

Otimizations Based on Population Methods

Carlos Matheus Barros da Silva, *Bachelor Student of ITA*
Prof. Marcos Ricardo Omena de Albuquerque Máximo

Abstract—In this paper is avaliated a population based optimization called Particle Swarm Optimization (PSO). In order evaluate it, it was tested on a test function, and on a control problem - optimization of proportional-integral-derivative controller parameters. It was seen that *PSO* can find fast a solution, although it can get easily stocked on a local maximum. Its soluitons however are usually good enough for a number of problems.

Index Terms—Particle Swarm Optimization, PSO, population based optimization, optimization, proportional-integral-derivative controller, PID

I. INTRODUCTION

IN computer science, particle cloud optimization or particle swarm optimization (known by its acronym in English: PSO, of particle swarm optimization) refers to a metaheuristic that evokes the behavior of particles in nature.

PSO allows to optimize a problem from a population of candidate solutions, denoted as "particles", moving them throughout the search space according to mathematical rules that take into account the position and velocity of the particles. The movement of each particle is influenced by its best local position found so far, as well as by the best global positions found by other particles as they travel through the search space. The theoretical foundation of this is to make the particle cloud converge rapidly towards the best solutions.

PSO is a metaheuristic, since it assumes few or no hypotheses about the problem to be optimized and can be applied in large spaces of candidate solutions. However, like all metaheuristics, PSO does not guarantee obtaining an optimal solution in all cases.

II. PARTICLE SWARM OPTIMIZATION (PSO) IMPLEMENTATION

The implementation was based on the file *particle swarm optimization*. The essence is on *ParticleSwarmOptimization* Class and *Particle* Class development. The first one provides the interface in which the simulation communicates with the PSO.

Based on this, the development of *ParticleSwarmOptimization* Class was oriented on receive inputs, provides information, and have methods that triggers the algorithm continuation.

In order to achieve this, it was developed the methods shown from the Code 1 to the Code 6.

```
def __init__(self, hyperparams, lower_bound,
upper_bound):
    self.hyperparams = hyperparams
    self.lower_bound = lower_bound
    self.upper_bound = upper_bound
    self._particles = [Particle(lower_bound,
upper_bound, hyperparams, i) for i in range(hyperparams
.num_particles)]
```

```
self._best_value = -inf
self._best_particle = None
self._evaluate_idx = 0
```

Code 1. Constructor of *PSO* Class.

```
return self._best_particle.position
```

Code 2. Method *get best position* of *PSO* Class.

```
return self._best_value
```

Code 3. Method *get best value* of *PSO* Class.

```
particle = self._particles[self._evaluate_idx]
return particle.position
```

Code 4. Method *get position to evaluate* of *PSO* Class.

```
best_position = self.get_best_position()
for particle in self._particles:
    particle.update_particle_velocity(best_position)
)
particle.update_particle_position()
```

Code 5. Method *advance generation* of *PSO* Class.

```
actual_particle = self._particles[self._evaluate_idx]
if value > actual_particle.particle_best_value:
    actual_particle.particle_best_value = value
    actual_particle.particle_best_position =
actual_particle.position

if value > self._best_value:
    self._best_value = value
    self._best_particle = actual_particle

self._evaluate_idx += 1
if self._evaluate_idx >= self._hyperparams.
num_particles:
    self.advance_generation()
    self._evaluate_idx = 0
```

Code 6. Method *notify evaluation* of *PSO* Class.

The two main methods during the execution are *get position to evaluate* that returns to simulation the position of on particle, and the *notify evaluation* that gives the simulation result of that position to the *PSO* algorithm. With the result the code check if it is the best of that specific particle, or if it is the best overall. Then advance the index to next iteration.

This advance index step is quite important, because if the index reaches the population size, it calls the method *advance generation* that will update the velocity and the position of every particle and reset the index.

A. Particle Class Implementation

Objects of this class stores its position, its velocity, and its best position. The class provides, as well, methods to the particle update its velocity and its position.

The implementation of *Particle* Class is shown from Code 7 to Code 9.

```

upper_bound_hyper = 1.5

self.hyperparams = hyperparams
self.lower_bound = lower_b
self.upper_bound = upper_b * upper_bound_hyper
self.idx = i

num_of_params = len(self.lower_bound)

position = [0]*num_of_params
velocity = [0]*num_of_params

max_bound = max(max(self.lower_bound), max(self.
upper_bound))

for idx in range(len(position)):
    position[idx] = random.uniform(self.lower_bound
[idx], self.upper_bound[idx])
    velocity[idx] = random.uniform(-1*max_bound,
max_bound)

self.position = np.array(position)
self.velocity = np.array(velocity)

self.particle_best_position = self.position.copy()
self.particle_best_value = -inf

```

Code 7. Constructor of *Particle* Class.

```

def update_particle_velocity(self, best_position):
    rp = random.uniform(0.0, 1.0)
    rg = random.uniform(0.0, 1.0)

    inertial_factor = self.hyperparams.inertia_weight *
self.velocity
    cognitive_factor = self.hyperparams.
cognitive_parameter * rp * (self.particle_best_position
- self.position)
    social_factor = self.hyperparams.social_parameter *
rg * (best_position - self.position)

    self.velocity = inertial_factor + cognitive_factor
+ social_factor

```

Code 8. Method *update particle velocity* of *PSO* Class.

```

def update_particle_position(self):
    new_position = self.position + self.velocity
    for idx in range(len(new_position)):
        if not self.lower_bound[idx] < new_position[idx]
< self.upper_bound[idx]:
            self.velocity[idx] *= -1
            new_position[idx] = self.position[idx]
    self.position = new_position

```

Code 9. Method *update particle position* of *PSO* Class.

1) *Update velocity*: The update velocity method uses Equation 1. Where r_p and r_g are two different random numbers between 0 and 1 generated on every time the equation is calculated.

The ϕ_p is called the cognitive parameter, and it is a constant that multiplies the vector $(b_i - x_i)$ which is the direction to the best position of that particle.

The ϕ_g is called the social parameter, and it is a constant that multiplies the vector $(b_g - x_i)$ which is the direction to the best position in all particles.

Therefore the constants ϕ_p and ϕ_g balance how much important will be the best position locally and the best position globally.

Basically this is a balance between exploration and exploitation. The bigger the importance of the global best, the more probable to converge to a not so good local maximum. The bigger the importance of local best, the more probable to find a better solution, but with a more slow convergence.

$$v_{i+1} = \omega v_i + \phi_p r_p (b_i - x_i) + \phi_g r_g (b_g - x_i) \quad (1)$$

2) *Update position*: In order to update the particle position. Its current position is added to its calculated new velocity. This can be seen in the Code 9.

As professor Maximo suggested during his lecture, it was implemented a particle reflection, when one of them fall out of the boundaries while updating its position.

III. PSO RESULT IN TEST FUNCTION

As suggested, the PSO implementation was first tested using an easy test function in order to verify its correctness.

The cost function used in the test was described on Equation 2, in which has a global maximum [123].

$$f(x, y, z) = -((x - 1)^2 + (y - 2)^2 + (z - 3)^2) \quad (2)$$

In tests conducted using 40 particles and 1000 generations, the PSO converged rapidly to the global maximum. The position and quality of the results can be seen in Code 10 and Code 11.

```

00001: ['4.244', '2.703', '3.659']
00002: ['1.655', '4.074', '2.454']
00003: ['0.243', '3.719', '2.822']
00004: ['1.988', '4.002', '3.336']
00005: ['3.954', '2.921', '4.386']
00006: ['0.094', '1.245', '2.298']
00007: ['0.058', '0.277', '3.016']
00008: ['0.681', '3.481', '2.713']
00009: ['2.698', '2.018', '0.586']
00010: ['3.865', '1.417', '1.424']
...
39991: ['1.000', '2.000', '3.000']
39992: ['1.000', '2.000', '3.000']
39993: ['1.000', '2.000', '3.000']
39994: ['1.000', '2.000', '3.000']
39995: ['1.000', '2.000', '3.000']
39996: ['1.000', '2.000', '3.000']
39997: ['1.000', '2.000', '3.000']
39998: ['1.000', '2.000', '3.000']
39999: ['1.000', '2.000', '3.000']
40000: ['1.000', '2.000', '3.000']

```

Code 10. Positions results on Test function, shown only the first 10 and the last 10.

```

00001: -11.452
00002: -5.030
00003: -3.561
00004: -5.097
00005: -11.499
00006: -1.884
00007: -3.857
00008: -2.376
00009: -8.708
00010: -11.031
...
39991: -0.000
39992: -0.000
39993: -0.000
39994: -0.000
39995: -0.000
39996: -0.000
39997: -0.000
39998: -0.000
39999: -0.000
40000: -0.000

```

Code 11. Quality result on Test function, shown only the first 10 and the last 10

It is possible to see with the Figure 1 to Figure 3, the convergence on the best position, the overall convergence, and the convergence of the parameters.

Thus it is clear that the algorithm works and converges accordingly. That said, the algorithm is ready to be used on the line follower problem.

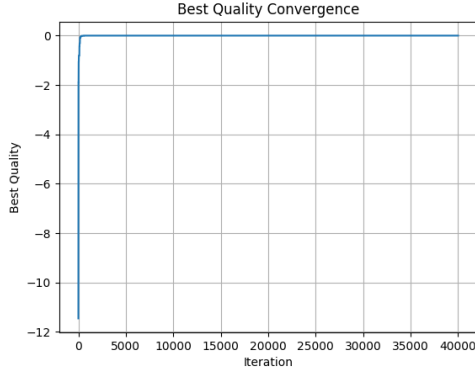


Fig. 1. Test best quality convergency

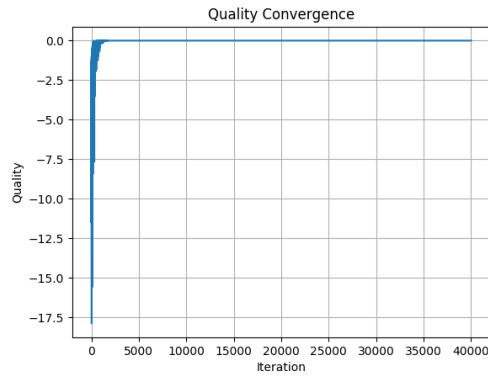


Fig. 2. Test quality convergency

IV. PSO ON LINE FOLLOWER PROBLEM

From a car circuit simulation, it was developed a line follower optimization on the *proportional–integral–derivative controller (PID controller)* Control technique.

From the Equation 3, the objective of this optimization is to optimize K_p , K_i , K_d , and the linear velocity of the car.

$$u(t) = K_p e(t) + K_i \int_0^t e(t') dt' + K_d \frac{de(t)}{dt} \quad (3)$$

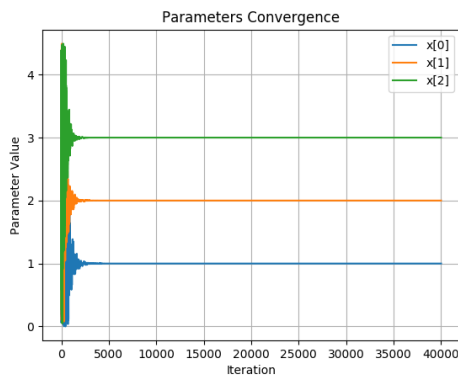


Fig. 3. Test parameters convergency

In order to do that, the *PSO* algorithm was run with 40 particles and it was developed the *evaluate* method. With this method every position of the car were evaluated and the sum of all evaluations was the score of the position of that particle.

The *evaluate* method can be seen in the Code 12, where by the end of the implementation was defined a high weight on linear speed and severe penalty for high errors and miss the track.

```
def evaluate(self):
    """
    Evaluates the current robot situation.

    :return: reward for the current robot situation.
    :rtype: float.
    """
    linear, angular = self.line_follower.get_velocity()
    # v_k, omega_k
    error, detection = self.line_follower.line_sensor.get_error() # e_k
    track_tangent = self.track.get_tangent(self.line_follower.pose.position) # t_k
    robot_direction = Vector2(cos(self.line_follower.pose.rotation), sin(self.line_follower.pose.rotation))
    # r_k
    dot_product = track_tangent.dot(robot_direction) # dot(r_k, t_k)

    penalty = 0
    if not detection:
        penalty -= 100
    if dot_product < 0.7:
        penalty -= 50
    penalty -= abs(error*100)
    bonus = (5*dot_product + 20*linear + 2*angular)
    score = bonus + penalty

    return score
```

Code 12. Method *evaluate* of *Simulation Class*, it is intended attribute a score for the situation of a particle.

In the firsts tests, it was clear that the convergency was always slow and always tending first to a quite slow velocity.

Therefore a higher weight was given to linear speed and the velocity upper bounder was increased.

A. Concise convergency

After the adjustment, the algorithm was run with 40 particles for about two hours, what resulted in 5477 iterations.

The quality history and the position history can be seen in the Code 13 and Code 14.

The convergencies can be seen in the figures from Figure 4 to Figure 5.

```
00001: [1.225676643187185, 56.211330519240406,
        653.1234529195855, 12.837701277825095]
00002: [0.3565902949694052, 75.3177289721953,
        504.57011457119165, 21.730278585480974]
00003: [1.2511739906303163, 187.9336877407604,
        76.18016394468776, 7.68891537765638]
00004: [1.2515616294712324, 223.23053980497562,
        579.0173040338675, 19.870242173536596]
00005: [0.3237209113182609, 48.78052763777478,
        592.4655948653118, 17.03450361256317]
00006: [0.039001448236550836, 104.43632762013195,
        1187.5784846482006, 41.05284055849808]
00007: [0.6717135859072237, 120.90764767410491,
        1515.628919697384, 22.580418511451903]
00008: [0.7612216082716723, 274.9311101686025,
        1374.9876058094464, 20.439664028944062]
00009: [0.8325782450789015, 224.35346113666452,
        1321.0070827502357, 0.7356200143216329]
00010: [1.3006801409202593, 249.00165418558345,
        1107.2709985663648, 36.60396593128962]
...
05468: [0.5986051313659557, 133.6024321626638,
        828.484865588594, 16.14557817693988]
```

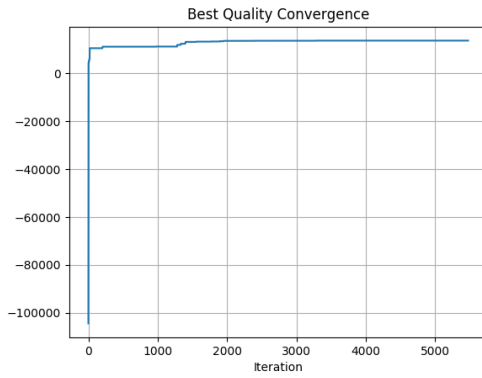


Fig. 4. Best quality convergence

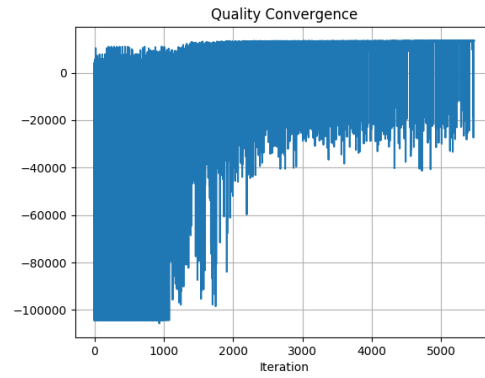


Fig. 5. Quality convergence

```

05469: [0.5986575496233267, 133.59077086754252,
      828.6199658936172, 16.149119326235567]
05470: [0.5987015088103654, 133.61826616052235,
      828.7509542242515, 16.149560200373198]
05471: [0.5986386846591139, 133.6251916579745,
      828.8159260688111, 16.147334508166956]
05472: [0.5986603085473429, 133.58180078582106,
      828.5789562111686, 16.149741017084704]
05473: [0.5986693753494389, 133.54926358408727,
      828.4026262079443, 16.1514112470073]
05474: [0.5986524240501516, 133.62263240517424,
      828.7944989571478, 16.147815898334713]
05475: [0.5975684100679194, 133.41589082512698,
      828.450393697157, 16.159370529654986]
05476: [0.598291399991801, 133.13915883700977,
      827.251992552975, 16.17686123514508]
05477: [0.598646086373311, 133.62198508696477,
      828.7952484332451, 16.147569255083646]

```

Code 13. Positions results on PSO optimization, shown only the first 10 and the last 10.

```

00001: -104352.23133052589
00002: 4214.807803543599
00003: -104352.23133052589
00004: -104352.23133052589
00005: -2090.7683044936607
00006: 4905.244143740709
00007: -77278.19907622323
00008: -66781.99714212856
00009: -64307.94226352465
00010: -104352.23133052589
...
05288: -19680.200896881557
05289: -22876.64329523349
05290: 13420.295096798776
05291: 13535.485396344737
05292: 13453.091696302565
05293: 13325.548757146422
05294: 13447.686307087304
05295: 13546.060933645334
05296: 13259.90533864725
05297: 13545.107624637552

```

Code 14. Quality result on PSO optimization, shown only the first 10 and the last 10

V. CONCLUSION

It was clear, therefore, that *PSO* method was implemented successfully. And by analysing its results, it can be seen that *PSO* has a fast conversion, but it does not converge to something necessarily great.

In fact, it finds local maximums and can easily be stocked on it. But for many cases it is good enough. In the case test, the solution found was able to complete the entire track in about 11 seconds, that is, it was really fast and precise at the same time.

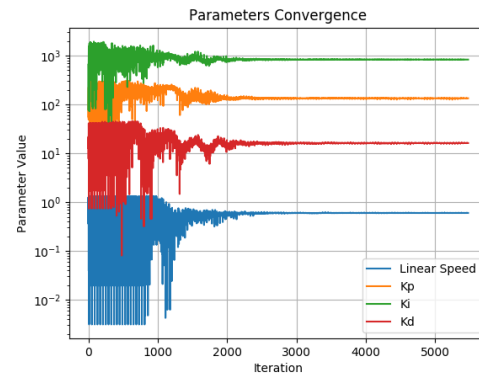


Fig. 6. Parameters convergency

As it is possible to see on the on Figure 7 and Figure 7 that at initials iterations the parameters are really bad, and with the time after many itarations it has a convergency to something good.

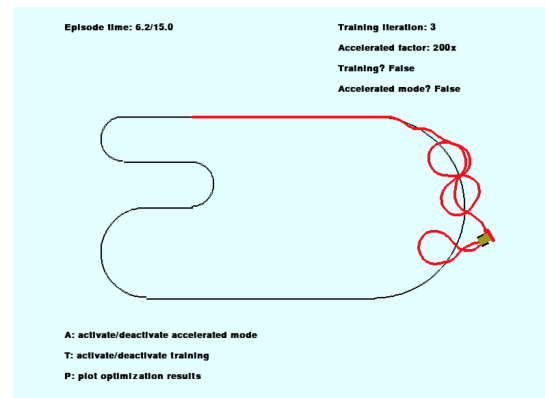


Fig. 7. Screenshot of the begging of the solution. Clearly the line follower was with bad parameter at this iteration.

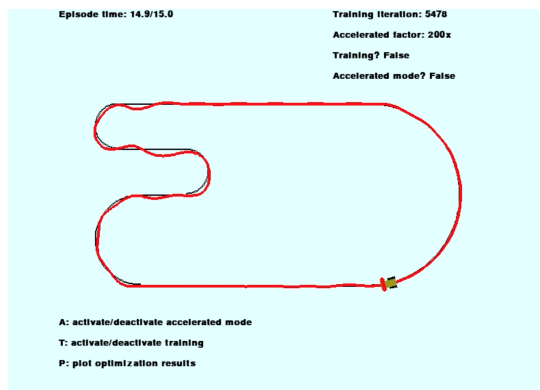


Fig. 8. Screenshot at iteration number 5478. Clearly the line follower is now with good parameters.