

Instituto Politécnico Nacional

ESCOM

“Escuela Superior de Cómputo”

Materia: Algoritmos Bionspirados

Profesora: Abril Valeria Uriarte Arcia

Practica 2: Programación Genética

Alumno: Higuera Pineda Angel Abraham

Grupo: 5BM1

Fecha: 16/10/2025

Introducción:

La programación genética (GP) es un método de programación de ordenadores que utiliza la selección natural o los algoritmos evolutivos. Se basa en la idea de que las soluciones a los problemas pueden encontrarse evolucionando en una simulación informática. Comienza con una población de programas aleatorios, que se evalúan en función de su capacidad para resolver el problema. Los programas con mejor rendimiento se seleccionan para formar la siguiente generación, que se muta y se evalúa de la misma manera. Este proceso se repite hasta que se encuentra una solución satisfactoria. [1]

Utilizando los principios de la selección natural darwiniana y operaciones inspiradas biológicamente.

Uno de los problemas que puede solucionar la programación genética es la regresión simbólica. Esto es una técnica de aprendizaje automático que tiene como objetivo identificar una expresión matemática subyacente que mejor describe una relación. Eso comienza construyendo una población de fórmulas aleatorias ingenuas para representar una relación entre las variables independientes conocidas y su variable dependiente objetivos para predecir nuevos datos. Cada generación sucesiva de programas es luego evolucionó a partir del anterior seleccionando al más apto individuo de la población para someterse a operaciones genéticas. [2]

En pocas palabras, se encarga de buscar una función matemática que resuelva el problema, sin alguna estructura previa.

Cabe mencionar que la manera de resolver esto, es por medio de la creación de árboles.

En Python existen algunas librerías para la programación genética, estas librerías se encargan de facilitar la implementación de este tipo de programas/algoritmos. Algunas de esas librerías son: GPLearn y Deap,

- **GPLearn:**

Es una librería en Python para realizar regresión simbólica usando programación genética. Para ello comienza construyendo una población de fórmulas aleatorias para representar una relación entre variables independientes conocidas y sus objetivos de variables dependientes con el fin de predecir los datos. Luego, cada generación sucesiva de programas evoluciona a partir de la anterior seleccionando a los individuos más aptos de la población, los que mejor predicen los datos. [3] =

- **Deap:**

Es un marco de trabajo (framework) de computación evolutiva que proporciona un conjunto de herramientas y estructuras de datos para implementar algoritmos genéticos y otras técnicas evolutivas de manera fácil y rápida. A diferencia de otras bibliotecas que ofrecen soluciones "listas para usar" para problemas específicos,

DEAP está diseñado para ser extremadamente flexible y transparente, dándole al programador un control total sobre cada componente del proceso evolutivo. [4]

Desarrollo:

Se nos pido revisar la biblioteca Deap y modificar la implementación de regresión simbólica propuesta (ejemplo) en la página principal de la documentación de Deap.

La propuesta dada está hecha para solucionar una distribución unidimensional, específicamente la función: $x^4 + x^3 + x^2 + x$

Como comentario visto, se debe tener encuenta las versiones de cada librería e incluso la versión de Python, ya que al intentar implementar el ejemplo de la librería GPLEarn, se tuvo muchos problemas debido a la incompatibilidad de nuevas versiones, como se pudo solucionar cambiando la versión de Python a la 3.10 y la versión de numpy, desconozco si se puede utilizar la versión más reciente de Deap con la última versión de Python (3.13.x). Lo menciono, ya que fue un problema bastante problemático.

Se debe modificar el ejemplo para que pueda soportar una función de 2 variables y que se acerque a la función: $x^3 * 5y^2 + \frac{x}{2}$

La modificación en si es sencilla, basta con solo modificar una línea de código en el apartado de la “Creación del conjunto de primitivas”



```
#Crea un conjunto de primitivas llamado "MAIN" que acepta 2 argumentos.  
pset = gp.PrimitiveSet("MAIN", 2)
```

- Creación del conjunto de primitivas

Dentro de este apartado, se agregan las operaciones que va a poder utilizar el algoritmo (árbol) como la suma, resta, división, negación, potencia cuadrada y cubica. También nombramos las variables a utilizar.

Se puede agregar cualquier operación que desee, pero entre más se tengan, puede llegar a tardar más en el proceso de la creación de las ramas (Generación de la función). En la captura que proporciono, se puede eliminar la operación de negación. Algunas operaciones tienen funciones aparte, para la prevención de errores, como la división en 0 y potencias que generan

resultados infinitos.



```
#Renombra los argumentos de entrada a 'x' e 'y' para mayor claridad.  
pset.renameArguments(ARG0='x', ARG1='y')  
#Anade la operacion de suma (2 argumentos) al conjunto.  
pset.addPrimitive(operator.add, 2)  
#Anade la operacion de resta (2 argumentos).  
pset.addPrimitive(operator.sub, 2)  
#Anade la operacion de multiplicacion (2 argumentos).  
pset.addPrimitive(operator.mul, 2)  
#Anade la funcion de division protegida (2 argumentos).  
pset.addPrimitive(protected_div, 2)  
#Anade la operacion de negacion (1 argumento).  
pset.addPrimitive(operator.neg, 1)  
#Anade la funcion de potencia al cuadrado protegida (1 argumento).  
pset.addPrimitive(protected_pow2, 1)  
#Anade la funcion de potencia al cubo protegida (1 argumento).  
pset.addPrimitive(protected_pow3, 1)  
#Anade una constante efimera que genera un numero aleatorio entre -2.0 y 2.0.  
pset.addEphemeralConstant("rand_small", functools.partial(random.uniform, -2.0, 2.0))
```

Las funciones fueron creadas por un tema persona, ya que cuando generaba algún error, mostraba en la ejecución un texto en rojo para indicar la alerta que sucedió, gracias a Numpy, cosa que no me agradaba. De paso se agregaron verificaciones y soluciones a dichas alertas.

- División: Verifica que el resultado sea un valor valido, en caso de que no regrese 1



```
#Define una funcion de division protegida para evitar errores de division por cero.  
def protected_div(left, right):  
    #Suprime temporalmente las advertencias de NumPy sobre division por cero o valores invalidos.  
    with np.errstate(divide='ignore', invalid='ignore'):  
        #Realiza la division.  
        x = np.divide(left, right)  
    #Si el resultado es infinito o no es un numero (NaN).  
    if np.isinf(x) or np.isnan(x):  
        #Devuelve un valor por defecto (1.0).  
        return 1.0  
    #Si el resultado es valido, lo devuelve.  
    return x
```

- Potencia al cuadrado: Verifica que el resultado sea un valor valido, en caso de que no regrese 0



```
#Define una funcion protegida para elevar al cuadrado.  
def protected_pow2(x):  
    #Suprime temporalmente las advertencias sobre desbordamiento (overflow).  
    with np.errstate(over='ignore', invalid='ignore'):  
        #Calcula la potencia.  
        val = np.power(x, 2)  
    #Si el resultado es infinito o NaN.  
    if np.isinf(val) or np.isnan(val):  
        #Devuelve un valor por defecto (0.0).  
        return 0.0  
    #Si es valido, devuelve el valor.  
    return val
```

- Potencia cubica: Verifica que el resultado sea un valor valido, en caso de que no, regresa 0



```
#Define una funcion protegida para elevar al cubo.  
def protected_pow3(x):  
    #Suprime temporalmente las advertencias sobre desbordamiento.  
    with np.errstate(over='ignore', invalid='ignore'):  
        #Calcula la potencia.  
        val = np.power(x, 3)  
    #Si el resultado es infinito o NaN.  
    if np.isinf(val) or np.isnan(val):  
        #Devuelve un valor por defecto (0.0).  
        return 0.0  
    #Si es valido, devuelve el valor.  
    return val
```

Como se puede observar en cada función suprime la alerta y realiza la operación y si se obtiene un valor no valido, dará un resultado por defecto, en caso de que sea válido, dará el resultado obtenido.

- Creación y Caja de herramientas

En este apartado se registran algunos parámetros específicos del proceso de evolución.

En la creación se define como va a trabajar el programa, es decir se define si el fitness trabajara como un problema de minimización, indicando por medio de un numero negativo = -1.0.

Se define como serán los individuos que tendrá cada generación y resuelvan el problema. Esta definido para que sea un árbol de expresión matemática, como add(mul(x, 5), y) y que trabaje siendo como un problema de minimización. A simple vista no se entiende como es la función, pero este “problema” se soluciona más adelante con una función que se encarga de simplificar el resultado obtenido.



```
# --- DEAP setup ---
#Crea una definicion de 'Fitness' para minimizacion (un solo objetivo, menor es mejor).
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
#Crea la estructura de un 'Individuo', que es un arbol de primitivas con el 'Fitness' definido.
creator.create("Individual", gp.PrimitiveTree, fitness=creator.FitnessMin)
```

En este apartado se definen como tal las herramientas que se utilizaran para la creación de las funciones, como:

- Toolbox.registrar(expr): Mezcla de los árboles o ramas de forma aleatoria, para obtener la función.
- Toolbox.registrar(individual): Estos resultados se convierten en un individuo oficial.
- Toolbox.registrar(population): Fabrica la población inicial completa.
- Toolbox.registrar(compile): Esta es muy importante. Su trabajo es convertir un árbol de expresión (que es solo una estructura de datos) en una función de Python real que pueda calcular cosas.

Es como tomar un plano (gp.PrimitiveTree) y construir un motor funcional a partir de él. Esta herramienta te permite ejecutar las fórmulas que evolucionan el algoritmo.



```
#Crea una 'Toolbox' para almacenar las herramientas y parametros de la evolucion.
toolbox = base.Toolbox()
#Registra una herramienta para generar expresiones (árboles) usando el metodo 'HalfAndHalf'.
toolbox.register("expr", gp.genHalfAndHalf, pset=pset, min_=1, max_=3)
#Registra una herramienta para crear un individuo completo a partir de una expresión.
toolbox.register("individual", tools.initIterate, creator.Individual, toolbox.expr)
#Registra una herramienta para crear una población como una lista de individuos.
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
#Registra una herramienta para compilar un árbol de expresión en una función de Python ejecutable.
toolbox.register("compile", gp.compile, pset=pset)
```

- Datos de entrenamiento:

Tenemos algunas operaciones dadas por Numpy, funcionan para evaluar las funciones que fuimos obteniendo por medio de valores numéricos dadas en un rango. En este caso se usa un rango de -1.0 a 1.0.

Primero se analizan los números obtenidos con la función que proporcionamos, dicha función es la que estamos esperando que obtenga el programa, al menos una que se acerquen.



```
# --- Datos de entrenamiento ---
#Genera 500 puntos de entrenamiento (x, y) con valores aleatorios entre -1.0 y 1.0.
train_points = np.random.uniform(-1.0, 1.0, (500, 2))
#Separa las columnas en arreglos 'xs' e 'ys'.
xs, ys = train_points[:, 0], train_points[:, 1]
#Calcula los valores objetivo usando la función real que queremos que el sistema descubra.
target_raw = 5 * xs ** 3 * ys ** 2 + xs / 2.0
```

- Analizador o calificador dependiendo del fitness

Esta función es el "juez" del programa. Su único trabajo es tomar una fórmula (un individuo), calificarla y devolver su puntuación de error (su fitness).

Lo hace en tres pasos principales:

- Convierte la formula en una herramienta funcional:

La primera línea toma el individuo, que es solo una estructura de datos en forma de árbol (como un plano), y usa la herramienta compile para convertirlo en una función de Python real y ejecutable que puede hacer cálculos.

```
#Compila el arbol del individuo en una función de Python.
func = toolbox.compile(expr=individual)
```

- Prueba y cálculo de errores:

Esta es la parte central. El código intenta hacer lo siguiente:

1. Aplica la función: Ejecuta la función en cada uno de los 500 puntos de entrenamiento (train_points) para obtener 500 predicciones (preds).
2. Revisa los Resultados: Comprueba si alguna de las predicciones es un valor matemático inválido como NaN (No es un Número) o inf (infinito). Si encuentra alguno, significa que la fórmula es inestable y le asigna una puntuación de error terrible (1e12, un billón) para descartarla inmediatamente. 🤦

3. Calcula el Error Real: Si todos los resultados son válidos, calcula el Error Cuadrático Medio (MSE). Compara sus predicciones (preds) con las respuestas correctas (target_raw), eleva las diferencias al cuadrado y saca el promedio. Este MSE es la calificación final de la fórmula. Un número más bajo significa una mejor calificación.

```
#Inicia un bloque try-except para manejar posibles errores durante la evaluacion.
try:
    #Aplica la funcion a cada punto de entrenamiento y obtiene las predicciones.
    preds = np.fromiter((func(x, y) for x, y in train_points), dtype=np.float64, count=len(train_points))
    #Si alguna prediccion es NaN o infinita, la solucion no es valida.
    if np.any(np.isnan(preds)) or np.any(np.isinf(preds)):
        #Devuelve una aptitud muy mala (un numero muy grande).
        return (1e12,)
    #Calcula el Error Cuadratico Medio (MSE) entre las predicciones y los valores objetivo.
    mse = np.mean((preds - target_raw) ** 2)
    #Devuelve el MSE como una tupla (formato requerido por DEAP).
    return (mse,)
```

- Manejo de errores:

Si genera un error, lo indicará o regresará el valor si no hay ningún error.

```
#Si ocurre cualquier otra excepcion durante la evaluacion.
except Exception:
    #Devuelve una aptitud muy mala.
    return (1e12,)
```

- Traducción del programa:

En este apartado se encarga de solucionar el problema de las funciones “raras”. Ej:

```
Gen 004 | Fitness promedio = 1.5395e+10 | Mejor = 2.1551e-01 | Tamaño promedio = 4.10
↳ Mejor función encontrada: mul(add(x, add(x, x)), protected_pow2(y))
```

Para que se vean algo así:

```
--- Expresión Simplificada ---
x*(4.932866666718*x**2*y**2 + 0.514365306753784)
```

Esta función utiliza la biblioteca SymPy.

Primero, crea las variables simbólicas x e y que usará la biblioteca de álgebra SymPy. Luego, define un diccionario de traducción (local_dict), que es el apartado más importante. Este diccionario le dice a Python cómo convertir los nombres de las funciones de DEAP (como la cadena de texto 'add') a las operaciones matemáticas reales que SymPy entiende (como el operador +).

```

#Crea los simbolos 'x' e 'y' para la expresion matematica.
x, y = sp.symbols('x y')
#Define un diccionario para mapear los nombres de las funciones de DEAP a sus equivalentes en SymPy.
local_dict = {
    'add': lambda a, b: a + b, 'sub': lambda a, b: a - b, 'mul': lambda a, b: a * b,
    'protected_div': lambda a, b: a / b, 'neg': lambda a: -a,
    'protected_pow2': lambda a: a ** 2, 'protected_pow3': lambda a: a ** 3,
    'x': x, 'y': y
}

```

Después, convierte la fórmula a texto, donde el individuo de DEAP es una estructura de árbol compleja. Esta parte primero lo convierte en una simple cadena de texto, algo como: add(mul(x, y), rand_small).

Luego, busca cualquier constante efímera (los números aleatorios que el algoritmo inventó) y reemplaza sus nombres genéricos (rand_small) por sus valores numéricos reales (ej. 1.7345). La cadena de texto ahora se ve así: add(mul(x, y), 1.7345).

```

#Convierte el arbol del individuo a una cadena de texto (ej. "add(mul(x, y), 2.5)").
code = str(individual)
#Busca todas las constantes efimeras en el individuo.
ephemerals = [p for p in individual if isinstance(p, gp.Terminal) and p.name.startswith("rand")]
#Reemplaza los nombres de las constantes (ej. "rand_small") por sus valores numericos en la cadena.
for eph in reversed(ephemerals):
    code = code.replace(eph.name, f"{{eph.value}}")

```

Por último, ejecuta y entrega el resultado final simplificado:

1. eval(code, ...): Ejecuta la cadena de texto como si fuera código. Usando el local_dict como guía, construye una expresión matemática real en el lenguaje de SymPy. Es un paso de ejecución muy seguro porque {"__builtins__": None} le prohíbe usar cualquier comando potencialmente peligroso.
2. sp.simplify(expr): Una vez que tiene la expresión matemática, usa toda la potencia de SymPy para simplificarla algebraicamente (por ejemplo, convertir $x + x$ en $2*x$).
3. return: Finalmente, devuelve la expresión limpia y simplificada.

```

#Inicia un bloque try-except para manejar errores en la conversion.
try:
    #Evalua de forma segura la cadena de texto para construir una expresion de SymPy.
    expr = eval(code, {"__builtins__": None}, local_dict)
    #Simplifica la expresion matematica resultante y la devuelve.
    return sp.simplify(expr)
#Si ocurre un error durante la simplificacion.
except Exception as e:
    #Devuelve un mensaje de error.
    return f"Error al simplificar: {e}"

```

Con esto terminamos las funciones agregadas, pasamos al main, que es donde se inicia las ejecuciones del programa y se decide que tipo de selección, crusa, mutación, etc. se usa.

- Principal:

Al principio se tienen la impresión de información básica de la práctica, en pocas palabras la mini presentación.

```
print("/// PROGRAMACION GENETICA ///")
print("By: Abraham Higuera")
print("-----")
print("Bienvenido a mi programa\n")
print("Funcion a buscar: 5 * xs ** 3 * ys ** 2 + xs / 2.0")
```

- o Aumento de velocidad:

Después se tiene un apartado super importante para mejorar la velocidad del programa. Esto debido a que en el apartado de la evaluación se evalúan alrededor de 1500 individuos para poder seleccionar al mejor de la generación, este proceso es algo lento debido a la cantidad de individuos y verificaciones que hace. Se puede mejorar la velocidad si disminuimos la cantidad de individuos, pero para tener un valor más cercano fue buena idea utilizar 1000 o 1500 individuos. Entonces para poder acelerar el programa es crucial utilizar toda la potencia de nuestro equipo, esto se obtiene al implementar un esquema de computación paralela, utilizando la biblioteca MULTIPROCESSING.

- Creación de un Pool de procesos:

La ejecución en paralelo se inicia con la siguiente instrucción:

```
#Crea un 'pool' de procesos para usar multiples nucleos del CPU.
pool = multiprocessing.Pool()
```

Esta línea de código instancia un objeto Pool, el cual crea un conjunto de procesos de trabajo (worker processes). Por defecto, el número de procesos creados es igual al número de núcleos lógicos de la CPU disponibles en el sistema. Cada uno de estos procesos opera de manera independiente y puede ejecutar tareas de forma simultánea.

- Integración con el Flujo:

```
#Registra la funcion 'map' del pool en la toolbox para que DEAP la use.
toolbox.register( alias: "map", pool.map)
```

Esta instrucción sobrescribe la función map por defecto que utiliza DEAP. La función map estándar de Python es secuencial, es decir, aplica una función a una lista de elementos uno

por uno. Al registrar pool.map en su lugar, se le instruye a DEAP para que, durante la fase de evaluación de la población, distribuya la tarea de evaluar a los individuos entre todos los procesos de trabajo disponibles en el pool.

Con esto se tiene un aumento de la velocidad significativamente. Es sorprendente que con pequeñas líneas de código puedes aumentar el rendimiento.

- Selección de parámetros:

Para guiar el proceso evolutivo, se configuraron los siguientes operadores genéticos en la Toolbox de DEAP, los cuales definen cómo la población de soluciones mejora a lo largo de las generaciones.

- Evaluación (evaluate): Se registró la función eval_symb_reg como el método para calcular la aptitud (fitness) de cada individuo. Esta función mide el Error Cuadrático Medio (MSE) entre las predicciones de una fórmula y los datos reales, siendo este el valor a minimizar.
- Selección (select): Se implementó una selección por torneo (selTournament) con un tamaño de torneo de 7 (tournsize=7). En este método, se seleccionan aleatoriamente 7 individuos de la población y solo el mejor de ellos (el de menor error) es elegido para convertirse en padre. Un tamaño de torneo grande incrementa la presión selectiva, favoreciendo la convergencia hacia soluciones de alta calidad.
- Cruce (mate): Se utilizó el operador de cruce de un punto (gp.cxOnePoint). Este método toma dos individuos padres, selecciona un punto de cruce aleatorio en cada uno e intercambia los subárboles resultantes para crear dos nuevos individuos descendientes, combinando así el material genético de ambos.
- Mutación (mutate): Se configuró una mutación uniforme (gp.mutUniform), la cual introduce diversidad genética. Este operador selecciona aleatoriamente un nodo dentro del árbol de un individuo y lo reemplaza por un subárbol completamente nuevo, generado aleatoriamente.
- Control de Complejidad (gp.staticLimit): Para prevenir el crecimiento descontrolado de los árboles de expresión (un fenómeno conocido como bloat), se aplicó un decorador para limitar la altura máxima de los árboles a 17 niveles. Esta restricción se impone después de aplicar tanto los operadores de cruce como los de mutación, asegurando que las soluciones se mantengan computacionalmente manejables.

```

#Registra la funcion de evaluacion en la toolbox.
toolbox.register("evaluate", eval_symb_reg)
#Registra el metodo de seleccion (torneo de tamano 7).
toolbox.register("select", tools.selTournament, tourysize=7)
#Registra el metodo de cruce (crossover de un punto).
toolbox.register("mate", gp.cxOnePoint)
#Registra un metodo para generar expresiones pequenas para la mutacion.
toolbox.register("expr_mut", gp.genFull, min_=0, max_=2)
#Registra el metodo de mutacion (mutacion uniforme, que reemplaza un subarbol).
toolbox.register("mutate", gp.mutUniform, expr=toolbox.expr_mut, pset=pset)
#Anade un "decorador" para limitar la altura maxima de los arboles a 17 despues del cruce.
toolbox.decorate("mate", gp.staticLimit(key=operator.attrgetter("height"), max_value=17))
#Anade un "decorador" para limitar la altura maxima de los arboles a 17 despues de la mutacion.
toolbox.decorate("mutate", gp.staticLimit(key=operator.attrgetter("height"), max_value=17))

```

Podremos considerar que la cantidad de individuos que tiene la población entra en el apartado de selección de parámetros. Y la creación del salón de la fama, para obtener el mejor individuo de cada generación.

```

#Crea la poblacion inicial con 1500 individuos.
pop = toolbox.population(n=1000)
#Crea un objeto 'Hall of Fame' para guardar al mejor individuo encontrado.
hof = tools.HallOfFame(1)

```

- Monitoreo y ejecución:

Para observar el progreso de la evolución, se configuró un sistema de recolección de estadísticas y se ejecutó el bucle evolutivo principal.

1. Recolección de Estadísticas

Se implementó un objeto MultiStatistics para registrar dos métricas clave en cada generación:

- Aptitud (fitness): Se monitoreó el valor del Error Cuadrático Medio (MSE) de los individuos.
- Tamaño (size): Se registró el número de nodos en el árbol de cada individuo para observar su complejidad.

Para ambas métricas, se calcularon el promedio (avg), la desviación estándar (std) y el valor mínimo (min) de la población. Este monitoreo es crucial para analizar la convergencia del algoritmo y el comportamiento de la población a lo largo del tiempo.

2. Ejecución del Bucle Evolutivo

El proceso evolutivo se llevó a cabo mediante el algoritmo algorithms.eaSimple, que orquesta el ciclo de selección, cruce y mutación. Se configuró con los siguientes hiperparámetros:

- Probabilidad de Cruce (cxb): 0.7 (70%), indicando una alta preferencia por recombinar soluciones existentes.
- Probabilidad de Mutación (mutpb): 0.2 (20%), para introducir nueva diversidad genética en la población.
- Número de Generaciones (ngen): 100, definiendo la duración total del proceso evolutivo.

Adicionalmente, se utilizó un objeto HallOfFame (hof) para preservar al mejor individuo encontrado en toda la ejecución, garantizando que la mejor solución nunca se pierda.

3. Finalización de Procesos Paralelos

Al concluir el algoritmo, se ejecutaron los comandos pool.close() y pool.join(). Estas instrucciones gestionan de forma segura el cierre del pool de procesos paralelos, asegurando que todas las tareas de evaluación distribuidas finalicen correctamente antes de que el programa termine.

```
● ● ●

#Configura la recolección de estadísticas sobre la aptitud (fitness).
stats_fit = tools.Statistics(lambda ind: ind.fitness.values)
#Configura la recolección de estadísticas sobre el tamaño (número de nodos) de los individuos.
stats_size = tools.Statistics(len)
#Combina ambas estadísticas en un solo objeto.
mstats = tools.MultiStatistics(fitness=stats_fit, size=stats_size)
#Registra las métricas a calcular: promedio, desviación estandar y mínimo.
mstats.register("avg", np.mean)
mstats.register("std", np.std)
mstats.register("min", np.min)

#Ejecuta el algoritmo evolutivo 'eaSimple' con los parámetros definidos.
pop, log = algorithms.eaSimple(
    pop, toolbox, cxpb=0.7, mutpb=0.2, ngen=100,
    stats=mstats, halloffame=hof, verbose=True
)

#Cierra el pool de procesos de forma segura.
pool.close()
#Espera a que todos los procesos terminen.
pool.join()
```

Como finalización tenemos la obtención del mejor individuo y muestra su respectivo fitness y muestra la función ya simplificada.

```
#Obtiene el mejor individuo del 'Hall of Fame'.
best = hof[0]
#Imprime la cabecera de los resultados.
print("\n--- Mejor Individuo Encontrado ---")
#Imprime la aptitud (MSE) del mejor individuo.
print(f"Fitness (MSE Real): {best.fitness.values[0]}")
#Imprime otra cabecera.
print("\n--- Expresion Simplificada ---")
#Llama a la funcion para simplificar e imprimir la expresion matematica encontrada.
print(compile_to_sympy(best))
```

Resultados:

```

/// PROGRAMACION GENETICA ///
By: Abraham Higuera
-----
Bienvenido a mi programa

Funcion a buscar: 5 * xs ** 3 * ys ** 2 + xs / 2.0
      fitness          size
-----
```

gen	nevals	avg	gen	min	nevals	std	avg	gen	min	nevals	std
0	1000	3.39294e+48	0	0.324187	1000	1.07235e+50	5.047	0	2	1000	3.01808
1	780	5.52226e+30	1	0.274481	780	1.74542e+32	4.346	1	1	780	2.63824
2	750	2.30258e+15	2	0.269447	750	7.27694e+16	4.354	2	1	750	2.9703
3	775	8.02601e+29	3	0.22478	775	2.53678e+31	5.061	3	1	775	3.57845
4	765	8.32534e+45	4	0.22478	765	2.63139e+47	5.908	4	1	765	3.90404
5	761	4.24878e+09	5	0.22478	761	1.34291e+11	6.543	5	1	761	3.86292
6	790	1.20727e+18	6	0.218315	790	3.8158e+19	6.391	6	1	790	3.44211
7	727	4.49522e+12	7	0.202032	727	1.10926e+14	6.457	7	1	727	3.26836
8	788	9.09668e+62	8	0.0873959	788	2.87518e+64	6.759	8	1	788	2.9358
9	760	1.00541e+26	9	0.0873959	760	3.17778e+27	7.488	9	1	760	3.14704
10	704	1.08241e+11	10	0.0873959	704	3.00065e+12	8.214	10	1	704	3.10068
11	750	4.89073e+14	11	0.0628233	750	1.54576e+16	8.785	11	1	750	3.40247
12	766	3.11717e+73	12	0.0608268	766	9.85244e+74	8.854	12	1	766	4.05397
86	791	4.2998e+56	86	5.55577e-05	791	1.35904e+58	16.809	86	1	791	3.87924
87	741	6.86336e+63	87	5.55577e-05	741	2.16881e+65	16.97	87	1	741	4.16667
88	794	3.0738e+63	88	5.55577e-05	794	9.71535e+64	16.804	88	1	794	4.02586
89	777	1.35972e+46	89	5.55577e-05	777	4.29764e+47	16.806	89	1	777	3.91463
90	744	1.169e+132	90	5.55577e-05	744	3.69487e+133	16.815	90	1	744	3.85289
91	782	9.07799e+77	91	5.55577e-05	782	2.86928e+79	16.837	91	1	782	3.97548
92	780	1.99585e+48	92	5.55577e-05	780	6.30766e+49	16.91	92	1	780	3.96408
93	758	3.26465e+58	93	5.55577e-05	758	1.03186e+60	16.643	93	1	758	4.16312
94	772	3.33166e+115	94	5.55577e-05	772	1.05304e+117	16.723	94	1	772	4.05984
95	755	1.67411e+108	95	5.55577e-05	755	5.29136e+109	16.832	95	1	755	3.82149
96	776	2.45855e+58	96	5.55577e-05	776	7.77072e+59	16.802	96	1	776	3.80063
97	780	4.28574e+72	97	5.55577e-05	780	1.35459e+74	16.778	97	1	780	4.06703
98	758	1.94791e+42	98	5.55577e-05	758	4.5984e+43	16.904	98	1	758	4.07023
99	769	1.1799e+71	99	5.55577e-05	769	3.72931e+72	17.07	99	1	769	3.87416
100	780	1.80769e+49	100	5.55577e-05	780	5.57897e+50	16.918	100	1	780	4.00491

```

--- Mejor Individuo Encontrado ---
Fitness (MSE Real): 5.555769329777095e-05

--- Expresión Simplificada ---
x*(4.9328666666718*x**2*y**2 + 0.514365306753784)

Process finished with exit code 0

```

Como podemos observar su fitness es sumamente pequeño. Durante las pruebas nunca pudo llegar a 0, solo valores muy cercanos a este. La función obtenida es casi idéntica a la buscada, solo varía en decimales y como se mostró la expresión.

Buscábamos: $x^3 * 5y^2 + \frac{x}{2}$

Y obtuvimos: $x^3 * 4.932866 y^2 + 0.51436 x$

Como observamos son valores cercanos y no pudo mostrar tal cual $x/2$, solo dio el decimal y que multiplica.

- Referencias:
 - [1] Fortin, F. A., De Rainville, F. M., Gardner, M. A., Parizeau, M., & Gagné, C. (2012). DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13, 2171-2175.
 - [2] MlearningLab. (2024, 24 de septiembre). Regresión Simbólica con gplearn. MlearningLab. <https://mlearninglab.com/2024/09/24/regresion-simbolica-con-gplearn/>
 - [3] Stephens, T. (2015). gplearn: Genetic Programming in Python. Read the Docs. <https://gplearn.readthedocs.io/en/stable/index.html>
 - [4] TechLib.net. (s.f.). Programación Genética. Recuperado el 16 de octubre de 2025, de <https://techlib.net/techedu/programacion-genetica>