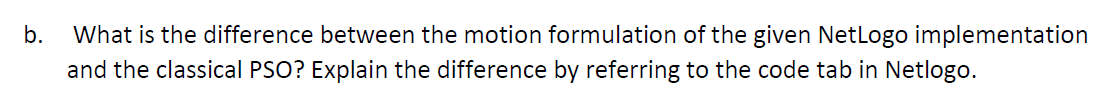
# Assignment 4

Text

Description automatically generated

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | **Impact** | |
| **No** | **Change** | **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 ca 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 faster 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 |



**Text, letter

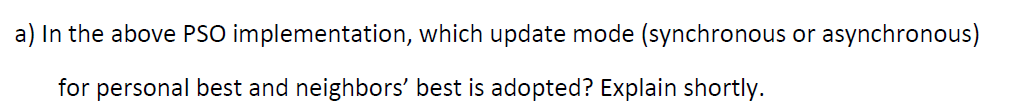
Description automatically generated**

One difference in this implementation is the (1 – 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 sped 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.

**Text

Description automatically generated**

****

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.

Text

Description automatically generated

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

**Text

Description automatically generated**

Text, letter

Description automatically generated

When C1 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.

Logo

Description automatically generated with medium confidence

When C2 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.

Shape

Description automatically generated with low confidence

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.

Chart

Description automatically generated

**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)

    #initalize 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

            #update the personal best value based on how it evaluates based on the funciton

            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 persoal 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

Key component of simple PSO

            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])

                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:**

**Text

Description automatically generated**

**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)

    #initalize 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 funciton

            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)

            # 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 persoal 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:**

Graphical user interface, text

Description automatically generated

**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\_ittr = 100

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