rssbus
SEE THE WORLD AS A DATABASE
ADO.NET • JDBC • ODBC • SQL (SSIS)
ODATA • MYSQL • EXCEL
SAP
QuickBooks
DOWNLOAD NOW

Artificial Intelligence

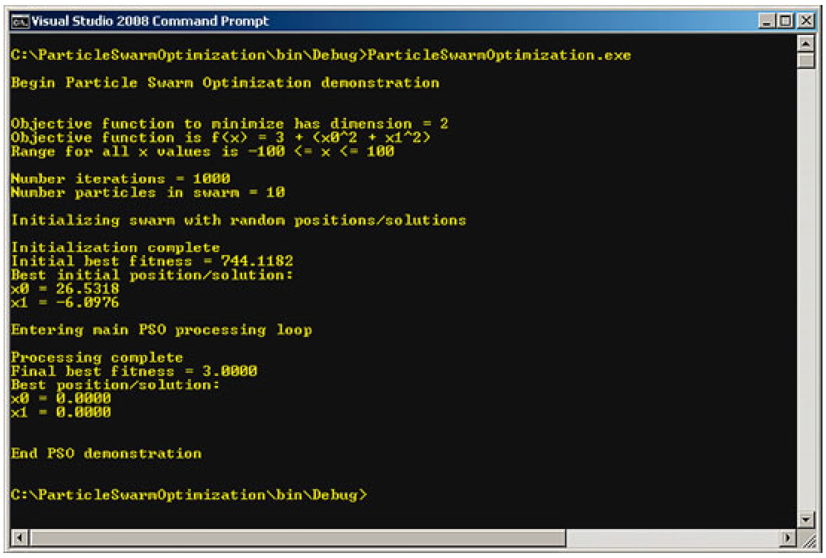
Particle Swarm Optimization

James McCaffrey

Download the Code Sample

Particle swarm optimization (PSO) is an artificial intelligence (AI) technique that can be used to find approximate solutions to extremely difficult or impossible numeric maximization and minimization problems. The version of PSO I describe in this article was first presented in a 1995 research paper by J. Kennedy and R. Eberhart. PSO is loosely modeled on group behavior, such as bird flocking and fish schooling. The best way for you to get a feel for what PSO is and to see where I'm heading here is to examine **Figure 1**.

The first part of the figure describes a dummy problem being solved by a demonstration PSO program. The goal is to find values $x_0$and x_1 so the value of the function $f = 3 + x_0^2 + x_1^2$ is minimized. In the print edition of the article, I use the notation ^2 to indicate the squaring operation. Notice that I've deliberately chosen an unrealistically simple problem to keep the ideas of PSO clear. It's obvious that the solution to this problem is $x_0 = 0.0$ and $x_1 = 0.0$, which yields a minimum function value of 3.0, so using PSO isn't really necessary. I discuss more realistic problems that can be solved by PSO later in this article. In this example, the dimension of the function to minimize is 2 because we need to solve for 2 numeric values. In general, PSO is well-suited to numeric problems with dimensions of 2 or larger. In most situations, PSO must have some constraints on the range of possible x values. Here x_0 and x_1 are arbitrarily limited to the range -100.0 to 100.0.



```
C:\ParticleSwarmOptimization\bin\Debug>ParticleSwarmOptimization.exe

Begin Particle Swarm Optimization demonstration

Objective function to minimize has dimension = 2
Objective function is f(x) = 3 + (x0^2 + x1^2)
Range for all x values is -100 <= x <= 100

Number iterations = 1000
Number particles in swarm = 10

Initializing swarm with random positions/solutions

Initialization complete
Initial best fitness = 744.1182
Best initial position/solution:
x0 = 26.5318
x1 = -6.0976

Entering main PSO processing loop

Processing complete
Final best fitness = 3.0000
Best position/solution:
x0 = 0.0000
x1 = 0.0000

End PSO demonstration

C:\ParticleSwarmOptimization\bin\Debug>
```

Figure 1 **Particle Swarm Optimization Demo Run**

The next part of **Figure 1** indicates that the PSO program is using 10 particles and that the program will

MSDN Magazine Blog

November Issue of MSDN Magazine
As I wrote earlier, the Government Special Issue of MSDN Magazine, currently live on our Web site, is worth checking out. The articles in the special... More...
Tuesday, Nov 5

Government Special Issue of MSDN Magazine
Visit the MSDN Magazine Web site and you'll see we've been a bit busier than usual of late. In addition to the regularly scheduled November issue, fea... More...
Friday, Nov 1

More MSDN Magazine Blog entries >

Current Issue

Browse All MSDN Magazines

Subscribe to MSDN Flash newsletter

Receive the MSDN Flash e-mail newsletter every other week, with news and information iterate 1,000 times. As you'll see shortly, each particle represents a possible solution to the PSO problem being solved. PSO is an iterative technique and in most cases it's not possible to know when an optimal solution has been found. Therefore, PSO algorithms usually must have some limit on the number of iterations to perform.

The next lines in **Figure 1** indicate that each of the 10 particles in the swarm is initialized to a random position. A particle's position represents a possible solution to the optimization problem to be solved. The best randomly generated initial position is x_0 = 26.53 and x_1 = -6.09, which corresponds to a fitness (the measure of solution quality) of $3 + 26.53^2 + (-6.09)^2 = 744.12$. The PSO algorithm then enters a main processing loop where each particle's position is updated on each pass through the loop. The update procedure is the heart of PSO and I'll explain it in detail later in this article. After 1,000 iterations, the PSO algorithm did in fact find the optimal solution of x_0 = 0.0 and x_1 = 0.0, but let me emphasize that in most situations you won't know whether a PSO program has found an optimal solution.

In this article, I'll explain in detail the PSO algorithm and walk you line by line through the program shown running in **Figure 1**. I coded the demo program in C#, but you should be able to easily adapt the code presented here to another language, such as Visual Basic .NET or Python. The complete source code for the program presented in this article is available at msdn.microsoft.com/magazine/msdnmag0811. This article assumes you have intermediate coding skills with a modern procedural language but does not assume you know anything about PSO or related AI techniques.

Particles

When using PSO, a possible solution to the numeric optimization problem under investigation is represented by the position of a particle. Additionally, each particle has a current velocity, which represents a magnitude and direction toward a new, presumably better, solution/position. A particle also has a measure of the quality of its current position, the particle's best known position (that is, a previous position with the best known quality), and the quality of the best known position. I coded a Particle class as shown in **Figure 2**.

Figure 2 **Particle Definition**

```
public class Particle
{
  public double[] position;
  public double fitness;
  public double[] velocity;

  public double[] bestPosition;
  public double bestFitness;

  public Particle(double[] position, double fitness,
   double[] velocity, double[] bestPosition, double bestFitness)
  {
    this.position = new double[position.Length];
    position.CopyTo(this.position, 0);
    this.fitness = fitness;
    this.velocity = new double[velocity.Length];
    velocity.CopyTo(this.velocity, 0);
    this.bestPosition = new double[bestPosition.Length];
    bestPosition.CopyTo(this.bestPosition, 0);
    this.bestFitness = bestFitness;
  }

  public override string ToString()
  {
    string s = "";
    s += "==========================\n";
    s += "Position: ";
    for (int i = 0; i < this.position.Length; ++i)
      s += this.position[i].ToString("F2") + " ";
    s += "\n";
    s += "Fitness = " + this.fitness.ToString("F4") + "\n";
    s += "Velocity: ";
    for (int i = 0; i < this.velocity.Length; ++i)
      s += this.velocity[i].ToString("F2") + " ";
    s += "\n";
    s += "Best Position: ";
    for (int i = 0; i < this.bestPosition.Length; ++i)
      s += this.bestPosition[i].ToString("F2") + " ";
    s += "\n";
    s += "Best Fitness = " + this.bestFitness.ToString("F4") + "\n";
    s += "==========================\n";
    return s;
  }
} // class Particle
```

The Particle class has five public data members: position, fitness, velocity, bestPosition and bestFitness. When using PSO, for simplicity I prefer using public scope fields, but you may want to use private fields along with get and set properties instead. The field named position is an array of type double and along with get and set properties instead. The field named position is an array of type double and represents a possible solution to the optimization problem under investigation. Although PSO can be used to solve non-numeric problems, it's generally best-suited for solving numeric problems. Field fitness is a measure of how good the solution represented by position is. For minimization problems, which are the most common types of problems solved by PSO, smaller values of the fitness field are better than larger values; for maximization problems, larger values of fitness are better.

Field velocity is an array of type double and represents the information necessary to update a particle's current position/solution. I'll explain particle velocity in detail shortly. The fourth and fifth fields in the Particle type are bestPosition and bestFitness. These fields hold the best position/solution found by the Particle object and the associated fitness of the best position.

The Particle class has a single constructor that accepts five parameters that correspond to each of the Particle's five data fields. The constructor simply copies each parameter value to its corresponding data field. Because all five Particle fields have public scope, I could have omitted the constructor and then just used field assignment statements in the PSO code, but I think the constructor leads to cleaner code.

The Particle class definition contains a ToString method that echoes the values of the five data fields. As with the constructor, because I declared the position, fitness, velocity, bestPosition and bestFitness fields with public scope, I don't really need a ToString method to view a Particle object's values, but including it simplifies viewing the fields and it's useful for WriteLine-style debugging during development. In the ToString method I use string concatenation rather than the more efficient StringBuilder class to make it easier for you to refactor my code to a non-Microsoft .NET Framework-based language if you wish.

The PSO Algorithm

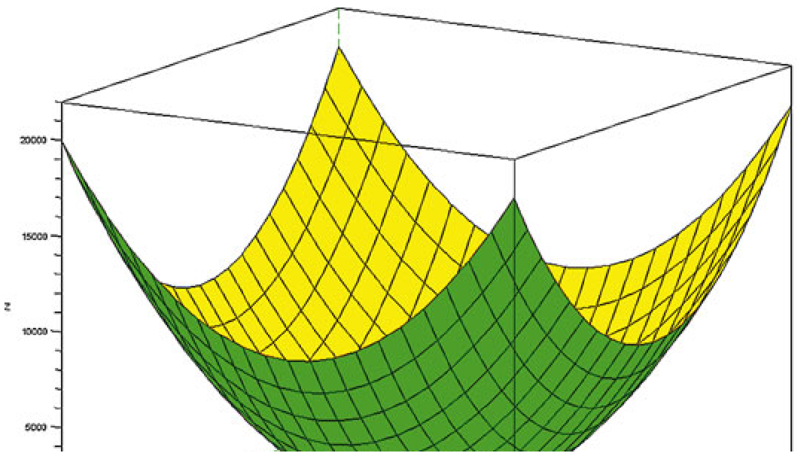
Although the heart of the PSO algorithm is rather simple, you'll need to understand it thoroughly in order to modify the code in this article to meet your own needs. PSO is an iterative process. On each iteration in the PSO main processing loop, each particle's current velocity is first updated based on the particle's current velocity, the particle's local information and global swarm information. Then, each particle's position is updated using the particle's new velocity. In math terms the two update equations are:

$$\boldsymbol{v}(t+1) = (w * \boldsymbol{v}(t)) + (c_1 * r_1 * (\boldsymbol{p}(t) - \boldsymbol{x}(t)) + (c_2 * r_2 * (\boldsymbol{g}(t) - \boldsymbol{x}(t))$$

$$\boldsymbol{x}(t+1) = \boldsymbol{x}(t) + \boldsymbol{v}(t+1)$$

Bear with me here; the position update process is actually much simpler than these equations suggest. The first equation updates a particle's velocity. The term $\boldsymbol{v}(t+1)$ means the velocity at time $t+1$. Notice that $\boldsymbol{v}$ is in bold, indicating that velocity is a vector value and has multiple components such as {1.55, -0.33}, rather than being a single scalar value. The new velocity depends on three terms. The first term is $w * \boldsymbol{v}(t)$. The w factor is called the inertia weight and is simply a constant like 0.73 (more on this shortly); $\boldsymbol{v}(t)$ is the current velocity at time t. The second term is $c_1 * r_1 * (\boldsymbol{p}(t) - \boldsymbol{x}(t))$. The c_1 factor is a constant called the cognitive (or personal or local) weight. The r_1 factor is a random variable in the range [0, 1), which is greater than or equal to 0 and strictly less than 1. The $\boldsymbol{p}(t)$ vector value is the particle's best position found so far. The $\boldsymbol{x}(t)$ vector value is the particle's current position. The third term in the velocity update equation is $(c_2 * r_2 * (\boldsymbol{g}(t) - \boldsymbol{x}(t))$. The c_2 factor is a constant called the social—or global—weight. The r_2 factor is a random variable in the range [0, 1). The $\boldsymbol{g}(t)$ vector value is the best known position found by any particle in the swarm so far. Once the new velocity, $\boldsymbol{v}(t+1)$, has been determined, it's used to compute the new particle position $\boldsymbol{x}(t+1)$.

A concrete example will help make the update process clear. Suppose you're trying to minimize $3 + x_0^2 + x_1^2$ as described in the introductory section of this article. The function is plotted in **Figure 3**. The base of the containing cube in **Figure 3** represents x_0 and x_1 values and the vertical axis represents the function value. Note that the plot surface is minimized with $f = 3$ when $x_0 = 0$ and $x_1 = 0$.



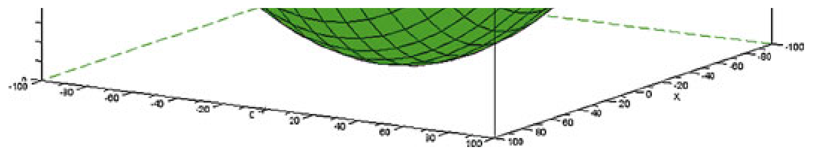


Figure 3 **Plot of *f* = 3 + x_0^2 + x_1^2**

Let’s say that a particle’s current position, ***x***(*t*), is {x_0, x_1} = {3.0, 4.0}, and that the particle’s current velocity, ***v***(*t*), is {-1.0, -1.5}. Let’s also assume that constant *w* = 0.7, constant c_1 = 1.4, constant c_2 = 1.4, and that random numbers r_1 and r_2 are 0.5 and 0.6 respectively. Finally, suppose the particle’s best known position is ***p***(*t*) = {2.5, 3.6} and the global best known position by any particle in the swarm is ***g***(*t*) = {2.3, 3.4}. Then the new velocity and position values are:

v(*t*+1) = (0.7 * {-1.0,-1.5}) +
        (1.4 * 0.5 * {2.5, 3.6} - {3.0, 4.0}) +
        (1.4 * 0.6 * {2.3, 3.4} – {3.0, 4.0})
      = {-0.70, -1.05} + {-0.35, -0.28} + {-0.59, -0.50}
      = {-1.64, -1.83}

x(*t*+1) = {3.0, 4.0} + {-1.64, -1.83}
      = {1.36, 2.17}

Recall that the optimal solution is {x_0, x_1} = (0.0, 0.0}. Observe that the update process has improved the old position/solution from (3.0, 4.0} to {1.36, 2.17}. If you mull over the update process a bit, you’ll see that the new velocity is the old velocity (times a weight) plus a factor that depends on a particle’s best known position, plus another factor that depends on the best known position from all particles in the swarm. Therefore, a particle’s new position tends to move toward a better position based on the particle’s best known position and the best known position of all particles. The graph in **Figure 4** shows the movement of one of the particles during the first eight iterations of the demo PSO run. The particle starts at x_0 = 100.0, x_1 = 80.4 and tends to move toward the optimal solution of x_0 = 0, x_1 = 0. The spiral motion is typical of PSO.

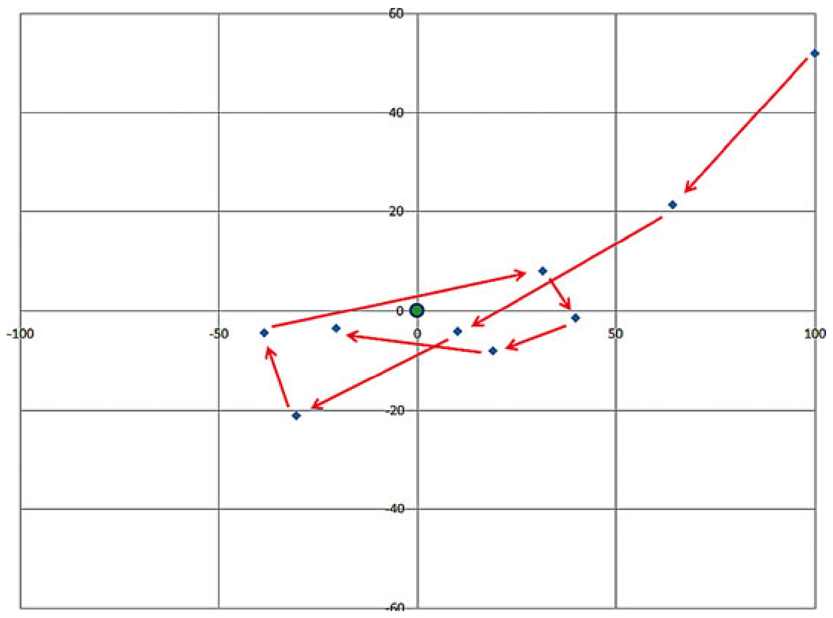


Figure 4 **Particle Motion Toward Optimal Solution**

Implementing the PSO Algorithm

Figure 5 presents the overall structure of the PSO program that produced the program run shown in **Figure 1**. I used Visual Studio to create a C# console application project named ParticleSwarmOptimization. PSO code is fairly basic, so any version of the .NET Framework (1.1 through 4) will work well. I removed all Visual Studio-generated using statements except for the reference to the core System namespace. I declared a class-scope object of type Random to generate the cognitive and social random numbers described in the previous section. I also used the Random object to generate random initial velocities and positions for each Particle object. Inside the Main method I wrap all my code in a single, high-level try statement to catch any exceptions.

Figure 5 **PSO Program Structure**

```
using System;
namespace ParticleSwarmOptimization
{
  class Program
  {
    static Random ran = null;
    static void Main(string[] args)
    {
      try
      {
        Console.WriteLine("\nBegin PSO demo\n");
        ran = new Random(0);

        int numberParticles = 10;
        int numberIterations = 1000;
        int iteration = 0;
        int Dim = 2; // dimensions

        double minX = -100.0;
        double maxX = 100.0;

        Particle[] swarm = new Particle[numberParticles];
        double[] bestGlobalPosition = new double[Dim];
        double bestGlobalFitness = double.MaxValue;

        double minV = -1.0 * maxX;
        double maxV = maxX;

        // Initialize all Particle objects

        double w = 0.729; // inertia weight
        double c1 = 1.49445; // cognitive weight
        double c2 = 1.49445; // social weight
        double r1, r2; // randomizations

        // Main processing loop

        // Display results
        Console.WriteLine("\nEnd PSO demo\n");
      }
      catch (Exception ex)
      {
        Console.WriteLine("Fatal error: " + ex.Message);
      }
    } // Main()

    static double ObjectiveFunction(double[] x)
    {
      return 3.0 + (x[0] * x[0]) + (x[1] * x[1]);
    }

  } // class Program

  public class Particle
  {
    // Definition here
  }

} // ns
```

After instantiating the Random object with an arbitrary seed value of 0, I initialize some key PSO variables:

```
int numberParticles = 10;
int numberIterations = 1000;
int iteration = 0;
int Dim = 2;
double minX = -100.0;
double maxX = 100.0;
```

I use 10 Particle objects. As a rule of thumb, more Particle objects are better than fewer, but more can significantly slow program performance. I set the number of main processing loop iterations to 1,000. The number of iterations you'll want to use will depend on the complexity of the problem you're trying to optimize and the processing power of your host machine. Typically, PSO programs use a value between 1,000 and 100,000. The variable named iteration is a counter to keep track of the number of main loop iterations. The Dim variable holds the number of *x* values in a solution/position. Because my example problem needs to find the values of x_0 and x_1 that minimize $3 + x_0^2 + x_1^2$, I set Dim to 2. As I mentioned earlier, in most PSO situations you'll want to limit the *x* values that make up the position/solution vector to some problem-dependent range. Without some limits, you're effectively searching from double.MinValue to double.MaxValue. Here I arbitrarily limit x_0 and x_1 to [-100.0, +100.0].

Next, I prepare to instantiate the particle swarm:

```
Particle[] swarm = new Particle[numberParticles];
double[] bestGlobalPosition = new double[Dim];
double bestGlobalFitness = double.MaxValue;
double minV = -1.0 * maxX;
double maxV = maxX;
```

I create an array of Particle objects named swarm. I also set up an array to hold the global best known position determined by any Particle—denoted by ***g***(*t*) in the algorithm—and the corresponding fitness of that position array. I set constraints for the maximum and minimum values for a new velocity. The idea here is that because a new velocity determines a particle's new position, I don't want the magnitude of any of the velocity components to be huge.

The code to initialize the swarm is as follows:

```
for (int i = 0; i < swarm.Length; ++i)
{
  double[] randomPosition = new double[Dim];
  for (int j = 0; j < randomPosition.Length; ++j) {
    double lo = minX;
    double hi = maxX;
    randomPosition[j] = (hi - lo) * ran.NextDouble() + lo;
  }
...
```

I iterate through each Particle object in the array named swarm. I declare an array of size Dim to hold a random position for the current Particle. Then for each *x*-value of the position I generate a random value between minX (-100.0) and maxX (+100.0). In many realistic PSO problems, the range for each *x*-value will be different, so you'll have to add code to deal with each *x*-value in the position array separately.

Now I continue the initialization process:

```
double fitness = ObjectiveFunction(randomPosition);
double[] randomVelocity = new double[Dim];
for (int j = 0; j < randomVelocity.Length; ++j) {
  double lo = -1.0 * Math.Abs(maxX - minX);
  double hi = Math.Abs(maxX - minX);
  randomVelocity[j] = (hi - lo) * ran.NextDouble() + lo;
}
swarm[i] = new Particle(randomPosition, fitness, randomVelocity,
  randomPosition, fitness);
...
```

First I compute the quality of the current random position array by passing that array to the method ObjectiveFunction. If you refer back to **Figure 5**, you'll see that the ObjectiveFunction method simply computes the value of the function I'm trying to minimize, namely $3 + x_0^2 + x_1^2$. Next I compute a random velocity for the current Particle object. After I have a random position, the fitness of the random position and a random velocity, I pass those values to the Particle constructor. Recall that the fourth and fifth parameters are the particle's best known position and its associated fitness, so when initializing a Particle the initial random position and fitness are the best known values.

The swarm initialization code finishes with:

```
...
  if (swarm[i].fitness < bestGlobalFitness) {
    bestGlobalFitness = swarm[i].fitness;
    swarm[i].position.CopyTo(bestGlobalPosition, 0);
  }
} // End initialization loop
```

I check to see if the fitness of the current Particle is the best (smallest in the case of a minimization problem) fitness found so far. If so, I update array bestGlobalPosition and the corresponding variable bestGlobalFitness.

Next, I prepare to enter the main PSO processing loop:

```
double w = 0.729; // inertia weight
double c1 = 1.49445; // cognitive weight
double c2 = 1.49445; // social weight
double r1, r2; // randomizers
```

I set the value for *w*, the inertia weight, to 0.729. This value was recommended by a research paper that investigated the effects of various PSO parameter values on a set of benchmark minimization problems. Instead of a single, constant value for *w*, an alternative approach is to vary the value of *w*. For example, if your PSO algorithm is set to iterate 10,000 times, you could initially set *w* to 0.90 and gradually decrease *w* to 0.40 by reducing w by 0.10 after every 2,000 iterations. The idea of a dynamic *w* is that early in the algorithm you want to explore larger changes in position, but later on you want smaller particle movements. I set the values for both c_1 and c_2, the cognitive and social weights, to 1.49445. Again, this movements. I set the values for both c_1 and c_2, the cognitive and social weights, to 1.49445. Again, this value was recommended by a research study. If you set the value of c_1 to be larger than the value of c_2, you place more weight on a particle's best known position than on the swarm's global best known position, and vice versa. The random variables r_1 and r_2 add a random component to the PSO algorithm and help prevent the algorithm from getting stuck at a non-optimal local minimum or maximum solution.

Next, I begin the main PSO processing loop:

```
for (int i = 0; i < swarm.Length; ++i)
{
  Particle currP= swarm[i];

  for (int j = 0; j < currP.velocity.Length; ++j)
  {
    r1 = ran.NextDouble();
    r2 = ran.NextDouble();

    newVelocity[j] = (w * currP.velocity[j]) +
      (c1 * r1* (currP.bestPosition[j] - currP.position[j])) +
      (c2 * r2 * (bestGlobalPosition[j] - currP.position[j]));
    ...
```

I iterate through each Particle object in the swarm array using i as an index variable. I create a reference to the current Particle object named currP to simplify my code, but I could have used swarm[i] directly. As explained in the previous section, the first step is to update each particle's velocity vector. For the current Particle object, I walk through each one of the values in the object's velocity array, generate random variables r_1 and r_2, and then update each velocity component as explained in the previous section.

After I compute a new velocity component for the current Particle object, I check to see if that component is between the minimum and maximum values for a velocity component:

```
if (newVelocity[j] < minV)
    newVelocity[j] = minV;
  else if (newVelocity[j] > maxV)
    newVelocity[j] = maxV;
} // each j
newVelocity.CopyTo(currP.velocity, 0);
...
```

If the component is out of range, I bring it back in range. The idea here is that I don't want extreme values for the velocity component because extreme values could cause my new position to spin out of bounds. After all velocity components have been computed, I update the current Particle object's velocity array using the handy .NET CopyTo method.

Once the velocity of the current Particle has been determined, I can use the new velocity to compute and update the current Particle's position:

```
for (int j = 0; j < currP.position.Length; ++j)
{
  newPosition[j] = currP.position[j] + newVelocity[j];
  if (newPosition[j] < minX)
    newPosition[j] = minX;
  else if (newPosition[j] > maxX)
    newPosition[j] = maxX;
}
newPosition.CopyTo(currP.position, 0);
...
```

Again I perform a range check, this time on each of the current particle's new position components. In a sense, this is a redundant check because I've already constrained the value of each velocity component, but in my opinion the extra check is warranted here.

Now that I have the current Particle object's new position, I compute the new fitness value and update the object's fitness field:

```
newFitness = ObjectiveFunction(newPosition);
    currP.fitness = newFitness;

    if (newFitness < currP.bestFitness) {
      newPosition.CopyTo(currP.bestPosition, 0);
      currP.bestFitness = newFitness;
    }
    if (newFitness < bestGlobalFitness) {
      newPosition.CopyTo(bestGlobalPosition, 0);
      bestGlobalFitness = newFitness;
    }

  } // each Particle
} // main PSO loop
```

```
...
```

After updating the current particle, I check to see if the new position is the best known position of the particle; I also check to see if the new position is a best global swarm position. Notice that logically, there can be a new global best position only if there's a best local position, so I could have nested the global best check inside the check for a local best position.

At this point my main PSO algorithm loop is finished and I can display my results:

```
Console.WriteLine("\nProcessing complete");
  Console.Write("Final best fitness = ");
  Console.WriteLine(bestGlobalFitness.ToString("F4"));
  Console.WriteLine("Best position/solution:");
  for (int i = 0; i < bestGlobalPosition.Length; ++i){
    Console.Write("x" + i + " = " );
    Console.WriteLine(bestGlobalPosition[i].ToString("F4") + " ");
  }
  Console.WriteLine("");
  Console.WriteLine("\nEnd PSO demonstration\n");
 }
 catch (Exception ex)
 {
  Console.WriteLine("Fatal error: " + ex.Message);
 }
} // Main()
```

Extending and Modifying

Now that you've seen how to write a basic PSO, let's discuss how you can extend and modify the code I've presented. The example problem I solved is artificial in the sense that there's no need to use PSO to find an approximate solution because the problem can be solved exactly. Where PSO is really useful is when the numeric problem under investigation is extremely difficult or impossible to solve using standard techniques. Consider the following problem. You want to predict the score of an (American) football game between teams A and B. You have historical data consisting of the previous results of A and B against other teams. You mathematically model the historical rating of a team X in such a way that if the team wins a game, the team's rating goes up by some fixed value (say 16 points) plus another value that depends on the difference between the teams' ratings (say 0.04 times the difference if the team X rating is less than the opposing team's). Furthermore, you model the predicted margin of victory of a team as some function of the difference in team ratings; for example, if team X is rated 1,720 and team Y is rated 1,620, your model predicts a margin of victory for X of 3.5 points. In short, you have a large amount of data and need to determine several numeric values (such as the 16 and the 0.04) that minimize your prediction errors. This data-driven parameter estimation is the type of problem that's right up PSO's alley.

PSO is just one of several AI techniques based on the behavior of natural systems. Perhaps the technique closest to PSO algorithms is Genetic Algorithms (GAs). Both techniques are well-suited to difficult numeric problems. GAs have been extensively studied for decades. An advantage of PSOs over GAs is that PSO algorithms are significantly simpler to implement than GAs. It's not clear at this time whether PSOs are more or less effective than GAs, or roughly equal to them.

The version of PSO I've presented here can be modified in many ways. One particularly interesting modification is to use several sub-swarms of particles rather than one global swarm. With such a design, each particle belongs to a sub-swarm and the new velocity of a particle could depend on four terms rather than three: the old velocity, the particle's best known position, the best known position of any particle in the sub-swarm, and the best known position of any particle. The idea of this sub-swarm design is to reduce the chances of the PSO algorithm getting stuck in a non-optimal solution. To the best of my knowledge such a design has not yet been thoroughly investigated.

Dr. James McCaffrey *works for Volt Information Sciences Inc., where he manages technical training for software engineers working at the Microsoft Redmond, Wash., campus. He's worked on several Microsoft products, including Internet Explorer and MSN Search. Dr. McCaffrey is the author of ".NET Test Automation Recipes" (Apress, 2006), and can be reached at jammc@microsoft.com.*

Thanks to the following Microsoft technical experts for reviewing this article: ***Paul Koch, Dan Liebling, Anne Loomis Thompson*** *and* ***Shane Williams***

Rate:

Share this content

Comments (5)

Leave a Comment

Mohamudsomali: Wednesday, January 15, 2014 12:37 PM

This article is very very helpful for students, thank you very much indeed.

Ruwan_T W: Tuesday, September 24, 2013 11:13 AM

This is an excellent article to understand how does PSO work , even if you don't have any prior knowledge. Basic, but very clear and straightforward implementation.
Thanks a lot.

Master Pheonix: Wednesday, August 29, 2012 9:47 PM

best possible position of 0,0 being the point say where food is or something like that? what about that as the worst possible position?

Master Pheonix: Wednesday, August 29, 2012 9:44 PM

very nice

Dupati: Saturday, August 11, 2012 3:51 AM

The Article presentation is very impressive for beginners. I am grateful to the author.

Sign in to Leave a Comment Report Abuse

Switch to Amyuni for quality PDF integration into your applications!
Easy Integration and Licensing. Certified for Windows Servers.
.NET - COM - WinRT - ASP.NET - Windows Phone ...
AMYUNI