# Laboratory of Evolutionary Algorithms

Laboratory 2:
**Evolution strategies**

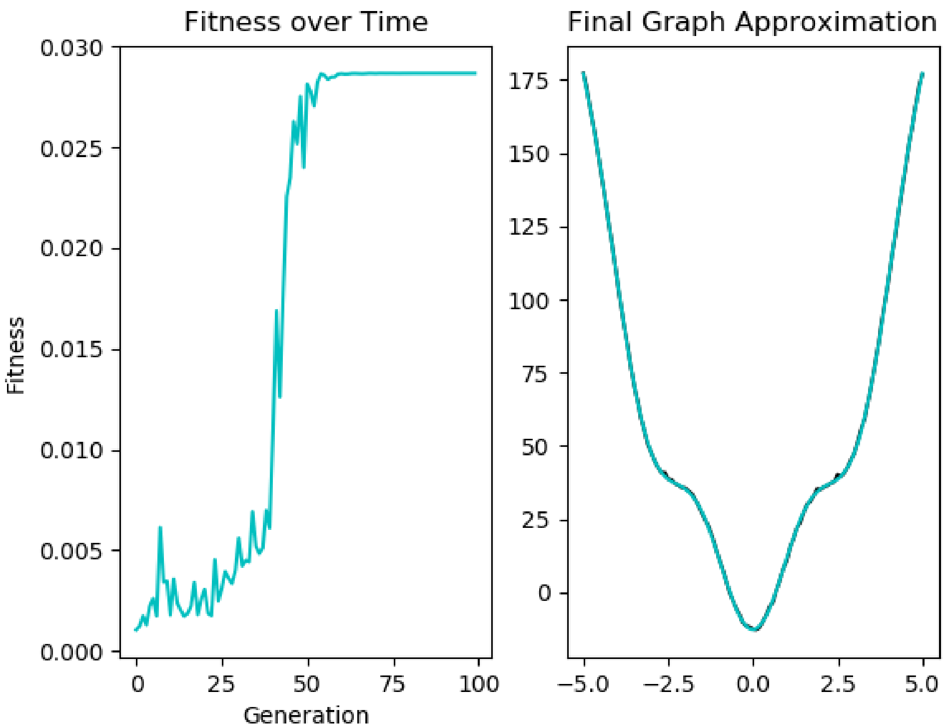
Author:
Agata Raczyńska

Tutor:
Robert Czabański, PhD

## 1. Tasks

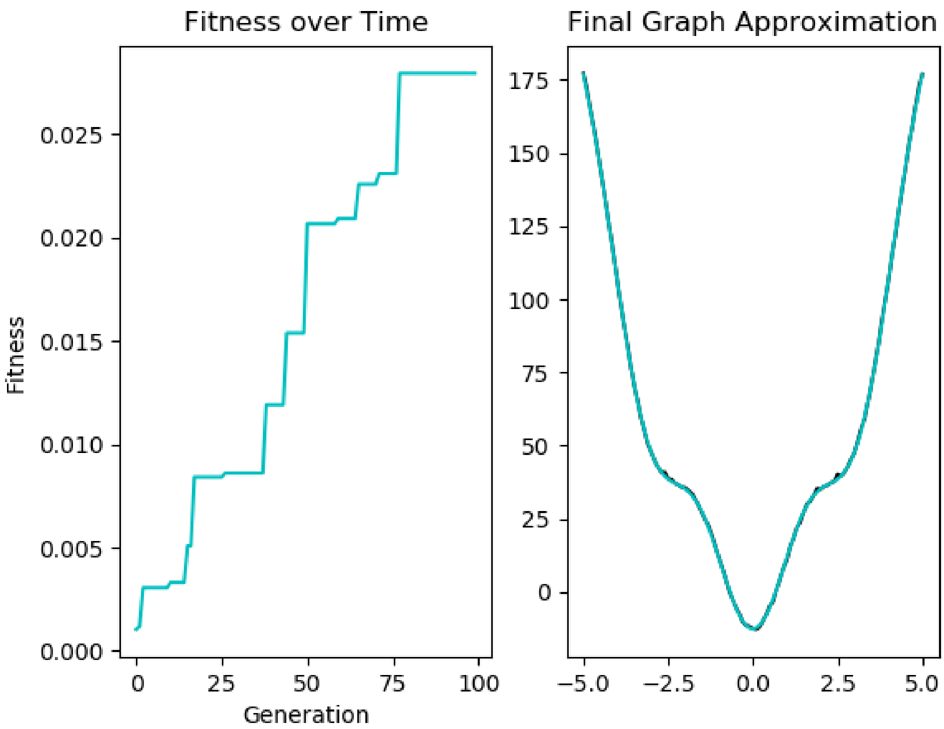
### 1.1. Write a computer program to solve the optimization problem provided by a tutor using the Evolution Strategy. Implement both, (μ, λ) and (μ + λ) approaches. As the population varying operator only mutation should be used.

- (μ, λ) approach:




Fitness of the winner: 0.0287
Time of calculations: 356.1019

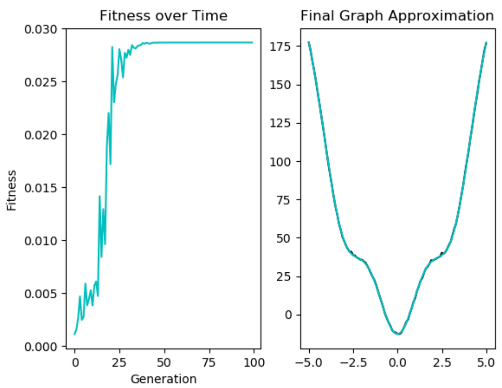
- (μ + λ) approach:



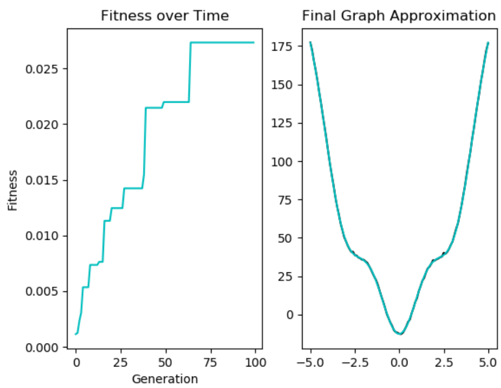

Fitness of the winner: 0.0280
Time of calculations: 363.3503

**1.2. Implement ES crossover operators: the discrete and intermediate crossover. Evaluate the influence of the ES parameters (number of individuals, number of offspring, selection and crossover method, ...) on the performance and the time of computations.**

- **intermediate crossover:**



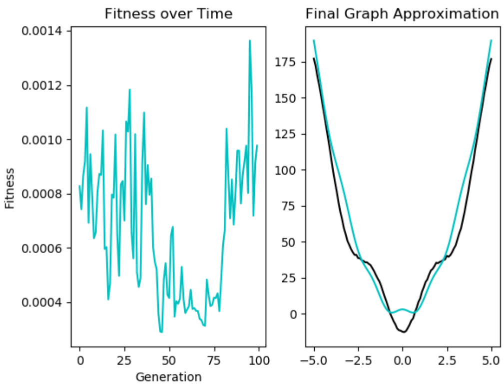

(μ,λ) approach
1μ:10 λ number of population
(100 parents, 1000 offspring)



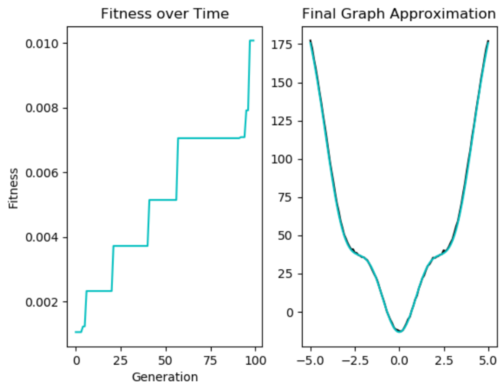

(μ + λ) approach
1μ:10 λ number of population
(100 parents, 1000 offspring)

Fitness of the winner: 0.02869
Time of calculations: 348.50

Fitness of the winner: 0.0273
Time of calculations: 384.60




$(\mu,\lambda)$ approach
$1\mu:1\ \lambda$ number of population
(100 parents, 100 offspring)

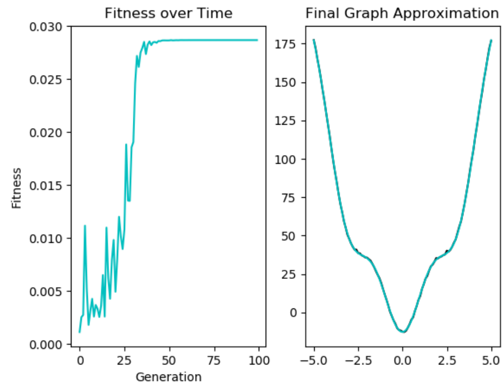



$(\mu + \lambda)$ approach
$1\mu:1\ \lambda$ number of population
(100 parents, 100 offspring)

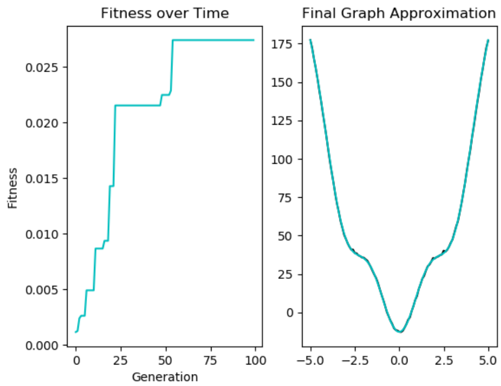
Fitness of the winner: 0.0010
Time of calculations: 38.24

Fitness of the winner: 0.0101
Time of calculations: 39.59

**- discrete crossover:**



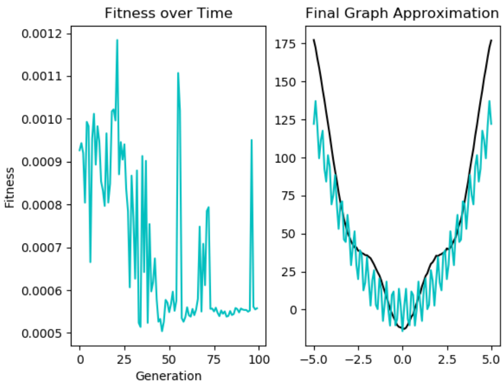

$(\mu,\lambda)$ approach
$1\mu:10\ \lambda$ number of population
(100 parents, 1000 offspring)



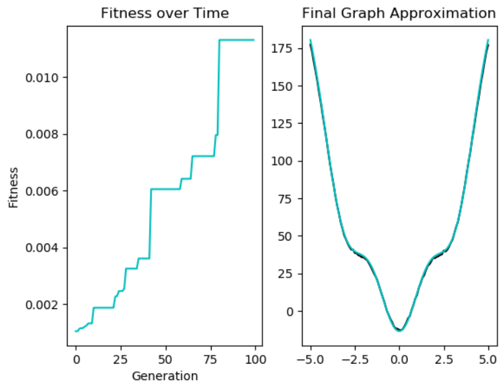

$(\mu + \lambda)$ approach
$1\mu:10\ \lambda$ number of population
(100 parents, 1000 offspring)

Fitness of the winner: 0.0287
Time of calculations: 349.03

Fitness of the winner: 0.0274
Time of calculations: 367.50




$(\mu,\lambda)$ approach




$(\mu + \lambda)$ approach

| 1μ:1 λ number of population (100 parents, 100 offspring) | 1μ:1 λ number of population (100 parents, 100 offspring) |
|---|---|
| Fitness of the winner: 0.0006<br>Time of calculations: 43.84 | Fitness of the winner: 0.0113<br>Time of calculations: 39.97 |

## 2. Conclusions

**Task1**

Implementation of evolution strategies enabled to fit function parameters to the given dataset. Both (μ, λ) and (μ+ λ) approaches fit the parameters of the function, although for (μ, λ) fitness value was oscillating, whereas for (μ+λ) approach fitness value was rising in steps.

**Task2**

Application of μ, λ approach did not lead to finding optimal solution in crossover methods when the number of parents and offspring population was 1μ:1λ. This type of population (1μ:1 λ) did not lead to finding optimal solution even with μ + λ approach. Best results were obtained for population 1μ:10λ, but this approach strongly prolonged time of calculations (8-10 times). For μ,λ and μ+λ similar results were gathered and an optimal solution was found when population model was 1μ:10λ. Calculations with discrete crossover were a bit faster than with intermediate crossover.

## 3. Implemented code

```
from math import pi, cos, sqrt, exp
import matplotlib.pyplot as plt
import pandas as pd
import random
import time
import numpy as np
random.seed(123)

#Functions

def Individual():
    abc = [random.uniform(-10, 10),random.uniform(-10, 10),random.uniform(-10,
10),]
    abc_var = [random.uniform(0, 10),random.uniform(0, 10),random.uniform(0, 10),]
    individual = np.zeros((1, 4), dtype=list, order='C')
    individual[0, 0] = abc
    individual[0, 1] = abc_var
    individual[0, 2] = random.gauss(0, 1) #r
    #individual[0, 3] = eval(abc) #fitness
    return individual

def Error(abc):
    err = 1 / sum([abs(Y[i] - origin_function(abc[0], abc[1], abc[2], X[i])) for i
in range(len(X))])
    return err

def origin_function(a, b, c, x):
    o_roof = a * ((x ** int(2)) - (b * cos(c * pi * x)))
    return o_roof

def init_pop(pop_size):
    pop = np.zeros((pop_size, 4), dtype=list, order='C')
```

```
    for i in range(pop_size):
        pop[i] = Individual()
    return pop

def eval_pop(pop):
    for i in range(len(pop)):
        pop[i,3] = Error(pop[i,0])

def u_lambda(mi, parents, offspring):
    next_gen = offspring[offspring[:,3].argsort()]
    next_gen = np.flip(next_gen, 0)
    print("Fitness: ", next_gen[0,3])
    return next_gen

def u_plus_lambda(mi, parents, offspring):
    xx0 = np.concatenate((parents, offspring), axis=0, out=None)
    xx = xx0[xx0[:,3].argsort()]
    xx = np.flip(xx, 0)
    print("Fitness: ", xx[0,3])
    return xx

def discrete_CX(pop, mi):
    x = 0
    off = np.zeros((mi, 4), dtype=list, order='C')
    for i in range(mi):
        p1 = pop[x]
        p2 = pop[x-1]
        r_select = [p1 if random.uniform(0, 1) > 0.5 else p2 for _ in range(6)]
        (off[i,0]) =
[((r_select[0])[0])[0],((r_select[1])[0])[1],((r_select[2])[0])[2]]
        (off[i,1]) =
[((r_select[3])[1])[0],((r_select[4])[1])[1],((r_select[5])[1])[2]]
        x += 1
        if x >= pop_size:
            x -= pop_size
    return off

def intermediate_CX(pop, mi):
    x = 0
    off = np.zeros((mi, 4), dtype=list, order='C')
    for i in range(mi):
        p1 = pop[x]
        p2 = pop[x-1]
        abc_CX = [((p1[0])[0]+(p2[0])[0])/2 , ((p1[0])[1]+(p2[0])[1])/2 ,
((p1[0])[2]+(p2[0])[2])/2] #abc from p1 and p2
        abc_var_CX = [((p1[1])[0] + (p2[1])[0]) / 2, ((p1[1])[1] + (p2[1])[1]) / 2,
                     ((p1[1])[2] + (p2[1])[2]) / 2]  # abc_var from p1 and p2
        off[i,0] = abc_CX
        off[i,1] = abc_var_CX
        x += 1
        if x >= pop_size:
            x -= pop_size
    return off

def gen_offspring(pop, mi):
    x = 0
    off = np.zeros((mi, 4), dtype=list, order='C')
    for i in range(mi):
        p = pop[x]
        off[i,0] = [((p[0])[0] + random.gauss(0, 1) * (p[1])[0]), #a + rand*var.a
                    ((p[0])[1] + random.gauss(0, 1) * (p[1])[1]),
                    ((p[0])[2] + random.gauss(0, 1) * (p[1])[2])]
        off[i,1] = [((p[1])[0] *exp(p[2]*tau1) * exp(random.gauss(0, 1) * tau2)),
#var.a*exp(r*tau1)*exp(rand*tau2)
                    ((p[1])[1] * exp(p[2] * tau1) * exp(random.gauss(0, 1) *
tau2)),
                    ((p[1])[2] * exp(p[2] * tau1) * exp(random.gauss(0, 1) *
tau2))]
```

```
        x += 1
        if x >= pop_size:
            x -= pop_size
    return off

#Task#########################################################################

start = time.time()

#Variables
t = 0
Tmax = 100
pop_size = 100
mi = 1000
data = pd.read_csv("model1.txt", delimiter = "  ", header=None, engine='python')
data.columns = ["X", "Y"]
X = data.X
Y = data.Y
tau1 = 1 / sqrt(2 * pop_size)
tau2 = 1 / sqrt(2 * sqrt(pop_size))

err = []
pop = init_pop(pop_size)
eval_pop(pop)

while t < Tmax:

    #Select offspring generation method:

    #offspring = intermediate_CX(pop, mi)
    #offspring = discrete_CX(pop, mi)
    #offspring = pop
    offspring = gen_offspring(pop, mi)

    eval_pop(offspring)

    # Select u,lambda or u+lambda approach

    #pop = u_lambda(mi, pop, offspring)
    pop = u_plus_lambda(mi, pop, offspring)

    t = t + 1
    err.append(pop[0,3])
    print("Iteration ", t)

yy = []
p = pop[0]
for x in X:
    yy.append(origin_function(p[0][0], p[0][1], p[0][2], x))

end = time.time()

print("Time: " , str(end - start))
print("Fitness: " , err[-1])
print()

#Plots

plt.subplot(1, 2, 1)
plt.title("Fitness over Time")
plt.plot(err, 'c-')
plt.ylabel('Fitness')
plt.xlabel('Generation')
plt.subplot(1, 2, 2)
plt.title("Final Graph Approximation")
plt.plot(X, Y, 'k-')
plt.plot(X, yy, 'c-')
plt.show()
```