Problem 6

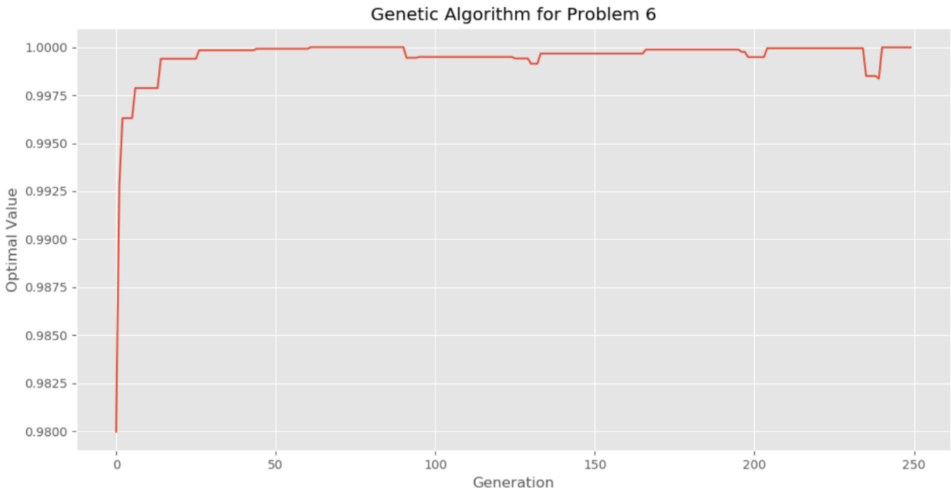
Maximize $f(x,y) = sin^2(5\pi(x^{3/4} - 0.1)) - (y-1)^4$ $2 \le x \le 4$; $-1 \le y \le 2$; $x+y \le 5$

Maximum=1 at (x,y)=(3.575,1)

Category	Item	Content
Parameters setting	Population Size	100
	Crossover Rate	90%
	Mutation Rate	10%
	Generation	**250** I think 250 is suitable for this problem. (compare to 1000 times, which only improve optimal value 10^{-7})
Composition of Chromosomes with precision being $\mathbf{10^{-3}}$		
Decision Variable X		11 bits
Decision Variable Y		12 bits
Total for a chromosome length		23 bits
Mechanisms		
Selection Mechanism		Roulette Wheel
Crossover Mechanism		One-point Crossover
Mutation Mechanism		Bit-by-bit base
Constraint handling		
Subject to	$2 \le x \le 4$; $-1 \le y \le 2$	Pre-censoring
	$x+y \le 5$	Penalty, which add a negative number
		Check Constraint post-crossover & post-mutation

The Final Optimization Results	
Optimal Value	0.9999996913865616 (For 1000 generation case: 0.9999997922565107)
Optimal for Decision Variable X	3.6912554958475816
Optimal for Decision Variable Y	1.221978021978022

The evolution history



Program Code:

Parameters setting and Pre-censoring of 2 Decision Variable domain

```
"""
Problem 6
@Aaron YuKu Chen
"""
import numpy as np
from matplotlib import pyplot as plt
import pandas as pd
import math as m
import random

population = 100  # set the number of chromosomes in each population
chs_length, x_length, y_length = 23, 11, 12  # by precision 10^-3 ;chromosomes total length for 23, and x is 11, y is 12
CR, MR = 0.9, 0.1  # Crossover Rate set 90% ; Mutation Rate set 10%
generation = 250  # set the number of generations
pi = m.pi  # set π , objective function will use it

XBound = [2, 4]  # pre-censoring for 2< x < 4
YBound = [-1, 2]  # pre-censoring for -1< y < 2
```

Objective Function

Also Check Constraint post-crossover & post- mutation.

Create a population of Initialization.

Set get fitness function and 2 transfer way make Binary to Decimal for X, Y.

```
def obj_function(x, y):  # check x + y <= 5 and then calculated the objective value
    if x + y <= 5:
        return m.sin(5 * pi * (x ** (3 / 4) - 0.1)) ** 2 - (y - 1) ** 4
    else:
        return -99  # if not , give the value -99


def initial():  # for first step, set up the 100(chromosomes) * 23(the length of each) as initial population
    pop = np.random.randint(0, 2, size=(population, chs_length))  # generate the random binary number of each bit
    return pop


def get_fitness(result):
    return result + 1e-5 - np.min(result)  # Convert the calculated objective value to a positive value
    # (plus a small value to avoid a negative denominator)


def transfer_x(x): # Binary to Decimal for x
    x_outcome = x.dot(2 ** np.arange(x_length)[::-1]) / float(2 ** x_length - 1) * (XBound[1] - XBound[0]) + XBound[
        0]
    return x_outcome


def transfer_y(y):  # Binary to Decimal for y
    y_outcome = y.dot(2 ** np.arange(y_length)[::-1]) / float(2 ** y_length - 1) * (YBound[1] - YBound[0]) + YBound[
        0]
```

Roullete Wheel Selection

Set 2 indexes for 2 parents.

The indexes choose one from population size, which depends on their proportion of total fitness.

Do Crossover

Set 2 temporary list to assist exchange.

```
# Roulette Wheel Selection
def selection(pop, fitness):
    # print("p", len(fitness))
    index01 = np.random.choice(a=100, size=1, replace=False, p=fitness / sum(fitness))
    # select a random number as index no.1, which chosen 1 from 100 by their fitness proportion
    parent01 = pop[index01]  # determine which chromosome as the parent no.1
    index02 = np.random.choice(a=100, size=1, replace=False, p=fitness / sum(fitness))
    # select a random number as index no.2, which chosen 1 from 100 by their fitness proportion
    parent02 = pop[index02]  # determine which chromosome as the parent no.2
    return parent01, parent02, index01, index02


# crossover
def crossover(parent01, parent02):
    if np.random.rand() < CR:  # Determine whether DO the crossover or Not
        crossover_location = m.ceil(
            random.random() * (chs_length - 1))  # Determine the crossover location in chromosome
        temp_ch01, temp_ch02 = list(range(1, chs_length)), list(range(1, chs_length))
        # Create 2 temporary lists to assist the crossover process
        temp_ch01[crossover_location:] = parent02[0][crossover_location:]
        # Make the exchange chromosome section in parent no.2 transfer save in temporary no.1
        temp_ch02[crossover_location:] = parent01[0][crossover_location:]
        # Make the exchange chromosome section in parent no.1 transfer save in temporary no.2
        parent01[0][crossover_location:] = temp_ch01[crossover_location:]
        # Make the exchange chromosome section in temporary no.1 transfer save in parent no.1
        parent02[0][crossover_location:] = temp_ch02[crossover_location:]
        # ake the exchange chromosome section in temporary no.2 transfer save in parent no.2
    return [parent01, parent02]
```

Do Mutation

Check do mutate or not to do it using bit-by-bit.

```
# mutate
def mutate(offspring):  # bit-by-bit
    list_mut = []  # Create a list to save the after-process offsprings
    for i in range(2):  # for two parent in processing
        for point in range(chs_length):  # mutate bit-by-bit mechanism
            if np.random.rand() < MR:  # Determine each bit will do mutate or not
                offspring[i][0][point] = 1 if offspring[i][0][point] == 0 else 0  # make 0 to 1 ; or 1 to 0
        list_mut.append(offspring[i])  # add after-process and no mutated to list
    return list_mut
```

Executive

Realization the initialization.

For each chromosome, divide into x & y parts, then transfer into real value.

Do 50 times for selection, crossover and mutation to make offspring achieve 100.

Do the 1000 generations, and show the optimal value for each generations.

```
# the processing flow
popch = initial()
target = []  # record the Max value of each generation
for j in range(generation):
    result = []  # the chromosome values of generation
    fitness = []  # the fitness value from each chromosome
    # print("check_first t", len(fitness))
    for k in range(100):
        x_ = popch[k][:11]  # pick up the x part in chromosome
        y_ = popch[k][11:]  # pick up the y part in chromosome
        if transfer_x(x_) + transfer_y(y_) > 5:  # if x + y > 5 , give the penalty
            obj_values = obj_function(transfer_x(x_), transfer_y(y_)) - 100
        else:
            obj_values = obj_function(transfer_x(x_), transfer_y(y_))  # compute their objective value
        result.append(obj_values)  # add the objective value into the result list
        fitness.append(get_fitness(result[k]))  # make the result get fitness

    for i in range(int(population / 2)):  # make the offsprings achieve 100

        parent01, parent02, index01, index02 = selection(popch, fitness)  # select 2 parents
        offspring_list = crossover(parent01, parent02)  # the crossover stage
        list_mut = mutate(offspring_list)  # the mutation stage
        popch[index01] = list_mut[0]  # offspring replace their parent
        popch[index02] = list_mut[1]  # offspring replace their parent

    target.append(np.max(result))  # add max value of result to the target list

    print('target(Max Value):', np.max(target))  # print the max value of each generation
```

Print the evolution history

Optimal value and their decision variables value.

```
print("Max Value: ", np.max(target))  # print the final optimal value
print("optimal x:", transfer_x(x_))  # print the optimal value of decision variable x
print("optimal y:", transfer_y(y_))  # print the optimal value of decision variable y
target = pd.DataFrame(target)
plt.style.use('ggplot')
plt.plot(range(generation), pd.DataFrame.rolling(target, window=30, min_periods=1).max())
plt.xlabel("Generation")
plt.ylabel("Optimal Value")
plt.title("Genetic Algorithm for Problem 6")
plt.figure()
plt.show()
```