Proceedings of the 2009 IEEE International Conference on Systems, Man, and Cybernetics
San Antonio, TX, USA - October 2009

Impact of the Quality of Random Numbers Generators on the Performance of Particle Swarm Optimization

Carmelo J. A. Bastos-Filho, Júlio D. Andrade, Marcelo R. S. Pita, Alex D. Ramos
Department of Computing and Systems
Polytechnic School of Pernambuco — University of Pernambuco
Recife, Brazil
cjabf@dsc.upe.br

***Abstract*—Intelligent search algorithms are highly efficient to solve problems when it is not possible to use exaustive search. Particle Swarm Optimization (PSO) is a bio-inspired technique to perform search in continuous and hyperdimensional spaces. Despite it is common used to solve real world problems, a deeper study on the impact of the quality of Random Number generators has not been made yet. In this paper, we compare the performance of four variations of PSO algorithms in several benchmark functions considering five different Random Number Generators. PSO with inertia and constricted were analyzed. Global and local topologies were explored as well. The five different Random Numbers Generators are derived from Linear Congruential Generator (LCG) and the Marsaglia´s algorithm. We showed that PSO algorithms need random number generators with a minimum quality. However, we also showed that no significative improvements were achieved when we compared high quality random number generators to medium quality Random Number Generators.**

***Index Terms*—Particle swarm optimization, Random number generators.**

I. Introduction

Particle Swarm Optimization (PSO) is a computational intelligence technique proposed by Kennedy and Ebehart [1]. This technique is commonly used to solve a great variety of optimization problems. It is inspired on the social behavior of bird flocks. The main idea is to create particles that simulate the movements made by birds to reach something inside a specific search space. The technique explores the social behavior inside a group of individuals and its communication capacity. Each particle represents a solution in a high-dimensional space. The entire swarm uses specific communication mechanism to reach a common sense solution. This kind of intelligence is not produced by an individual of the swarm, but the synergy of their local interactions. The intelligence that emerges of local interactions is commonly referred as swarm intelligence.

The stochastic behaviour involved in these interactions helps to guide a more effective exploration in the search process. In algorithmic terms, it means that PSO uses random numbers generation to influences the behavior (*i.e.* movement) of particles.

Many PSO researchers normally do not care about the random number generators quality. In general, random number generators implemented in some standard library of the programming language are used without any investigation about its quality. The standard library of C and Java programming languages, for example, use the Linear Congruential Generator (LCG), that contains well known drawbacks.

PSO algorithms depend on the quality of the random numbers generator to mantain the capacity to explore efficiently the search space. In this paper, we explore the influence of some random number generators in PSO algorithms variations. This paper uses variations of two random number generators, the LCG and the Marsaglia´s generator [2]. The latter is known to be better than the former in randomness quality.

Sections II and III explain the variations of the PSO algorithm and random number generators analyzed in this work. Section IV and V show the experiments and results obtained. In section VI we give our conclusions.

II. Particle Swarm Optimization

PSO is a class of bio-inspired algorithms that can be used for optimization, mainly in continuous domains [3]. PSO is a population based algorithm and performs a distributed search regarding the memory achieved during the search process. In the most common PSO implementations, particles move through the search space using a combination of an attraction to the best solution that they have found individually, and an attraction to the best solution that any particle in the neighborhood has found.

Each particle contains three vectors: its current position in the *D*-dimensional search space $\vec{x}_i = (x_{i1}, x_{i2}, \cdots, x_{iD})$, its best position found in the search process so far $\vec{p}_i = (p_{i1}, p_{i2}, \cdots, p_{iD})$ and its current velocity $\vec{v}_i = (v_{i1}, v_{i2}, \cdots, v_{iD})$. Initial positions and velocities of particles are initialized randomly. In the original PSO, particles velocities and positions are updated iteratively, according to equations (1) and (2), respectively.

$$v_{id} = v_{id} + c_1 \cdot r_1 \cdot (p_{id} - x_{id}) + c_2 \cdot r_2 \cdot (p_{gd} - x_{id}), \quad (1)$$

$$x_{id} = x_{id} + v_{id}. \quad (2)$$

The constants c_1 and c_2 are the cognitive and social acceleration constants, respectively. They are used to weight the contribution of the cognitive and social components, respectively. The values r_1 and r_2 are random numbers, randomly generated using a uniform distribution in the interval $[0, 1]$. This updating process is performed for each dimension of the search space, $d = 1$ to D. The vector $\vec{p}_g$ is the best position inside the particle´ neighborhood.

Particles velocity can be clamped to a maximum value to avoid a search explosion state. Particles positions are also limited by search space bounds, avoiding inutile exploration of space, enhancing the probability of stagnation in local minima which is an undesirable event.

Each swarm can present its own communication mechanism to distribute the information of the best positions found in the search process. This information is extremely important in the swarm status updating process (velocity and position) and is exchanged based on the neighbourhood model. The neighborhood defines the part of the swarm which a particle is able to communicate with. Fig. 1 illustrates the most known approaches: global and local topologies.

In the global topology (or *gbest*), each particle communicate with all others. By using *gbest*, particles can spread information quickly through the swarm. It is analogous to a large community where all decisions taken are instantaneously known by everyone. Each particle in this topology is attracted towards the best solution found by the entire swarm.

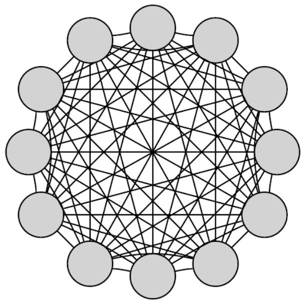
The most known information exchange mechanism based on local neighborhood is known as *lbest*. The particles only share information with their own neighbors. The structure used by this mechanism is also known as *ring* topology. Different regions of the search space can be explored at the same time. It occurs because the successful information from these regions takes a longer time to be sent to all other particles. Each particle communicates with its 2 immediate neighbors and attempts to imitate the best neighbor by moving closer to the best solution found within the neighborhood. Generally, this structure provides a better quality of solutions for multi-modal problems than the global topology, but takes a longer time to converge.

A. PSO with Inertia

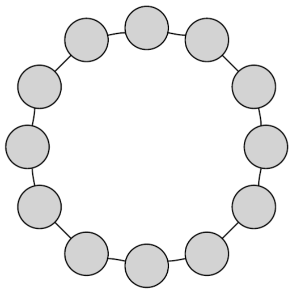
The original PSO updates the velocities fully considering the previous velocity of the particles. A variation of this updating process uses an inertia factor (ω), which considers only a fraction of the previous velocity in the update process.

The inertia factor (ω) [4] balances the exploration-exploitation ability by controlling the momentum of the particle by weighting the contribution of the velocity in the previous instant. It works by decreasing the value of velocity at each time step in order to increase the PSO exploitation level along the iterations. Equation (3) shows the resulting velocity update equation. The inertia factor must be in the interval $[0, 1]$. The most used strategy is to linearly decrease the inertia along the iterations.

$$v_{id} = \omega \cdot v_{id} + c_1 \cdot r_1 \cdot (p_{id} - x_{id}) + c_2 \cdot r_2 \cdot (p_{gd} - x_{id}). \quad (3)$$



(*a*)



(*b*)

Fig. 1: PSO topologies: (*a*) global; (*b*) local.

B. Constricted PSO

Another approach to update the particles velocities was proposed by [5]. In this approach, the constriction factor χ is defined according to equation (4).

$$\chi = \frac{2}{\left|2 - c - \sqrt{c^2 - 4c}\right|}, c = c_1 + c_2, c > 4. \quad (4)$$

The resulting velocity update equation for the constricted PSO is defined by equation (5).

$$v_{id} = \chi \cdot [v_{id} + c_1 \cdot r_1 \cdot (p_{id} - x_{id}) + c_2 \cdot r_2 \cdot (p_{gd} - x_{id})]. \quad (5)$$

The parameter χ controls the exploration-exploitation abilities of the swarm. $\chi \approx 1$ values are used to provide a high degree of exploration but with a slow convergence. For fast convergence, low values of χ are used.

III. Random Number Generators

Random number generators (RNG) are systems with the ability to generate sequences of random numbers according to a probability density function. These systems may be physical devices capable of generating real random sequence of numbers from physical phenomena. Mathematical algorithms can also be used as pseudo-random numbers generators when implemented in a computational device.

A sequence of numbers $X_0, X_1, X_2, \cdots, X_{N-1}$ is random when the number X_N cannot be predicted even if the previous numbers of the sequence $(X_0, \cdots, X_{N-1})$ are known. Therefore, computational implementations of random numbers generators based on mathematical algorithms do not generate truly random numbers, because if the initial numbers of a generated sequence and the algorithm behind the process are known, it is easy to predict the next numbers of the sequence. Despite that, numbers generated by mathematical algorithms are good enough for the majority of application because of their good statistical properties. Thus, these numbers can be informally referred as random numbers [6].

Mathematical algorithms to generate random numbers are very useful in simulations and other algorithms which depend on stochastic behaviours (*e.g.* PSO, for the initialization of particles and iterative generation of r_1 and r_2 values for velocity updating). In general, the values are uniformly distributed in a known real interval, but others types of distribution can be explored, like the normal distribution.

In the next subsections, we analyze two implementations of random numbers generators: the Linear Congruential Generator (LCG) and the George Marsaglia´s generator. The former is implemented in the majority of programming languages, while the last one is less popular but have better statistical properties than the first one.

A. Linear Congruential Generator

Linear Congruential Generator (LCG) is a classical random number generator that is implemented in many mathematical standard libraries of programming languages, such as C (`rand()` function) and Java (`java.lang.Random` class). The next number of a sequence for a seed X_0 generated by the LCG is defined by equation (6) [7]. The m value defines the period of the generator (*i.e.* the maximum number of generated elements without repetition). The LCG is very sensitive to the a (multiplier), c (increment) and m (modulus) parameters.

$$X_N = (a \cdot X_{N-1} + c) \mod m. \quad (6)$$

LCG are not suitable for applications where high-quality randomness is required (*e.g.* Monte Carlo simulations, cryptography), because it exhibits the following drawbacks: (*i*) the serial correlation between successive numbers of a sequence in a *n*-dimensional space lie on, at most, $m^{1/n}$ hyperplanes; (*ii*) if m is a power of 2, lower-order bits of the sequence have a very short period than the entire sequence.

B. Marsaglia's Generator

The Marsaglia´s 64-bit random number generator algorithm [2] used in this paper is an extension of the original 32-bit universal random number generator also proposed by Marsaglia [8]. The algorithm is based on two binary operations described by equations (7) and (8).

$$x \bullet y = \text{if } x \geq y \text{ then } (x - y), \text{ else } (x - y + 1); \quad (7)$$

$$c \circ d = \text{if } c \geq d \text{ then } (c - d), \text{ else } (c - d + r). \quad (8)$$

The method requires 97 initial seed values $x_1, x_2, \cdots, x_{97}$ and a parameter c_1 to generate the next seeds according to equations (9) and (10). A sequence of random numbers $(U_1, U_2, U_3, \cdots)$ can be obtained using equation (11).

$$x_n = x_{n-97} \bullet x_{n-33}, \quad (9)$$

$$c_n = x_{n-1} \circ r, \quad (10)$$

$$U_n = x_n \bullet c_n. \quad (11)$$

The period of the Marsaglia´s generator depends directly on the number of initial seeds. For scientific computational applications, two random initial integer seeds are sufficient to guarantee good statistical properties. In this case, the other seeds can be obtained from initial seeds. This approach was tested and passed, for example, on the Diehard Battery problem [9].

IV. Simulation Setup

The following benchmarks functions are all minimizing problems and were used in our experiments.

$$F_{Rosenbrock}(\vec{x}) = \sum_{i=1}^{n-1} \left[100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2\right], \quad (12)$$

$$F_{Rastrigin}(\vec{x}) = 10n + \sum_{i=1}^{n} \left[x_i^2 - 10\cos(2\pi x_i)\right], \quad (13)$$

$$F_{Griewank}(\vec{x}) = 1 + \sum_{i=1}^{n} \frac{x_i^2}{4000} - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right), \quad (14)$$

$$F_{Sphere}(\vec{x}) = \sum_{i=1}^{n} x_i^2, \quad (15)$$

$$F_{Schwefel1.2}(\vec{x}) = \sum_{i=1}^{n} \left(\sum_{j=1}^{i} x_j\right)^2, \quad (16)$$

$$F_{Schwefel2.6}(\vec{x}) = -\sum_{i=1}^{n} x_i sin\sqrt{x_i}, \quad (17)$$

$$\begin{aligned} F_{P8}(\vec{x}) &= \tfrac{\pi}{n}\Big\{10 \cdot sin^2(\pi \cdot y_1) \\ &+ \textstyle\sum_{i=1}^{n-1}(y_i - 1)^2 \cdot [1 + 10 \cdot sin(\pi \cdot y_i)] \\ &+ (y_n - 1)^2\Big\} + \textstyle\sum_{i=1}^{n} \mu(x_i, 10, 100, 4). \\ , y_i &= 1 + \tfrac{1}{4} \cdot (x_i + 1), \end{aligned}$$

$$\mu(x_i, a, k, m) = \begin{cases} k(x_i - a)^m, & x_i > a \\ 0, & -a \leq x_i \leq a \\ k(-x_i - a)^m, & x_i < a. \end{cases}$$

$$\begin{aligned} F_{P16}(\vec{x}) = \; & 0.1 \cdot \Big\{ sin^2(3\pi x_1) \\ & + \textstyle\sum_{i=1}^{n-1}(x_i - 1)^2 \cdot \{1 + sin(3\pi x_{i+1})\} \\ & + (x_n - 1)^2 \cdot \{1 + sin^2(2\pi x_n)\}\Big\} \\ & + \textstyle\sum_{i=1}^{2} \mu(x_i, 5, 100, 4). \end{aligned}$$

The search space, the initialization subspace and optimum point in the search space for all the benchmark functions are shown in Table I. Also on the tables, x^D is denoting a D-dimentional vector, where all the dimensions are real values. One can note that the optimum point for all functions is always far from the initialization space.

TABLE I: The search space, the initialization subspace and the optimum point in the search space for the benchmark functions used in the simulation.

Function	Search space	Initialization subspace	Optimum point
Rosenbrock	$-30 \leq x_i \leq 30$	$15 \leq x_i \leq 30$	1.0^D
Rastrigin	$-5.12 \leq x_i \leq 5.12$	$2.56 \leq x_i \leq 5.12$	0.0^D
Griewank	$-600 \leq x_i \leq 600$	$300 \leq x_i \leq 600$	0.0^D
Sphere	$-100 \leq x_i \leq 100$	$50 \leq x_i \leq 100$	0.0^D
Schwefel 1.2	$-100 \leq x_i \leq 100$	$50 \leq x_i \leq 100$	0.0^D
Schwefel 2.6	$-500 \leq x_i \leq 500$	$-500 \leq x_i \leq -250$	0.0^D
Penalized Function P8	$-50 \leq x_i \leq 50$	$25 \leq x_i \leq 50$	-1.0^D
Penalized Function P16	$-50 \leq x_i \leq 50$	$25 \leq x_i \leq 50$	1.0^D

In all the simulations, we performed 30 trials per function, where 10,000 iterations were executed for each trial in a 30 dimensions search space. We used 30 particles in all the simulations. The total number of fitness functions evaluations per trial was 300,000.

We simulate four variations of the Particle Swarm Optimization algorithm. We tested two types of velocity update equations, constricted and inertia. In both cases, we used $c_1 = 2.05$ and $c_2 = 2.05$. In the simulations using Inertia, we used w linearly decreasing from 0.9 to 0.4. We also evaluated the PSO performance in two different communication topologies, star and ring (also known as G_{best} and L_{best}), as shown in Fig. 1.

We compared five different Random Number Generators for all the situations and the case where r_1 and r_2 are static. The Random Number Generators vary in their statistical properties. We used the following Random Number Generators:

1. Marsaglia — Marsaglia generator using two seed with 10 digits. It is the best Random Number Generator used in this paper.

2. JAVA — Generator used by JAVA in Eclipse IDE based on Linear Congruential Generator (LCG).

3. LCG Excellent — A high quality Linear Congruential Generator (LCG) using $m = 2^{31} - 1$, $a = 163490618$ and $c = 0$.

4. LCG Good — A medium quality Linear Congruential Generator (LCG) using $m = 2^{32}$, $a = 398347535$ and $c = 0$. It is worse than LCG Excellent because it uses a square power m value.

5. LCG Bad — A poor quality Linear Congruency Generator (LCG) using $m = 2^{32}$, $a = 1$ and $c = 1$. It is even worse than LCG good. It generates a sequential series with regular intervals.

6. Static — In this case, we use a fixed number equal to 0.5. We used this option in order to assess the necessity to use Random Number Generators in particle swarm optimization algorithms.

V. Simulation Results

This section describes the results of the simulations performed in two situations for six different Ramdom Number approaches. The simulations were performed using Global and Local topologies, and Inertia and Constricted update velocity equations. Two analysis were performed: the analysis of the quality of random number generator on the update velocity equation and the analysis of the quality of random number generator on the particles initialization.

A. Analysis of the Quality of Random Number Generator on the Velocity Update Equation

Tables II, III, IV, V, VI, VII, VIII and IX present the results in terms of average and standard deviation values when we tested different Random Number Generator on the update velocity Equation for functions Rosenbrock, Rastrigin, Griewank, Sphere, Schwefel 1.2, Schwefel 2.6, Penalized 8 and Penalized 16, respectively. All the particles were initialized using the JAVA random number generator. Since we are interested to measure the influence of different RNGs, as lower are the values obtained for the minimization function for a particular RNG, as better is its performance.

In general, the four former Random Number Generators achieved similar performance. In all the cases (except for Penalized 16 function in static case for inertia velocity update equation and global topology), they achieved better results than LCG bad and static case. The LCG bad achieved bad results such as the static case. However, the LCG bad approach achieved even worse results when compared to the static case for inertia velocity update equation for functions Griewank, Sphere, Penalized 8 and Penalized 16. The LCG Good approach did not achieved a good performance for function Penalized 8 in the case where the global topology and the constricted update equation is used. We conclude that one should choose carefully the Random Number Generator, mainly when the inertia update equation is being used.

TABLE II: Results for Six Random Number Generators on the Velocity Update Equation for Rosenbrock function.

RNG	**Global Const.**	**Global Inertia**	**Local Const.**	**Local Inertia**
Marsaglia	2.73 (9.90)	26.36 (1450.39)	9.97 (38.67)	18.66 (453.06)
JAVA	7.16 (311.65)	40.17 (1094.02)	10.94 (200.18)	21.68 (609.34)
LCG excellent	2.49 (12.33)	33.47 (1309.25)	17.46 (467.27)	24.98 (554.02)
LCG good	2.314 (8.33)	32.38 (1076.99)	9.93 (48.85)	22.50 (607.77)
LCG bad	$2.11 \cdot 10^{6}$ $(2.88 \cdot 10^{12})$	$6.5 \cdot 10^{5}$ $(3.55 \cdot 10^{11})$	$7.4 \cdot 10^{5}$ $(4.02 \cdot 10^{11})$	$1.38 \cdot 10^{5}$ $(2.55 \cdot 10^{10})$
Static	$4.51 \cdot 10^{6}$ $(8.9 \cdot 10^{12})$	7372.67 $(4.18 \cdot 10^{7})$	$2.62 \cdot 10^{5}$ $(1.78 \cdot 10^{10})$	2233.04 $(5.82 \cdot 10^{6})$

TABLE III: Results for Six Random Number Generators on the Velocity Update Equation for Rastrigin function.

RNG	**Global Const.**	**Global Inertia**	**Local Const.**	**Local Inertia**
Marsaglia	50.17 (196.46)	14.36 (29.67)	41.75 (99.63)	22.3 (36.4)
JAVA	50.61 (191.41)	18.07 (38.78)	41.15 (55.94)	20.13 (26.07)
LCG excellent	53.32 (199.25)	15.25 (34.02)	41.68 (68.63)	21.26 (19.91)
LCG good	55.38 (289.9)	14.75 (33.31)	39.49 (64.39)	21.09 (20.34)
LCG bad	181.67 (637.1)	120.63 (671.09)	204.88 (417.30)	176.93 (1444.76)
Static	170.74 (743.2)	118.98 (851.85)	195.65 (291.58)	122.60 (648.39)

TABLE IV: Results for Six Random Number Generators on the Velocity Update Equation for Griewank function.

RNG	**Global Const.**	**Global Inertia**	**Local Const.**	**Local Inertia**
Marsaglia	0.028 (0.0012)	0.013 (0.00026)	0.0013 $(1 \cdot 10^{-5})$	0.0022 $(3.6 \cdot 10^{-5})$
JAVA	0.033 (0.00096)	0.015 (0.0004)	0.0014 $(1.5 \cdot 10^{-5})$	0.0032 $(3.2 \cdot 10^{-5})$
LCG excellent	0.033 (0.0012)	0.023 (0.00057)	0.0016 $(1.5 \cdot 10^{-5})$	0.00059 $(4.9 \cdot 10^{-6})$
LCG good	0.043 (0.0011)	0.021 (0.00024)	0.0018 $(2 \cdot 10^{-5})$	0.0029 $(4.2 \cdot 10^{-5})$
LCG bad	53.03 (540.95)	22.81 (183.35)	30.76 (88.86)	9.65 (29.23)
Static	73.52 (703.8)	3.4 (2.38)	17.14 (25.74)	2.08 (0.15)

TABLE V: Results for Six Random Number Generators on the Velocity Update Equation for Sphere function.

RNG	**Global Const.**	**Global Inertia**	**Local Const.**	**Local Inertia**
Marsaglia	$2.8 \cdot 10^{-154}$ $(2 \cdot 10^{-306})$	$1.41 \cdot 10^{-43}$ $(8 \cdot 10^{-86})$	$6.62 \cdot 10^{-70}$ $(3 \cdot 10^{-138})$	$9.8 \cdot 10^{-14}$ $(1.4 \cdot 10^{-26})$
JAVA	$8.2 \cdot 10^{-157}$ $(2 \cdot 10^{-317})$	$3.34 \cdot 10^{-43}$ $(2.6 \cdot 10^{-84})$	$1.7 \cdot 10^{-69}$ $(6 \cdot 10^{-137})$	$9.5 \cdot 10^{-14}$ $(2 \cdot 10^{-26})$
LCG excellent	$2.3 \cdot 10^{-156}$ $(2 \cdot 10^{-317})$	$6.32 \cdot 10^{-44}$ $(1.7 \cdot 10^{-86})$	$3.6 \cdot 10^{-70}$ $(5 \cdot 10^{-139})$	$1.57 \cdot 10^{-13}$ $(1.5 \cdot 10^{-25})$
LCG good	$2.6 \cdot 10^{-156}$ $(2 \cdot 10^{-317})$	$2.78 \cdot 10^{-43}$ $(5 \cdot 10^{-85})$	$1 \cdot 10^{-69}$ $(5 \cdot 10^{-138})$	$8.88 \cdot 10^{-14}$ $(1.1 \cdot 10^{-26})$
LCG bad	6028.87 $(6.74 \cdot 10^{6}))$	2407.19 $(2.73 \cdot 10^{6})$	3028.91 $(2.14 \cdot 10^{6})$	998.80 (359174)
Static	7253.56 $(7.63 \cdot 10^{-6})$	294.19 (25903.3)	1743.26 $(2.6 \cdot 10^{5})$	129.42 (2163.81)

TABLE VI: Results for Six Random Number Generators on the Velocity Update Equation for Schwefel 1.2 function.

RNG	**Global Const.**	**Global Inertia**	**Local Const.**	**Local Inertia**
Marsaglia	$4.11 \cdot 10^{-21}$ $(3 \cdot 10^{-40})$	4.41 (11.37)	0.00025 $(8.5 \cdot 10^{-8})$	148.13 (5483.16)
JAVA	$5.80 \cdot 10^{-21}$ $(3 \cdot 10^{-40})$	4.06 (13.60)	0.00046 $(2.3 \cdot 10^{-7})$	148.29 (7736.67)
LCG excellent	$4.75 \cdot 10^{-21}$ $(2 \cdot 10^{-40})$	5.65 (54.69)	0.00031 $(8.8 \cdot 10^{-8})$	133.99 (4295.81)
LCG good	$5.58 \cdot 10^{-21}$ $(3 \cdot 10^{-40})$	6.11 (53.53)	0.00038 $(1.4 \cdot 10^{-7})$	138.39 (4079.06)
LCG bad	27965.3 $(4.05 \cdot 10^{8})$	7644.33 $(2.36 \cdot 10^{7})$	5466.6 $(4.93 \cdot 10^{7})$	1550.7 (705320)
Static	15865.5 $(4.93 \cdot 10^{7})$	4035.81 $(7.05 \cdot 10^{6})$	2137.21 (878627)	401.86 (48670.9)

TABLE VII: Results for Six Random Number Generators on the Velocity Update Equation for Schwefel 2.6 function.

RNG	**Global Const.**	**Global Inertia**	**Local Const.**	**Local Inertia**
Marsaglia	4557.85 (261751)	2334.56 (119502)	4621.08 (113270)	3047.25 (170951)
JAVA	4604.32 (192287)	2223.36 (111698)	4371.94 (192169)	2906.29 (174526)
LCG excellent	4599.2 (307773)	2277.97 (92378.9)	4352.55 (311005)	2888.08 (187176)
LCG good	4620.24 (315258)	2210.2 (193096)	4535.76 (225459)	2996.92 (207537)
LCG bad	6657.23 $(2.5 \cdot 10^{6})$	7198.72 $(1.6 \cdot 10^{6})$	7692.03 $(1.8 \cdot 10^{6})$	7247.4 $(1 \cdot 10^{6})$
Static	6666.64 $(2.2 \cdot 10^{6})$	6480.94 $(1.8 \cdot 10^{6})$	6567.36 (956441)	5133.23 (791043)

B. Analysis of the Quality of Random Number Generator on the Particles Initialization

We tested different five Random Number Generators on the particles initialization for functions Rosenbrock, Rastrigin, Griewank, Sphere, Schwefel 1.2, Schwefel 2.6, Penalized 8 and Penalized 16, respectively. We did not analyze the static case because we think it is not fair to initialize all the particles in the same point with the same velocity.

In general, all the Random Number Generators achieved similar performance. However, LCG bad Random Number Generator failed for Griewank and Rastrigin function when we used the constricted velocity update equation. We think it happened because of the periodical behaviour of these functions. LCG bad Random Number Generator also failed

TABLE VIII: Results for Six Random Number Generators on the Velocity Update Equation for Penalized 8 function.

RNG	Global Const.	Global Inertia	Local Const.	Local Inertia
Marsaglia	$4.6 \cdot 10^{-32}$ $(3 \cdot 10^{-63})$	$5.13 \cdot 10^{-32}$ $(3 \cdot 10^{-63})$	$3.14 \cdot 10^{-33}$ $(4 \cdot 10^{-65})$	$3.14 \cdot 10^{-33}$ $(4 \cdot 10^{-65})$
JAVA	$6 \cdot 10^{-32}$ $(3 \cdot 10^{-63})$	$7.38 \cdot 10^{-32}$ $(3 \cdot 10^{-63})$	$8.37 \cdot 10^{-33}$ $(2 \cdot 10^{-64})$	$4.71 \cdot 10^{-33}$ $(5 \cdot 10^{-65})$
LCG excellent	$4.5 \cdot 10^{-32}$ $(2 \cdot 10^{-63})$	$4.13 \cdot 10^{-32}$ $(2 \cdot 10^{-63})$	$1.83 \cdot 10^{-32}$ $(1 \cdot 10^{-63})$	$5.4 \cdot 10^{-33}$ $(5 \cdot 10^{-65})$
LCG good	0.014 (0.0066)	$4.55 \cdot 10^{-32}$ $(2 \cdot 10^{-63})$	$9.42.10^{-33}$ $(2 \cdot 10^{-64})$	$6.8 \cdot 10^{-33}$ $(2 \cdot 10^{-64})$
LCG bad	126268 $(8.25 \cdot 10^{10})$	6852.16 $(2.08 \cdot 10^{8})$	4098.43 $(2.27 \cdot 10^{8})$	99.04 $(3 \cdot 10^{5})$
Static	$1.28 \cdot 10^{6}$ $(4.21 \cdot 10^{12})$	0.0013 $(3.8 \cdot 10^{-5})$	0.15 (0.10)	$5.75 \cdot 10^{-33}$ $(6 \cdot 10^{-65})$

TABLE IX: Results for Six Random Number Generators on the Velocity Update Equation for Penalized 16 function.

RNG	Global Const.	Global Inertia	Local Const.	Local Inertia
Marsaglia	1.615 (7.05)	7.61 (78.33)	0.20 (0.09)	1.38 (3.16)
JAVA	2.83 (35.29)	6.11 (115.20)	0.21 (0.21)	0.87 (1.31)
LCG excellent	0.84 (2.38)	7.37 (141.03)	0.12 (0.04)	1.01 (1.83)
LCG good	1.28 (7.08)	9.10 (211.07)	0.20 (0.10)	1.89 (5.34)
LCG bad	129.45 (7355.93)	49.40 (1915.2)	138.16 (4377.96)	23.42 (297.60)
Static	122.85 (2937.67)	2.73 (5.21)	48.0452 (303.31)	4.05 (3.94)

for Penalized 16 function.

VI. Conclusion

We investigated the influence of the quality of the Random Number Generators on the performance of Particle Swarm Optimization algorithms. The simulation results showed that the Particle Swarm Optimization algorithms analyzed in this paper need random number generators with a minimum quality. However, we also showed that no significative improvements were achieved when we compared high quality random number generators to medium quality random number generators. The medium quality Linear Congruential Generator (LCG) just failed in one case.

We think that this study was important to state that the most common used Random Number Generator (High Quality LCG) is sufficient in terms of statistical properties to guarantee the quality of Particle Swarm Optimization algorithms.

It is also important to notice that one should care about the quality of the Random Numbers to implementing Particle Swarm Optimization algorithms in embeeded systems.

Acknowledgment

The authors acknowledge the financial support from CNPq and FACEPE for scholarships and grant.

References

[1] J. Kennedy and R. Eberhart, *Particle Swarm Optimization*, Proceedings of the IEEE International Conference on Neural Network, 1995, pp. 1942-1948.

[2] G. Marsaglia and W. Tsang, *The 64-bit Universal RNG*, Statistics & Probability Letters, v. 66, 2004, pp. 183-187.

[3] D. Bratton and J. Kennedy, *Defining a Standard for Particle Swarm Optimization*, Proceedings of the IEEE Swarm Intelligence Symposium, 2007.

[4] R. C. Eberhart and Y. Shi, *Comparing inertia weights and constriction factors in particle swarm optimization* Proceedings of the Congress on Evolutionary Computation, 2000, pp. 84-88.

[5] M. Clerc J. and Kennedy, *The particle swarm - explosion, stability, and convergence in a multidimensional complex space*. IEEE Transactions On Evolutionary Computation, v. 6, 2002, pp. 58-73.

[6] R. Cassia-Moura, C. Sousa, A. Ramos, L. Coelho and M. Valena, *Yet another Application of the Monte Carlo Method for Modeling in the Field of Biomedicine*, Computer Methods and Programs in Biomedicine, v. 78, 2005, pp. 223-235.

[7] E. Weisstein, *Linear Congruence Method*, From Mathword — A Wolfram Web Resource. Available: http://mathworld.wolfram.com/LinearCongruenceMethod.html. Last access: 2009/3/6.

[8] G. Marsaglia, A. Zaman and W. Tsang, *Toward a Universal Random Number Generator*, Statistics and Probability Letters, v. 8, n. 5, 1990, pp. 35-39.

[9] G. Marsaglia, *The Marsaglia Random Number CDROM, with the Diehard Battery of Tests of Randomness*. Florida State University, The National Science Foundation. Available: http://www.stat.fsu.edu/pub/diehard. Last access: 2009/3/6.