

## Assignment 4

### Question1 (5 Points)

NetLogo has a model that implements a simple **PSO** in 2-D searching space [Models Library → Computer Science → Particle Swarm Optimization].

- a. Run experiments with different populations (30, 80), speed limits (2 and 6), particle's inertia (0.60, 0.729), personal-best (1.7, 1.494) and the global-best factor the same as personal factor. Examine the PSO algorithm's characteristics (speed of converge and ability to find global optima), report your observations.

No	Change	Impact	
		Speed of convergence	Ability to find global optima
1	Effect of Increasing Population	Convergence is achieved faster when the population size is increased	More consistently finds the accurate values for the global maxima
2	Effect of Increasing the Speed Limit	Increasing the speed limit decreases the speed of convergence because now the swarm is exploring a larger search space, leading to a slower convergence	Increasing the speed limit makes it more difficult to find the global optima. Hence, particles do not find the global optima
3	Effect of Increasing the particle's inertia	Increasing the particles' inertia results in minimal local exploitation, and so, this leads to a slower convergence	Increasing the particles' inertia results in more exploration in the direction of search of the current solution which will increase the ability to find the global maxima
4	Effect of increasing the personal best factor	Since the personal best factor contains more information from past moves, they can faster convergence based on their own biases	Increasing the personal best factor means you are less susceptible to be trapped in a local minimum
5	Effect of increasing the global factor	Since the global factor contains more information from the general swarm, they can decrease convergence	Increasing the global factor can also result in larger diversity, and the PSO having a greater chance of finding the global optima

- b. What is the difference between the motion formulation of the given NetLogo implementation and the classical PSO? Explain the difference by referring to the code tab in Netlogo.

```
; Technical note:  
; In the canonical PSO, the "(1 - particle-inertia)" term isn't present in the  
; mathematical expressions below. It was added because it allows the  
; "particle-inertia" slider to vary particles motion on the the full spectrum  
; from moving in a straight line (1.0) to always moving towards the "best" spots  
; and ignoring its previous velocity (0.0).  
  
; change my velocity by being attracted to the "personal best" value I've found so far  
facexy personal-best-x personal-best-y  
let dist distancexy personal-best-x personal-best-y  
set vx vx + (1 - particle-inertia) * attraction-to-personal-best * (random-float 1.0) * dist * dx  
set vy vy + (1 - particle-inertia) * attraction-to-personal-best * (random-float 1.0) * dist * dy
```

One difference in this implementation is the  $(1 - \text{particle-inertia})$  component. Normally, this term isn't in the classical PSO implementation. It was added because it allows the "particle-inertia" slider to vary particles motion on the full spectrum from moving in a straight line (1.0) to always moving towards the "best" spots and ignoring its previous velocity (0.0) – as stated in one of the technical notes.

As well, the particle speed limit was an addition in this code from the classical PSO implementation because as stated in the code. We are dealing with a toroidal (wrapping) world, which means that particles can start warping around the world at ridiculous speeds. To restrict this, we limit the particle speed. In this case, we set constraints to the velocity.

### Question2 (5 Points)

Consider below an implementation of **Particle Swarm Optimization** (PSO) algorithm

```
procedure [X] = PS(max_it, AC1, AC2, vmax, vmin)
  initialize X //usually  $\mathbf{x}_i$ ,  $\forall i$ , is initialized at random
  initialize  $\Delta \mathbf{x}_i$  //at random,  $\Delta \mathbf{x}_i \in [v_{min}, v_{max}]$ 
  t  $\leftarrow$  1
  while t < max_it do,
    for i = 1 to N do, //for each particle
      if  $g(\mathbf{x}_i) > g(\mathbf{p}_i)$ ,
        then  $\mathbf{p}_i = \mathbf{x}_i$ , //best indiv. performance
      end if
      g = i //arbitrary
      //for all neighbors
      for j = indexes of neighbors
        if  $g(\mathbf{p}_j) > g(\mathbf{p}_g)$ ,
          then g = j, //index of best neighbor
        end if
      end for
       $\Delta \mathbf{x}_i \leftarrow \Delta \mathbf{x}_i + \phi_1 \odot (\mathbf{p}_i - \mathbf{x}_i) + \phi_2 \odot (\mathbf{p}_g - \mathbf{x}_i)$ 
       $\Delta \mathbf{x}_i \in [v_{min}, v_{max}]$ 
       $\mathbf{x}_i \leftarrow \mathbf{x}_i + \Delta \mathbf{x}_i$ 
    end for
    t  $\leftarrow$  t + 1
  end while
end procedure
```

a) In the above PSO implementation, which update mode (synchronous or asynchronous) for personal best and neighbors' best is adopted? Explain shortly.

This is an **asynchronous update mode**. This is because the neighborhood best update is in the particles loop, to be updated individually. We can see this in the outlined orange section above. In this case each particle's individual solution state, whenever the neighbor is accessed, we change the best and adopt the neighbor.

Same with the personal best solution, we can see from the blue section that the individual performance is updated in every iteration for each particle.

- b) Comment on how to change the algorithm to work on asynchronous mode if your answer to part (a) is “synchronous” otherwise to work on synchronous if your answer to part (a) is “asynchronous”.

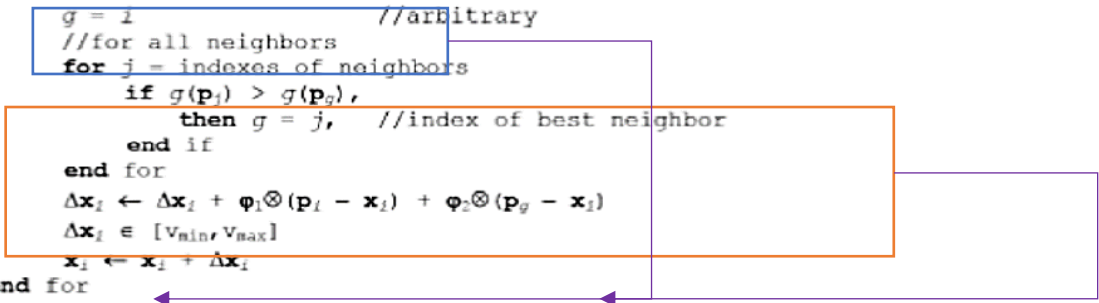
If you want to change the algorithm to work on a synchronous mode, we need to move the code in the blue and orange blocks till after the “for all particles” loop, so we update the neighborhood and personal best once for each particle

Consider below an implementation of **Particle Swarm Optimization (PSO)** algorithm

```

procedure [X] = PS(max_it, AC1, AC2, vmax, vmin)
  initialize X //usually  $\mathbf{x}_i$ ,  $\forall i$ , is initialized at random
  initialize  $\Delta \mathbf{x}_i$  //at random,  $\Delta \mathbf{x}_i \in [v_{min}, v_{max}]$ 
  t  $\leftarrow$  1
  while t < max_it do,
    for i = 1 to N do, //for each particle
      if g( $\mathbf{x}_i$ ) > g( $\mathbf{p}_i$ ),
        then  $\mathbf{p}_i = \mathbf{x}_i$ , //best indiv. performance
      end if
      g = 1 //arbitrary
      //for all neighbors
      for j = indexes of neighbors
        if g( $\mathbf{p}_j$ ) > g( $\mathbf{p}_i$ ),
          then g = j, //index of best neighbor
        end if
      end for
       $\Delta \mathbf{x}_i \leftarrow \Delta \mathbf{x}_i + \phi_1 \otimes (\mathbf{p}_i - \mathbf{x}_i) + \phi_2 \otimes (\mathbf{p}_g - \mathbf{x}_i)$ 
       $\Delta \mathbf{x}_i \in [v_{min}, v_{max}]$ 
       $\mathbf{x}_i \leftarrow \mathbf{x}_i + \Delta \mathbf{x}_i$ 
    end for
    t  $\leftarrow$  t + 1
  end while
end procedure

```



The diagram illustrates the modification of the PSO algorithm for asynchronous mode. A blue box highlights the initialization of  $\mathbf{p}_i$  and the selection of the best neighbor  $\mathbf{p}_g$  within the inner loop for each particle  $i$ . An orange box highlights the velocity update and position update steps. A purple arrow points from the end of the orange box back to the start of the blue box, indicating that these steps are repeated for each particle. Another purple arrow points from the end of the orange box to the  $\mathbf{p}_g$  update step in the blue box, indicating that the global best is updated after each particle's position is updated.

is asynchronous .

c) Considering the effect of Parameters  $W, C_1, C_2$  in the PSO model:

$$v_{t+1}^{id} = w * v_t^{id} + c_1 r_1^{id} (pbest_t^{id} - x_t^{id}) + c_2 r_2^{id} (Nbest_t^{id} - x_t^{id})$$

i. What happens when  $C_1$  is set to zero?

When  $C_1$  is set to 0, then the velocity model reduces to just the social component, and **so all particles will be attracted to Nbest** which is either the local or global best value. This can cause certain areas to be unexplored, and particles might end-up converge at a sub-optimal solution.

ii. What happens when  $C_2$  set to zero?

When  $C_2$  is set to 0, then the velocity model reduces to just the cognitive component, and so all **particles will be attracted to Pbest** which is each particle relying solely on its previous best value obtained. This can cause each particle to act as independent hill-climbers and may not result in them converging to a solution.

iii. What is the importance of the  $W$  parameter?

The  $W$  parameter is important because it is used to **balance exploration and exploitation**. Large values of  $W$  promote exploration, allowing more potential solutions to be explored, and more optimal solutions to be reached. Whereas smaller values of  $W$  promote more exploitation, allowing the algorithm to place more control on cognitive and social components, which can lead to converging at a potential solution.

### Question3 (10 Points)

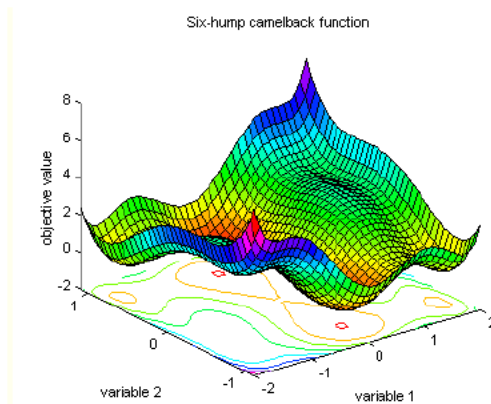
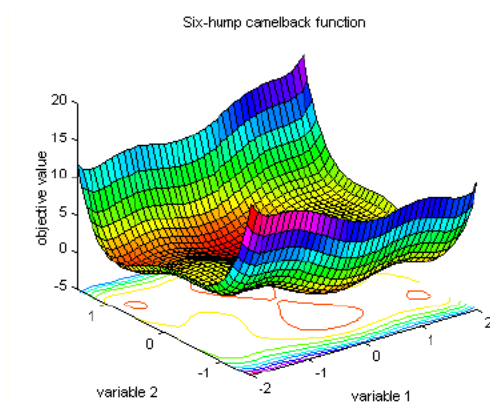
The six-hump camelback problem is given as:

$$z = (4 - 2.1x^2 + \frac{x^4}{3})x^2 + xy + (-4 + 4y^2)y^2$$

where x and y lie between +/-5. The objective is to minimize z. The global minimum lies at (-0.089840, 0.712659) or (0.089840, -0.712659) where  $z = -1.0316285$ . The global optimum to the problem is given for reference purpose. Do not use it in your solution methodology.

Code a **simple PSO** and a **linear PSO** to solve the problem. To do this you need to encode the problem, initialize a population, select a velocity update equation, and select a stopping criterion.

Run your code and report: final solution, plot the progress of the average fitness and the best particle fitness for the simple and linear versions. In your report, describe the choices you made to set the parameters of the populations, and your observations on the performance of simple PSO vs linear PSO.



### Simple PSO

```
from dataclasses import dataclass
from collections import deque
import random

#particle struct
@dataclass
class particle:
    velocity: list[float] #this will be the size of the dimensions list
    position: list[float]
    personal_best: list[float]
    function_eval_best: int
    function_eval: int

swarm = [] #swarm will contain particles
```

```
def function_evaluate(position):
    x = position[0]
    y = position[1]

    z1 = (4 - (2.1*x**2) + (x**4)/3) * (x**2)
    z2 = x * y
    z3 = (-4 + (4 * (y**2))) * (y**2)

    z = z1 + z2 + z3
    return z

def init_swarm(dim, num_particles, bounds):

    # print(dim)
    # print(num_particles)
    # print(bounds)

    #initialize the particles
    for i in range(num_particles):
        #create a particle

        vel = [random.uniform(0,1), random.uniform(0,1)] #celocities can be
        #initialized to 0 or small values
        pos = [random.uniform(bounds[0][0],bounds[0][1]),
        random.uniform(bounds[1][0],bounds[1][1])] #particle positions can be initialized
        #randomly in range
        new_particle = particle(vel, pos, pos, -1, -1) #personal best position is
        #initialized to the particle's initial position

        swarm.append(new_particle)

def perform_optimization(max_itr, num_particles, bounds):

    global_best_val = -1
    global_best_pos = []

    for i in range(max_itr):
        print(global_best_val)
```

```
# update the personal best based on evaluation of the fitness function
for j in range(num_particles):

    fun_val = function_evaluate(swarm[j].position)
    swarm[j].function_eval = fun_val

    #update the personal best value based on how it evaluates based on
the function
    if (fun_val < swarm[j].function_eval_best
or swarm[j].function_eval_best == -1):
        swarm[j].function_eval_best = fun_val
        swarm[j].personal_best = swarm[j].position

    # if(swarm[j].function_eval < global_best_val or global_best_val == -
1):

        # global_best_val = float(swarm[j].function_eval)
        # global_best_pos = list(swarm[j].position)

#update the global best based on the personal best of all particles
for j in range(num_particles):

    if(swarm[j].function_eval_best < global_best_val or global_best_val
== -1):

        global_best_val = float(swarm[j].function_eval_best)
        global_best_pos = list(swarm[j].personal_best)

#update velocities and position for each of the particles
for j in range(num_particles):

    w = 0.5
    c1 = 1
    c2 = 2

    r1 = random.random()
    r2 = random.random()
    #update the velocity of each particle for each dimension according to
the formula
    for i in range(2):

        v_cog = c1 * r1 * (swarm[j].personal_best[i] -
swarm[j].position[i])
        v_soc = c2 * r2 * (global_best_pos[i] - swarm[j].position[i])
```

Key component of simple PSO



```
        swarm[j].velocity[i] = w * swarm[j].velocity[i] + v_cog + v_soc

        #update the velocity of each particle for each dimension according to
the formula
        for i in range(2):
            swarm[j].position[i] = swarm[j].position[i] +
swarm[j].velocity[i]

            if swarm[j].position[i] > bounds[i][1]:
                swarm[j].position[i] = bounds[i][1]

            if swarm[j].position[i] < bounds[i][0]:
                swarm[j].position[i] = bounds[i][0]

        # print(global_best_val)
        # print(swarm[50].position[0])
        # print(swarm[50].position[1]) #the swarm converges

dim = 2
num_particles = 500
bounds = [[-5, 5], [-5, 5]]
max_ittr = 100

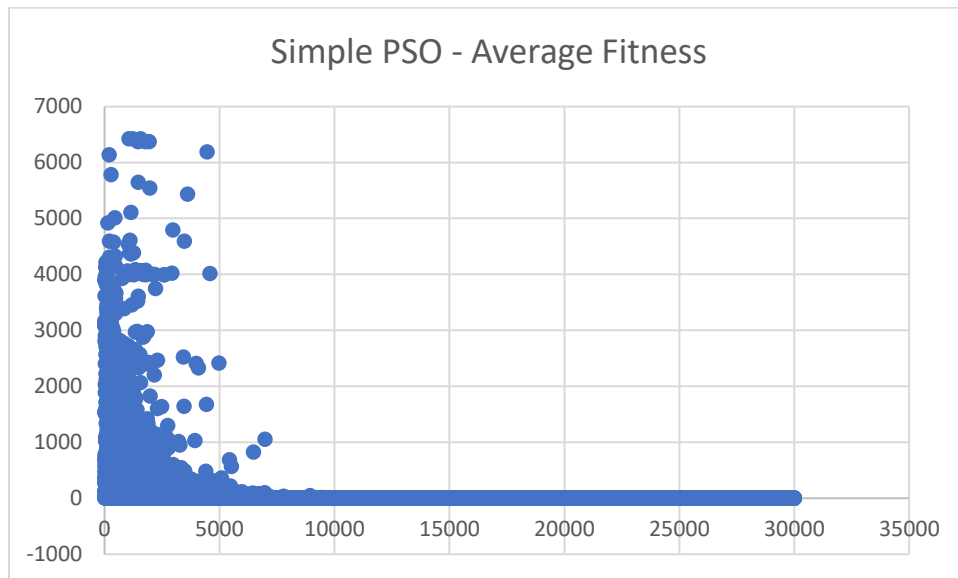
init_swarm(dim, num_particles, bounds)
#print(swarm)

#perform optimization
perform_optimization(max_ittr, num_particles, bounds)
```

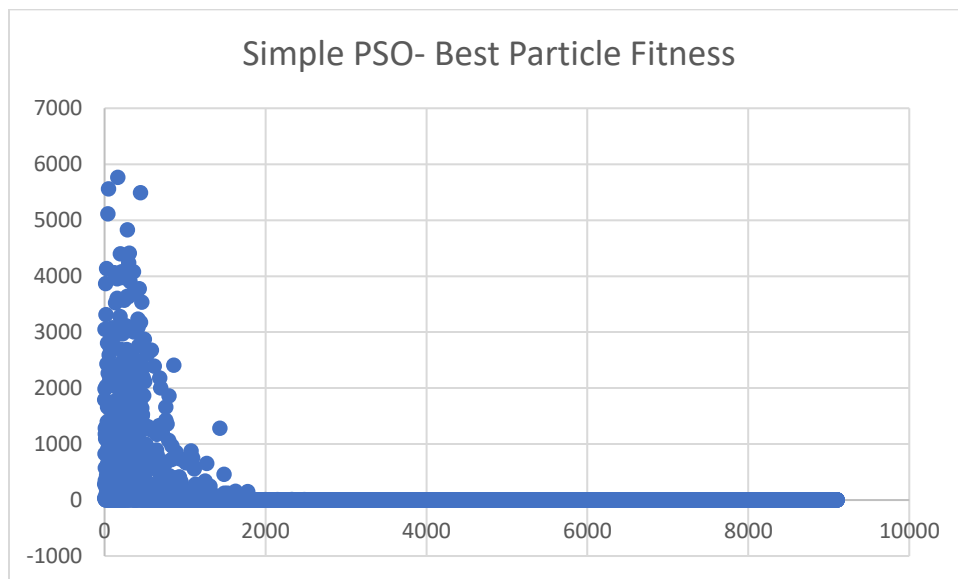
#### Output:

```
PROBLEMS 1 OUTPUT JUPYTER DEBUG CONSOLE TERMINAL
[Running] python -u "c:\Users\rupak\Documents\ECE457A_A4\simple_pso.py"
-1.0316284534898774
-0.0898420108088042
0.7126564062053776
```

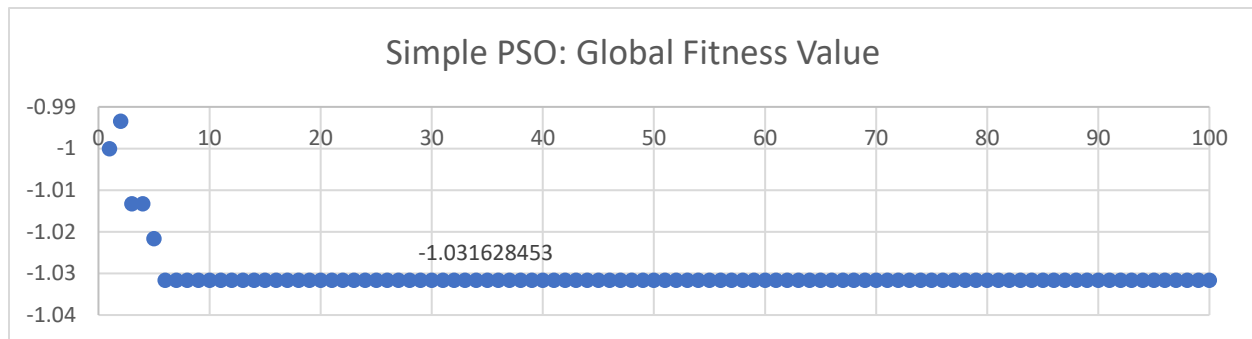
**Results from Simple PSO – Average Fitness based on current particle's solution. Our goal is to minimize this.**



**Results from Simple PSO –Best particle Fitness based on current particle solution. Our goal is to minimize the fitness function value.**



### Global Fitness Value for Simple PSO:



### Linear PSO

```
from dataclasses import dataclass
from collections import deque
import random

#particle struct
@dataclass
class particle:
    velocity: list[float] #this will be the size of the dimensions list
    position: list[float]
    personal_best: list[float]
    function_eval_best: int
    function_eval: int

swarm = [] #swarm will contain particles

def function_evaluate(position):
    x = position[0]
    y = position[1]

    z1 = (4 - (2.1*x**2) + (x**4)/3) * (x**2)
    z2 = x * y
    z3 = (-4 + (4 * (y**2))) * (y**2)

    z = z1 + z2 + z3
    return z

def init_swarm(dim, num_particles, bounds):

    # print(dim)
```

```
# print(num_particles)
# print(bounds)

#initialize the particles
for i in range(num_particles):
    #create a particle

    vel = [random.uniform(0,1), random.uniform(0,1)] #celocities can be
initialiozed to 0 or small values
    pos = [random.uniform(bounds[0][0],bounds[0][1]),
random.uniform(bounds[1][0],bounds[1][1])] #particle positions cna be initialized
randomly in range
    new_particle = particle(vel, pos, pos, -1, -1) #personal best position is
initialized to the particle's inital position

    swarm.append(new_particle)

def perform_optimization(max_ittr, num_particles, bounds):

    global_best_val = -1
    global_best_pos = []

    for i in range(max_ittr):
        print(global_best_val)
        # update the personal best based on evaluation of the fitness function
        for j in range(num_particles):

            fun_val = function_evaluate(swarm[j].position)
            swarm[j].function_eval = fun_val
            # print(fun_val)

            #update the personal best value based on how it evaluates based on
the functon
            if (fun_val < swarm[j].function_eval_best
or swarm[j].function_eval_best == -1):
                swarm[j].function_eval_best = fun_val
                swarm[j].personal_best = swarm[j].position
                print(fun_val)
```

```
1):
    # if(swarm[j].function_eval < global_best_val or global_best_val == -
    #
    #     global_best_val = float(swarm[j].function_eval)
    #     global_best_pos = list(swarm[j].position)

    #update the global best based on the personal best of all particles
    for j in range(num_particles):

        if(swarm[j].function_eval_best < global_best_val or global_best_val
        == -1):
            global_best_val = float(swarm[j].function_eval_best)
            global_best_pos = list(swarm[j].personal_best)

    #update velocities and position for each of the particles
    for j in range(num_particles):

        w = 0.5
        c1 = 1
        c2 = 2

        #update the velocity of each particle for each dimension according to
        the formula
        for i in range(2):
            r1 = random.random()
            r2 = random.random()

            v_cog = c1 * r1 * (swarm[j].personal_best[i] -
            swarm[j].position[i])
            v_soc = c2 * r2 * (global_best_pos[i] - swarm[j].position[i])

            swarm[j].velocity[i] = w * swarm[j].velocity[i] + v_cog + v_soc

        #update the velocity of each particle for each dimension according to
        the formula
        for i in range(2):
            swarm[j].position[i] = swarm[j].position[i] +
            swarm[j].velocity[i]
```

```
        if swarm[j].position[i] > bounds[i][1]:
            swarm[j].position[i] = bounds[i][1]

        if swarm[j].position[i] < bounds[i][0]:
            swarm[j].position[i] = bounds[i][0]

    # print(global_best_val)
    # print(swarm[50].position[0])
    # print(swarm[50].position[1]) #the swarm converges

dim = 2
num_particles = 500
bounds = [[-5, 5], [-5, 5]]
max_ittr = 100

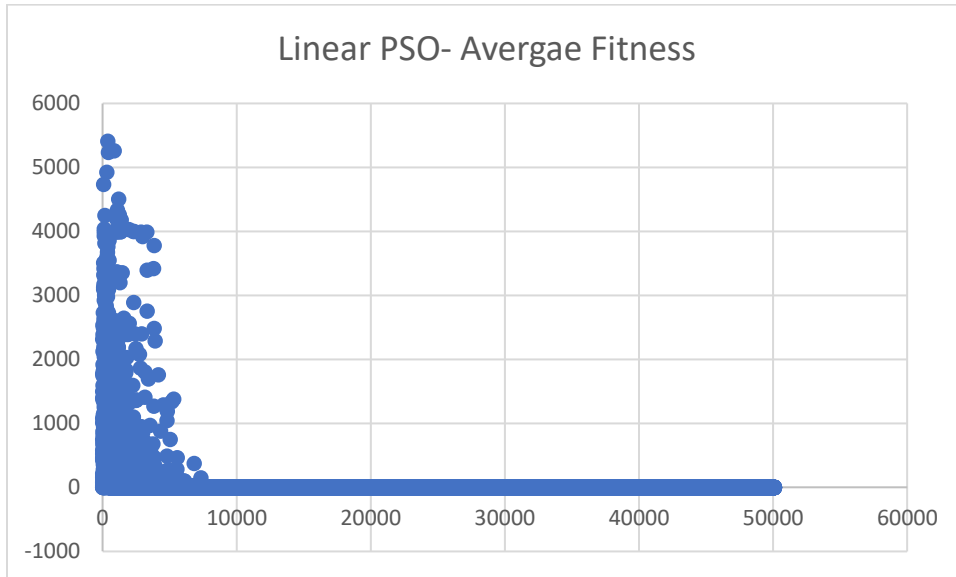
init_swarm(dim, num_particles, bounds)
#print(swarm)

#perform optimization
perform_optimization(max_ittr, num_particles, bounds)
```

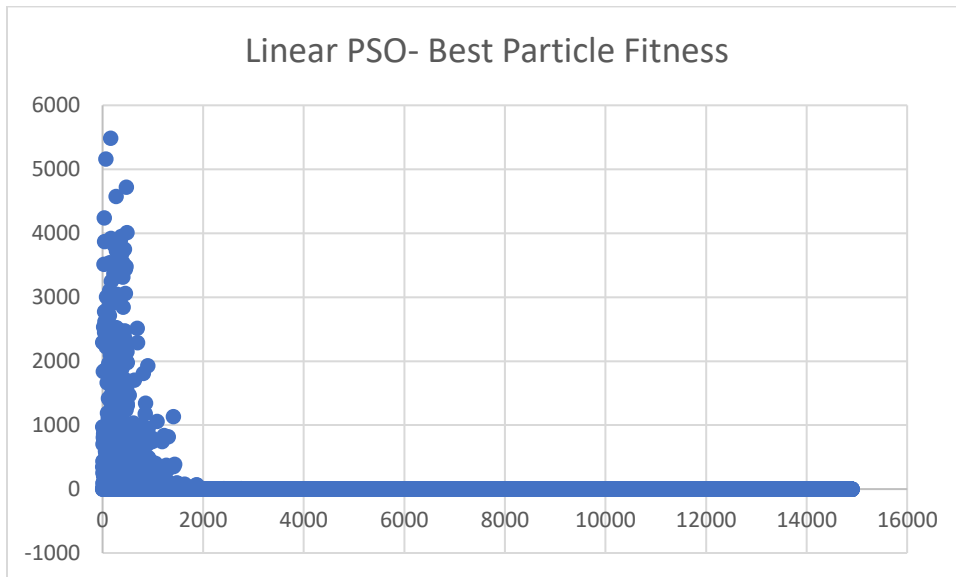
**Output:**

```
-1.0316284534898774
0.08984201464407837
-0.7126564018970498
```

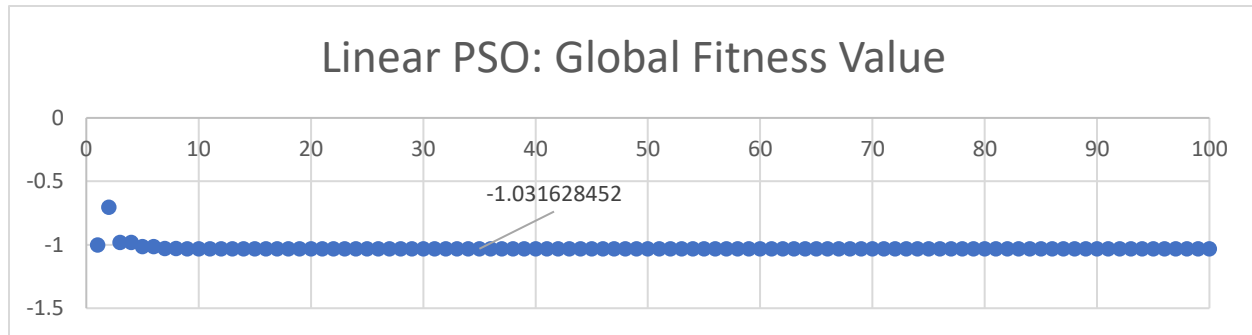
**Results from Linear PSO – Average Fitness based on current particle's solution. Our goal is to minimize this.**



**Results from Linear PSO –Best particle Fitness based on current particle solution. Our goal is to minimize the fitness function value.**



### Global Fitness Value for Linear PSO



#### Observations:

- There is more randomness to simple PSO, whereas for linear we have most points cluttered in one region
- The simple PSO converges faster than the linear PSO
- Both converge to the optimal solution eventually

#### Parameter choices:

```
w = 0.5  
c1 = 1  
c2 = 2  
num_particles = 500  
max_itr = 100
```

I observed that these parameters helped achieve the optimal solution for both algorithms