Implement Particle swarm optimization for benchmark function (eg. Square, Rosenbrock function). Initialize the population from the Standard Normal Distribution. Evaluate fitness of all particles. Use : c1=c2 = 2 Inertia weight is linearly varied between 0.9 to 0.4. Global best variation

```
from random import random
from random import uniform
from numpy.random import normal
import math
```

```
# functions to optimize (minimize)
def square(x):
    total=0
    for i in range(len(x)):
        total+=x[i]**2
    return total

def rosenbrock(x):
    a = 1
    b = 15
    return ((a - x[0]**2)+b*((x[1]-x[0]**2)**2))
```

```
class Particle:
    def __init__(self, initial_pos):
        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(float(normal(0.5,0.175,1)))
            self.position_i.append(initial_pos[i])

    def evaluate(self,cost_function):
        '''
        evaluate current fitness

        :params
         cost_function : function to optimize
        '''
        self.err_i=cost_function(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

```

```
    def update_velocity(self,pos_best_g):
        '''
        update new particle velocity

        :params
         pos_best_g : global best position
        '''
        w=uniform(0.4,0.9)       #linearly varied b/w 0.9 to 0.4
        c1=2
        c2=2

        for i in range(0,num_dimensions):
            r1=random()
            r2=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

    def update_position(self,bounds):
        '''
        updates the particle position based on new velocity updates

        :params
         bounds
        '''
        for i in range(0,num_dimensions):
            self.position_i[i]=self.position_i[i]+self.velocity_i[i]

            # check boundary conditions
            if self.position_i[i]>bounds[i][1]:
                self.position_i[i]=bounds[i][1]
            if self.position_i[i]<bounds[i][0]:
                self.position_i[i]=bounds[i][0]
```

```
def minimize(cost_function, initial_pos, bounds, num_particles, max_iterations, ve
    global num_dimensions

    num_dimensions=len(initial_pos)
    err_best_g=-1                   # best error for group
    pos_best_g=[]                   # best position for group

    # create the swarm
    swarm=[]
    for i in range(0,num_particles):
        swarm.append(Particle(initial_pos))

    i=0
    while i<max_iterations:
        if verbose: print(f'iteration: {i:>4d}, best solution: {err_best_g:10.6f}'

```

```
        # evaluate fitness
        for j in range(0,num_particles):
            swarm[j].evaluate(cost_function)

            # 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)

        # 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

    return err_best_g, pos_best_g
```

```
initial=[5,5]               # initial starting location [x1,x2...]
bounds=[(-10,10),(-10,10)]  # input bounds [(x1_min,x1_max),(x2_min,x2_max)...]

# for rosenbrock function
minima, best_position = minimize(rosenbrock, initial, bounds, num_particles=15, ma
print('\n\nBest Position:',best_position)
print('Best Solution:',minima)
```

```
iteration:    0, best solution:  -1.000000
iteration:    1, best solution: 5976.000000
iteration:    2, best solution: 5976.000000
iteration:    3, best solution: 1951.321343
iteration:    4, best solution:  61.556599
iteration:    5, best solution:   3.752611
iteration:    6, best solution:   3.752611
iteration:    7, best solution:  -3.899044
iteration:    8, best solution:  -3.920926
iteration:    9, best solution:  -3.920926
iteration:   10, best solution:  -3.920926
iteration:   11, best solution:  -4.964809
iteration:   12, best solution:  -5.064727
iteration:   13, best solution:  -5.064727
iteration:   14, best solution:  -5.064727
iteration:   15, best solution:  -5.064727
iteration:   16, best solution:  -5.064727
iteration:   17, best solution:  -5.064727
iteration:   18, best solution:  -5.274907
iteration:   19, best solution:  -5.274907
iteration:   20, best solution:  -5.424370
iteration:   21, best solution:  -5.608808
iteration:   22, best solution:  -5.608808
iteration:   23, best solution:  -5.608808
iteration:   24, best solution:  -5.608808
iteration:   25, best solution:  -5.608808
iteration:   26, best solution:  -5.608808
iteration:   27, best solution:  -5.608808
iteration:   28, best solution:  -5.608808
```

```
iteration:   29, best solution:  -5.608808


Best Position: [2.586473862228981, 6.61634458572222]
Best Solution: -5.6088078782347335
```

```
# for square function
minima_sq, best_position_sq = minimize(square, initial, bounds, num_particles=15,
print('\n\nBest Position:',best_position_sq)
print('Best Solution:',minima_sq)
```

```
iteration:    0, best solution:  -1.000000
iteration:    1, best solution:  50.000000
iteration:    2, best solution:  50.000000
iteration:    3, best solution:  40.451234
iteration:    4, best solution:  26.225596
iteration:    5, best solution:  13.825216
iteration:    6, best solution:   5.020039
iteration:    7, best solution:   1.778763
iteration:    8, best solution:   1.318052
iteration:    9, best solution:   0.652904
iteration:   10, best solution:   0.416916
iteration:   11, best solution:   0.129198
iteration:   12, best solution:   0.008798
iteration:   13, best solution:   0.008336
iteration:   14, best solution:   0.008336
iteration:   15, best solution:   0.006516
iteration:   16, best solution:   0.002443
iteration:   17, best solution:   0.002443
iteration:   18, best solution:   0.001535
iteration:   19, best solution:   0.001535
iteration:   20, best solution:   0.001535
iteration:   21, best solution:   0.001535
iteration:   22, best solution:   0.000919
iteration:   23, best solution:   0.000919
iteration:   24, best solution:   0.000919
iteration:   25, best solution:   0.000919
iteration:   26, best solution:   0.000919
iteration:   27, best solution:   0.000919
iteration:   28, best solution:   0.000102
iteration:   29, best solution:   0.000035


Best Position: [-0.003899158934036445, 0.004423475758581159]
Best Solution: 3.477057817963139e-05
```