**Name :** M. Saad Bin Shafiq

**ABSTRACT:**

Our Project is based on **PSO (Particle Swam Optimization)** algorithm. Basically this is population based algorithm mainly used for the optimal solution of a particular problem. PSO algorithm is inspired by the social behavior of bird flocking and fish schooling the concept of social interaction is used for solving a problem it uses a number of particles that constitute a swarm moving around in the search space for the best solution each particle in the swarm looks for its positional coordinates in the solution space ,which are associated with the best solution that has been achieved so far by that particle .Another Best value known as or global best is tracked by **PSO.** This is best possible value obtained so far by any particle in the neighborhood of that particle. PSO Focus on the group of birds. This group is called ‘**Swarm’.** The main focus of PSO was to graphically was to graphically simulate the graceful choreography of bird flock.

**INTRODUCTION:**

In our life we always heard **“work as a team”** it will benefit you Particle Swarm Optimization (PSO) Algorithm is based on that. In 1995, Kennedy and Eberhart wrote a research paper based on the social behavior of animal groups, where they had stated that sharing information among the group increases survival advantage. Like while a bird searching for food randomly can optimize her searching if she works with the flock. The advantage of working is mutual sharing of the best information, which can help a flock to discover the best place to hunt. There are two types of Optimization algorithms in swarm intelligence. The first one is Ant Colony Optimization. In this case the algorithm is based on the collective behavior of ants of their colony. The Second technique is particle Swarm Optimization. In this case the main focus is group of birds This Group of bird is referred to as ‘**Swarm’.** This algorithm has experienced lots of enhancements as researchers have developed new versions for different problems and demands thus they have presented different variants of the algorithm.

**ALGORITHM:**

For each particle

Initialize particle

END

Do

For each particle

Calculate fitness value

If the fitness value is better than the Pbest(Personal Best) in history

Set current value as the new Pbest.

END

Choose the particle with the best fitness value of all particles as the Gbest(Global Best)

For each particle

1. Calculate particle velocity
2. Update particle position

END

While max iteration or min error criteria is not attained

**Formulas for calculating particle velocity and updating particle position:**

1. Calculate particle velocity

(V[] = V[] + C1\*rand()\*(Pbest[] – present[]) + C2\*rand()\*(Gbest[] – present[])

1. Update particle position

Present[] = present[] + V[]

**EXPLANATION:**

First of all there is a FOR loop which will initialize each particle with some velocity and position. Now there is a Do-While loop in which For loop is used where we will calculate the fitness value for each particle. After then we will compare that if the fitness value of particle is better than the Pbest in history then we will set the current fitness value as the Pbest. After then we will choose the particle with best fitness value and declare it as a Gbest. Now with the help of Pbest and Gbest, we will calculate each particle velocity and its position. At the last if you have reached the maximum number of iteration, then the program will terminate otherwise this Do-While loop will run again and again until all iterations are not completed.

**FLOWCHART:**

**Diagram

Description automatically generated**

**COMPLEXITY:**

Until we have a new theory, every metaheuristic has a complexity of infinity. Since there is no theory developed that establishes that any metaheuristic finds and identifies an optimal solution for a large enough class of problems. Unfortunately.

That does not mean that they are not useful, of course, with the above remark in mind, that you cannot take for granted – for your particular problem – that you can even find a near optimal solution and since the methodology is not based on optimality criteria, it is also very hard to assess whether a given vector is optimal or at least satisfies necessary optimality conditions.

But again, they can be very handy indeed.

As per the number of iteration is based on the user choice which can be up to infinity. hence time complexity may reach to infinity.

**CONCLUSION:**

The most exciting part of PSO is there is a stable topology where particles are able to communicate with each other and increase the learning rate to achieve global optimum. The metaheuristic nature of this optimization algorithm gives us lots of opportunities as it optimizes a problem by iteratively trying to improve a candidate solution. Applicability of it will increase more with the ongoing research work in Ensemble Learning.

**REFERENCES:**

1. <https://youtu.be/Rn1kcmG9AUU>
2. <https://youtu.be/efVwSGuK4J0>
3. <https://www.researchgate.net/figure/Flow-chart-of-the-PSO-LMS-algorithm-for-complex-valued-data_fig5_265730681>
4. Complexity( Chalmers University of Technology )
5. <https://github.com/maystrovyy/Particle-Swarm-Optimization-with-Python/blob/master/simple-particle-swarm-optimization.py>

**CODE:**

from \_\_future\_\_ import division

import random

import math

#--- COST FUNCTION ----

# function we are attempting to optimize (minimize)

def func1(x):

total=0

for i in range(len(x)):

total+=x[i]\*\*2

return total

#--- MAIN ----

class Particle:

def \_\_init\_\_(self,x0):

self.position\_i=[] # particle position

self.velocity\_i=[] # particle velocity

self.pos\_best\_i=[] # best position individual

self.err\_best\_i=-1 # best error individual

self.err\_i=-1 # error individual

for i in range(0,num\_dimensions):

self.velocity\_i.append(random.uniform(-1,1))

self.position\_i.append(x0[i])

# evaluate current fitness

def evaluate(self,costFunc):

self.err\_i=costFunc(self.position\_i)

# check to see if the current position is an individual best

if self.err\_i<self.err\_best\_i or self.err\_best\_i==-1:

self.pos\_best\_i=self.position\_i.copy()

self.err\_best\_i=self.err\_i

# update new particle velocity

def update\_velocity(self,pos\_best\_g):

w=0.5 # constant inertia weight (how much to weigh the previous velocity)

c1=1 # cognative constant

c2=2 # social constant

for i in range(0,num\_dimensions):

r1=random.random()

r2=random.random()

vel\_cognitive=c1\*r1\*(self.pos\_best\_i[i]-self.position\_i[i])

vel\_social=c2\*r2\*(pos\_best\_g[i]-self.position\_i[i])

self.velocity\_i[i]=w\*self.velocity\_i[i]+vel\_cognitive+vel\_social

# update the particle position based off new velocity updates

def update\_position(self,bounds):

for i in range(0,num\_dimensions):

self.position\_i[i]=self.position\_i[i]+self.velocity\_i[i]

# adjust maximum position if necessary

if self.position\_i[i]>bounds[i][1]:

self.position\_i[i]=bounds[i][1]

# adjust minimum position if neseccary

if self.position\_i[i]<bounds[i][0]:

self.position\_i[i]=bounds[i][0]

class PSO():

def \_\_init\_\_(self, costFunc, x0, bounds, num\_particles, maxiter, verbose=False):

global num\_dimensions

num\_dimensions=len(x0)

err\_best\_g=-1 # best error for group

pos\_best\_g=[] # best position for group

# establish the swarm

swarm=[]

for i in range(0,num\_particles):

swarm.append(Particle(x0))

# begin optimization loop

i=0

while i<maxiter:

if verbose: print(f'iter: {i:>4d}, best solution: {err\_best\_g:10.6f}')

# cycle through particles in swarm and evaluate fitness

for j in range(0,num\_particles):

swarm[j].evaluate(costFunc)

# determine if current particle is the best (globally)

if swarm[j].err\_i<err\_best\_g or err\_best\_g==-1:

pos\_best\_g=list(swarm[j].position\_i)

err\_best\_g=float(swarm[j].err\_i)

# cycle through swarm and update velocities and position

for j in range(0,num\_particles):

swarm[j].update\_velocity(pos\_best\_g)

swarm[j].update\_position(bounds)

i+=1

# print final results

print('\nFINAL SOLUTION:')

print(f' > {pos\_best\_g}')

print(f' > {err\_best\_g}\n')

if \_\_name\_\_ == "\_\_PSO\_\_":

main()

#--- RUN ---

initial=[5,5] # initial starting location [x1,x2...]

bounds=[(-10,10),(-10,10)] # input bounds [(x1\_min,x1\_max),(x2\_min,x2\_max)...]

PSO(func1, initial, bounds, num\_particles=15, maxiter=30, verbose=True)

#--- END ---

**OUTPUTS:**

If number of particles=15 and maximum iteration=30

A picture containing graphical user interface

Description automatically generated

If number of particles=20 and maximum iteration=25

Text

Description automatically generated with medium confidence

If number of particles=5 and maximum iteration=10

Text

Description automatically generated