Random Numbers and their Effect on Particle Swarm Optimization

Mark Rodgers

School of Computer Science & Informatics
University College Dublin

Abstract

Particle Swarm Optimization is a relatively new evolutionary computation technique. It is based on the social behaviour of birds flocking or fish schooling. It was designed to find optimal regions in a search space, with the biological idea of swarms in mind. In the Particle Swarm Optimization algorithm, particles which 'fly' around the search space have velocities associated with them. Updating these particles velocities can be described in one succinct equation which essentially updates the velocities of the particles in the system. A large part of this formula is affected by random numbers, as is it is clear from the formula that depending on the size, the random number generated can have a big say in the change of the velocity. The aim of this paper is to explore the effects of using different random number generation techniques, and specifically if it has an impact on the performance of Particle Swarm Optimization on three common optimization problems.

1. Introduction

Particle swarm optimization (PSO) was developed by Kennedy and Eberhart [1, 2]. It is an evolutionary computation technique, based on the idea of the social behavior of swarms of fish, birds or insects. It was discovered that these swarms had some method of sharing information with each other, e.g. the location of food, or the approach of danger. Hence there is the idea of individual knowledge and also a sense of swarm-wide knowledge.

The system is initialized with a *population of particles* in a search space. Each particle in the swarm represents a possible solution to the optimization problem being dealt with. These particles 'fly' through the search space, and thus have a position and a velocity attached to them. Particles update their velocity according to equation (1). Each particle has associated with it its current position, velocity and personal best location. This personal (local) best position (lbest) is the best location that any individual particle has come across in the run thus far. It may be updated as the particle progresses, depending on whether the lbest value is exceeded. There is also a global best (gbest) position, which is the best solution that any particle in the entire swarm has discovered i.e. the best value among all the lbest values. The equation for updating the velocity is:

$$v_i(t) = wv_i(t-1)+c_1r_1(p_i- x_i(t-1))+ c_2r_2(p_g - x_i(t-1)) \qquad (1)$$

$$x_i(t) = v_i(t)+ x_i(t-1) \qquad (2)$$

Where v_i is the previous velocity, w is the inertia weight, c_1 and c_2 are positive constants, p_i represents the personal best position of the i^{th} particle and p_g denotes the global best position of the swarm, r_1 and r_2 are random numbers in the region 0 to 1. In (1) w is the inertia weight. This was not in the original PSO formula; it was introduced as an improvement later by Eberhart and Shi [4]. It is generally incorporated into the PSO equation since it is advantageous to the performance [3]. Since the search space is D-dimensional, the i^{th} particle can be represented by a D-dimensional vector, (2). So x_i would be the position of a particle, and v_i is the velocity being added. This allows us to update the particle and determine its position.

Velocities of particles are typically limited to a maximum velocity V_{max}. The maximum velocity can be set by the user and can affect the performance of PSO. It has been shown that a larger V_{max} helps towards global exploration, and conversely a smaller V_{max} promotes local exploitation [3]. The parameter must be carefully set since too little velocity means the swarm will converge on the first good solution that is found, while too much means the swarm may never converge on a solution. Since this is a relatively new research area, several other improvements have been documented including convergence insurance [4] and research into parameter selection [5].

The PSO approach is different to Genetic Algorithms (GA) in several ways. Firstly, it traditionally has no crossover between individual particles. Neither does it have mutation. Adding these features into PSO are ongoing research topics [6]. Also particles are never substituted for other individuals during a run; instead they are continually updated in the hope of finding an optimum solution.

Looking at equation (1) it is clear to see that the random number element has an influence on the overall outcome of $v_i(t)$. I propose to investigate whether using different methods of random number generation will have an affect on the performance of the PSO equation, and which method gives the best results. Firstly I will give a brief introduction to the concept of random number generation.

2. Random Number Generation

Today there are many reasons for people to harness random number generation, such as in statistical methods. In computing, the area of cryptography and modeling physical and biological processes uses random numbers extensively [7]. Since computers are based on a fixed hardware, and algorithms are deterministic, they cannot be used to create truly random numbers. When we talk about random numbers generated by computers we actually mean *pseudorandom numbers*, since they are not entirely random. These pseudorandom numbers are typically generated by a program which takes random bits as an input. However, if the seed (initial bit sequence entered) is the same, then the algorithm will usually return a similar sequence of numbers.

Truly random numbers are generated by random processes. E.g. a fair die roll, atmospheric noise, radioactive decay, or even picking a ball from an urn. There are several sites online dedicated to providing true random numbers by sampling a physical process. One such site, random.org, samples atmospheric noise [8]. Mr. Haahr provides a server which lets people access these truly random numbers through a variety of interfaces. In this experiment I connect to the server using the Java interface [9] which lets me use these numbers.

Java provides methods for random number generation (RNG) (java.util.Random, Math.random()). Old versions (Math.random()) of this implementation used the computers clock as the seed. A problem would be that since time is only measured to a granularity of 1 millisecond if you generate 2 or more random number sequences within one millisecond of each other they generate the exact same sequence of numbers [10]. The new version (java.util.Random) uses a 48-bit seed, which is modified using a linear congruential formula [11]. Though it is still possible for 2 instances of Random to have the same seed, the same sequence of method calls must be made for the sequences to be the same.

Ideally, these numbers would portray no recognizable repetitions, but since their creation is derived from an algorithm, they are not truly random. Math.random() is the default method of number generation used in the JSwarm software for experimenting with PSO. JSwarm will be discussed shortly.

3. Experimental Studies

With the aim of discovering what effect different methods of number generation have on the performance of the PSO equation, three benchmark test functions were used: Rosenbrock, Schaffer and Sphere. These are well known problems and have been used before in the field of evolutionary computations [12, 13]. The sphere function is a continuous, strongly convex unimodel function. It is a good measure of convergence velocity. In terms of a formula,

$$f(x) = \sum_{i=1}^{D} x_i^2 ,$$

sums up the sphere function.

The Schaffer function is a simple optimization problem, stated as:

$$f(x) = 0.5 + \frac{\sin^2 \sqrt{x^2 + y^2} - 0.5}{\left(1 + 0.001 \cdot \left(x^2 + y^2\right)\right)^2},$$

The Schaffer equation can be made quite difficult to solve [14], but in this case it is a Schaffer function that has to be minimized, which is easier to solve.

Finally, the Rosenbrock function has a global minimum of zero at the point (1, 1), in a parabolic shaped narrow valley [15]. Its general form is;

$$f(x) = \sum_{i=1}^{D-1} 100 \cdot \left(x_{i+1} - x_i^2\right)^2 + \left(1 - x_i\right)^2,$$

3.1 A Working Introduction to JSwarm-PSO

JSwarm is a PSO package written in Java. It is freely available for download [16]. It is simple to use, and comes with a range of sample equations (Alpine, Rastrijin's function etc.) and examples. It is also highly editable, in that anybody with a decent knowledge of Java can change the code to suit their own needs. In the case of this experiment the ParticleUpdate.java file was edited to modify the random number generator being used. Likewise, the fitness function, particle, swarm and evolve etc. classes can be changed. After changing the files, they must be recompiled and run them from the command line. A possible output looked like this:

```
Begin: Example Rosenbrock
Best fitness: 8.90954651565993E-6
Best position:  [0.9499273711223972, 0.92334509482122]
Number of evaluations: 250000
End: Example Rosenbrock
```

From this output, the best fitness value for that run can be extracted and likewise for the other experiments.

3.2 Starting the Experiments

Four different situations to test the hypothesis of this paper, the first being the equation with no random number element, the second the using the standard Math.random(), the third using the improved java.util.Random, and the fourth being the true random number generation provided by the random.org server. In the case of there being no random number generator I set the value in the equation to be 1, since multiplying any number by 1 will not effect its value. Also, maximum number of iterations is set to 10,000.

Each test function was run 30 times with the different configurations to give reliable results, although this could be improved on if more time were available. Then the average fitness at the end of the run was calculated and the best fitness that was obtained during the whole run was recorded.

4. Experimental Results & Analysis

ROSENBROCK	Average Fitness	Best Fitness
No RNG	1.67E-3	1.77E-3
Math.random()	4.30E-5	4.23E-5
java.util.Random	4.34E-5	4.19E-5
Random.org	4.40E-5	4.36E-5

Table 1. Results for Rosenbrock experiments.

SPHERE	Average Fitness	Best Fitness
No RNG	2.0E-4	2.21E-4
Math.random()	5.63E-5	5.03E-5
java.util.Random	5.50E-5	5.06E-5
Random.org	5.01E-5	4.78E-5

Table 2. Results for Sphere experiments.

SCHAFFER	Average Fitness	Best Fitness
No RNG	1.91E-4	2.51E-4
Math.random()	3.37E-5	3.28E-5
java.util.Random	3.45E-5	3.30E-5
Random.org	3.29E-5	3.14E-5

Table 3. Results for Schaffer experiments.

The results obtained from the experiments are tabulated in tables 1 to 3. On inspection, it is clear to see that running the PSO algorithm with r_1 and r_2 set to 1 gives relatively poor results, in comparison to when a random number is inserted. Meanwhile, the three random number generators give fairly similar results throughout.

In the Rosenbrock experiments, the run using Math.random() ended up having the best average fitness. All of the functions were to be 'minimized' which means results closer to zero are more desirable. The java.util.Random had worse average fitness than the Random.org implementation, but both were much better than running the experiments without a random number generator. I speculate that it could be possible that the PSO algorithm takes advantage of possible slight repetitions in the numbers being generated. Unfortunately, not enough is currently known about the internal workings of the algorithm to explain this [17]. It is also worth noticing that the best fitness was achieved by a particle in the java.util.Random experiments, even though on average it was outperformed by Math.random().

In the Sphere experiments, the results show that the Random.org number generator gives the best result. It is a good bit better than the java.util.Random method in this case. In these experiments there are the largest gaps between the average fitness and the best fitness values. This could be because of the nature of the function, with some particles finding better optima but due to the swarming technique, they have to move back from it slightly. The java.util.Random method does slightly better than the Math.random() this time around, but they achieve very similar results. As expected, the experiments with no random number element fail to produce decent results compared to those experiments which use random numbers.

In the final experiment with the Schaffer function, the averages of the 3 random number generators are fairly close together. However the Random.org generator produces slightly better results than the others. It also gives the best fitness for all the experiments for this function. Math.random() does a little better than java.util.Random again this time, again possibly due to the nature of the function being used.

5. Conclusions & Future Work

This paper looked at the effects using different random number generators have on the performance of the PSO algorithm by testing three different benchmark functions. After looking at the results we can see that there exist some discrepancies in performance when different random number generators are used. Twice in these experiments the truly random number generator performed better in the average fitness results than both Java's random number generators. Both of the Java RNG's performed quite similar in the tests, with the older Math.random() proving better on two occasions. It looks conclusive that having a random number generator is many times better than not having one.

It would be beneficial to do further testing and for more runs to thoroughly verify these results. It would certainly help incorporating more benchmark functions for further testing such as the Griewank function and the Rastrijin's function. In addition, variables could be changed away from the normally accepted values [18] to other values to try and discover a larger difference between these RNG's. Also perhaps by sensitivity analysis stronger evidence could emerge [19].

References

1. Kennedy, J., and Eberhart, R. C. (1995), Particle swarm optimization, Proc. IEEE International Conference on Neural Networks (Perth, Australia), IEEE Service Center, Piscataway, NJ, IV: 1942-1948.
2. Eberhart, R. C., and Kennedy, J. (1995), A new optimizer using particle swarm theory, Proc. Sixth International Symposium on Micro Machine and Human Science (Nagoya, Japan), IEEE Service Center, Piscataway, NJ, 39-43.
3. K. E. Parsopoulos and M. N. Vrahatis. Recent approaches to global optimization problems through particle swarm optimization. Natural Computing 1 (2002) 235–306.
4. Y. Shi and R. Eberhart. A modified particle swarm optimizer. IEEE Int. Conf. on Evolutionary Computation (1997) 303–308
5. Shi, Y. and Eberhart, R. (1998) Parameter selection in particle swarm optimization. In Evolutionary Programming VIZ: Proc. EP98, New York: Springer Verlag pp. 591-600.
6. Eberhart, R. and Shi, Y. (1998). Comparison between Genetic Algorithms and Particle Swarm Optimization, The 7th Annual Conference on Evolutionary Programming, San Diego, USA.
7. James E. Gentle, Random Number Generation and Monte Carlo Methods (2000). Springer.
8. Mads Haahr, Department of Computer Science, Trinity College Dublin : www.random.org
9. See http://random.org/clients/soap/RandomDotOrg.tar.gz
10. Jonathan Knudsen, Java Cryptography (1998) (Book, pg 38). O'Reilly.
11. See the Java API.
12. K. A. De Jong, An analysis of the behavior of a class of genetic adaptive systems, Ph.D. dissertation, Univ. Michigan, Ann Arbor, 1975.
13. Carlos Artemio Coello, Gary B. Lamont, David A. Van Veldhuizen, Evolutionary Algorithms for Solving Multi-Objective Problems. Kluwer Academic Publishers, New York, May 2002. ISBN 0-3064-6762-3.
14. C.Wei, Z. He,Y. Zhang, and W. Pei, Swarm directions embedded in fast evolutionary programming, in Proc. Congr. Evolutionary Computation (CEC 02), vol. 2, 2002, pp. 1278, 1283.
15. Rosenbrock, H. H. "An Automatic Method for Finding the Greatest or Least Value of a Function." Computer J. 3, 175-184, 1960.
16. See http://sourceforge.net/projects/jswarm-pso/
17. M. Clerc and J. Kennedy, The Particle Swarm: Explosion, Stability and convergence in a Multi-Dimensional Complex Space. IEEE Transactions on Evolutionary Computation, in press.
18. A. Carlisle and G. Dozier. An Off-The-Shelf-PSO. In Proceedings of the Particle Swarm Optimization Workshop, pages 1–6, 2001.
19. T. Beielstein, K.E. Parsopoulos, and M.N. Vrahatis. Tuning PSO parameters through sensitivity analysis. Technical report of the Collaborative Research Centre 531 Computational Intelligence CI-124/02, University of Dortmund, Jan 2002