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
Tuning Parameters and Setting Condition	Inertia Weight	1.0
	Acceleration Constants (Full model)	Self confidence factor = 2
		Swarm confidence factor = 2
	Velocity Limit	K_V = **0.7**
	Swarm Size	**40**
	Termination Condition	Achieve Max **250** Iterations
Constraint Handling		
Penalty Static		MINUS 999
Adhere Strategy		Repair Outlier to Boundary X [2,4] Y [-1,2]
The Final Optimization Results		
Optimal Value		0.9999820659659432
Optimal for Decision Variable X		3.5750214245286918
Optimal for Decision Variable Y		1.0633541344148891

Using Wilcoxon rank sum test 混和排序 GA & PSO RANK

Part one : compare the performance in optimal value.

Compare Final Optimization Results

	1	2	3	4	5	6	7	8
GA Optimal	0.999976578	0.99999	0.9999746	0.9999959	0.9999997	0.999997	0.999997	0.999987137
PSO Optimal	0.999999137	0.999997	0.9999884	0.9989892	0.9999809	0.999974	0.999997	0.999999868
	9	10	11	12	13	14	15	
GA Optimal	0.999996	0.999998636	0.999994055	0.99999675	0.999997	0.999997	0.999952046	
PSO Optimal	0.999992	0.99999970758	0.999988493	0.99997665	0.999999	0.999999	0.999982066	

Rank of Wilcoxon Rank Sum Test

Rank	1	2	3	4	5	6	7	8
GA Optimal	5	13	4	16	30	24	23	9
PSO Optimal	26	21	10	1	7	3	18	30
	9	10	11	12	13	14	15	
GA Optimal	17	25	15	20	22	19	2	
PSO Optimal	14	29	12	6	28	27	8	

Hypothesis Test 1:

H_0: $\eta_1 \neq \eta_2$ V.S H_1: $\eta_1 = \eta_2$

CR:$\{Z < -Z_{\alpha/2}$ or $Z > Z_{\alpha/2}\}$

Because $n_1 > 10$, $n_2 > 10$ Approximate normalization (Z distribution)

$W_1 = 5+13+4+16+30+24+23+9+17+25+15+20+22+19+2 = 244$

$W_2 = 26+21+10+1+7+3+18+30+14+29+12+6+28+27+8 = 240$

$E(W_1) = n_1(n_1 + n_2 + 1)/2 = 232.5$

$Var(W_1) = n_1 n_2 (n_1 + n_2 + 1)/12 = 581.25$

$$Z = \frac{W1 - E(W1)}{\sqrt{Var(W1)}} = (244-232.5)/24.1091 = 0.47699$$

Reject H_0 at $\alpha = 0.05$ if $Z_0 > Z_{0.025} = 1.96$ or $Z_0 < -Z_{0.025} = -1.96$

Because $Z_0 = 0.47699$ not locate at critical region

Thus, do not reject H_0 Hypothesis Test 1 at $\alpha = 0.05$, For my 2 algorithms, there are no enough evidence to show that GA & PSO have significant different.

Hypothesis Test 2:

H_0: $\eta_1 >= \eta_2$ V.S H_1: $\eta_1 < \eta_2$

CR: $\{Z < -Z_\alpha\}$

$W_1 = 5+13+4+16+30+24+23+9+17+25+15+20+22+19+2 = 244$

$W_2 = 26+21+10+1+7+3+18+30+14+29+12+6+28+27+8 = 240$

Because $n_1 > 10$, $n_2 > 10$ Approximate normalization (Z distribution)

$E(W_1) = n_1(n_1 + n_2 + 1)/2 = 232.5$

$Var(W_1) = n_1\ n_2(n_1 + n_2 + 1)/12 = 581.25$

$$Z = \frac{W1 - E(W1)}{\sqrt{Var(W1)}} = (244-232.5)/24.1091 = 0.47699$$

Reject H_0 at $\alpha = 0.05$ if $Z_0 < -Z_{0.05} = -1.645$

Because $Z_0 = 0.47699$ not locate at critical region

Thus, do not reject H_0 Hypothesis Test 2 at $\alpha = 0.05$, For my 2 algorithms, there are no enough evidence to show that PSO performance significance greater than GA performance.

Part two: compare the performance in solution Time

Compare Solution Time

	1	2	3	4	5	6	7	8
GA TIME	8.952366	8.158956	6.0435960	9.7287270	6.3310800	6.2068670	6.7334180	9.7724420
PSO TIME	3.71808	3.91612	3.678529	4.826299	3.517157	3.816037	3.415198	3.652631
	9	10	11	12	13	14	15	
GA TIME	9.855072	8.947202	9.519951	9.183107	9.80964	8.506256	9.564501	
PSO TIME	3.944484	3.473737	3.722482	3.666855	3.435037	4.214749	4.189138	

Rank of Wilcoxon Rank Sum Test

	1	2	3	4	5	6	7	8
GA TIME	8	11	15	4	13	14	12	3
PSO TIME	23	20	24	16	27	21	30	26
	9	10	11	12	13	14	15	

GA TIME	1	9	6	7	2	10	5	
PSO TIME	19	28	22	25	29	17	18	

Hypothesis Test 1:

H_0: $\eta_1 \neq \eta_2$ V.S H_1: $\eta_1 = \eta_2$

CR:$\{Z < -Z_{\alpha/2}$ or $Z > Z_{\alpha/2}\}$

Because $n_1 > 10$, $n_2 > 10$ Approximate normalization (Z distribution)

$W_1 = 8+11+15+4+13+14+12+3+1+9+6+7+2+10+5 = 120$

$W_2 = 23+20+24+16+27+21+30+26+19+28+22+25+29+17+18 = 345$

$E(W_1) = n_1(n_1+ n_2+1)/2 = 232.5$

$Var(W_1) = n_1 n_2(n_1 + n_2 +1)/12 = 581.25$

$Z = \frac{W1 - E(W1)}{\sqrt{Var(W1)}}$ = (120-232.5)/24.1091 = -4.666

Reject H_0 at α= 0.05 if $Z_0 > Z_{0.025} = 1.96$ or $Z_0 < -Z_{0.025} = -1.96$

Because $Z_0 = -4.666$ ***locate at*** critical region

Thus, **Reject H_0** Hypothesis Test 1 at α= 0.05 , For my 2 algorithms, there are enough evidence to show that GA & PSO have significant different.

Hypothesis Test 2:

H_0: $\eta_1 >= \eta_2$ V.S H_1: $\eta_1 < \eta_2$

CR:$\{Z < -Z_\alpha\}$

Because $n_1 > 10$, $n_2 > 10$ Approximate normalization (Z distribution)

$W_1 = 8+11+15+4+13+14+12+3+1+9+6+7+2+10+5 = 120$

$W_2 = 23+20+24+16+27+21+30+26+19+28+22+25+29+17+18 = 345$

$E(W_1) = n_1(n_1+ n_2+1)/2 = 232.5$

$Var(W_1) = n_1 n_2(n_1 + n_2 +1)/12 = 581.25$

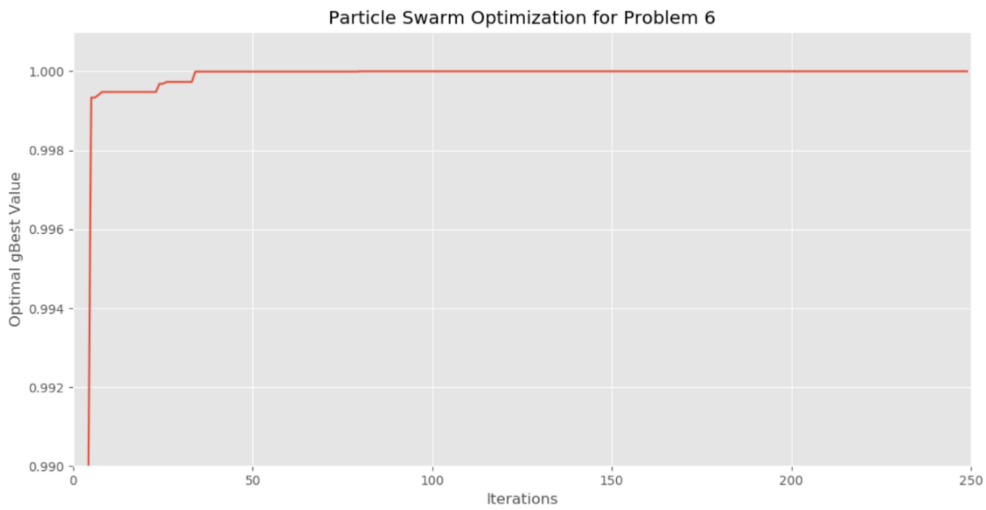
$Z = \frac{W1 - E(W1)}{\sqrt{Var(W1)}}$ = (120-232.5)/24.1091 = -4.666

Reject H_0 at α= 0.05 if $Z_0 < -Z_{0.05} = -1.645$

Because $Z_0 = -4.666$ ***locate at*** critical region

Thus, **Reject H_0** Hypothesis Test 2 at α= 0.05, For my 2 algorithms, there are enough evidence to show that compare THE SOULTION TIME PSO is superior to GA.

The evolution history



For the evolution history part, majority PSO convergence at 20~30 iterations But majority GA convergence at 50~70 iterations.

Program Code:

1

```
"""
Problem 6
@Aaron YuKu Chen M10801108
"""
import pandas as pd
import numpy as np
from numpy.random import randn
import random
import math as m
from matplotlib import pyplot as plt
import datetime

start_time = datetime.datetime.now()  # record time
pi = m.pi  # set π , objective function will use it
swarm_size = 40  # swarm size set 40
max_iterations = 250  # set max run 250 time iterations
x_domain = [2, 4]  # set the domain for decision variable X
y_domain = [-1, 2]  # set the domain for decision variable Y
initial_velocity_constraint = [-1, 1]  # set constraint for initial velocity
# set parameters
w = 1
r1, r2 = random.uniform(0, 1), random.uniform(0, 1)
c1, c2 = 2, 2
k_v = 0.7
```

2

```python
# set this function to generate initial swarm
def initialization():
    particles = []  # set empty list in order to save the value of particles
    velocity = []  # set empty list in order to save the value of velocity
    for i in range(swarm_size):  # compute the locations and velocities for each one
        s = random.uniform(0, 1)
        i_location = [
            s * (x_domain[1] - x_domain[0]) + x_domain[0], s * (y_domain[1] - y_domain[0]) + y_domain[0]]
        particles.append(i_location)  # compute the value of  locations, and set as array
        initialization_locations = np.array(particles)
        i_velocity = [
            s * (initial_velocity_constraint[1] - initial_velocity_constraint[0]) + initial_velocity_constraint[0],
            s * (initial_velocity_constraint[1] - initial_velocity_constraint[0]) + initial_velocity_constraint[0]]
        velocity.append(i_velocity)  # compute the value of velocities , and set as array
        initialization_velocities = np.array(velocity)
    return initialization_locations, initialization_velocities


def updateCurrentLoc(new_locations):
    current_locations = new_locations
    return current_locations  # make the new locations from iterations and transfer as current for next iterations
```

3

```python
def computeFitness(current_locations, pBest_fitness, pBest_locations, gBest_fitness,
                   gBest_location):  # compute the fitness then update pBest and determine gBest
    temp_fitnessValue = []
    for particle in range(swarm_size):  # compute the value of decision variable X.Y
        x_value = current_locations[particle][0]
        y_value = current_locations[particle][1]
        if x_value + y_value <= 5:  # Check the third constrains
            fitness_value = m.sin(5 * pi * (x_value ** (3 / 4) - 0.1)) ** 2 - (y_value - 1) ** 4
        else:  # if not follow third constraints , give penalty
            fitness_value = m.sin(5 * pi * (x_value ** (3 / 4) - 0.1)) ** 2 - (y_value - 1) ** 4 - 999

        temp_fitnessValue.append(fitness_value)
        current_fitness = temp_fitnessValue  # set it as current fitness value
    for order in range(swarm_size):  # update pBest
        if current_fitness[order] > pBest_fitness[order]:
            pBest_fitness[order] = current_fitness[order]  # if current one superior than pBest fitness, update it.
            pBest_locations[order] = current_locations[order]  # update their locations , too.
        else:
            pass
    for order_gBest in range(swarm_size):  # determine gBest
        if current_fitness[order_gBest] > gBest_fitness:
            gBest_fitness = current_fitness[order_gBest]  # if find someone better than gBest fitness, update it.
            gBest_location = current_locations[order_gBest]  # determine the location , too.
        else:
            pass  # if do not better than both of pBest or gBest, pass it.
    return pBest_fitness, pBest_locations, gBest_fitness, gBest_location
```

4

```python
def computeVelociteis(current_locations):  # compute the velocities
    for order_vel in range(swarm_size):
        current_velocities[order_vel][0] = w * current_velocities[order_vel][0] + r1 * c1 * (
                pBest_locations[order_vel][0] - current_locations[order_vel][0]) + r2 * c2 * (
                                                   gBest_location[0] - current_locations[order_vel][0])
        current_velocities[order_vel][1] = w * current_velocities[order_vel][1] + r1 * c1 * (
            pBest_locations[order_vel][1]) - \
                                           current_locations[order_vel][1] + r2 * c2 * (
                                                   gBest_location[1] - current_locations[order_vel][1])
        x_vMax = k_v * (x_domain[1] - x_domain[0])  # set v_max for X range
        y_vMax = k_v * (y_domain[1] - y_domain[0])  # set v_max for Y range
        if current_velocities[order_vel][0] > x_vMax:  # Velocity damping for x when exceed upper bound
            current_velocities[order_vel][0] = x_vMax
        if current_velocities[order_vel][0] < -x_vMax:  # Velocity damping for x when lower than bottom bound
            current_velocities[order_vel][0] = -x_vMax
        elif current_velocities[order_vel][1] > y_vMax:  # Velocity damping for y when exceed upper bound
            current_velocities[order_vel][1] = y_vMax
        elif current_velocities[order_vel][1] < -y_vMax:  # Velocity damping for y when lower than bottom bound
            current_velocities[order_vel][1] = -y_vMax
        else:
            pass
    new_velocities = current_velocities  # save current velocity as new velocity in order to update new locations
    print(new_velocities)
    return new_velocities
```

5

```python
def computeNewLocation(current_locations, new_velocities):  # update the outcome as new locations
    new_locations = current_locations + new_velocities
    return new_locations
```

6

```
# Run the PSO procedure
current_locations, current_velocities = initialization() # Generate the initial particle swarm

pBest_locations = current_locations  # set current locations as pBest location

initialization_fitnessValue = [] # set a empty list to save the fitness value
for particle in range(swarm_size):  # determine the initial fitness
    x_value = pBest_locations[particle][0]
    y_value = pBest_locations[particle][1]
    if x_value + y_value <= 5:
        fitness_value = m.sin(5 * pi * (x_value ** (3 / 4) - 0.1)) ** 2 - (y_value - 1) ** 4
    else:
        fitness_value = m.sin(5 * pi * (x_value ** (3 / 4) - 0.1)) ** 2 - (y_value - 1) ** 4 - 999  # give penalty
    initialization_fitnessValue.append(fitness_value)
pBest_fitness = initialization_fitnessValue  # initial fitness are the initial pBest fitness

gBest_fitness = max(pBest_fitness)  # select the max one as the initial gBest fitness
gBest_location = pBest_locations[pBest_fitness.index(gBest_fitness)]  # determine gBest location

new_locations = pBest_locations
history_gBest_fitness = []
```

7

```
for iteration in range(max_iterations):  # run the PSO
    current_locations = updateCurrentLoc(new_locations)
    pBest_fitness, pBest_locations, gBest_fitness, gBest_location = \
        computeFitness(current_locations, pBest_fitness, pBest_locations, gBest_fitness, gBest_location)
    history_gBest_fitness.append(gBest_fitness)  # record the history highest , in order to graphic it
    new_velocities = computeVelociteis(current_locations)
    new_locations = computeNewLocation(current_locations, new_velocities)
    for i in range(swarm_size):  # repair the locations if they are outlier
        if new_locations[i][0] < 2:
            new_locations[i][0] = 2
        elif new_locations[i][0] > 4:
            new_locations[i][0] = 4
        elif new_locations[i][1] < -1:
            new_locations[i][1] = -1
        elif new_locations[i][1] > 2:
            new_locations[i][1] = 2
        else:
            pass

print("Final gBest Value is:  ", gBest_fitness)  # print the final optimal value
print("Optimal X value: ", gBest_location[0])
print("Optimal Y value: ", gBest_location[1])
```

8

```
history_gBest_fitness = pd.DataFrame(history_gBest_fitness)
plt.style.use('ggplot')
plt.plot(range(max_iterations), pd.DataFrame.rolling(history_gBest_fitness, window=30, min_periods=1).max())
plt.xlim(0, 250)
plt.ylim(0.99, 1.001)

plt.xlabel("Iterations")
plt.ylabel("Optimal gBest Value")
plt.title("Particle Swarm Optimization for Problem 6")

plt.show()

end_time = datetime.datetime.now()
print(end_time - start_time).second
```