ASSIGNMENT #4

Computational Intelligence: PSO
(Particle Swarm Optimization)

Marwan Mohamed Nabil
20200512

Table of Contents

PART 1: Problem formulation & Objectives

1. The Problem

-We have a certain problem/ function: $f(X_1, X_2) = \sin(2X_1 - 0.5\pi) + 3\cos(X_2) + 0.5X_1$, Where:

- $-2 \leq X_1 \leq 3$
- $-2 \leq X_2 \leq 1$

Where $f(X_1, X_2)$ is a maximization problem.

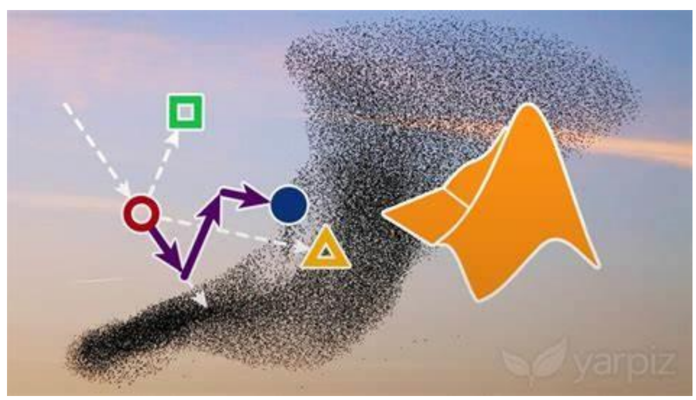
2. Objective:

-It is required to solve this problem and try to find its optimal solution using the PSO (Particle Swarm Optimization) algorithm, which is:

- The best value for both X_1 *and* X_2, which is the position of the particles.
- The value of the function at such positions

3. About PSO:

- Particle Swarm Optimization, or PSO, is a nature-inspired optimization algorithm that involves simulating a swarm of particles moving through a search space to find the optimal solution to a given problem.



-The algorithm starts by initializing a population of particles, each with a random position and velocity in the search space.

-The particles then move through the search space, updating their velocity and position according to a velocity update rule that combines their current velocity with a cognitive and social component.

-The cognitive component pulls the particle towards its best-known position, while the social component pulls the particle towards the best-known position in the swarm.

-The particles continue to move and update their velocity and position until a stopping criterion is met, such as a maximum number of iterations or a satisfactory level of fitness.

-The best-known position found by any particle in the swarm is recorded as the global best position, and the process is repeated for several iterations or until the desired level of accuracy is achieved.

-PSO is a metaheuristic algorithm that is widely used in optimization problems, such as in engineering design, machine learning, and data mining.

PART 2: System Overview

1. System components:

-Let's first look at our system environment:

System	Entity	Attributes	Activities	Events	State Variable

Maximization Function	Particle	Velocity
		
		Position
		
		Cognitive and Social Components
		
		Fitness
			Updating Particle's Position and Velocity
			
			Calculating Fitness
			
			Comparing Fitness Values
				Initializing Population
				
				Updating Values
				
				Reaching Optimal solution
					`global_bestFit`
					
					`p_g`

2. System Analysis:

-To solve this problem, we're going to use Python programming language on a Jupyter Notebook.

-We're using the NumPy library to ease some of the calculations and with initializing random numbers for the initial population.

-Here is a list of the variables used and their usage:

a. Initializing the initial population.:

```
def initpop(npop,x_max,x_min,v_max,dim):
```

-In this part of the code, we initialize the initial population using the NumPy library's random function, where x_id is the initial population positions and v_id is the initial population velocities.

-There is also a for loop that creates a list full of zeros of size 1*dim, which is used to indicate the lower bounds of the velocities of the particles which is zero, to be added as a parameter in the random function.

b. Calculating Fitness for each particle.

```
def fitCalc(x_i)
```

-In this part of the code, we calculate the fitness value for each particle, by substituting its X1 and X2 values (It's position) in the Objective function mentioned in the beginning of the report.

c. Comparing and Updating parameters according to fitness value.

-In this part, we use 2 functions to decide whether to update the fitness values or not:

```
def updatePid(x_i,x_fitness,p_i,particle_bestFit)
```

-In this function, we check the history of the particle's previous fitness's "particle_bestFit" and compare it to the new fitness "x_fitness".
-If it turns out that this new fitness is indeed the best new fitness across this particle, we assign it as the best new fitness value "particle_bestFit".

-Else, move on to the velocity-update function.

```
def updatePgd(p_i,particle_bestFit,p_g,global_bestFit)
```

-In this function, we check to see if the local best fitness value we just achieved "particle_bestFit" is better than the global best fitness value "global_bestFit".
-If so, assign this new value as the new global best value "global_bestFit".
Else, move on to the velocity-update function.

d. Updating Position and Velocity of Particle.

```
def updateVidXid(p_i,p_g,x_i,v_i,c_cog,c_soc,dim):
```

-Here, we update the values of the velocities for each particle using the equation:

$vij(t + 1) = vij(t) + c1r1j(t)[yij(t) - xij(t)] + c2r2j(t)[\hat{}yj(t) - xij(t)]$

-Where:

1. $vij(t)$ Previous Velocity (Inertia Component):

It can be seen as a momentum, which prevents the particle from drastically changing direction.

2. $c1r1j(t)[yij(t)$– xi(t)] Cognitive Component (Nostalgia):

Particles are drawn back to their own best positions, resembling the tendency of particles to return to places that satisfied them most in the past.

3. $c2r2j(t)[\hat{}yj(t) - xij(t)]$ Social Component (Envy):

Particle is drawn towards the best position found by the particle's neighborhood.

-We then calculate the new positions by adding the new velocity to the old positions vector.

e. Main PSO Function.

```
def PSO(numItr,npop,x_max,x_min,v_max,dim,c_cog,c_soc)
```

-This is the main function that combines the functionality of each function stated to successfully implement the PSO algorithm.

-We follow the following steps using the functions created previously until one of the stopping criteria are met (Which is reaching the max number of iterations here):

1. Initialize the population of particles with random positions and velocities on D dimensions.

2. For each particle, evaluate the fitness based on the above function.

3. Find the maximum fitness & compare it with the best fitness found so far pest'. If it is better than pbest', set 'pbest' to the maximum fitness in the population and set to the location of the particle with the maximum fitness.

4. Update the velocities and the positions of the particles according to the set of equations described in the lecture.

5. Loop to step 2, until the stopping criteria is met.

-Also, most of the parameters are set to be input by the user, to allow for dynamic use of the code instead of being hard coded.

PART 3: Results and Conclusions

1. Results

-Using the following values for our parameters and constraints:

```
numItr = 200
npop = 50
x_max = [3,1]
x_min = [-2,-2]
v_max = [0.1,0.1]
dim = 2
c_cog = 1.7
c_soc = 1.7
```

-We reach the following Optimal positions and fitness values:

```
0s [34] p_g

        array([1.45939831, 0.03198601])

0s      global_bestFit

        4.703448083000622
```

2. Conclusion

-From these answers we can conclude that:

The best X1 value is 1.45939831.

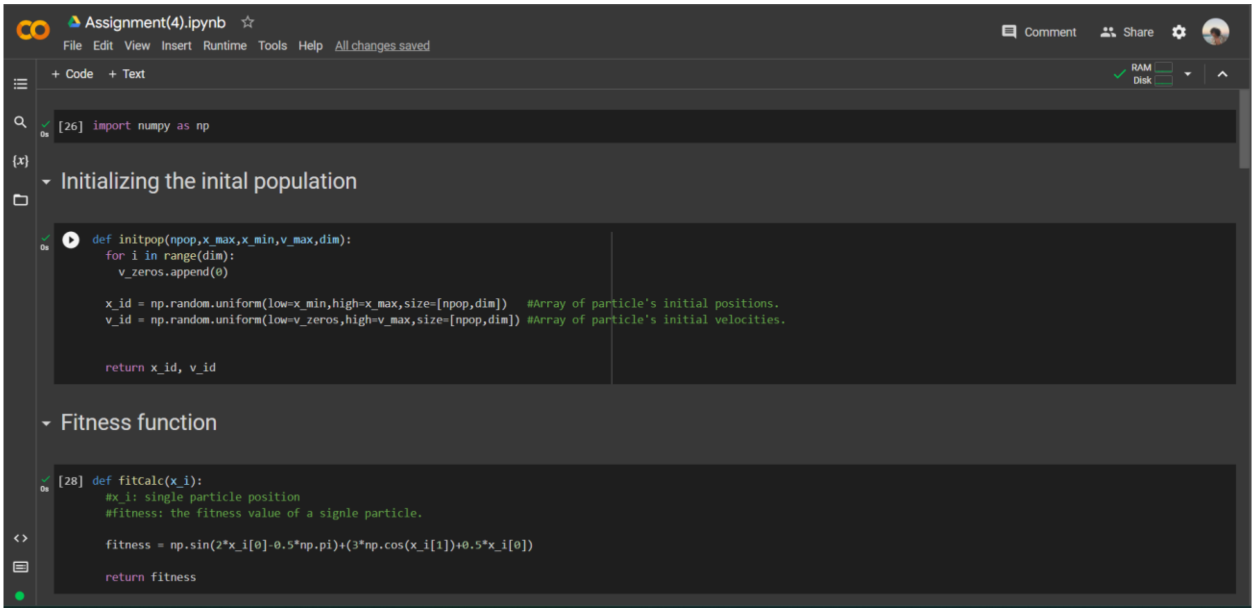
The best X2 Value is 0.03198601.

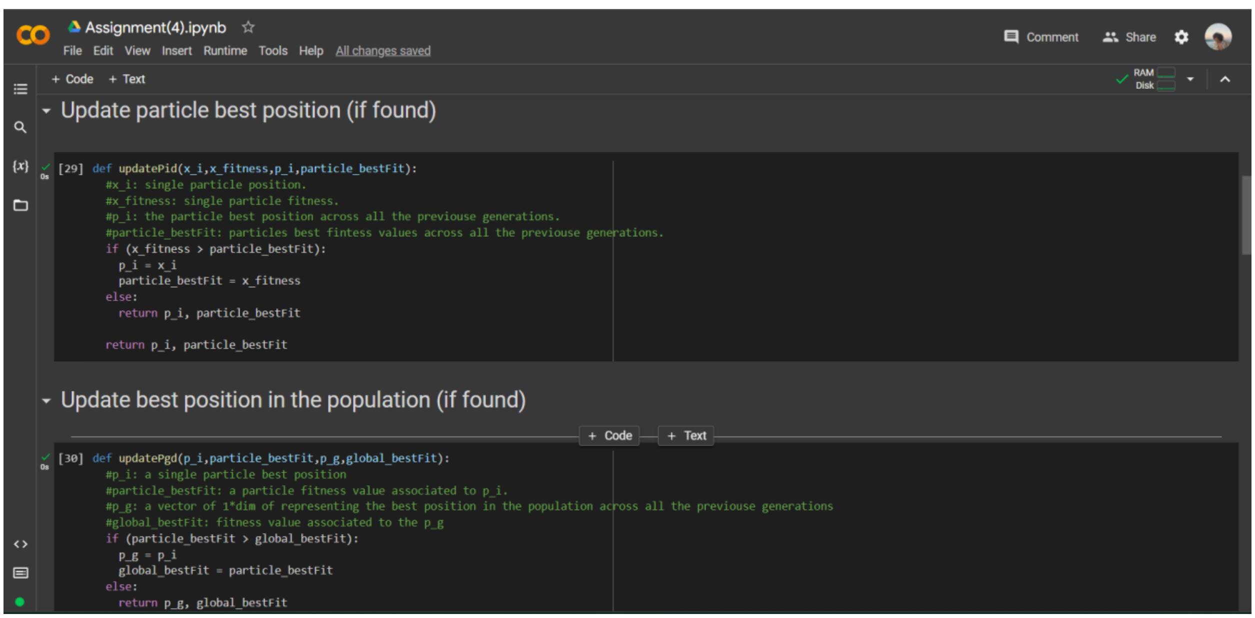
The optimal solution to the objective function $f(X_1, X_2)$ is 4.7034448083000622.

PART 4: Code link and additional Screenshots:

Link:
https://colab.research.google.com/drive/1zNLiL_0BFNyqHR6qzR3OCC3O6jVOGV77?usp=sharing

Screenshots:





Assignment(4).ipynb

File Edit View Insert Runtime Tools Help All changes saved

```
    global_bestFit = particle_bestFit
[30]   else:
         return p_g, global_bestFit

       return p_g, global_bestFit
```

Updating Position and Velocity of Particle.

```
[31] def updateVidXid(p_i,p_g,x_i,v_i,c_cog,c_soc,dim):
       #v_i: single particle velocity.
       #c_cog: cognitive component accerlaration constant
       #c_soc: social component accerlaration constant
       r_cog = np.random.random(size=2)
       r_soc = np.random.random(size=2)

       cognitive = c_cog * r_cog * (p_i - x_i)
       social = c_soc * r_soc * (p_g - x_i)

       v_i = v_i + cognitive + social
       x_i = x_i + v_i

       return x_i, v_i
```

MAIN PSO FUNCTION

```
[32] def PSO(numItr,npop,x_max,x_min,v_max,dim,c_cog,c_soc):
         #Use this function to put all the PSO algorithm together for number of iterations
         #numItr: number of iterations.(generations)
```

Assignment(4).ipynb

File Edit View Insert Runtime Tools Help All changes saved

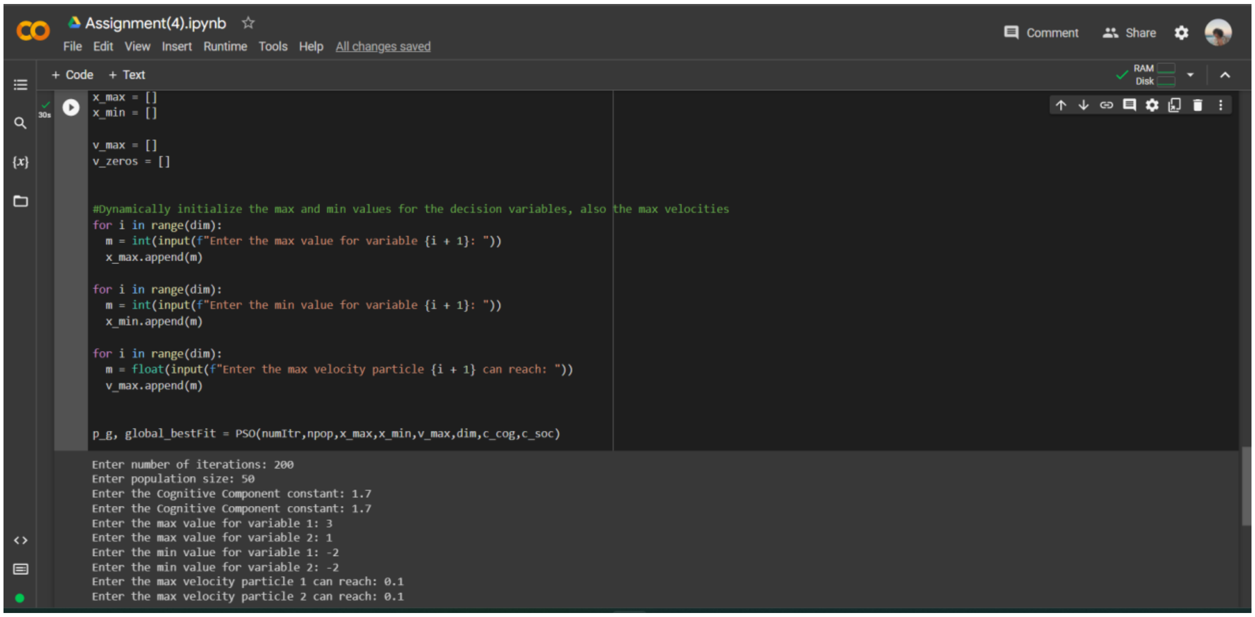
```
def PSO(numItr,npop,x_max,x_min,v_max,dim,c_cog,c_soc):
    #Use this function to put all the PSO algorithm together for number of iterations
    #numItr: number of iterations.(generations)
    #npop: population size
    #x_max: the upper limit for each decision variable (positions). [10,12]
    #x_min: the lower limit for each decision variable (positions). [1,2]
    #v_max: the upper limit for each decision variable (velocity). [2,4]
    #c_cog: cognitive constant (c1)
    #c_soc: social constant (c2)
    #dim: the number of decision variable.

    #Intialize
    population = []
    population = initpop(npop,x_max,x_min,v_max,dim)

    positions = population[0]    #array that stores the list of positions of all particles
    velocities = population[1]   #array that stores the list of velocitites of all particles

    p_i = []
    p_g = []
    particle_bestFit = 0
    global_bestFit = 0

    for i in range(dim):
      p_i.append(0)
      p_g.append(0)

    #repeat till number of iterations
    for iterations in range(numItr):
      for i in range(npop):
        x_i = positions[i]
        v_i = velocities[i]
```

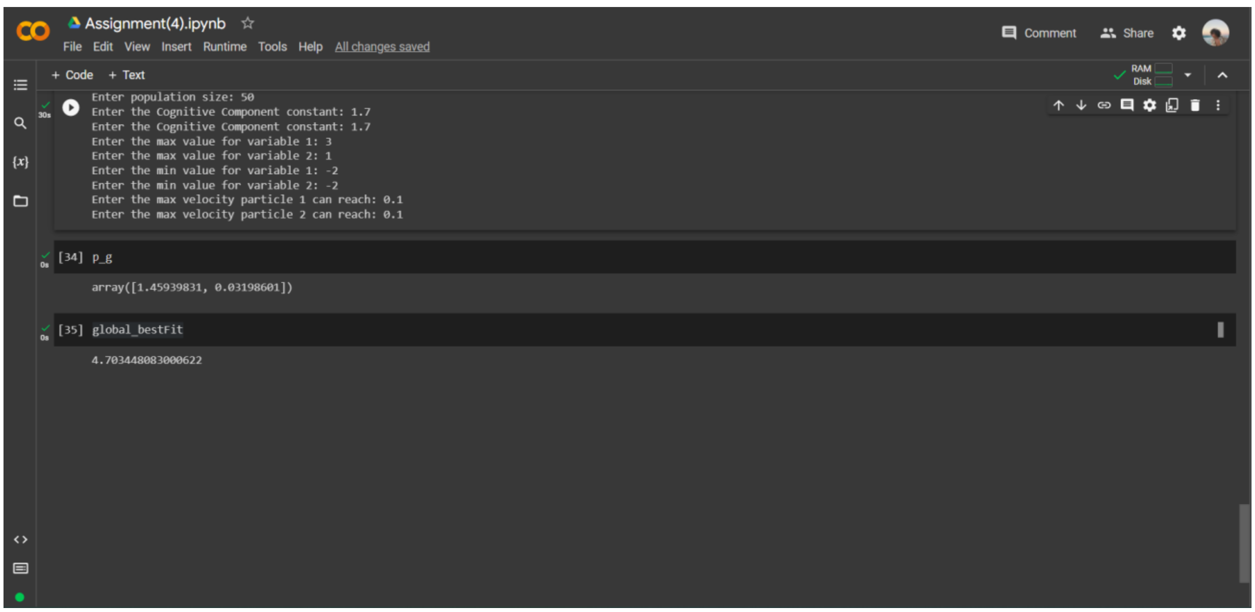
Assignment(4).ipynb

File Edit View Insert Runtime Tools Help All changes saved

```
    #repeat till number of iterations
    for iterations in range(numItr):
      for i in range(npop):
        x_i = positions[i]
        v_i = velocities[i]
        x_fitness = fitCalc(x_i)

        #Update particle best position and global best position
        p_i, particle_bestFit = updatePid(x_i,x_fitness,p_i,particle_bestFit)
        p_g, global_bestFit = updatePgd(p_i,particle_bestFit,p_g,global_bestFit)

        #Update velocity and position
        x_i, v_i = updateVidXid(p_i,p_g,x_i,v_i,c_cog,c_soc,dim)

    return  p_g, global_bestFit
    #p_g: the position with the best fitness in the final generation.
    #global_bestFit: value associated to p_g
```

```
numItr = int(input("Enter number of iterations: "))
npop = int(input("Enter population size: "))
dim = 2
c_cog = float(input("Enter the Cognitive Component constant: "))
c_soc = float(input("Enter the Cognitive Component constant: "))

x_max = []
x_min = []

v_max = []
v_zeros = []
```


Assignment(4).ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment Share
+ Code + Text
x_max = []
x_min = []
v_max = []
v_zeros = []
#Dynamically initialize the max and min values for the decision variables, also the max velocities
for i in range(dim):
m = int(input(f"Enter the max value for variable {i + 1}: "))
x_max.append(m)
for i in range(dim):
m = int(input(f"Enter the min value for variable {i + 1}: "))
x_min.append(m)
for i in range(dim):
m = float(input(f"Enter the max velocity particle {i + 1} can reach: "))
v_max.append(m)
p_g, global_bestFit = PSO(numItr,npop,x_max,x_min,v_max,dim,c_cog,c_soc)
Enter number of iterations: 200
Enter population size: 50
Enter the Cognitive Component constant: 1.7
Enter the Cognitive Component constant: 1.7
Enter the max value for variable 1: 3
Enter the max value for variable 2: 1
Enter the min value for variable 1: -2
Enter the min value for variable 2: -2
Enter the max velocity particle 1 can reach: 0.1
Enter the max velocity particle 2 can reach: 0.1


Assignment(4).ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment Share
+ Code + Text
Enter population size: 50
Enter the Cognitive Component constant: 1.7
Enter the Cognitive Component constant: 1.7
Enter the max value for variable 1: 3
Enter the max value for variable 2: 1
Enter the min value for variable 1: -2
Enter the min value for variable 2: -2
Enter the max velocity particle 1 can reach: 0.1
Enter the max velocity particle 2 can reach: 0.1
[34] p_g
array([1.45939831, 0.03198601])
[35] global_bestFit
4.703448083000622