Computación evolutiva: grammatical evolution

Andrés Mañas Mañas

April 19, 2017

Abstract

En este trabajo se aborda el problema de aproximar la derivada de una función utilizando técnicas de evolución gramatical. Se incluye una introducción teórica a la evolución gramatical.

Se implementa en python un algoritmo áltamente parametrizable que permite ejecutar experimentos evolutivos orientados a obtener la mejor solución a un conjunto de 7 distintos problemas.

Se describe el método de búsqueda de los parámetros óptimos para ejecutar el programa de modo que se obtengan los mejores resultados en los 7 problemas.

Se presentan los resultados de todos los experimentos realizados basándose en métricas SR, MBF y AES junto con plots de progreso y representaciones gráficas de las funciones objetivo y las mejores aproximaciones encontradas.

Tabla de contenidos

1 Descripción del problema a resolver

Remitiéndome a las indicaciones dadas en el documento de la actividad, el problema consiste en implementar un algoritmo evolutivo para calcular la derivada simbólica de una función

$$f : X \subseteq \mathcal{R} \rightarrow \mathcal{R}$$

Disponemos de las siguientes dos definiciones:

Definición de derivada de una función en un punto: Sea $X \subseteq \mathcal{R}$ un intervalo abierto. Diremos que $f : X \subseteq \mathcal{R} \rightarrow \mathcal{R}$ es derivable en $x_0 \in X$, denotado por $f'(x_0)$, si existe y es finito el límite:

$$f'(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h} \tag{1}$$

Definición de derivada de una función en un intervalo: Sea $X \subseteq \mathcal{R}$ un intervalo abierto. Diremos que $f : X \subseteq \mathcal{R} \rightarrow \mathcal{R}$ es derivable en el intervalo $[a, b] \subseteq X$, si f es derivable en cada uno de los puntos de dicho intervalo, es decir, si:

$$f'(x) = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}, \forall x \in [a, b] \tag{2}$$

Suponiendo que f sea derivable en $[a, b]$, el problema de calcular la derivada lo vamos a transformar en un nuevo problema de optimización consistente en encontrar una función $g(x)$ que minimice la expresión:

$$\min_{g(x)} \frac{1}{b-a} \int_a^b error[f'(x), g(x)]dx \tag{3}$$

dónde $f'(x)$ se calcularía utilizando la expresión (2).

No obstante, el problema anterior se puede resolver de forma aproximada discretizando el intervalo de definición, es decir, cambiando el operador integral por un sumatorio:

$$\min_{g(x)} \frac{1}{N+1} \sum_{i=0}^{N} error_i[f'(a + i * h), g(a + i * h)] \tag{4}$$

dónde $h = \frac{b-a}{N}$ es la anchura del subintervalo de muestreo para conseguir muestrear $N+1$ puntos en el intervalo $[a, b]$, y $f'(a + i * h)$ viene dado por:

$$f'(a + i * h) = \frac{f(a + (i + 1) * h) - f(a + i * h)}{h}, \forall i \in \{0, 1, ..., N\} \tag{5}$$

2 Breve introducción a la GE

La evolución gramatical codifica un conjunto de números pseudo aleatorios **(codones)** en un cromosoma que consiste en un número variable de genes binarios de 8 bits.

Estos números se usan para seleccionar una regla apropiada a partir de una definición de gramática con **notación Backus-Naur (BNF)**.

Una gramática de BNF consiste en la tupla

$$\{N, T, P, S\}$$

dónde:

- **N** es el conjunto de no terminales,
- **T** es el conjunto de terminales
- **P** es un conjunto de reglas de producción que mapean los elementos de N a T
- **S** es un símbolo de inicio que es un miembro de N.

Los no terminales de la gramática se mapean en los terminales de la gramática mediante la aplicación recursiva de las reglas dictadas por los valores de los genes. Al finalizar el proceso de mapeo, el código final producido (fenotipo) está formado sólo por terminales.

Por ejemplo, si consideramos esta gramática BNF:

```
N = { expr, op, pre_op }
T = { Sin, Cos, Tan, Log, +, -, /, *, X, () }
S = <expr>
```

Y representamos P por:

```
(1) <expr> ::= <expr> <op> <expr>      (A)
             | ( <expr> <op> <expr> )  (B)
             | <pre-op> ( <expr> )     (C)
             | <var>                   (D)
(2) <op> ::= + (A)
           | - (B)
           | / (C)
           | * (D)
(3) <pre-op> ::= Sin (A)
               | Cos (B)
               | Tan (C)
               | Log (D)
(4) <var> ::= X
```

Consideremos la regla (1):

```
(1) <expr> ::= <expr> <op> <expr>
             | ( <expr> <op> <expr> )
             | <pre-op> ( <expr> )
             | <var>
```

En este caso, el no terminal puede producir uno de cuatro resultados diferentes, para decidir cuál utilizar nuestro sistema toma el siguiente número aleatorio disponible del cromosoma y, en este caso obtiene el módulo cuatro del número para decidir qué regal de producción toma.

Cada vez que se tiene que tomar una decisión, se lee otro número pseudo aleatorio del cromosoma, y de esta manera, el sistema atraviesa el cromosoma.

En GE es posible que los individuos se queden sin genes durante el proceso de mapeo, y en este caso hay dos alternativas: - La primera es declarar al individuo inválido y castigarlos con un valor de fitness adecuado - La segunda es envolver al individuo y reutilizar los genes. Este es un enfoque bastante inusual en EAs, ya que es completamente posible que ciertos genes se usen dos o más veces.

Lo que es crucial, sin embargo, es que cada vez que un individuo en particular es mapeado de su genotipo a su fenotipo, se genera la misma salida. Esto se garantiza por el proceso de mapeo descrito anteriormente.

3 Método para resolverlo

3.1 Idoneidad de GE para resolver el problema

Nuestro problema consiste en encontrar la derivada de una función probando/evolucionando distintas combinaciones de otras dadas.

Según leemos en grammatical-evolution.org:

> *GE has proved successful when applied to a symbolic regression problem [Ryan 98a], and finding trigonometric identities [Ryan 98b], here we apply GE to a symbolic integration problem taken from the literature [Koza 92]. This involves finding a function which is an integral of Cos(X)+2X+1. In each of these cases we take a subset of C as our target language which is described in Backus Naur Form definition. A Steady State selection mechanism [Syswerda 89] has been employed and was found to reduce the number of generations required to achieve a correct solution. Using this selection mechanism we reapplied our system to the two problems previously tackled and again found an improvement in performance for both of these problems.*

Por lo tanto, según esta referencia, parece que utilizar una técnica evolutiva basada en Evolución Gramatical podría ser la forma más indicada.

Además, teniendo en cuenta que el fundamente de la evolución gramatical se basa en definir una gramática que representa las formas válidas que pueden adoptar los individuos, y teniendo en cuenta que una función objetivo puede alcanzarse como combinación de operadores, números y otras funciones básicas, parece bastante natural que el problema de encontrar la derivadda de una función pueda abordar con técnicas de GE.

3.2 Expresión matemática de la función de evaluación

Primero necesitamos una función que nos permita evaluar en N puntos del intervalo de definición $D = [a, b]$ de una función f cómo de bien queda aproximada la derivada de f por una función

g que a la postre será el fenotipo de un individuo generado por los procesos evolutivos que se implementan más tarde en esta práctica.

```
% matplotlib inline

from math import sin, cos, exp, log
import numpy as np
import matplotlib.pyplot as plt
import threading

np.seterr(all='ignore')


def F(f, g, D, N=50., U=0.1, K_0=1., K_1=10., EVAL_PENALTY=1e6):
    """
    Función de evaluación.
    """
    a, b = D
    h = float(b - a) / N
    cum_error = 0.
    hitting = True
    for x in np.linspace(a, b, N):
        delta = .001
        try:
            e = abs((f(x + delta) - f(x)) / delta - g(x))
        except:
            e = EVAL_PENALTY
        if e <= U:
            w = K_0
        else:
            w = K_1
            hitting = False
        cum_error += (w * e)

    return hitting, cum_error / N

F(sin,cos,[0,1])
```

```
(True, 0.00022959139054304467)
```

Siguiendo las indicaciones del enunciado de la práctica, en lugar de implementar desde cero el código que interpretase la gramática y realizara la decodificación de los individuos, prefiero apoyarme en una librería ya existante, ponyge.

De ésta extraigo sólo la funcionalidad que permite interpretar la gramática y decodificar un individuo. No utilizo la librería tal cuál está disponible en la red, sino que me quedo sólo con la parte relevante, la cual incluyo en el archivo python ponyge.py. De este modo, puedo tener un objecto que interprete una gramática dada con una simple llamada:

```
from ponyge import Grammar

G = Grammar(
    """<expr>    ::= <expr><op><expr> \
                   | (<expr><op><expr>) \
                   | <pre_op>(<expr>) \
                   | <var>
       <op>      ::= + | - | * | /
       <pre_op>  ::= sin | cos | exp | log
       <var>     ::= x | 1.0
    """,
    MAX_WRAPS=0)
```

Y puedo obtener el fenotipo de un individuo cualquiera llamando al método **generate** del objeto que contiene la gramática. Nótese que el método generate devuelve:

- el fenotipo del individuo de entrada si puede ser decodificado según la gramática. None en otro caso
- el número de codones que se necesitan para la decodificación

No todos los codones se corresponden con individuos que puedan ser decodificados según la gramática, según se ve en el siguiente experimento:

```
i = 0
while True:
    i += 1
    genome = [np.random.randint(0, 10) for _ in range(5)]
    fen = G.generate(genome)
    if fen[0]:
        print fen, 'generated in %s rounds' % i
        break
```

```
('1.0', 2) generated in 2 rounds
```

3.3 Operadores de inicialización, variación y selección

Para gestionar cómodamente los individuos utilizo la clase Individual. Esta clase contiene el genoma del individuo. Llamando al método **evaluate** con una gramática y una función fitness, se calcula el fenotipo, el fenotipo compilado (función lista para ser usada) y el fitness del individuo.

```
import copy
import random
from functools import partial


class Individual(object):
    """Clase que representa los individuos"""

    def __init__(self, codon_size, genome=None, LENGTH=100):
        if genome == None:
            self.genome = [
                random.randint(0, codon_size) for _ in range(LENGTH)
            ]
        else:
            self.genome = genome[:]
        self.codon_size = codon_size

    def __lt__(self, other):
        return self.fitness < other.fitness

    def __str__(self):
        return 'Individual: ' + str(self.phenotype) + \
                ';\t' + str(self.fitness) + \
                ';\t' + str(self.hitting)

    def evaluate(self, G, fitness_f, UNDECODABLE_PENALTY=1e6):
        """La función fitness_f admite como único
        parámetro el fenotipo compilado
        """
        self.phenotype, self.used_codons = G.generate(self.genome)
        if self.phenotype is None:
```

```
            self.hitting, self.fitness = False, UNDECODABLE_PENALTY
        else:
            a, b = fitness_f(eval('lambda x:' + self.phenotype))
            self.hitting, self.fitness = a, b
```

Para la **selección**, la podemos hacer por torneo:

```
def tournament_selection(population, GENERATION_SIZE, TOURNAMENT_SIZE=3):
    """Selecciona por torneo."""
    winners = []
    while len(winners) < GENERATION_SIZE:
        competitors = random.sample(population, TOURNAMENT_SIZE)
        winners.append(sorted(competitors)[0])
    return winners
```

Las mutaciones las realizo a nivel de codón, no de bit, permitiendo que se parametrice la probabilidad de mutación. Nótese que garantizo que el individuo resultado de una mutación siempre es un individuo válido según la gramática considerada. De este modo, según mis experimentos consigo mejores resultados.

```
def mutate(ind, G, fitnessF, UNDECODABLE_PENALTY=1e6,
           MUTATION_PROBABILITY=.1):
    """Muta un individuo a nivel de codón con
    probabilidad MUTATION_PROBABILITY."""
    for i in range(min(ind.used_codons, len(ind.genome))):
        if random.random() < MUTATION_PROBABILITY:
            gen = ind.genome[i]
            vals = range(ind.codon_size)
            np.random.shuffle(vals)
            for j in vals:
                ind.genome[i] = j
                ind.evaluate(G, fitnessF,
                             UNDECODABLE_PENALTY=UNDECODABLE_PENALTY)
                if ind.phenotype is not None:
                    break
                else:
                    ind.genome[i] = gen
    ind.evaluate(G, fitnessF,
                 UNDECODABLE_PENALTY=UNDECODABLE_PENALTY)
```

Y **el cruce lo realizo por un único punto**. Conviene observar que permito parametrizar que el cruce se realize o no por partes del genoma que pertenecen a la sección del mismo que es relevante en su decodificación. Igualmente, la probabilidad de que se produzca el cruze o no (que se devuelvan clones de los padres) es también parametrizable.

En algoritmo principal que gobierna la evolución, me garantizo de que sólo individuos válidos según la gramática que se considera son utilizados tras la operación de cruce.

```
def onepoint_crossover(p_0, p_1, CROSSOVER_PROBABILITY=.1,
                       WITHIN_USED=True):
    """Dados dos individuos, crea un hijo cruzándolos
    por punto único. Si WITHIN_USED es True, el cruce
    se produce por partes del genoma usadas en la
    decodificación de los padres"""
    c_p_0, c_p_1 = p_0.genome, p_1.genome
    if WITHIN_USED:
        max_p_0, max_p_1 = p_0.used_codons, p_1.used_codons
    else:
        max_p_0, max_p_1 = len(c_p_0), len(c_p_1)

    pt_p_0 = random.randint(1, max(1,max_p_0))
```

```
    pt_p_1 = random.randint(1, max(1,max_p_1))
    if random.random() < CROSSOVER_PROBABILITY:
        c_0 = c_p_0[:pt_p_0] + c_p_1[pt_p_1:]
        c_1 = c_p_1[:pt_p_1] + c_p_0[pt_p_0:]
    else:
        c_0, c_1 = c_p_0[:], c_p_1[:]

    ind_0 = Individual(p_0.codon_size, genome=c_0)
    ind_1 = Individual(p_1.codon_size, genome=c_1)
    return ind_0, ind_1
```

Y por último, el método de **reemplazo, generacional con elitismo 2 por defecto,** aunque es configurable:

```
def generational_replacement(new_pop, pre_pop, GENERATION_SIZE,
                             ELITE_SIZE=2):
    """Devuelve la nueva población a partir
    de la actual y la anterior aplicando elitismo"""
    pre_pop.sort()
    new_pop.extend(copy.deepcopy(pre_pop[:ELITE_SIZE]))
    new_pop.sort()
    return new_pop[:GENERATION_SIZE]
```

3.4 Implementación del algoritmo principal

Por último, podemos implementar, con todas las funcionalidades anteriores, el algoritmo que gobernará nuestro programa de evolución gramátical.

```
import functools

def live(f, D, config):
    """Ejecuta el programa de evolución gramatical
    Args:
        - f: la función cuya derivada buscamos
        - D: el intervalo de definición de f
        - config: la configuración del algoritmo
    Devuelve el histórico de mejores individuos
    por generación.
    """

    N = config['N']
    GRAMMAR = config['GRAMMAR']
    MAX_WRAPS = config['MAX_WRAPS']
    CODON_SIZE = config['CODON_SIZE']
    INITIAL_IND_LENGTH = config['INITIAL_IND_LENGTH']
    EVAL_PENALTY = config['EVAL_PENALTY']
    UNDECODABLE_PENALTY = config['UNDECODABLE_PENALTY']
    U = config['U']
    K_0 = config['K_0']
    K_1 = config['K_1']
    POPULATION_SIZE = config['POPULATION_SIZE']
    GENERATION_SIZE = config['GENERATION_SIZE']
    MAX_GENERATIONS = config['MAX_GENERATIONS']
    MUTATION_PROBABILITY = config['MUTATION_PROBABILITY']
    CROSSOVER_PROBABILITY = config['CROSSOVER_PROBABILITY']
    WITHIN_USED = config['WITHIN_USED']
    TOURNAMENT_SIZE = config['TOURNAMENT_SIZE']
    ELITE_SIZE = config['ELITE_SIZE']

    # Definimos la gramática
    G = Grammar(GRAMMAR, MAX_WRAPS)
```

```
# y la función fitness
def fitnessF(fen):
    return F(f, fen, D, N=N, U=U, K_0=K_0, K_1=K_1,
             EVAL_PENALTY=EVAL_PENALTY)

# Población inicial (válida)
population = []
while len(population) < POPULATION_SIZE:
    ind = Individual(CODON_SIZE,LENGTH=INITIAL_IND_LENGTH)
    ind.evaluate(G, fitnessF,
                 UNDECODABLE_PENALTY=UNDECODABLE_PENALTY)
    if ind.phenotype is not None:
        population.append(ind)

population.sort()
best = copy.deepcopy(population[0])
history = [best]

generation = 0
while generation < MAX_GENERATIONS and not best.hitting:

    pre_pop = [copy.deepcopy(i) for i in population]

    # Seleccionamos los padres
    parents = tournament_selection(
        population, GENERATION_SIZE,
        TOURNAMENT_SIZE=TOURNAMENT_SIZE)

    # Cruzamos los padres -> nueva población
    new_pop = []
    while len(new_pop) < GENERATION_SIZE:
        A, B = random.sample(parents, 2)
        C_1, C_2 = onepoint_crossover(A, B,
            CROSSOVER_PROBABILITY=CROSSOVER_PROBABILITY,
            WITHIN_USED=WITHIN_USED)
        # Solo admitimos si algun individuo es válido
        C_1.evaluate(G, fitnessF,
                     UNDECODABLE_PENALTY=UNDECODABLE_PENALTY)
        if C_1.phenotype is not None:
            new_pop.append(C_1)
        C_2.evaluate(G, fitnessF,
                     UNDECODABLE_PENALTY=UNDECODABLE_PENALTY)
        if C_2.phenotype is not None:
            new_pop.append(C_2)


    # Mutamos la nueva población
    for ind in new_pop:
        mutate(ind, G, fitnessF,
               UNDECODABLE_PENALTY=UNDECODABLE_PENALTY,
               MUTATION_PROBABILITY=MUTATION_PROBABILITY)

    # Procedemos al reemplazo generacional
    population = generational_replacement(
        new_pop,
        pre_pop,
        GENERATION_SIZE=GENERATION_SIZE,
        ELITE_SIZE=ELITE_SIZE)

    population.sort()
    best = copy.deepcopy(population[0])
    history.append(best)
```

```
        generation += 1

    return history
```

3.5 Parametrización

Una ejecución del algoritmo puede ser parametrizada utilizando un diccionario con las siguientes propiedades:

- **N**: número de puntos de muestreo en el intervalo de definición del problema
- **GRAMMAR**: un texto representando la gramática a usar
- **MAX_WRAPS**: máximo número de vueltas que se le permite dar al codón para obtener el fenotipo por el proceso de aplicar la gramática
- **CODON_SIZE**: valor máximo permitido de los enteros que componen el codón de un individuo
- **INITIAL_IND_LENGTH**: tamaño de los codones de los individuos que se generan inicialmente
- **UNDECODABLE_PENALTY**: valor del fitnes (alto para que penalice) que se aplica a individuos que no pueden decodificarse en funciones válidas según la gramática del problema y el valor MAX_WRAPS
- **EVAL_PENALTY**: penalización que se apliaca en el fitness de aquellos individuos que representan funciones no definidas en algún punto de muestreo en el intervalo de definición del problema
- **U**: el umbral mínimo por debajo del cual el error i-ésimo es admisible. Determina cuándo se alcanza un hit.
- **K_0**: valor de ponderación cuando se alcanza un hit
- **K_1**: valor de ponderación cuando no se alcanza un hit (penalización)
- **POPULATION_SIZE**: tamaño de la población inicial
- **GENERATION_SIZE**: tamaño de las generaciones
- **MAX_GENERATIONS**: número máximo de generaciones que si se alcanza da por acabado el experimento
- **MUTATION_PROBABILITY**: probabilidad de que se produzca la mutación en un individio
- **CROSSOVER_PROBABILITY**: probabilidad de que se produzca el cruce entre dos individuos
- **WITHIN_USED**: parámetro que si es True provoca que los cruces entre individuos se produzcan entre partes de sus codones relevantes en su construcción del fenotipo
- **TOURNAMENT_SIZE**: tamaño del torneo en el proceso de selección de los individuos de la siguiente generación
- **ELITE_SIZE**: tamaño en la selección de la élite

Por ejemplo, para probar que todo compila bien y que podemos hacer experimentos con distintas configuraciones, vamos a hacer una ejecución mínima. Nótese que la siguiente función nos ayudará a generar configuraciones basadas en una por defecto, sobre la que podremos ir modificando parámetros.

```
DEFAULT_GRAMMAR = """
<expr>   ::= <expr><op><expr> \
           | (<expr><op><expr>) \
           | <pre_op>(<expr>) \
```

```
            | <var>
<op>     ::= + | - | * | / | **
<pre_op> ::= sin | cos | exp | log
<var>    ::= x | 1.0
"""

def default_config(N=50,
                   MAX_WRAPS=0,
                   CODON_SIZE=32,
                   INITIAL_IND_LENGTH=20,
                   EVAL_PENALTY=1e6,
                   UNDECODABLE_PENALTY=1e6,
                   U=0.1,
                   K_0=1,
                   K_1= 10,
                   POPULATION_SIZE=100,
                   GENERATION_SIZE=100,
                   MAX_GENERATIONS=200,
                   MUTATION_PROBABILITY=0.1,
                   CROSSOVER_PROBABILITY=0.1,
                   WITHIN_USED=True,
                   TOURNAMENT_SIZE=2,
                   ELITE_SIZE=2,
                   GRAMMAR=DEFAULT_GRAMMAR):
    return {
        'N': N,
        'MAX_WRAPS': MAX_WRAPS,
        'CODON_SIZE': CODON_SIZE,
        'INITIAL_IND_LENGTH': INITIAL_IND_LENGTH,
        'EVAL_PENALTY': EVAL_PENALTY,
        'UNDECODABLE_PENALTY': UNDECODABLE_PENALTY,
        'U': U,
        'K_0': K_0,
        'K_1': K_1,
        'POPULATION_SIZE': POPULATION_SIZE,
        'GENERATION_SIZE': GENERATION_SIZE,
        'MAX_GENERATIONS': MAX_GENERATIONS,
        'MUTATION_PROBABILITY': MUTATION_PROBABILITY,
        'CROSSOVER_PROBABILITY': CROSSOVER_PROBABILITY,
        'WITHIN_USED': WITHIN_USED,
        'TOURNAMENT_SIZE': TOURNAMENT_SIZE,
        'ELITE_SIZE': ELITE_SIZE,
        'GRAMMAR': DEFAULT_GRAMMAR
    }
```

Podemos hacer un primer ejemplo para ver que todo va bien:

```
f = lambda x: x**2
sample_histories = [live(f, [0, 1],
                         default_config(MAX_GENERATIONS=10,
                                        GENERATION_SIZE=100)) \
                    for _ in range(5)]
for h in sample_histories[0]:
    print h
```

```
Individual: (x*exp(x));         1.79040373628;         False
Individual: (x*exp(x));         1.79040373628;         False
Individual: (x*exp(x));         1.79040373628;         False
Individual: x*exp(x);         1.79040373628;         False
Individual: (x*exp(cos(cos(x))));         0.542066130219;         False
Individual: (x*exp(cos(cos(x))));         0.542066130219;         False
Individual: (x+x);         0.000999999999993;         True
```

Resultado que, para ser el primero, sin optimización de parámetros ninguna, ofrece un comportamiento bastante bueno.

3.6 Manejo de restricciones y mecanismos de control de parámetros adaptativos o auto-adaptativos

En cuanto al manejo de restricciones, como he indicado más arriba, procedo de estos modos:

- **en la población inicial me aseguro de que todos los individuos son válidos**, es decir, pueden decodificarse en un fenotipo válido
- **en el cálculos del fitness de los individuos, aquellos cuyo fenotipo acaba siendo una función no definida en el intervalo de estudio, son penalizados** con un fitness muy alto por lo que están condenados a la extinción
- **en la operación de mutación garantizo que el individuo resultado de la mutación sigue siendo válido** según la gramática en uso
- **en la función de cruce, igualmente garantizo que sólo los resultados del cruce válidos según la gramática se incorporan a la nueva población**.
- **la propia función de evaluación penaliza con pesos altos aquellos individuos que se corresponden con funciones no definidas en el intervalo de definición de la función** que estamos buscando su derivada.

Probablemente no es buena práctica descartar individuos que no son válidos según la gramática en uso. Sin embargo, **según mis pruebas, los resultados siempre fueron peores en aquellos casos en los que no me aseguraba de estos descartes tempranos**.

Por lo demás, sinceramente no he tenido tiempo de profundizar en hacer que los parámetros del algoritmo sean adaptativos. Ya en la segunda práctica trabajé esta parte con profundidad y, como digo, por cuestiones de tiempo no me veo capaz de incluirlo en ésta.

4 Utilidades de interpretación de resultados

4.1 Plots de progreso en escala logarítmica

En primer lugar es necesario poder dibujar las gráficos de progreso de la convergencia a la solución.

Nótese el uso de la **escala logarítmica** para que las diferencias entre los individuos de las primeras generaciones (típicamente muy lejanos al óptimo) y los de las últimas no aberren los gráficos.

```
import matplotlib.pyplot as plt
import math

def log_plot(histories, figsize=(8,5), title=None):
    """ Dibuja el logaritmo del fitness de una
    lista de ejecuciones."""
    plt.figure(figsize=figsize)
    plt.ylabel('log (best individual fitness)')
```

```
    plt.xlabel('generation')
    for h in histories:
        d = [math.log(i.fitness + 1) for i in h]
        for i in range(len(d))[1:]:
            d[i] = min(d[i-1],d[i])
        plt.plot(d)
    plt.legend([i for i in range(len(histories))])
    if title:
        plt.title(title)
    plt.tight_layout()
    plt.show()

log_plot(sample_histories)
```

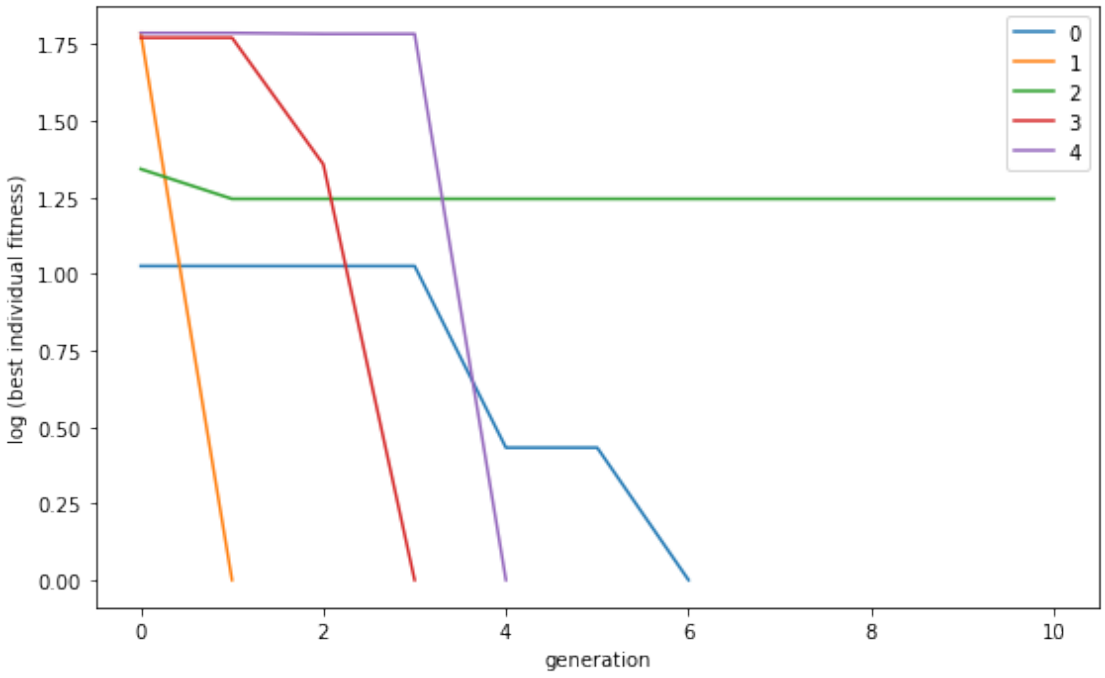


Figura 1: Prueba de representacion de progreso del fitness en escala logaritmica

4.2 Índices SR, MBF y AES

Para las métricas SR y AES, un experimento se considera exitoso si se encuentra un individuo que alcanza un hit en todos los puntos de muestreo.

```
import numpy

def SR(histories):
    """
    Calcula el porcentaje de experimentos que
    acaban con éxito.
    Un experimento es considerado como exitoso
    si encuentra un individuo que alcanza un hit
    en todos los puntos de muestreo.
```

```
    """
    exitos = filter(lambda h: h[-1].hitting, histories)
    return float(len(exitos)) / len(histories)


def AES(histories, GENERATION_SIZE):
    """
    Calcula el número medio de evaluaciones
    y su desviación típica para aquellos
    experimentos que fueron exitosos.
    Un experimento es considerado como exitoso
    si encuentra un individuo que alcanza un hit
    en todos los puntos de muestreo.
    """
    exitos = filter(lambda h: h[-1].hitting, histories)
    totals = [len(e) * GENERATION_SIZE for e in exitos]
    return np.mean(totals), np.std(totals)

def MBF(histories):
    """
    Calcula la media y la desciación típica
    del fitness del mejor individuo encontrado
    """
    A = [h[-1].fitness for h in histories]
    return np.mean(A), np.std(A)
```

Y añado además una función que me facilita la presentación de resultados en forma de tabla:

```
import pandas as pd

pd.set_option('display.notebook_repr_html', True)

def _repr_latex_(self):
    return self.to_latex()

pd.DataFrame._repr_latex_ = _repr_latex_

def print_metrics(named_histories, config):
    """
    Dibuja de forma tabulada las métricas de
    rendimiento de una lista de experimentos
    dados por el histórico de mejores individuos.
    """
    indexes=[]
    aes_means=[]
    aes_stds=[]
    mbf_means=[]
    mbf_stds=[]
    srs=[]
    for name,histories in named_histories:
        indexes = indexes + [name]
        aes = AES(histories, config['GENERATION_SIZE'])
        aes_means = aes_means + [aes[0]]
        aes_stds = aes_stds + [aes[1]]
        mbf = MBF(histories)
        mbf_means = mbf_means + [mbf[0]]
        mbf_stds = mbf_stds + [mbf[1]]
        srs = srs + [SR(histories)]
    return pd.DataFrame(data={'AES mean': aes_means,
                              'AES std': aes_stds,
                              'MBF mean': mbf_means,
                              'MBF std': mbf_stds,
                              'SR' : srs},
                        index=indexes)
```

```
print_metrics([['Exp. 1', sample_histories],
               ['Exp. 2', sample_histories]],
              default_config())
```

	AES mean	AES std	MBF mean	MBF std	SR
Exp. 1	450.0	180.277564	0.49566	0.98932	0.8
Exp. 2	450.0	180.277564	0.49566	0.98932	0.8

Figura 2. Tabla de prueba de representación de métricas

Añado además una funcionalidad que me permita ejecutar experimentos en paralelo:

```
from functools import partial
from multiprocessing import Pool

def i_live(f, D, config, i):
    return live(f, D, config)

def p_live(f, D, config, n):
    """
        Ejecuta en paralelo una batería
        de experimentos.
    """
    p = Pool(4)
    result = p.map(partial(i_live, f, D, config),
                   range(n))
    p.close()
    p.terminate()
    return result
```

5 Los 7 problemas

```
def f_1(x): return 2 * x**3 + 5
def g_1(x): return 6 * x**2

def f_2(x): return (x-1)/(x+1)
def g_2(x): return 2/((x+1)**2)

def f_3(x): return ((x**2 + 1)*(x-1))/4
def g_3(x): return (3*x**2 - 2*x + 1)/4

def f_4(x): return -exp(-(x**2) + 3)
def g_4(x): return 2 * x * exp(-(x**2) + 3)

def f_5(x): return (exp(2*x) + exp(-6*x))/6
def g_5(x): return exp(2*x)/3 - exp(-6*x)

def f_6(x): return x * log(1+x)
def g_6(x): return log(1+x) + x/(1+x)

def f_7(x): return exp(x) * sin(x)
def g_7(x): return exp(x) * (sin(x) + cos(x))

problemas = [['Problema 1', f_1, g_1, [0,5]],
```

```
            ['Problema 2', f_2, g_2, [0,5]],
            ['Problema 3', f_3, g_3, [-2,2]],
            ['Problema 4', f_4, g_4, [0,3]],
            ['Problema 5', f_5, g_5, [0,2]],
            ['Problema 6', f_6, g_6, [0,5]],
            ['Problema 7', f_7, g_7, [-2,2]]]
```

6 Justificación de la selección de parámetros

En primer lugar, vamos a evaluar el rendimiento del algoritmo **con los parámetros por defecto** en los 7 problemas:

```
def DEFAULT_CONFIG():
    config = default_config()
    results = []
    for nombre,f,_,D in problemas:
        results = results + [[nombre,
                              p_live(f, D, config, 4)]]
    df = print_metrics(results, config)
    df.to_csv('data/DEFAULT_CONFIG.csv')
    return df


# DEFAULT_CONFIG()
pd.read_csv('data/DEFAULT_CONFIG.csv')
```

	Unnamed: 0	AES mean	AES std	MBF mean	MBF std	SR
0	Problema 1	NaN	NaN	76.440944	13.066541	0.0
1	Problema 2	NaN	NaN	0.520916	0.057359	0.0
2	Problema 3	NaN	NaN	5.739239	1.778175	0.0
3	Problema 4	NaN	NaN	24.683961	7.889679	0.0
4	Problema 5	NaN	NaN	4.504148	0.000000	0.0
5	Problema 6	NaN	NaN	1.460803	0.897694	0.0
6	Problema 7	NaN	NaN	5.703284	1.079057	0.0

Figura 3: Métricas del algoritmo en los 7 problemas con los parámetros por defecto

6.1 MAX_WRAPS: aumentar vs no aumentar

```
def MAX_WRAPS_2():
    config = default_config(MAX_WRAPS=2)
    results = []
    for nombre,f,_,D in problemas:
        results = results + [[nombre,
                              p_live(f, D, config, 4)]]
    df = print_metrics(results, config)
    df.to_csv('data/MAX_WRAPS_2.csv')
    return df

# MAX_WRAPS_2()
pd.read_csv('data/MAX_WRAPS_2.csv')
```

	Unnamed: 0	AES mean	AES std	MBF mean	MBF std	SR
0	Problema 1	NaN	NaN	202.176047	178.124694	0.0
1	Problema 2	NaN	NaN	2.243169	2.980641	0.0
2	Problema 3	NaN	NaN	6.141329	1.420357	0.0
3	Problema 4	NaN	NaN	5.659845	1.659585	0.0
4	Problema 5	NaN	NaN	4.504148	0.000000	0.0
5	Problema 6	NaN	NaN	1.765540	0.984108	0.0
6	Problema 7	NaN	NaN	5.703284	1.079057	0.0

Figura 4: Métricas del algoritmo en los 7 problemas con los parámetros por defecto excepto MAX_WRAPS=2

6.2 CODON_SIZE: aumentar vs no aumentar

```
def CODON_SIZE_127():
    config = default_config(CODON_SIZE=127)
    results = []
    for nombre,f,_,D in problemas:
        results = results + [[nombre,
                              p_live(f, D, config, 4)]]
    df = print_metrics(results, config)
    df.to_csv('data/CODON_SIZE_127.csv')
    return df

# CODON_SIZE_127()
pd.read_csv('data/CODON_SIZE_127.csv')
```

	Unnamed: 0	AES mean	AES std	MBF mean	MBF std	SR
0	Problema 1	NaN	NaN	138.315005	53.751085	0.00
1	Problema 2	NaN	NaN	0.448503	0.171879	0.00
2	Problema 3	NaN	NaN	6.235648	0.659585	0.00
3	Problema 4	NaN	NaN	27.260550	9.798821	0.00
4	Problema 5	NaN	NaN	4.498904	0.009084	0.00
5	Problema 6	10300.0	0.0	2.545330	3.227546	0.25
6	Problema 7	NaN	NaN	5.210351	1.132653	0.00

Figura 5: Métricas del algoritmo en los 7 problemas con los parámetros por defecto excepto CODON_SIZE=127

6.3 INITIAL_IND_LENGTH: aumentar vs no aumentar

```
def INITIAL_IND_LENGTH_50():
    config = default_config(INITIAL_IND_LENGTH=50)
    results = []
    for nombre,f,_,D in problemas:
        results = results + [[nombre,
                              p_live(f, D, config, 4)]]
    df = print_metrics(results, config)
```

```
    df.to_csv('data/INITIAL_IND_LENGTH_50.csv')
    return df

# INITIAL_IND_LENGTH_50()
pd.read_csv('data/INITIAL_IND_LENGTH_50.csv')
```

	Unnamed: 0	AES mean	AES std	MBF mean	MBF std	SR
0	Problema 1	NaN	NaN	120.754563	29.605520	0.0
1	Problema 2	NaN	NaN	6.526256	2.597845	0.0
2	Problema 3	NaN	NaN	7.175647	4.958697	0.0
3	Problema 4	NaN	NaN	22.573063	6.076698	0.0
4	Problema 5	NaN	NaN	8.655649	7.190610	0.0
5	Problema 6	NaN	NaN	2.204840	1.899545	0.0
6	Problema 7	NaN	NaN	5.699749	1.085179	0.0

Figura 6: Métricas del algoritmo en los 7 problemas con los parámetros por defecto excepto INITIAL_IND_LENGTH=50

```
def INITIAL_IND_LENGTH_100():
    config = default_config(INITIAL_IND_LENGTH=100)
    results = []
    for nombre,f,_,D in problemas:
        results = results + [[nombre,
                              p_live(f, D, config, 4)]]
    df = print_metrics(results, config)
    df.to_csv('data/INITIAL_IND_LENGTH_100.csv')
    return df

# INITIAL_IND_LENGTH_100()
pd.read_csv('data/INITIAL_IND_LENGTH_100.csv')
```

	Unnamed: 0	AES mean	AES std	MBF mean	MBF std	SR
0	Problema 1	NaN	NaN	127.321797	48.427209	0.0
1	Problema 2	NaN	NaN	0.526056	0.291813	0.0
2	Problema 3	NaN	NaN	4.514631	1.653491	0.0
3	Problema 4	NaN	NaN	22.893938	9.922468	0.0
4	Problema 5	NaN	NaN	4.209435	0.510458	0.0
5	Problema 6	NaN	NaN	1.097086	0.021654	0.0
6	Problema 7	NaN	NaN	6.326278	0.000000	0.0

Figura 7: Métricas del algoritmo en los 7 problemas con los parámetros por defecto excepto INITIAL_IND_LENGTH=100

6.4 EVAL_PENALTY: disminuir vs no disminuir

```
def EVAL_PENALTY_1000():
    config = default_config(EVAL_PENALTY=1e3)
    results = []
    for nombre,f,_,D in problemas:
```

```
        results = results + [[nombre,
                              p_live(f, D, config, 4)]]
    df = print_metrics(results, config)
    df.to_csv('data/EVAL_PENALTY_1000.csv')
    return df

# EVAL_PENALTY_1000()
pd.read_csv('data/EVAL_PENALTY_1000.csv')
```

	Unnamed: 0	AES mean	AES std	MBF mean	MBF std	SR
0	Problema 1	NaN	NaN	118.353203	49.229141	0.00
1	Problema 2	NaN	NaN	0.511694	0.294401	0.00
2	Problema 3	NaN	NaN	7.265695	0.269585	0.00
3	Problema 4	NaN	NaN	19.806252	5.178271	0.00
4	Problema 5	NaN	NaN	8.309719	6.591441	0.00
5	Problema 6	15200.0	0.0	1.217874	0.900386	0.25
6	Problema 7	NaN	NaN	5.171454	1.163150	0.00

Figura 8: Métricas del algoritmo en los 7 problemas con EVAL_PENALTY=1000

6.5 UNDECODABLE_PENALTY: disminuir vs no disminuir

```
def UNDECODABLE_PENALTY_1000():
    config = default_config(UNDECODABLE_PENALTY=1e3)
    results = []
    for nombre,f,_,D in problemas:
        results = results + [[nombre,
                              p_live(f, D, config, 4)]]
    df = print_metrics(results, config)
    df.to_csv('data/UNDECODABLE_PENALTY_1000.csv')
    return df

# UNDECODABLE_PENALTY_1000()
pd.read_csv('data/UNDECODABLE_PENALTY_1000.csv')
```

	Unnamed: 0	AES mean	AES std	MBF mean	MBF std	SR
0	Problema 1	NaN	NaN	102.765417	27.996020	0.00
1	Problema 2	7800.0	0.0	0.356413	0.221934	0.25
2	Problema 3	NaN	NaN	6.993765	5.036116	0.00
3	Problema 4	NaN	NaN	20.331532	8.722510	0.00
4	Problema 5	NaN	NaN	7.625984	1.625984	0.00
5	Problema 6	16100.0	0.0	2.867688	3.118105	0.25
6	Problema 7	NaN	NaN	6.326278	0.000000	0.00

Figura 9: Métricas del algoritmo en los 7 problemas con UNDECODABLE_PENALTY=1000

6.6 POPULATION_SIZE: aumentar vs no aumentar

```python
def POPULATION_SIZE_200():
    config = default_config(POPULATION_SIZE=200)
    results = []
    for nombre,f,_,D in problemas:
        results = results + [[nombre,
                              p_live(f, D, config, 4)]]
    df = print_metrics(results, config)
    df.to_csv('data/POPULATION_SIZE_200.csv')
    return df

# POPULATION_SIZE_200()
pd.read_csv('data/POPULATION_SIZE_200.csv')
```

	Unnamed: 0	AES mean	AES std	MBF mean	MBF std	SR
0	Problema 1	NaN	NaN	4.584878	2.365487	0.00
1	Problema 2	NaN	NaN	0.617010	0.241911	0.00
2	Problema 3	NaN	NaN	4.622505	1.210324	0.00
3	Problema 4	NaN	NaN	19.049894	2.621906	0.00
4	Problema 5	NaN	NaN	4.487781	0.028348	0.00
5	Problema 6	11800.0	0.0	1.663882	1.221455	0.25
6	Problema 7	NaN	NaN	5.829720	0.860062	0.00

Figura 10: Métricas del algoritmo en los 7 problemas con POPULATION_SIZE=200

6.7 GENERATION_SIZE: aumentar vs no aumentar

```python
def GENERATION_SIZE_200():
    config = default_config(GENERATION_SIZE=200)
    results = []
    for nombre,f,_,D in problemas:
        results = results + [[nombre,
                              p_live(f, D, config, 4)]]
    df = print_metrics(results, config)
    df.to_csv('data/GENERATION_SIZE_200.csv')
    return df

# GENERATION_SIZE_200()
pd.read_csv('data/GENERATION_SIZE_200.csv')
```

	Unnamed: 0	AES mean	AES std	MBF mean	MBF std	SR
0	Problema 1	NaN	NaN	96.015671	28.857168	0.0
1	Problema 2	NaN	NaN	0.628210	0.081713	0.0
2	Problema 3	NaN	NaN	4.237838	1.170980	0.0
3	Problema 4	NaN	NaN	21.909495	2.990251	0.0
4	Problema 5	NaN	NaN	4.498904	0.009084	0.0
5	Problema 6	NaN	NaN	1.009452	0.188136	0.0
6	Problema 7	NaN	NaN	6.326278	0.000000	0.0

Figura 11: Métricas del algoritmo en los 7 problemas con GENERATION_SIZE=200

6.8 MUTATION_PROBABILITY: aumentar vs no aumentar

```
def MUTATION_PROBABILITY_0_25():
    config = default_config(MUTATION_PROBABILITY=0.25)
    results = []
    for nombre,f,_,D in problemas:
        results = results + [[nombre,
                              p_live(f, D, config, 4)]]
    df = print_metrics(results, config)
    df.to_csv('data/MUTATION_PROBABILITY_0_25.csv')
    return df

# MUTATION_PROBABILITY_0_25()
pd.read_csv('data/MUTATION_PROBABILITY_0_25.csv')
```

	Unnamed: 0	AES mean	AES std	MBF mean	MBF std	SR
0	Problema 1	NaN	NaN	161.177005	25.765716	0.0
1	Problema 2	NaN	NaN	0.760196	0.204378	0.0
2	Problema 3	NaN	NaN	5.321286	0.000000	0.0
3	Problema 4	NaN	NaN	43.898807	6.275042	0.0
4	Problema 5	NaN	NaN	8.655649	7.190610	0.0
5	Problema 6	NaN	NaN	4.300223	2.679289	0.0
6	Problema 7	NaN	NaN	9.622902	5.709921	0.0

Figura 12: Métricas del algoritmo en los 7 problemas con MUTATION_PROBABILITY=0.25

6.9 CROSSOVER_PROBABILITY: aumentar vs no aumentar

```
def CROSSOVER_PROBABILITY_0_25():
    config = default_config(CROSSOVER_PROBABILITY=0.25)
    results = []
    for nombre,f,_,D in problemas:
        results = results + [[nombre,
                              p_live(f, D, config, 4)]]
    df = print_metrics(results, config)
    df.to_csv('data/CROSSOVER_PROBABILITY_0_25.csv')
    return df

# CROSSOVER_PROBABILITY_0_25()
pd.read_csv('data/CROSSOVER_PROBABILITY_0_25.csv')
```

	Unnamed: 0	AES mean	AES std	MBF mean	MBF std	SR
0	Problema 1	NaN	NaN	1.458545	0.124578	0.0
1	Problema 2	NaN	NaN	0.465524	0.231369	0.0
2	Problema 3	NaN	NaN	5.379464	2.088075	0.0
3	Problema 4	NaN	NaN	34.225103	14.449560	0.0
4	Problema 5	NaN	NaN	4.504148	0.000000	0.0
5	Problema 6	NaN	NaN	1.535667	0.735963	0.0
6	Problema 7	NaN	NaN	6.326278	0.000000	0.0

Figura 13: Métricas del algoritmo en los 7 problemas con CROSSOVER_PROBABILITY=0.25

6.10 WITHIN_USED: True vs False

```
def WITHIN_USED_False():
    config = default_config(WITHIN_USED=False)
    results = []
    for nombre,f,_,D in problemas:
        results = results + [[nombre,
                              p_live(f, D, config, 4)]]
    df = print_metrics(results, config)
    df.to_csv('data/WITHIN_USED_False.csv')
    return df

# WITHIN_USED_False()
pd.read_csv('data/WITHIN_USED_False.csv')
```

	Unnamed: 0	AES mean	AES std	MBF mean	MBF std	SR
0	Problema 1	11700.0	0.0	152.176390	87.850418	0.25
1	Problema 2	NaN	NaN	1.085447	1.276805	0.00
2	Problema 3	NaN	NaN	5.321286	0.000000	0.00
3	Problema 4	NaN	NaN	22.737546	8.408634	0.00
4	Problema 5	NaN	NaN	6.549064	3.876771	0.00
5	Problema 6	NaN	NaN	1.627905	0.780074	0.00
6	Problema 7	NaN	NaN	6.326278	0.000000	0.00

Figura 14: Métricas del algoritmo en los 7 problemas con WITHIN_USED=False

6.11 TOURNAMENT_SIZE: aumentar vs no aumentar¶

```
def TOURNAMENT_SIZE_5():
    config = default_config(TOURNAMENT_SIZE=5)
    results = []
    for nombre,f,_,D in problemas:
        results = results + [[nombre,
                              p_live(f, D, config, 4)]]
    df = print_metrics(results, config)
    df.to_csv('data/TOURNAMENT_SIZE_5.csv')
    return df

# TOURNAMENT_SIZE_5()
pd.read_csv('data/TOURNAMENT_SIZE_5.csv')
```

	Unnamed: 0	AES mean	AES std	MBF mean	MBF std	SR
0	Problema 1	NaN	NaN	27.244712	13.364633	0.00
1	Problema 2	NaN	NaN	0.419992	0.205983	0.00
2	Problema 3	NaN	NaN	6.141329	1.420357	0.00
3	Problema 4	NaN	NaN	14.164393	3.692284	0.00
4	Problema 5	NaN	NaN	4.504148	0.000000	0.00
5	Problema 6	18300.0	0.0	4.514585	0.215154	0.25
6	Problema 7	NaN	NaN	6.326278	0.000000	0.00

Figura 15: Métricas del algoritmo en los 7 problemas con TOURNAMENT_SIZE=5

6.12 ELITE_SIZE: aumentar vs no aumentar¶

```
def ELITE_SIZE_5():
    config = default_config(ELITE_SIZE=5)
    results = []
    for nombre,f,_,D in problemas:
        results = results + [[nombre,
                              p_live(f, D, config, 4)]]
    df = print_metrics(results, config)
    df.to_csv('data/ELITE_SIZE_5.csv')
    return df

# ELITE_SIZE_5()
pd.read_csv('data/ELITE_SIZE_5.csv')
```

	Unnamed: 0	AES mean	AES std	MBF mean	MBF std	SR
0	Problema 1	NaN	NaN	0.265986	0.569563	0.0
1	Problema 2	NaN	NaN	0.545314	0.310666	0.0
2	Problema 3	NaN	NaN	5.577757	2.525620	0.0
3	Problema 4	NaN	NaN	3.265985	0.326595	0.0
4	Problema 5	NaN	NaN	4.451345	0.091458	0.0
5	Problema 6	NaN	NaN	1.204962	0.409646	0.0
6	Problema 7	NaN	NaN	6.103429	0.385985	0.0

Figura 16: Métricas del algoritmo en los 7 problemas con ELITE_SIZE=5

6.13 Comparación de resultados y selección final de parámetros

Observando los resultados de los experimentos parciales anteriores (consistentes en modificar un sólo parámetro cada vez), podemos constatar:

```
names = ['DEFAULT_CONFIG.csv',
         'CODON_SIZE_127.csv',
         'CROSSOVER_PROBABILITY_0_25.csv',
         'ELITE_SIZE_5.csv',
         'EVAL_PENALTY_1000.csv',
         'GENERATION_SIZE_200.csv',
```

```
          'INITIAL_IND_LENGTH_100.csv',
          'INITIAL_IND_LENGTH_50.csv',
          'MAX_WRAPS_2.csv',
          'MUTATION_PROBABILITY_0_25.csv',
          'POPULATION_SIZE_200.csv',
          'TOURNAMENT_SIZE_5.csv',
          'UNDECODABLE_PENALTY_1000.csv',
          'WITHIN_USED_False.csv']

mbf_means=[]
mbf_stds=[]
srs_means=[]
for name in names:
    df = pd.read_csv('data/' + name)
    mbf_means = mbf_means + [df['MBF mean'].mean()]
    mbf_stds = mbf_stds + [df['MBF std'].mean()]
    srs_means = srs_means + [df['SR'].mean()]

pd.DataFrame(data={'MBF means': mbf_means,
                   'MBF stds': mbf_stds,
                   'SR means': srs_means},
             index=names)
```

	MBF means	MBF stds	SR means
DEFAULT_CONFIG.csv	17.007614	3.538358	0.000000
CODON_SIZE_127.csv	29.713107	11.348511	0.035714
CROSSOVER_PROBABILITY_0_25.csv	8.739364	2.917494	0.000000
ELITE_SIZE_5.csv	3.576561	0.744675	0.000000
EVAL_PENALTY_1000.csv	25.561699	10.559465	0.035714
GENERATION_SIZE_200.csv	19.232264	4.756762	0.000000
INITIAL_IND_LENGTH_100.csv	23.841317	8.689585	0.000000
INITIAL_IND_LENGTH_50.csv	27.843919	8.469375	0.000000
MAX_WRAPS_2.csv	37.088919	30.764809	0.000000
MUTATION_PROBABILITY_0_25.csv	33.390867	6.832137	0.000000
POPULATION_SIZE_200.csv	6.045132	1.030668	0.035714
TOURNAMENT_SIZE_5.csv	9.800142	3.113876	0.035714
UNDECODABLE_PENALTY_1000.csv	23.273515	7.515781	0.071429
WITHIN_USED_False.csv	27.974845	14.598957	0.035714

Figura 17: Métricas medias obtenidas en los 7 problemas para cada configuración individual probada

La tabla anterior nos conduce a pensar que las siguientes modificaciones en la configuración por defecto **conducen a mejores resultados**:

- **CODON_SIZE: incrementar a 127 mejora la SR**
- **CROSSOVER_PROBABILITY: incrementar a 0.25 mejora la MBF**
- **ELITE_SIZE: incrementar a 5 mejora la MBF**
- **EVAL_PENALTY: disminuir a 1000 mejora la SR**
- **POPULATION_SIZE: incrementar a 200 mejora la SR y la MBF**
- **TOURNAMENT_SIZE: incrementar a 5 mejora la MBF y la SR**
- **UNDECODABLE_PENALTY: disminuir a 1000 mejora la SR**

- **WITHIN_USED: establecer a False mejora la SR**

Las demás modificacines que hemos probado no parecen ofrecer, de forma aislada, mejores resultados que los de la configuración por defecto.

Con estos datos podemos definir la configuración "óptima" con la que continuaremos el resto de experimentos de esta práctica.

```
FINAL_CONFIG = default_config(CODON_SIZE=127,
                              CROSSOVER_PROBABILITY=0.25,
                              ELITE_SIZE=5,
                              EVAL_PENALTY=1000,
                              POPULATION_SIZE=200,
                              TOURNAMENT_SIZE=5,
                              UNDECODABLE_PENALTY=1000,
                              WITHIN_USED=False,
                              MAX_GENERATIONS=500)
```

7 Ejecución con la configuración "óptima" en los 7 problemas

```
import pickle


def run_full(config):
    results = []
    for nombre, f, _, D in problemas:
        print nombre
        results = results + [[nombre, p_live(f, D, config, 10)]]
    with open('data/full_experiment_500_10.pkl', 'wb') as output:
        pickle.dump(results, output, pickle.HIGHEST_PROTOCOL)
    df = print_metrics(results, config)
    df.to_csv('data/full_experiment_500_10.csv')


# run_full(FINAL_CONFIG)
```

Como esta ejecución del experimento es bastante pesada guardo los resultados en **'data/full_experiment_500_10.pkl'**, de manera que puedo trabajar con ellos sin tener que re-ejecutar todo.

Con estos resultados podemos ver los **plots de progreso para cada problema**. Nótese que normalmente se analizan los plots de progreso para determinar si una u otra configuración del algoritmo ofrece mejores resultados a no. Así lo he hecho en las secciones anteriores de determinación de mejores parámetros, aunque no lo haya adjuntado en esta práctica. Sin embargo, **muestro estos plots ahora en esta sección, sencillamente por curiosidad sobre el comportamiento del algoritmo con la configuración final en cada uno de los 7 problemas**.

Por cierto, **limito el máximo de generaciones a 500 porque no tengo máquina suficiente para experimentos en otro orden de computación** (por ejemplo, 2000 generaciones). Según las pruebas que he estado haciendo, 500 generaciones parecen suficientes para mostrar que el algoritmo evoluciona en el buen sentido en los distintos problemas.

Nota: el archivo 'full_experiment_500_10.pkl' es demasiado grande para entregarlo vía la plataforma aLF. Si se quiere consultar puede obtenerse aquí: https://github.com/amanas/ce/blob/master/p3/data/full_experiment_500_10.pkl.

```
def get_full():
    with open('data/full_experiment_500_10.pkl', 'rb') as input:
        return pickle.load(input)

i=18
for name, histories in get_full():
    log_plot(histories,title='Figura' + str(i) + \
             ': plots de progreso para el ' + name)
    i += 1
```

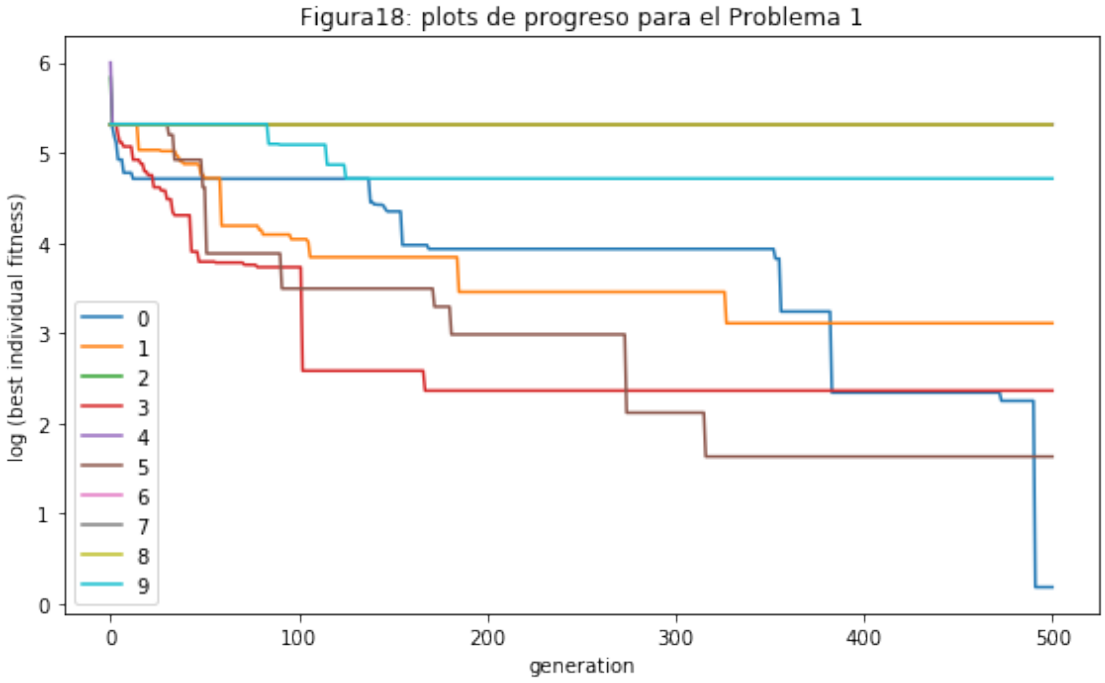


Figura19: plots de progreso para el Problema 2

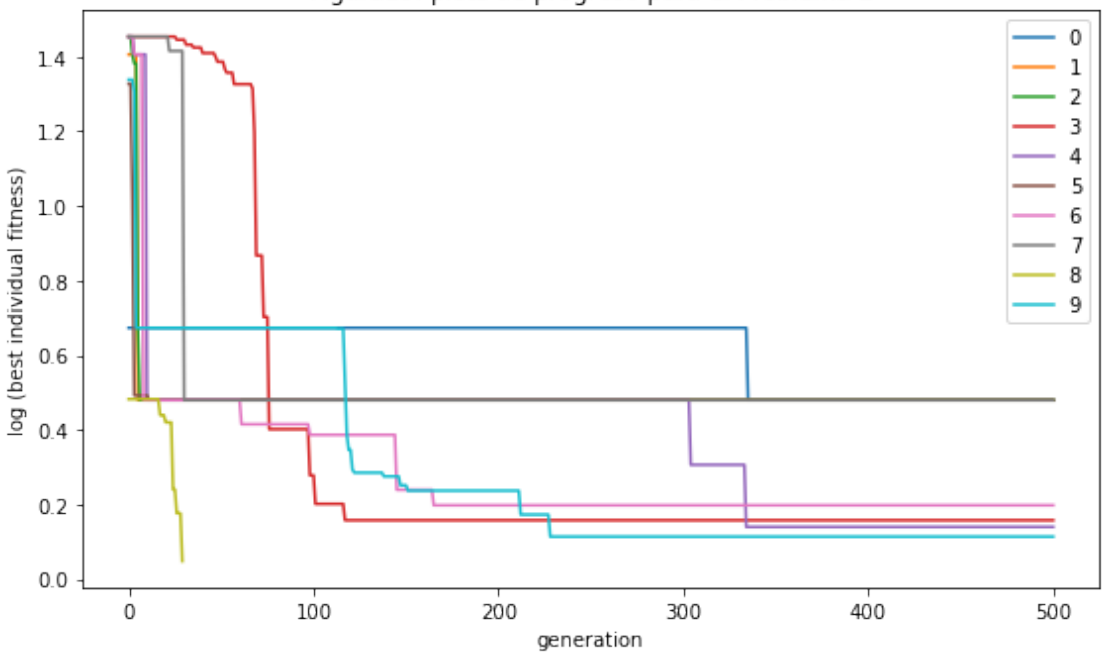


Figura20: plots de progreso para el Problema 3

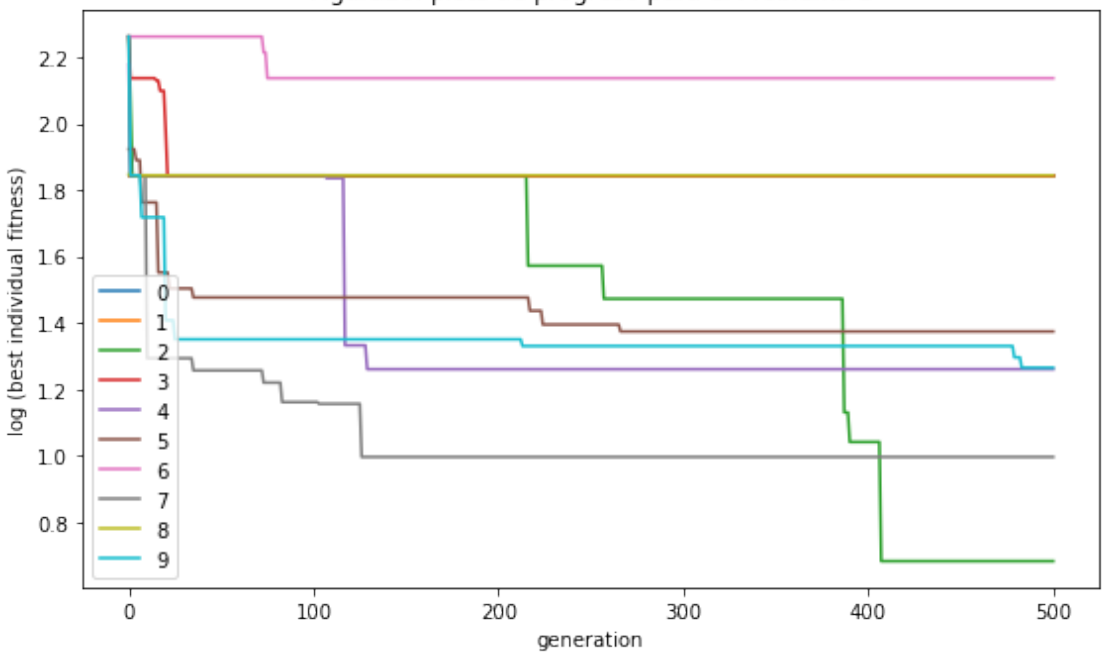


Figura21: plots de progreso para el Problema 4

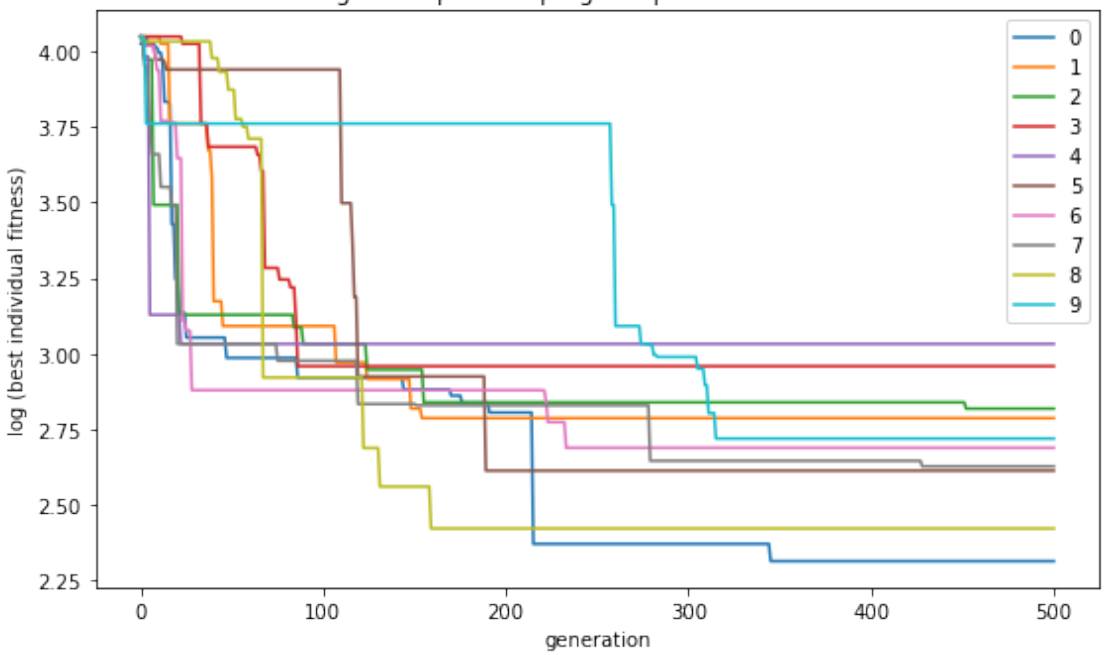


Figura22: plots de progreso para el Problema 5

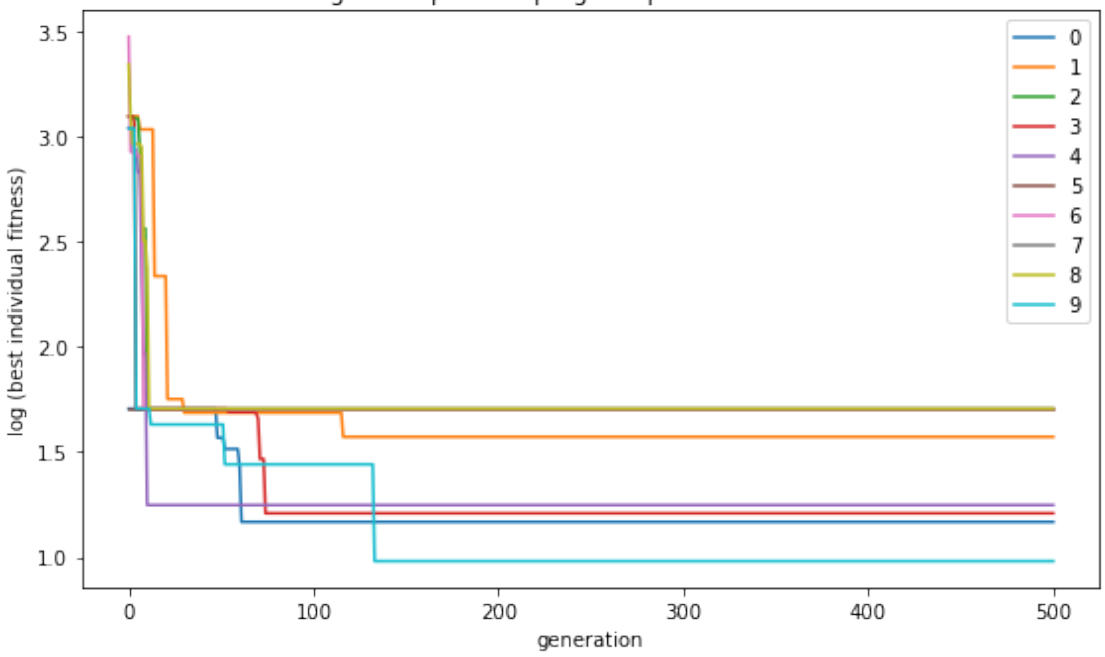


Figura23: plots de progreso para el Problema 6

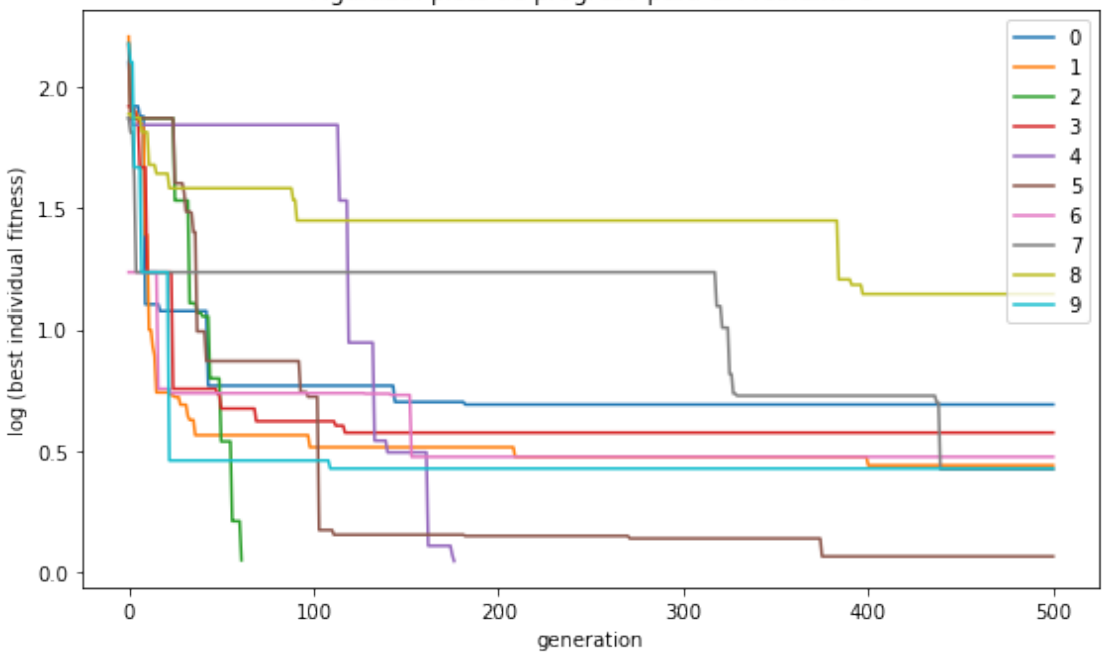
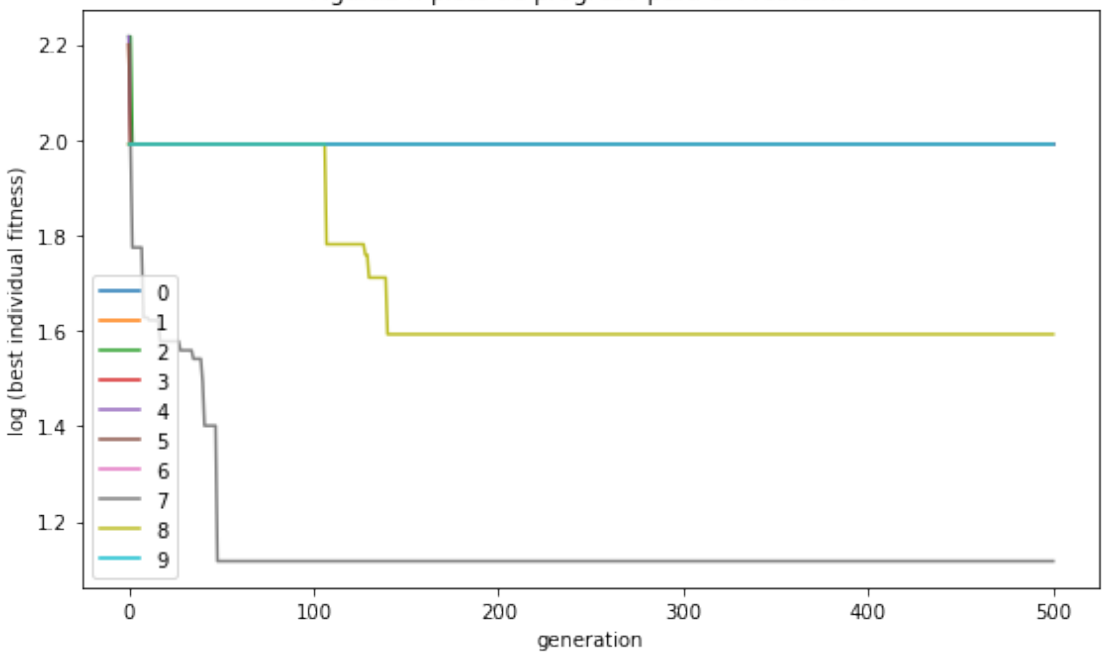


Figura24: plots de progreso para el Problema 7



8 Resultados y soluciones encontradas

Cada uno de los 7 problemas anteriores se ha atacado con 10 ejecuciones del algoritmo, compartiendo la configuración que se encontró en el punto anterior.

Ésta es la tabla de métricas resultantes para cada problema:

```
pd.read_csv('data/full_experiment_500_10.csv')
```

	Unnamed: 0	AES mean	AES std	MBF mean	MBF std	SR
0	Problema 1	NaN	NaN	141.238389	91.005918	0.0
1	Problema 2	3000.0	0.0	0.404015	0.235114	0.1
2	Problema 3	NaN	NaN	3.945871	1.969500	0.0
3	Problema 4	NaN	NaN	14.565040	3.199974	0.0
4	Problema 5	NaN	NaN	11.265345	1.598654	0.0
5	Problema 6	11950.0	5750.0	1.265986	1.598654	0.2
6	Problema 7	NaN	NaN	5.760762	1.301774	0.0

Figura 25: Resultados del experimento con la configuración "óptima" para todos los problemas.

Para determinar la mejor solución encontrada para cada problema nos quedaremos con el mejor individuo de la mejor ejecución de las 4 por problema. Este individuo, para cada problema, podremos finalmente representarlo junto con la derivada que se supone que aproxima.

Aquí tenemos los mejores individuos encontrados para cada problema:

```
best_found = {}
for name, histories in get_full():
    best_found[name] = min([min(h) for h in histories])

for n in sorted(best_found):
    print best_found[n]
```

```
Individual: exp((cos(x-x)/sin(1.0+1.0)))*(x*(x+x)); 0.197317349366;
            False
Individual: exp((cos(sin(1.0))/(x+1.0)-x)); 0.0494328298681; True
Individual: cos(1.0)*((sin(1.0*x)*1.0-1.0+x)*x); 0.978014157915; False
Individual: exp((x+(cos(x)+cos(x))+(x*cos(x)))); 9.12467473737; False
Individual: exp(sin(1.0))*x***x*x; 1.66642756071; False
Individual: 1.0+log((1.0+log(cos(1.0))**1.0+x*cos(cos(1.0))));
            0.0481099423371; True
Individual: exp(x)+x*(x+1.0)*cos(x)*1.0; 2.05754877701; False
```

9 Representación de las soluciones encontradas y las esperadas

NÓTESE que para las soluciones que alcanzan un hit en todos los puntos de muestreo, la aproximación encontrada de la derivada no se vé en las gráficas porque queda oculta por la representación de la función derivada objetivo

```
import inspect


def src(f):
    return str(inspect.getsourcelines(f)[0][0] \
               .split('return ')[1].split('\n')[0])


i = 26
for nombre, f, g, D in problemas:
    t = np.linspace(D[0], D[1], 400)
    best = best_found[nombre]
    fen = eval('lambda x: ' + best.phenotype)
    plt.figure(figsize=(10, 6))
    plt.plot(t,[f(x) for x in t],'m',
             label='Funcion: ' + src(f))
    plt.plot(t,[g(x) for x in t],'g',
             label='Derivada: ' + src(g))
    plt.plot(t,[fen(x) for x in t],'b',
             label='Aproximada: ' + best.phenotype)
    plt.title('Figura %s: encontrada vs esperada para %s' % (i,nombre))
    plt.legend()
    plt.ylabel('y')
    plt.xlabel('x')
    plt.tight_layout()
    plt.show()
    i += 1
```

Figura 26: encontrada vs esperada para Problema 1

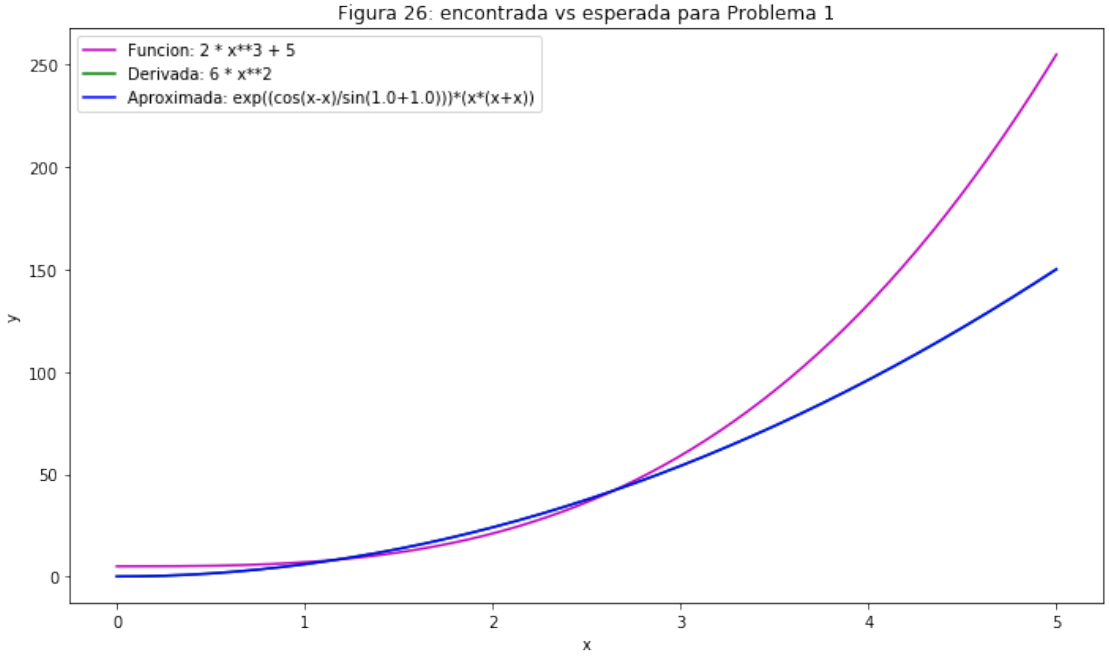


Figura 27: encontrada vs esperada para Problema 2

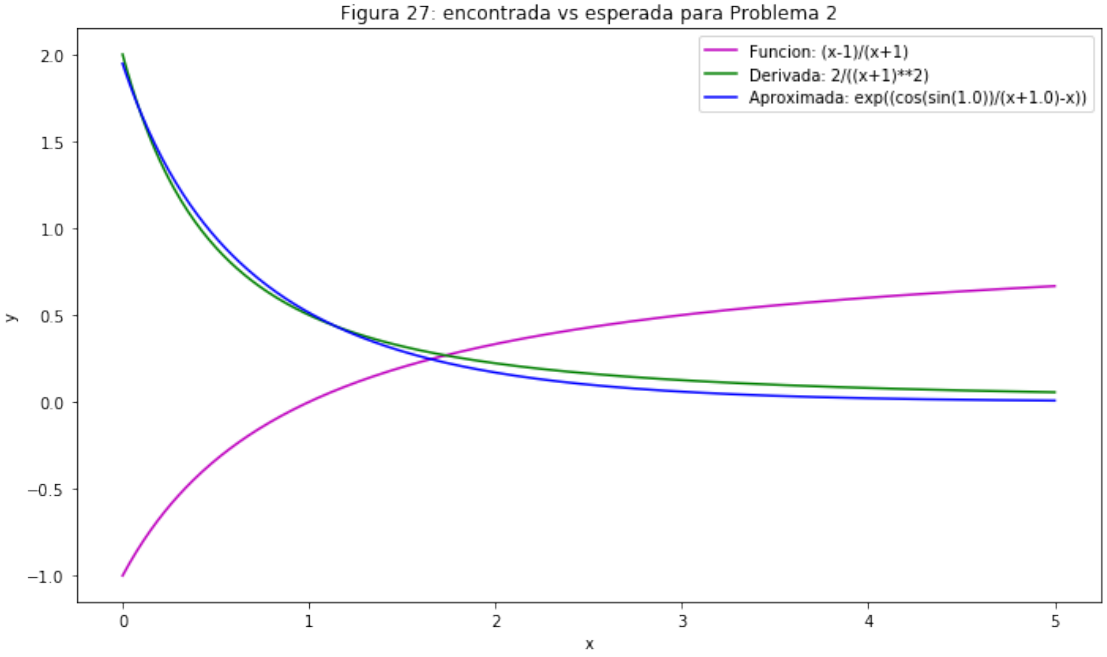


Figura 28: encontrada vs esperada para Problema 3

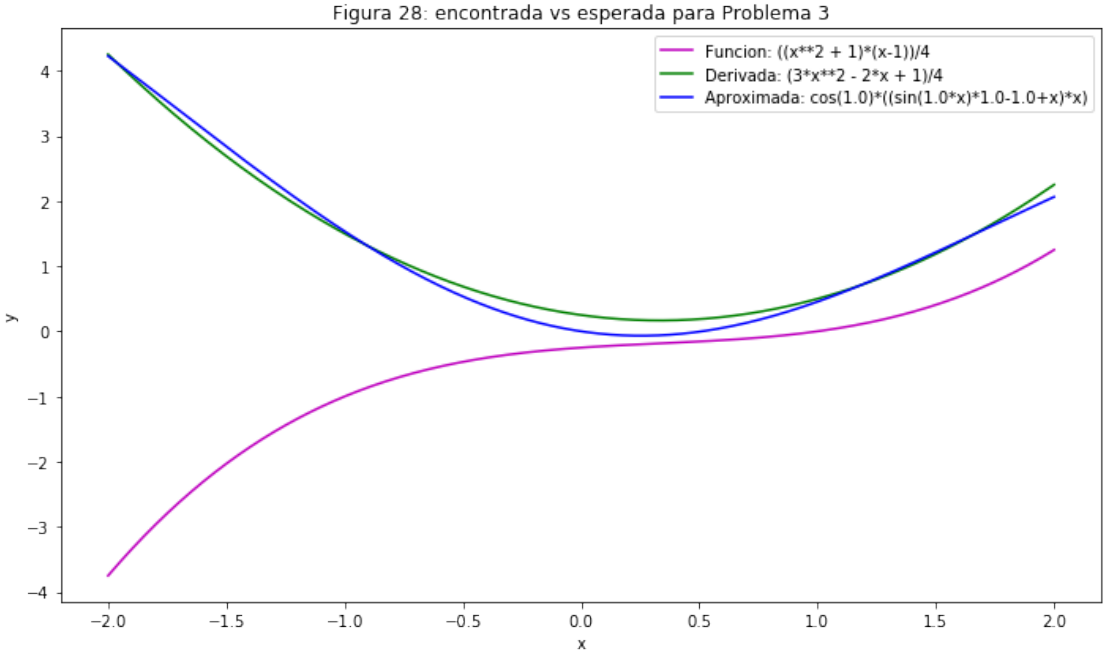


Figura 29: encontrada vs esperada para Problema 4

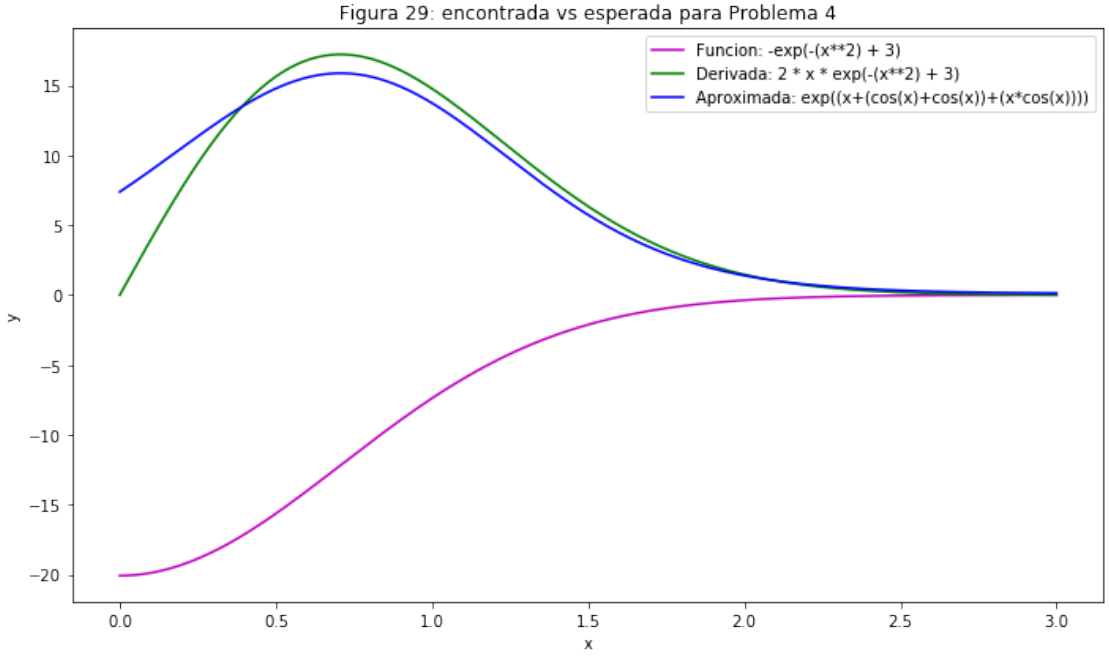


Figura 30: encontrada vs esperada para Problema 5

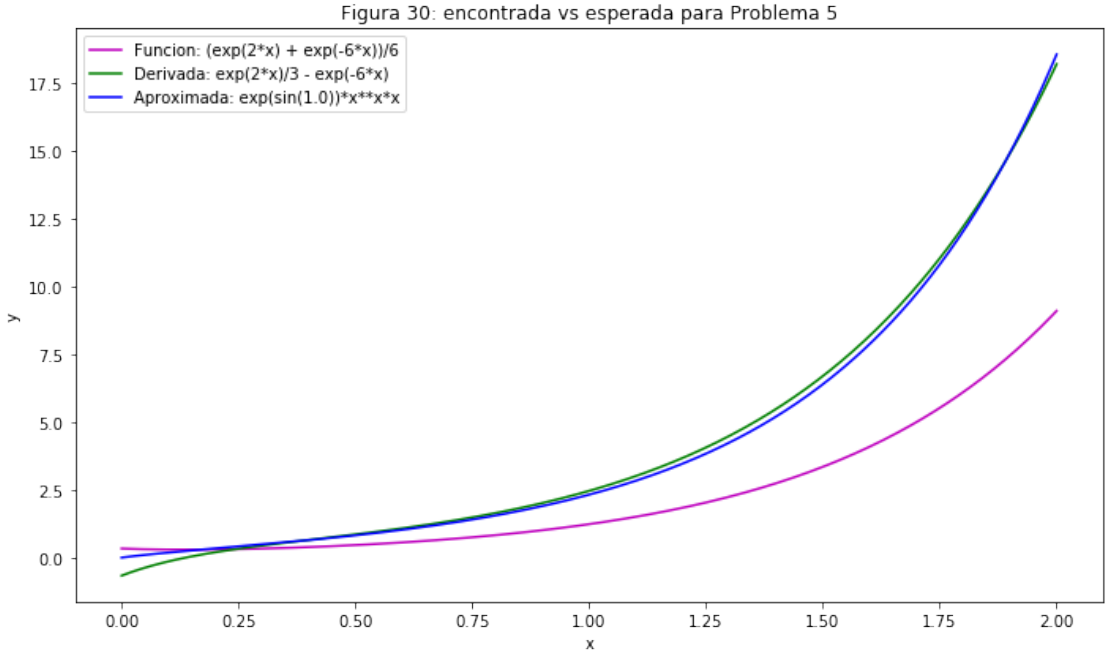


Figura 31: encontrada vs esperada para Problema 6

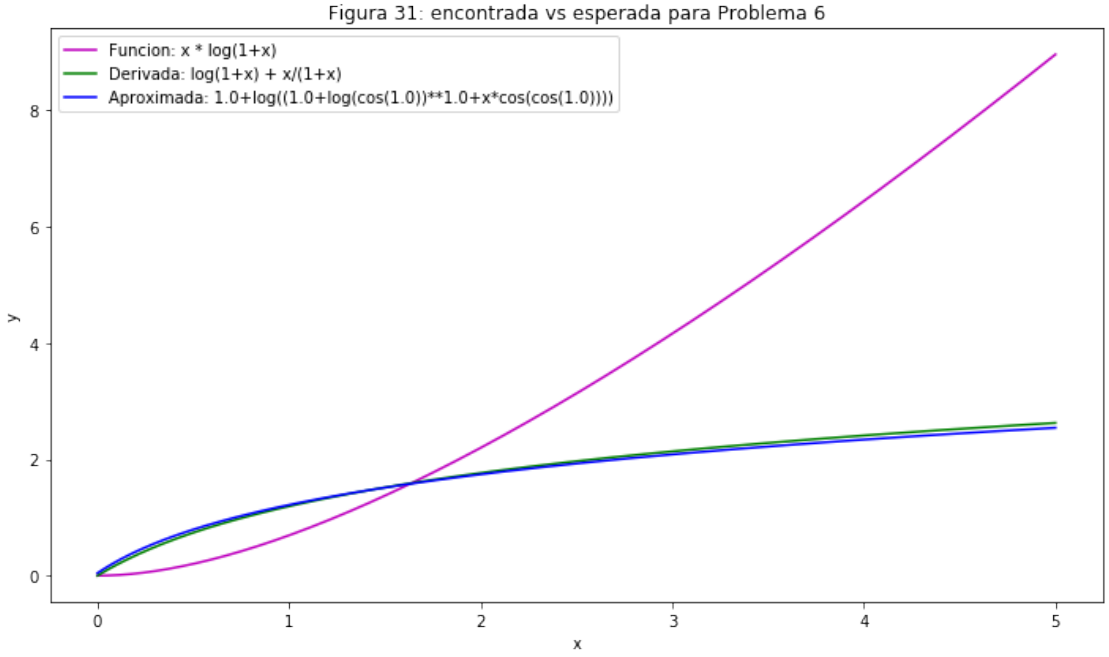
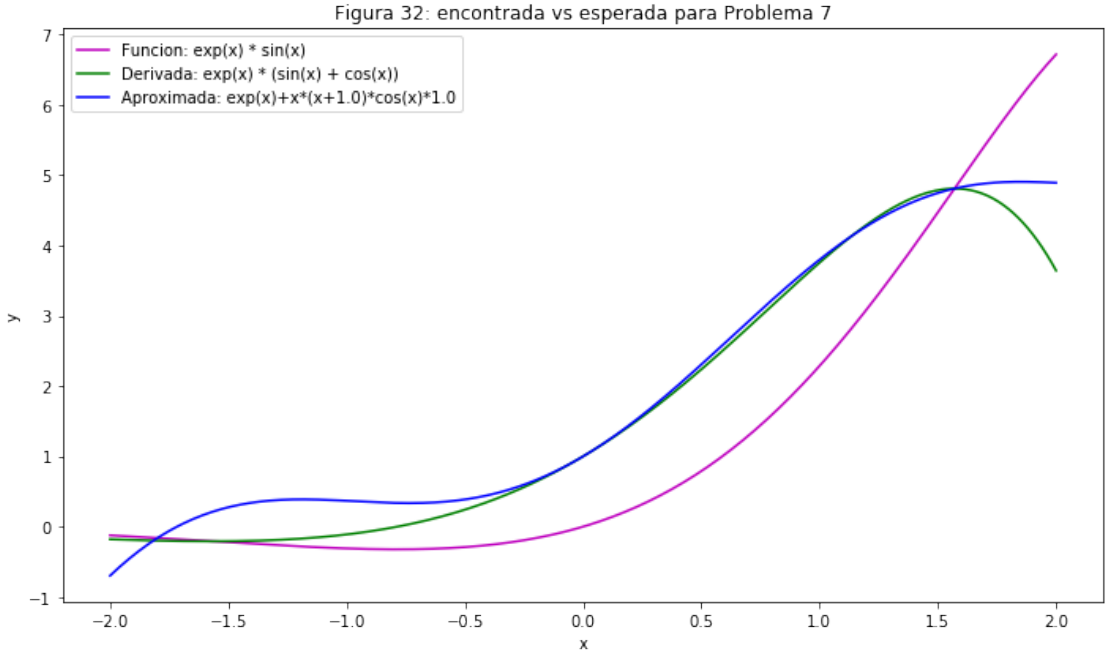


Figura 32: encontrada vs esperada para Problema 7



10 Conclusiones

Me tomo la libertad de añadir observaciones del todo subjetivas en esta sección en lugar de remitirme únicamente a comentar los resultados objetivos obtenidos.

Después de realizar esta práctica tengo la sensación de que la evolución gramatical resuelve problemas que no son nada triviales y que, ahora mismo, no sabría muy bien de qué otro modo atacar. Ha sido todo un descubrimiento este tipo de técnicas.

En particular, el concepto básico de gramática era desconocido para mí en el modo como se ha tratado en este trabajo y confieso que me resulta de lo más interesante. La representación de gramáticas en nomenclatura Backus-Naur ha sido también un conocimiento del todo nuevo y muy interesante.

Entiendo que los resultados que he obtenido seguramente son bastante mejorables. No obstante, en sólo 500 generaciones se han resuelto con éxito 2 de los 7 problemas. Como dije en otro punto, he limitado el número de generaciones por cuestión de los recursos informáticos de los que dispongo.

En cuanto a la mejor configuración de parámetros que he utilizado para atacar de forma común los 7 problemas entiendo que también podría afinarse más. Como indicaba más arriba, el algoritmo que he presentado aquí permite configurar 17 parámetros. Y la forma de seleccionarlos ha sido investigar aisladamente uno a uno cómo afectan al rendimiento general en los 7 problemas. Claro, es normal pensar que el hecho de que el rendimiento mejore cuando se modifican dos parámetros de forma aislada no necesariamente implique que el rendimiento mejore cuando esos parámetros se modifican de forma conjunta. Ésta es una crítica perfectamente válida al modo de búsqueda de parámetros que he hecho. Pero, ciertamente, con tanto grado de libertad y el tiempo que le he dedicado a esta práctica (bastante más del estimado inicialmente) me ha parecido que lo mejor era dar por buena la configuración encontrada.

La gramática que se usa he observado que es un factor importantísimo para el éxito de la búsqueda de soluciones. De hecho, pronto me dí cuenta que el algoritmo tenía grandes dificultades en aproximar polinomios de grado 3 o superior. Sin embargo, al añadir el operador exponencial (**) a la gramática, este tipo de problemas empezaron a resolverse con mayor facilidad.

Muy resaltable es también la forma como se aproxima la derivada en un punto. Tal y como han comentado varios compañeros en el foro de la asignatura, si se toma como épsilon para aproximar la derivada en un punto la longitud del intervalo dividida por el número de puntos de muestreo, los resultados que se obtienen son muy malos y el algoritmo casi nunca converge. He observado que épsilons del orden de 0.001 funcionan mucho mejor.

Y por lo demás, creo que la mejor manera de tener una idea de los resultados obtenidos es observar las gráficas del punto anterior. Los resultados no son perfectos pero si que me resultan bastante satisfactorios. De hecho, es bastante curioso observar como funciones que se definen de formas tan sumamente diferentes a veces pueden acabar correspondiéndose con curvas tan similares. Por ejemplo, en la Figura 27, que se corresponde con el problema 6, tenemos:

- función: $ x * log(1+x) $
- derivada: $log(1 + x) + x/(1 + x)$
- aproximación: $1.0 + log((1.0 + log(cos(1.0)) * *1.0 + x * cos(cos(1.0))))$

Curiosidades como ésta son las que me invitan a pensar que la evolución gramatical, en el contexto de problemas como el investigado en esta práctica, es una herramienta de trabajo muy potente.