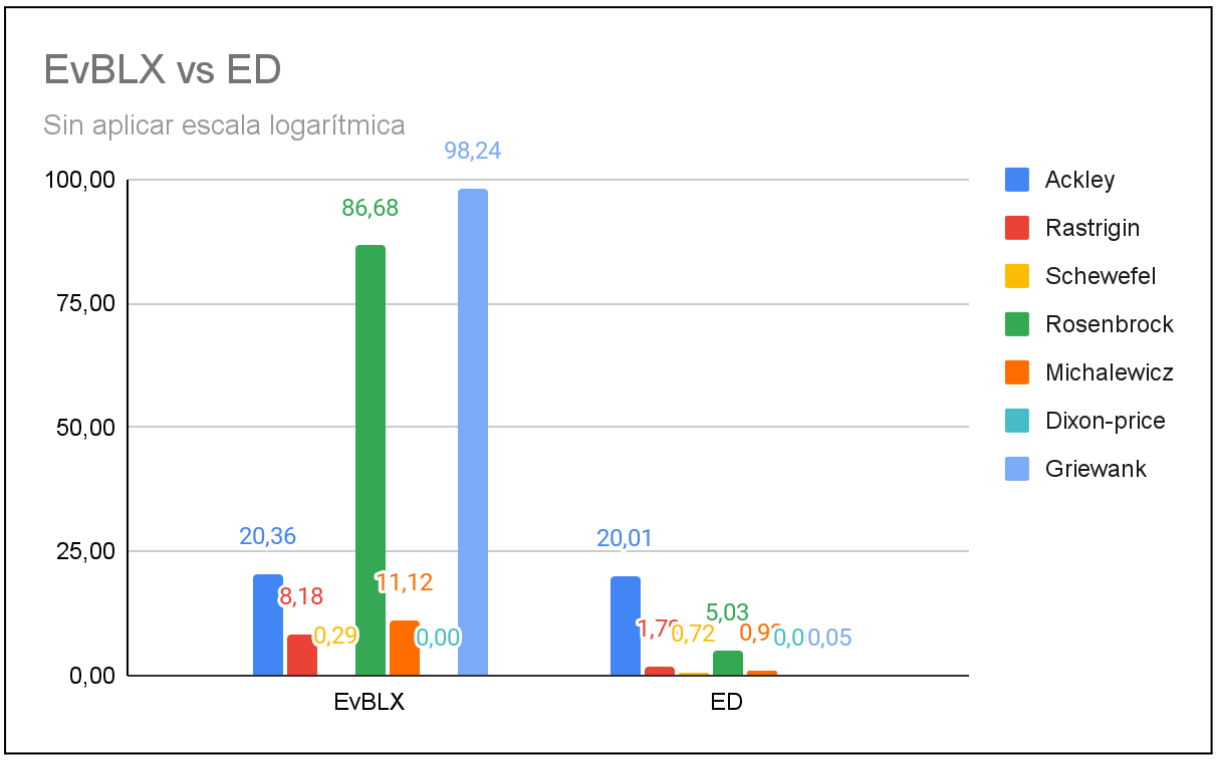
Si vamos comprando los gráficos de los resultados medios devueltos por cada operador, **podemos ver que el cruce BLX-Alfa es siempre superior al operador de la media**. En cambio, lo que ganamos en resultados se pierde en tiempo pues bien el cruce BLX-Alfa tarda más en ejecutarse.

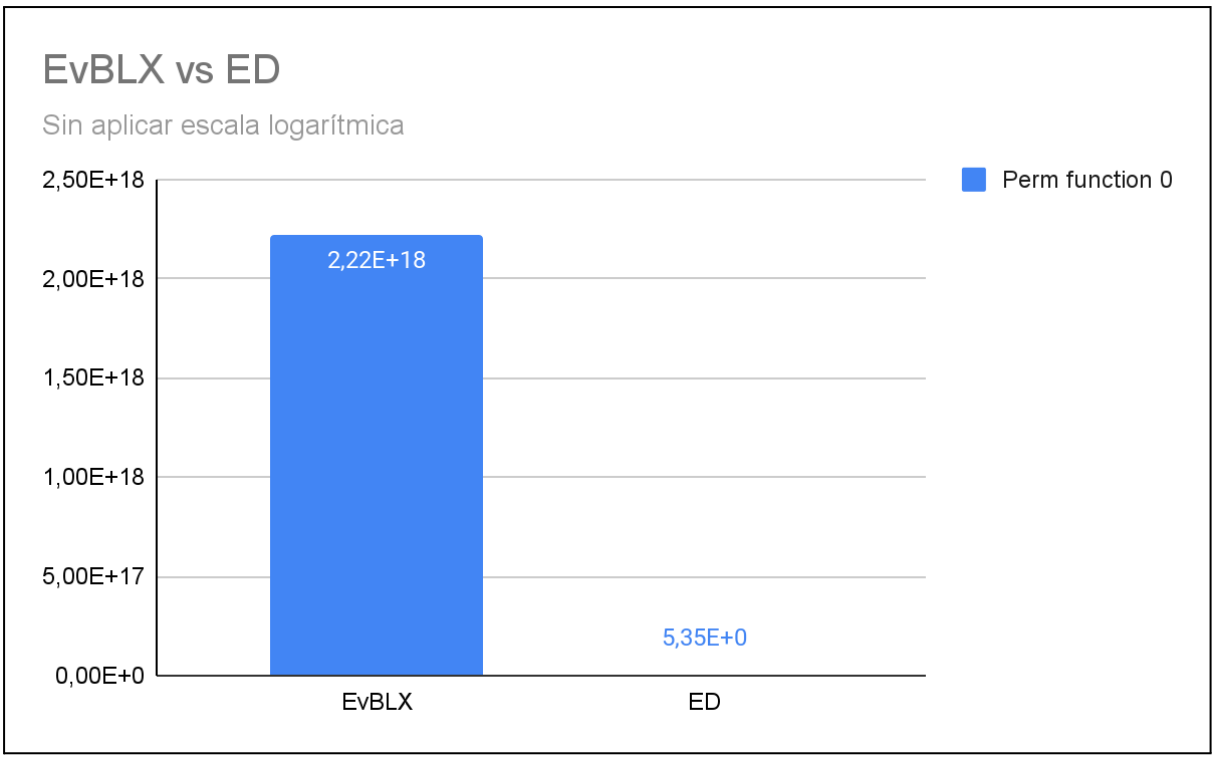
¿Por qué este aumento del tiempo?

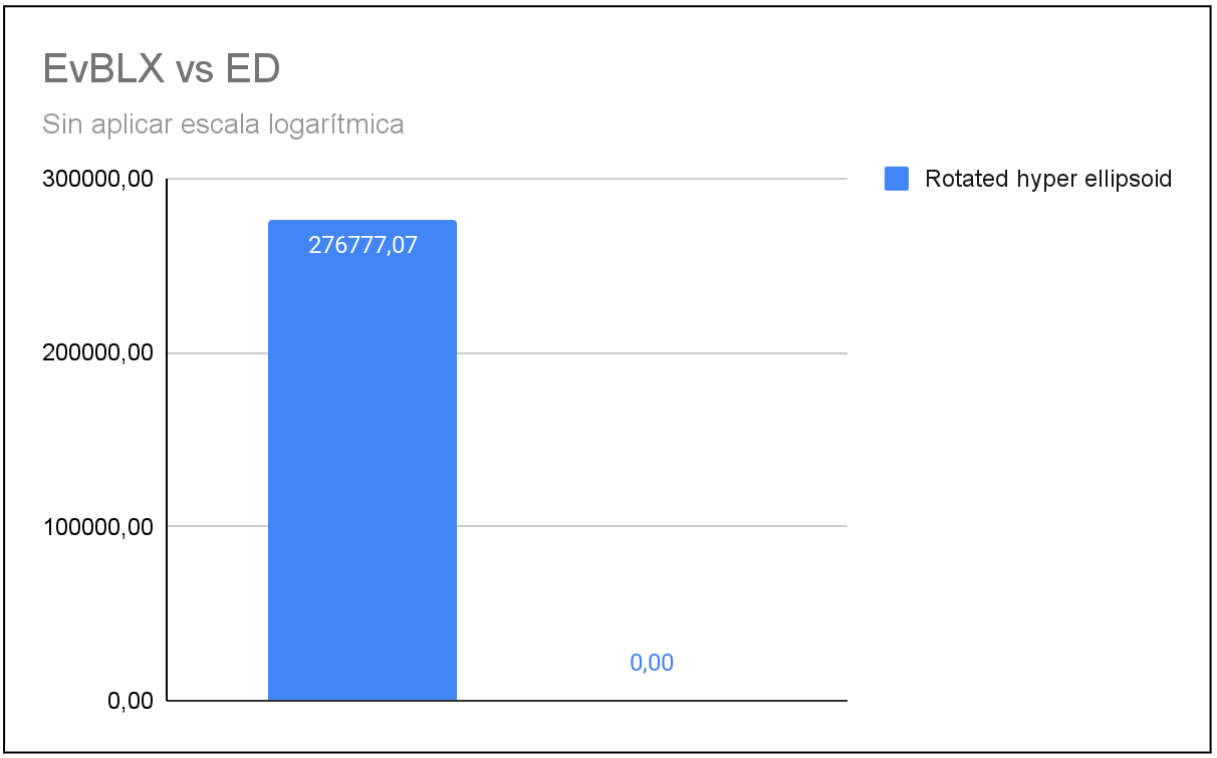
Aunque ambos operadores recorran el mismo bucle, **el cruce BLX-Alfa realiza un número de cálculos superior** ya que hay que hallar el intervalo y además hallar un aleatorio en dicho intervalo.

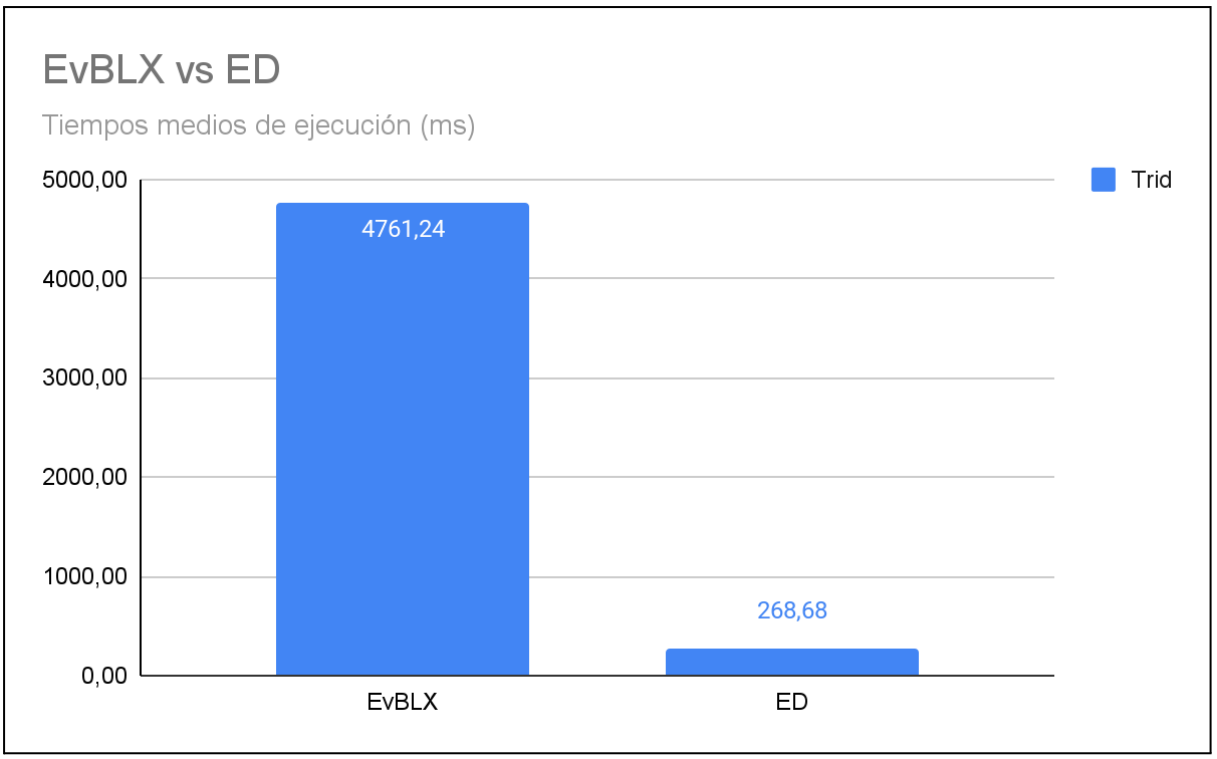
Realmente, estamos hablando de milisegundos extras, además de que las mejoras entre un operador y otro son aplastantes; por lo que podemos ignorar el aumento de tiempo y **podemos decir con seguridad que EvBLX es mejor que EvM**.

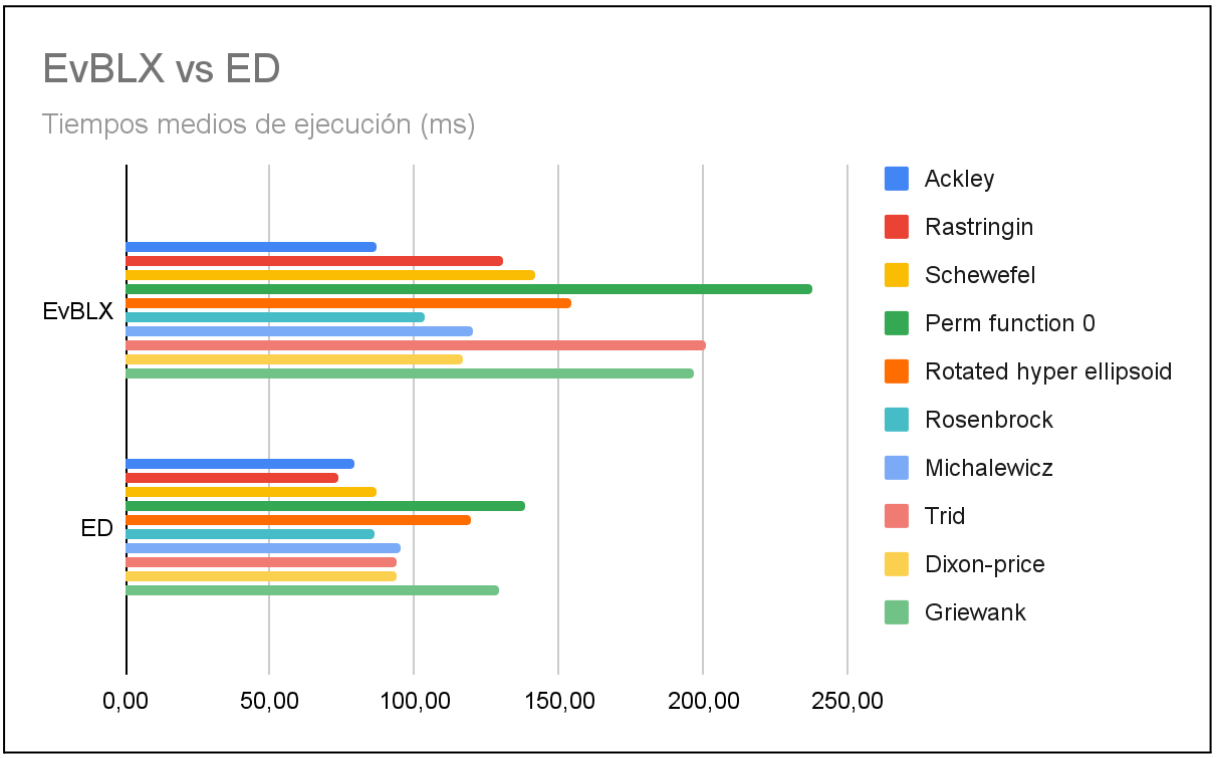
EvBLX vs ED




EvBLX vs ED
Sin aplicar escala logarítmica
2,50E+18
2,00E+18
1,50E+18
1,00E+18
5,00E+17
0,00E+0
2,22E+18
5,35E+0
EvBLX
ED
Perm function 0


EvBLX vs ED
Sin aplicar escala logarítmica
300000,00
200000,00
100000,00
0,00
276777,07
0,00
Rotated hyper ellipsoid





Si volvemos a hacer un análisis de los resultados medios obtenidos para cada función matemática, **observamos claramente que el algoritmo diferencial es mejor que el mejor de los dos evolutivos (BLX-Alfa)**.

Además, también mejora en tiempo; se puede observar cómo se mejora en los resultados y además en un menor tiempo de ejecución. Por lo que sin lugar a duda diremos que **el algoritmo de evolución diferencial es mejor que el algoritmo evolutivo BLX-Alfa**.

ED vs VNS

Si hacemos un rápido análisis de resultados comparando con el mejor algoritmo implementado en la práctica anterior (La búsqueda tabú usando VNS). **Podemos ver como el diferencial mejora todos los resultados para cada función matemática**. Además de emplear menos tiempo.

Cabe recalcar que el tiempo usado en la ejecución de los algoritmos es orientativo. Se obtienen dichos tiempos debido a la escritura de la ejecución del algoritmo en fichero log; de todas maneras, al escribir lo mismo en cada fichero, se puede usar este tiempo para comparar la rapidez en la ejecución.

Evaluación con ficheros de texto

Análisis del problema

Tenemos que hallar la mejor configuración para poder sacar mayor partido a las potencias generadas por un módulo, la manera de calcular la potencia es la siguiente:

$$P_m = DNI(a_1 + a_2 DNI + a_3 T_A + a_4 W_S + a_5 SMR) \ \forall \ a_i \in [-1, 1]$$

Donde DNI, T_A, W_S y SMR son valores constantes extraídos de una serie de observaciones.

Dentro del fichero de texto podemos encontrar dichos valores y además la potencia teórica del propio módulo. Tenemos bastantes observaciones por lo que **habrá que calcular la potencia real (P_m) para cada una de estas observaciones**.

El objetivo es **minimizar** el error generado entre la potencia media y la potencia real. Para evaluar dichos errores usaremos MAPE y RMSE.

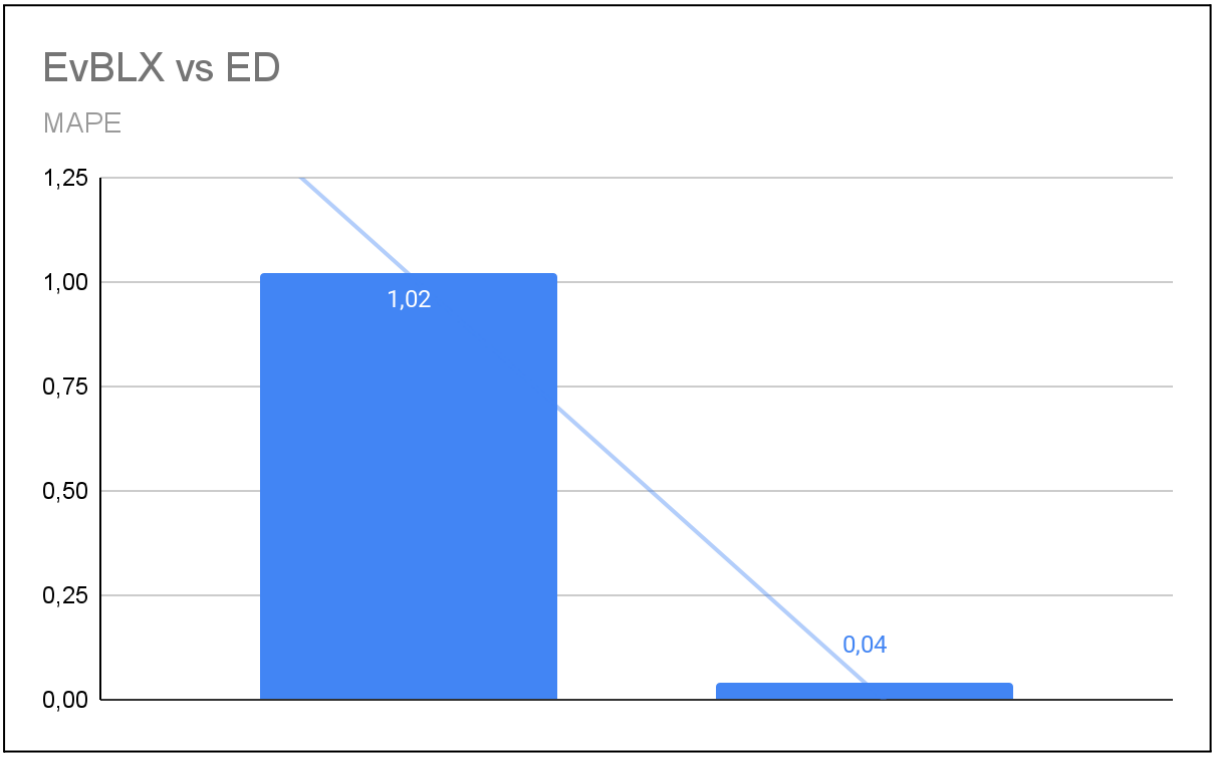
Para poder resolver este problema, usaremos los mejores 2 algoritmos de entre los probados anteriormente con funciones matemáticas, es decir; usaremos algoritmos evolutivos convencionales usando un cruce blx-alfa y otro algoritmo basado en evolución diferencial.

Nuestros individuo ahora contarán con 5 alelos generados en los intervalos mencionados anteriormente, nuestro fitness ahora se basará en el resultado generados por las funciones de evaluación de error MAPE y RMSE, nuestro objetivo será acercarnos a 0; es decir, **minimizar** el error.

Evaluación

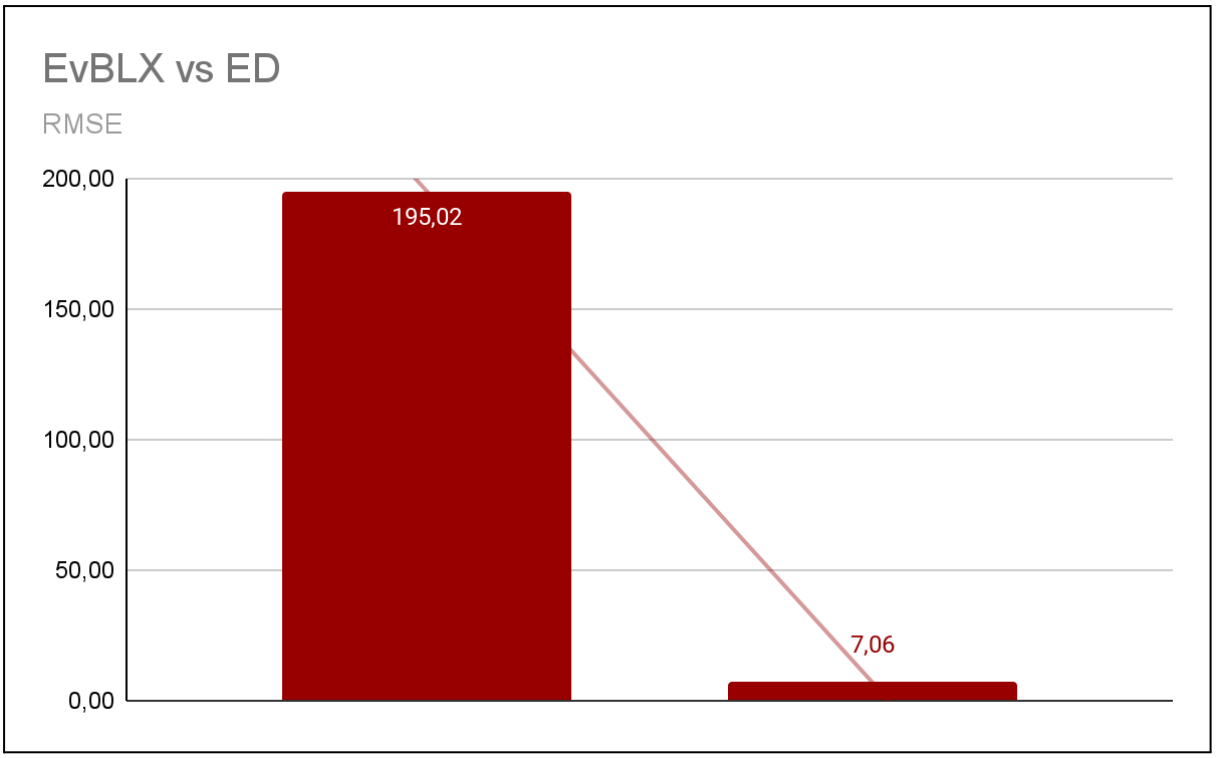
Tras realizar 5 ejecuciones para cada algoritmo con una función de evaluación en concreto, obtenemos los siguientes **resultados**:

MAPE



Podemos observar que **el algoritmo evolutivo da peor error que el algoritmo diferencial**. Al igual que en las pruebas realizadas con las funciones matemáticas.

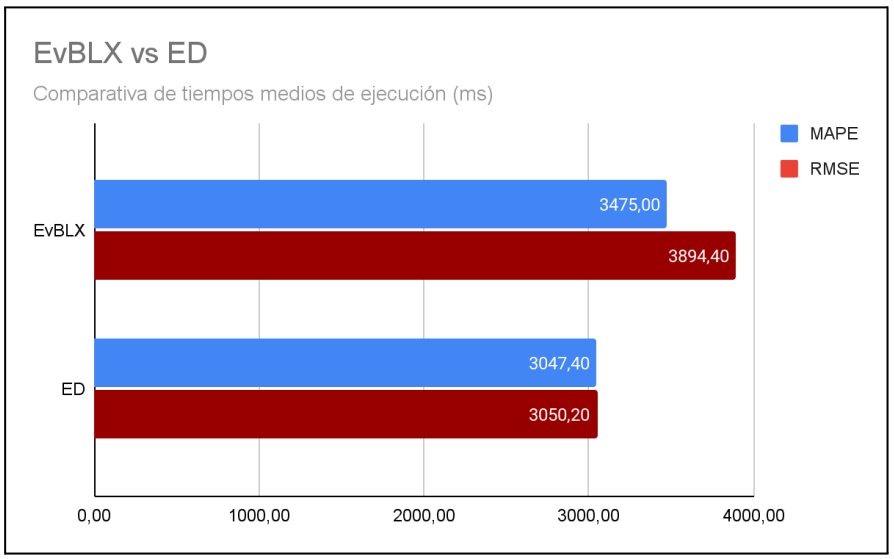
RMSE



Podemos sacar las mismas conclusiones que en el apartado anterior, **el algoritmo diferencial es mejor que el evolutivo**.

Tiempos de ejecución

Podemos echarle un vistazo a los tiempos de ejecución de cada algoritmos aplicando cada función de error:



Podemos ver que el **algoritmo diferencial** emplea menos tiempo y da mejores resultados, por lo que se usará este algoritmo para hallar la combinación de valores $\mathbf{a}_i$ óptima para este problema.