Instituto Politécnico Nacional

Escuela Superior de Cómputo

Evolutionary Computing

Laboratory session #06: Particle Swarm Optimization (PSO)

Student: Vargas Romero Erick Efrain

Professor: Rosas Trigueros Jorge Luis

Practice completion date: November 5, 2021

Report delivery date: October 14, 2021

1 Theoretical framework

1.1 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a non-derivate, nature inspired evolutionary optimization algorithm to solve the complex real time problem. It is a robust stochastic optimization technique based on the movement and intelligence of swarms. It was developed by *James Knnedy* and *Russ Ebenhart* in 1997 based on the social behaviour of biological organisms that move in groups (swarms) such as birds and fishes. It has been applied successfully in wide variety of search and optimization problems by abstracting the working mechanism of natural phenomenon. Since PSO is population-based (swam) evolutionary algorithms, which has some similarities with GA. However a fundamental difference between these paradigms is that the evolutionary are based on a competitive philosophy, it means only the fittest individuals tends to survive. Conversely, PSO incorporates a cooperative approach to solve a problem, given that all individuals (particles), which can survive, change themselves over time and one particle's sucessfull adaptation is shared and reflected in the performance of its neighbours. The basic element of a PSO is a particle, which can fly throught search space towards an optimum by using its own information as well as the information provided by other particles comprising its neighbourhood. In PSO, a swarn of n particles (individuals) communicate either directly or indirectly with one another using search directions (gradients). The algorithm adopted was a "set of particles" flying over a search space to locate global optimum. During iteration of PSO, each particle updates according to its previous experience and experience of its neighbours. PSO has numerical applications in many areas such as geolgy, agriculture, finance, climate and ecology, electrical science, etc [1].

1.2 Ackley function

In mathematical optimization the *Ackley function* is a non-convex function used as a performance test for optimization algorithms. It was proposed by *David Ackley* in his 1987 PhD Dissertation [2].

$$f(x,y) = -20e^{-0.2\sqrt{0.5(x^2+y^2)}} - e^{0.5(cos(2\pi x)+cos(2\pi y))} + e + 20 \tag{1.1}$$

1.3 Rastrigin function

In mathematical optimization, the *Rastrigin function* is share some similarities with the *Ackley function*. It is a non-convex function used as a performance test problem for optimization algorithms. It is a typical example of non-linear multimodal function. This function was proposed in 1974 by *Rastrigin* as a 2-dimensional function and has been generalized by *Rudolph* as we can see in the eq. (1.2)[2].

$$f(x_1, x_2, \ldots, x_n) = 10n + \sum_{i=1}^{n}(x_i^2 - 10cos(2\pi x_i)) \tag{1.2}$$

Where where $-5.12 \leq x_i \leq 5.12$ and this function has the global minimum at $f(0,0,\ldots,0) = 0$.

The generalized version was popularized by *Hoffmeister*, *Bäck* and *Mühlenbein*. Finding the minimum of this function is a fairly difficult problem due to its large search space and its large number of local minimal.

For our case we are using the *Rastrigin function* in three dimensions so we get as result the next equation 1.3.

$$f(x,y,z) = 30 + x^2 + y^2 + z^2 - 10cos(2\pi x) - 10cos(2\pi y) - 10cos(2\pi z) \tag{1.3}$$

2 Material and equipment

The necessary material for this practice is:

- A computer with the latest *Python* stable version installed
- A text editor

Or is possible to use the google site `https://colab.research.google.com/` that allows us to use a virtual machine with an *Ubuntu* operative system with *Python* installed.

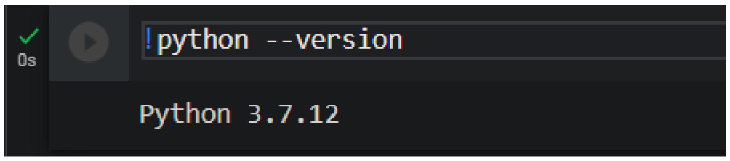
3 Practice development

3.1 Particle Swarm Optimization

To develop this practice we used the Google platform called *Colab* as this platform uses a virtual machine with linux (specifically Ubuntu) we can install some packets and of course we can verify if we have already *Python* installed. To check it we must use the command:

```
python --version
```

If we run this command in our linux terminal using *Colab* we can see the next as the result:



```
!python --version

Python 3.7.12
```

Figure 3.1: Verifying python version

The first step and the most important (in my humble opinion) is to understand the functionality of the algorithm. Particles move around the search space but according to specific behaviours, in our case it is an equation. As we reviewes in the theory the particles are guied by their own best-known position but also exists something called best-known position of the entire swarm.

Once that the algorithm finds the best position is mandatory to try to find something even better using that current best position to try to guide the swarm to an optimal solution. This process is repeated until we find a satisfactory solution.

Now lets talk about the function that we pretend to minimize, formally $f:\mathbb{R}^n \rightarrow R$ is the cost function. This function takes a possible solution and gets a real value as the output (at least for our case). The value that we get is used to find a solution for $f(a) \leq f(b)$ for all the b values in the search space, in other words we are looking for the global minimum.

Now that we known in general the most important about the algorithm we can discuss in depth about the work done in this practice. As we known from the professor we got a template about the PSO to implement the algorithm using two different fucntions, the first one is the Aukley's function and the second one was the Rastrigin's function. In both cases we tryied to find a way to avoid rewrite a lot of code, so we only modified the function called $f(x)$ as you can see below:

```python
def f(params):
    x = params[0]
    y = params[1]
    return -20 * math.exp(-0.2 * math.sqrt(0.5 * (x**2 + y**2))) - math.exp(0.5 * (
    math.cos(2 * math.pi * x) + math.cos(2 * math.pi * y))) + math.e + 20
```

As we known *params* is a *numpy* array so we only get each value to our function if we have a 3-dimensional function we only need to get other value, it makes it easier to manipulate, it is the case if we use the *Rastrigin's function*

```python
def f(params):
    x = params[0]
    y = params[1]
    z = params[2]
    return 30 + x**2 + y**2 + z**2 - 10 * math.cos(2 * math.pi * x) - 10 * math.cos
    (2 * math.pi * y) - 10 * math.cos(2 * math.pi * z)
```

As you can see of course we modified the equation but we are only adding an extra value called z and that is all.

In order to make it clear we add here the code for both programs, first of all we have the version for the Ackley's function

```python
#Aukley's function 2-dimensions

import ipywidgets as widgets
from IPython import display as display

import matplotlib.pyplot as plt
import numpy as np
import math

def createButton():
    button = widgets.Button(
        description = "Next iteration",
        disabled = False,
        button_style = "",  # 'success', 'info', 'warning', 'danger' or ''
        tooltip = "Next iteration",
        icon = "check",  # (FontAwesome names without the `fa-` prefix)
    )
    return button

def createButton10Iterations():
    button = widgets.Button(
        description = "Next 10 iterations",
        disabled = False,
        button_style = "",  # 'success', 'info', 'warning', 'danger' or ''
        tooltip = "Next iteration",
        icon = "check",  # (FontAwesome names without the ‘fa-‘ prefix)
    )
    return button

lowerLimit = -5
upperLimit = 5

nParticles = 30
nDimensions = 2

def f(params):
    x = params[0]
    y = params[1]
    return -20 * math.exp(-0.2 * math.sqrt(0.5 * (x**2 + y**2))) - math.exp(0.5 * (
    math.cos(2 * math.pi * x) + math.cos(2 * math.pi * y))) + math.e + 20

# initialize the particle positions and their velocities
# Bias the initial population
position = lowerLimit + 0.25 * (upperLimit - lowerLimit) * np.random.rand(nParticles
    , nDimensions)
speed = -(upperLimit - lowerLimit) / 2 + (upperLimit - lowerLimit) * np.random.rand(
    nParticles, nDimensions)

# initialize the global and local fitness to the worst possible
fitnessGlobalBest = np.inf
fitnessLocalBest = fitnessGlobalBest * np.ones(nParticles)

positionLocalBest = 1 * position
positionGlobalBest = 1 * positionLocalBest[0]

for i in range(0, nParticles):
    if f(positionLocalBest[i]) < f(positionGlobalBest):
        positionGlobalBest = 1 * positionLocalBest[i]

count = 0

def iteration(b):
    global count
    global position, positionLocalBest, positionGlobalBest, speed

    # Loop until convergence, in this example a finite number of iterations chosen
    weight = 0.5
    C1 = 0.3
    C2 = 0.2

    display.clear_output(wait = True)
    display.display(button)
    display.display(button10)
    count += 1

    print(count, "Best particle in:", positionGlobalBest, " gbest: ", f(
    positionGlobalBest))

    # Update the particle velocity and position
    for i in range(0, nParticles):
        for j in range(0, nDimensions):
            R1 = np.random.rand()  # uniform_random_number()
            R2 = np.random.rand()  # uniform_random_number()
            speed[i][j] = (
                weight * speed[i][j]
                + C1 * R1 * (positionLocalBest[i][j] - position[i][j])
                + C2 * R2 * (positionGlobalBest[j] - position[i][j])
            )
            position[i][j] = position[i][j] + speed[i][j]
        if f(position[i]) < f(positionLocalBest[i]):
            positionLocalBest[i] = 1 * position[i]
            if f(positionLocalBest[i]) < f(positionGlobalBest):
                positionGlobalBest = 1 * positionLocalBest[i]

def move10Interations(param):
    for i in range(0, 10):
        iteration(param)

button = createButton()
button.on_click(iteration)

button10 = createButton10Iterations()
button10.on_click(move10Interations)

display.display(button)
display.display(button10)
```

And the other program for the Rastrigin function:

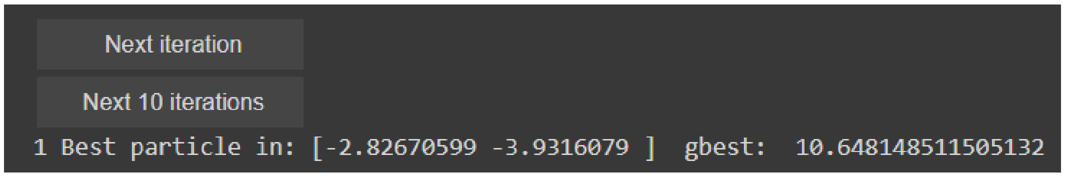
```
#Rastringin's function 3-dimensions

import ipywidgets as widgets
from IPython import display as display

import matplotlib.pyplot as plt
import numpy as np
import math

def createButton():
    button = widgets.Button(
        description = "Next iteration",
        disabled = False,
        button_style = "",  # 'success', 'info', 'warning', 'danger' or ''
        tooltip = "Next iteration",
        icon = "check",  # (FontAwesome names without the `fa-` prefix)
    )
    return button

def createButton10Iterations():
    button = widgets.Button(
        description = "Next 10 iterations",
        disabled = False,
        button_style = "",  # 'success', 'info', 'warning', 'danger' or ''
        tooltip = "Next iteration",
        icon = "check",  # (FontAwesome names without the `fa-` prefix)
    )
    return button

lowerLimit = -4
upperLimit = 4

```

```python
nParticles = 200
nDimensions = 3

def f(params):
    x = params[0]
    y = params[1]
    z = params[2]
    return 30 + x**2 + y**2 + z**2 - 10 * math.cos(2 * math.pi * x) - 10 * math.cos
    (2 * math.pi * y) - 10 * math.cos(2 * math.pi * z)

# initialize the particle positions and their velocities
# Bias the initial population
position = lowerLimit + 0.25 * (upperLimit - lowerLimit) * np.random.rand(nParticles
    , nDimensions)
speed = -(upperLimit - lowerLimit) + 3 * (upperLimit - lowerLimit) * np.random.rand(
    nParticles, nDimensions)

# initialize the global and local fitness to the worst possible
fitnessGlobalBest = np.inf
fitnessLocalBest = fitnessGlobalBest * np.ones(nParticles)

positionLocalBest = 1 * position
positionGlobalBest = 1 * positionLocalBest[0]

for i in range(0, nParticles):
    if f(positionLocalBest[i]) < f(positionGlobalBest):
        positionGlobalBest = 1 * positionLocalBest[i]

count = 0

def iteration(b):
    global count
    global position, positionLocalBest, positionGlobalBest, speed

    # Loop until convergence, in this example a finite number of iterations chosen
    weight = 0.5
    C1 = 0.2
    C2 = 0.3

    display.clear_output(wait = True)
    display.display(button)
    display.display(button10)
    count += 1

    print(count, "Best particle in:", positionGlobalBest, " gbest: ", f(
    positionGlobalBest))

    # Update the particle velocity and position
    for i in range(0, nParticles):
        for j in range(0, nDimensions):
            R1 = np.random.rand()  # uniform_random_number()
            R2 = np.random.rand()  # uniform_random_number()
            speed[i][j] = (
                weight * speed[i][j]
                + C1 * R1 * (positionLocalBest[i][j] - position[i][j])
                + C2 * R2 * (positionGlobalBest[j] - position[i][j])
            )
            position[i][j] = position[i][j] + speed[i][j]
        if f(position[i]) < f(positionLocalBest[i]):
            positionLocalBest[i] = 1 * position[i]
            if f(positionLocalBest[i]) < f(positionGlobalBest):
                positionGlobalBest = 1 * positionLocalBest[i]

def move10Interations(param):
    for i in range(0, 10):
        iteration(param)

button = createButton()
button.on_click(iteration)

button10 = createButton10Iterations()
button10.on_click(move10Interations)

display.display(button)
display.display(button10)
```

4 Screens, graphs and diagrams

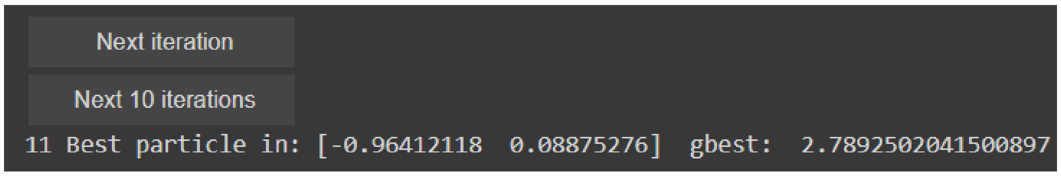
4.1 Aukley function

As we already known the global minimum for this function is in the position (0,0,0). As you can see in img. 4.1 we found a candidate to be the best answer, but if we continue iterating in the 11th generation img. 4.2 we found a particle even better and if we continue iterating in the iteration number 50 img. 4.3 we found something that is better than the previous particle, finally in the iteration 100 we found something so much better than the particle found in the iteration 50 img. 4.4.



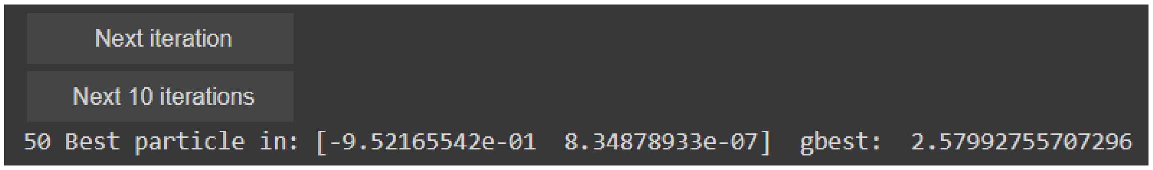
```
1 Best particle in: [-2.82670599 -3.9316079 ]  gbest:  10.648148511505132
```

Figure 4.1: PSO using Ackley's function iteration 1



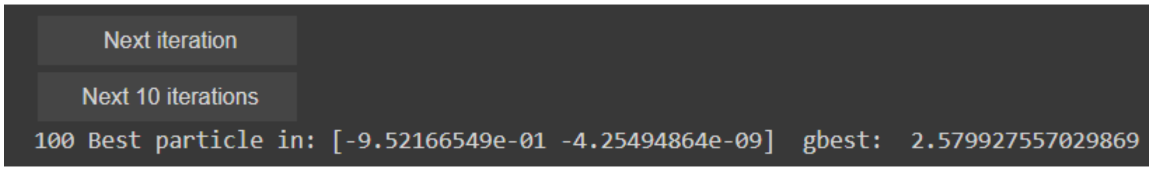
```
11 Best particle in: [-0.96412118  0.08875276]  gbest:  2.7892502041500897
```

Figure 4.2: PSO using Ackley's function iteration 11



```
50 Best particle in: [-9.52165542e-01  8.34878933e-07]  gbest:  2.57992755707296
```

Figure 4.3: PSO using Ackley's function iteration 50

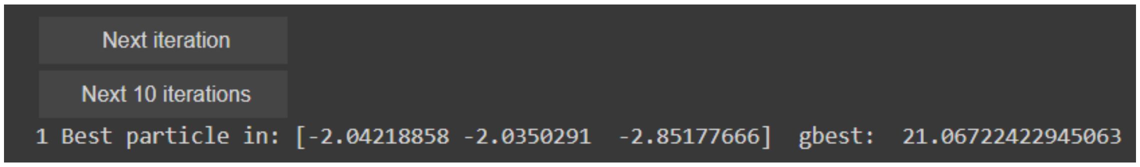


```
100 Best particle in: [-9.52166549e-01 -4.25494864e-09]  gbest:  2.579927557029869
```

Figure 4.4: PSO using Ackley's function iteration 100

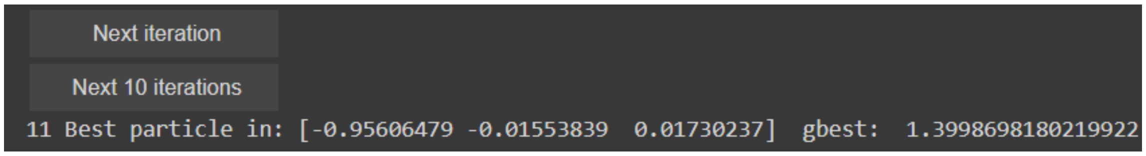
4.2 Rastrigin function

For the Rastrigin function the process has been very similar, first in the iteration number one img. 4.5 we found a possible solution but if we continue iterating the algorithn we found something better in the 11th generation img. 4.6 and if we keep going in generation 50 we found something better than in iteration 11 img. 4.7 but in the iterations 100 and 200 (img. 4.8 img. 4.9) you can see that we found something that probably is the best solution so far right now. Maybe if we continue repeating this process we will find something closer to $(0,0,0)$



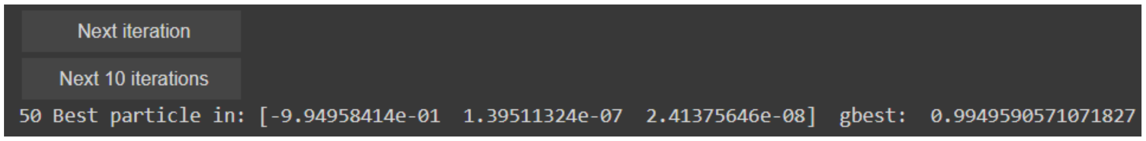
```
1 Best particle in: [-2.04218858 -2.0350291  -2.85177666]  gbest:  21.06722422945063
```

Figure 4.5: PSO using Rastrigin's function iteration 1



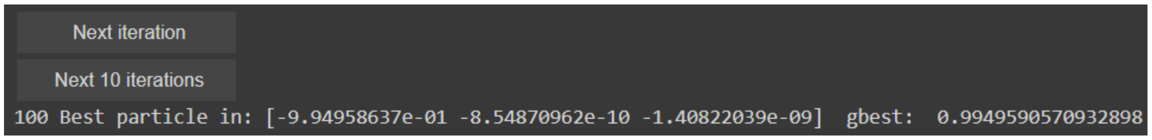
```
11 Best particle in: [-0.95606479 -0.01553839  0.01730237]  gbest:  1.3998698180219922
```

Figure 4.6: PSO using Rastrigin's function iteration 11



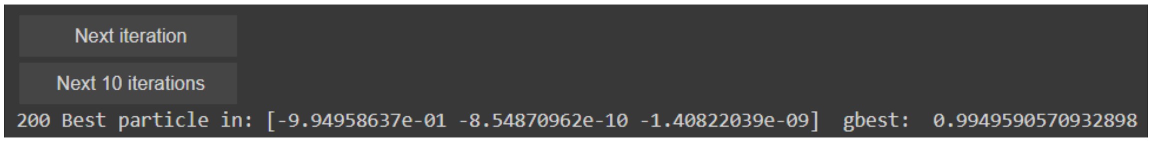
```
50 Best particle in: [-9.94958414e-01  1.39511324e-07  2.41375646e-08]  gbest:  0.9949590571071827
```

Figure 4.7: PSO using Rastrigin's function iteration 50



```
100 Best particle in: [-9.94958637e-01 -8.54870962e-10 -1.40822039e-09]  gbest:  0.9949590570932898
```

Figure 4.8: PSO using Rastrigin's function iteration 100



```
200 Best particle in: [-9.94958637e-01 -8.54870962e-10 -1.40822039e-09]  gbest:  0.9949590570932898
```

Figure 4.9: PSO using Rastrigin's function iteration 200

5 Conclusions

This practice the process has been a little bit different to the *Laboratory session 03: Genetic Algorithms Introduction*, first of all because we used a template to be sure if we find a solution to the *Rastrigin's function* nad *Ackley's function*. Something that is important to mention is that the PSO is very powerful and helps us to solve problems with a different technique that using a "common" genetic algorithm but both share that are nature behaviour based something that at least for me is amazing for example for this practice the swarm intelligence is something that probably we have already seen in our life for example in TV but try to understand the complexity of how a group of birds or fishes works to survive and try to implement something similar with an algorithm is very interesting.

Bibliography

[1] S. Rathod, K. Sinha and A. Saha *Particle Swarm Optimization and its applications in agricultural research*. November, 2014. Accessed on November 4th, 2021. Available online: `https://www.researchgate.net/publication/345415338_particle_swarm_optimization/link/5fa6412e92851cc2869ce1c7/download`

[2] L. Idoumghar, R. Schott and M. Melkemi. *A NOVEL HYBRID EVOLUTIONARY ALGORITHM FOR MULTI-MODAL FUNCTION OPTIMIZATION AND ENGINEERING APPLICATIONS*. September 2009. Accessed on November 04th, 2021. Available online: `https://www.researchgate.net/publication/235950622_A_novel_hybrid_evolutionary_algorithm_for_multi-modal_function_optimization_and_engineering_applications/download`