

In [1]:

```

1  # Import PySwarms
2  import numpy as np
3  import pyswarms as ps
4  from pyswarms.utils.functions import single_obj as fx
5  import random
6  import matplotlib.pyplot as plt
7  from matplotlib.animation import FuncAnimation
8
9  import statistics # added for the mean computation
10 from collections import defaultdict # added to compare elements of the list
11 from itertools import tee # to allow pairwise comparisons
12 from scipy.spatial.distance import cosine # to compute cosine distance

```

In []:

1

In [2]:

```
1  # ... I also made some experiments with PySwarm
```

In [3]:

```
1  # Adapted from: https://machinelearningmastery.com/a-gentle-introduction-to-part
```

In [4]:

```

1  #n_particles = 10
2  #X = np.random.rand(2, n_particles)
3  #V = np.random.randn(2, n_particles)

```

In [5]:

```

1  #n_particles = 3
2  #print(np.random.rand(2, n_particles)*0.1 + 0.2)

```

In [6]:

```

1  n_particles = 3
2  print(np.random.rand(2, n_particles)*0.1 + 0.2)
3  print(np.random.rand(2, n_particles)*0.1 + 0.5)

```

```

[[0.26196383 0.27054804 0.23470743]
 [0.23396162 0.22369122 0.26346497]]
[[0.53385654 0.58111676 0.55029171]
 [0.59624815 0.53083626 0.50239047]]

```

In [7]:

```

1
2 def f(x,y):
3     "Objective function"
4     #return (x-0.9)**2 + (y-0.5)**2 # new
5     return (x-0.8)**2 + (y-0.9)**2 # September 14, tests 1, m, n
6
7 # Compute and plot the function in 3D within [0,5]x[0,5]
8 x, y = np.array(np.meshgrid(np.linspace(0,1,100), np.linspace(0,1,100))) # 1, n
9 z = f(x, y)
10
11 # Find the global minimum
12 x_min = x.ravel()[z.argmin()]
13 y_min = y.ravel()[z.argmin()]
14
15 # Hyper-parameter of the algorithm
16 c1 = c2 = 0.1 # 0.1
17 w = 0.8 # 0.8
18
19 # Create particles
20 n_particles = 10 # 20
21 np.random.seed(1000) # take away or leave it here?
22 X = np.random.rand(2, n_particles)*0.9 # I can generate them randomly but clos
23 V = np.random.rand(2, n_particles)*0.01
24 #X = np.random.rand(2, n_particles)*0.1 + 0.2
25 #V = np.random.rand(2, n_particles)*0.1 + 0.2
26
27
28 #X = np.random.rand(2, n_particles) * 5
29 #V = np.random.randn(2, n_particles) * 0.1
30
31
32
33 # 0.2 + 0.2; 0.01 + 0.5
34
35 # with these parameters, we are already on the target:
36 # X = np.random.rand(2, n_particles)* 0.9
37 # V = np.random.rand(2, n_particles)*0.01
38 # also with 0.2, 0.4
39
40
41 #X = np.random.rand(2, n_particles) * 5
42 #V = np.random.randn(2, n_particles) * 0.1
43
44 # Initialize data
45 pbest = X
46 pbest_obj = f(X[0], X[1])
47 gbest = pbest[:, pbest_obj.argmin()]
48 gbest_obj = pbest_obj.min()
49
50 def update():
51     "Function to do one iteration of particle swarm optimization"
52     global V, X, pbest, pbest_obj, gbest, gbest_obj
53     # Update params
54     # r1, r2 = np.random.rand(2)
55     r1, r2 = np.random.rand(2)
56     V = w * V + c1*r1*(pbest - X) + c2*r2*(gbest.reshape(-1,1)-X)
57     X = X + V
58     obj = f(X[0], X[1])
59     pbest[:, (pbest_obj >= obj)] = X[:, (pbest_obj >= obj)]

```

```

60     pbest_obj = np.array([pbest_obj, obj]).min(axis=0)
61     gbest = pbest[:, pbest_obj.argmin()]
62     gbest_obj = pbest_obj.min()
63
64     # Set up base figure: The contour map
65     fig, ax = plt.subplots(figsize=(8,6))
66     fig.set_tight_layout(True)
67     img = ax.imshow(z, extent=[0, 1, 0, 1], origin='lower', cmap='viridis', alpha=0.5)
68     fig.colorbar(img, ax=ax)
69     ax.plot([x_min], [y_min], marker='x', markersize=5, color="white")
70     contours = ax.contour(x, y, z, 10, colors='black', alpha=0.4)
71     ax.clabel(contours, inline=True, fontsize=8, fmt="%.0f")
72     pbest_plot = ax.scatter(pbest[0], pbest[1], marker='o', color='black', alpha=0.5)
73     p_plot = ax.scatter(X[0], X[1], marker='o', color='blue', alpha=0.5)
74     p_arrow = ax.quiver(X[0], X[1], V[0], V[1], color='blue', width=0.005, angles='xy')
75     gbest_plot = plt.scatter([gbest[0]], [gbest[1]], marker='*', s=100, color='black')
76     ax.set_xlim([0,1])
77     ax.set_ylim([0,1])
78
79
80     def animate(i):
81         "Steps of PSO: algorithm update and show in plot"
82         title = 'Iteration {:02d}'.format(i)
83         # Update params
84         update()
85         # Set picture
86         ax.set_title(title)
87         pbest_plot.set_offsets(pbest.T)
88         p_plot.set_offsets(X.T)
89         p_arrow.set_offsets(X.T)
90         p_arrow.set_UVC(V[0], V[1])
91         gbest_plot.set_offsets(gbest.reshape(1,-1))
92         return ax, pbest_plot, p_plot, p_arrow, gbest_plot
93
94     anim = FuncAnimation(fig, animate, frames=list(range(1,50)), interval=500, blit=False)
95     anim.save("PSO.gif", dpi=120, writer="imagemagick")
96
97     print("PSO found best solution at f({})={}".format(gbest, gbest_obj))
98     print("Global optimal at f({})={}".format([x_min,y_min], f(x_min,y_min)))
99
100
101
102     # putting these commands over there, we get the values at the end of the simulation
103     print("The X-coordinates are: ", X[0]) # Added on September 14
104     print("The Y-coordinates are: ", X[1]) # Added on September 14

```

2022-09-14 19:22:57,988 - matplotlib.animation - WARNING - MovieWriter
 imagemagick unavailable; using Pillow instead.

2022-09-14 19:22:57,990 - matplotlib.animation - INFO - Animation.save
 using <class 'matplotlib.animation.PillowWriter'>

PSO found best solution at $f([0.81399111 \ 0.87709839])=0.0007202346509739927$

Global optimal at $f([0.797979797979798, 0.8989898989898991])=5.101520253035246e-06$

the X coordinates are: $x=0.81467607 \ 0.81500604 \ 0.81300111 \ 0.81500770$

In []:

1
2

In [8]:

```
1 # define a class Robot_PSO with the x, y instances... or, directly work with the
2 # Not needed classes here, we already have the position outputs, and the X[0], X[1]
3
4 # class of the target (here: minimum of the objective function)
5
6 class Target:
7     def __init__(self,name,x,y): # no indetermination in the target's position
8         self.name = name
9         self.x = x
10        self.y = y
11
12 T = Target("T", 0.9, 0.5) # deep in the ocean
13
```

In [9]:

```
1  #listX = list(k.betax for k in Robot._registry)
2  #listY = list(k.betay for k in Robotx._registry)
3
4  listX = X[0]
5  listY = X[1]
6
7  num_of_robots = 10
8
9  def Euclidean_distance(T, listX, listY): # the same as distance_A
10     sum_x = sum(listX)
11     sum_y = sum(listY)
12     center_x = sum_x/num_of_robots
13     center_y = sum_y/num_of_robots
14     return ((T.x - center_x)**2 + (T.y - center_y)**2)**0.5
15
16 print("Euclidean", Euclidean_distance(T, listX, listY))
17
18 def Manhattan_distance(T, listX, listY):
19     sum_x = sum(listX)
20     sum_y = sum(listY)
21     center_x = sum_x/num_of_robots
22     center_y = sum_y/num_of_robots
23     return (abs(T.x - center_x) + abs(T.y - center_y))
24
25 print("Manhattan", Manhattan_distance(T, listX, listY))
26
27 def Cosine_distance(T, listX, listY):
28     sum_x = sum(listX)
29     sum_y = sum(listY)
30     center_x = sum_x/num_of_robots
31     center_y = sum_y/num_of_robots
32     array_1 = np.array([center_x, T.x])
33     array_2 = np.array([center_y, T.y])
34     return cosine(array_1, array_2)
35
36 print("Cosine", Cosine_distance(T, listX, listY))
```

Euclidean 0.3862294716670414

Manhattan 0.4618173714674332

Cosine 0.04972432184349462

In []:

1