

## Test Run

## Bacterial Foraging Optimization

James McCaffrey

[Download the Code Sample](#)

Bacterial Foraging Optimization (BFO) is a fascinating artificial intelligence (AI) technique that can be used to find approximate solutions to extremely difficult or impossible numeric maximization or minimization problems. The version of BFO I describe in this article is based on the 2002 paper, "Biomimicry of Bacterial Foraging for Distributed Optimization and Control," by Dr. Kevin Passino. (This paper can be found via Internet search, but subscription is required.) BFO is a probabilistic technique that models the food-seeking and reproductive behavior of common bacteria such as *E. coli* in order to solve numeric optimization problems where there's no effective deterministic approach. The best way for you to get a feel for what BFO is and to see where I'm headed in this article is to examine **Figure 1** and **Figure 2**.

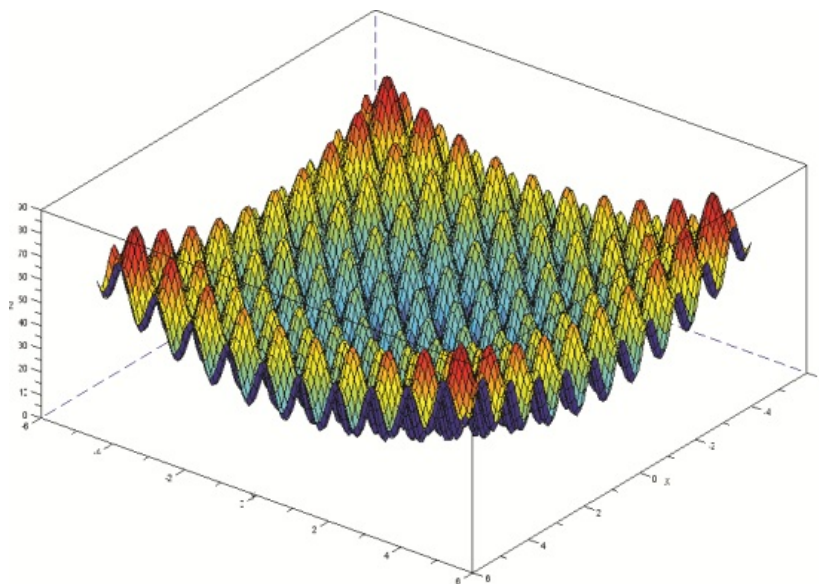


Figure 1 The Rastrigin Function Problem

```

file:///C:/BacterialForagingOptimization/bin/Debug/BacterialForagingOptimization.EXE
Begin Bacterial Foraging Optimization demo
Target function to minimize: f(x,y) = (x^2 - 10*cos(2*PI*x)) + (y^2 - 10*cos(2*PI*y)) + 20
Target function has known optimum value = 0.0 at x = 0.0 and y = 0.0
Colony size = 100
Chemotactic step count = 20
Reproduction step count = 8
Dispersal step count = 4
Maximum swim step count = 5
Death probability = 0.25
Base swim length = 0.05
Initializing bacteria colony to random positions
Computing the cost value for each bacterium
Best initial cost = 12.9343
Entering main BFO tumble-swim-reproduce-disperse algorithm loop
New best solution found by bacteria 9 at time = 0
New best solution found by bacteria 9 at time = 0
New best solution found by bacteria 89 at time = 0
New best solution found by bacteria 64 at time = 2
New best solution found by bacteria 64 at time = 2
New best solution found by bacteria 64 at time = 3
New best solution found by bacteria 6 at time = 15
New best solution found by bacteria 3 at time = 20
New best solution found by bacteria 1 at time = 35
New best solution found by bacteria 2 at time = 53
New best solution found by bacteria 1 at time = 159
All BFO processing complete
Best cost (minimum function value) found = 0.0002
Best position/solution = [ -0.0010 0.0005 ]
End BFO demo

```

One SDK, endless possibilities.



DOCUMENT | MEDICAL | MULTIMEDIA | IMAGING

Total Data Quality  
From A to SQL

Free Trial

MELISSA DATA

## MSDN Magazine Blog

[Charles Petzold Departing](#)

As I describe in this month's Editor's Note column, the October issue of MSDN Magazine is Charles Petzold's last as a regular columnist. He is leav... [More...](#)

Wednesday, Oct 1

[MSDN Magazine July Issue Preview: All About Azure Web Sites](#)

The July issue of MSDN Magazine is themed around the updated features and functionality of Microsoft Azure Web Sites, and reflects the rapid-fire p... [More...](#)

Monday, Jun 30

[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

## Figure 2 Using Bacterial Foraging Optimization

**Figure 1** is a graph of the Rastrigin function, which is often used as a standard benchmark problem to test the effectiveness of optimization algorithms. The goal is to find the values of  $x$  and  $y$  that minimize the function. You can see that there are many valleys that are local minima solutions. However, there's only one global solution at  $x = 0.0$  and  $y = 0.0$  where the value of the function is 0.0.

**Figure 2** is a screenshot of BFO attempting to solve the Rastrigin function. The program sets up several parameters, including the number of simulated bacteria (100 in this example). Each bacterium has a position that represents a possible solution. Initially, all bacteria are set to random positions. Each position has an associated cost that represents the value of the Rastrigin function. As the main processing loop executes, different bacteria find successively better solutions. At the end of processing, the best solution found is 0.0002 when  $x = -0.0010$  and  $y = 0.0005$ —extremely close to but not quite the optimal solution.

In the rest of this article I'll explain in detail the BFO algorithm and walk you through the program shown running in **Figure 2**. 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 Windows PowerShell. The complete source code for the program is available at [msdn.microsoft.com/magazine/msdnmag0412](https://msdn.microsoft.com/magazine/msdnmag0412). This article assumes you have intermediate or advanced coding skills with a modern procedural language but doesn't assume you know anything about BFO or related AI techniques.

## Real Bacteria Behavior

Bacteria such as *E. coli* are among the most successful organisms on the planet. Bacteria have semi-rigid appendages called flagella. When all flagella rotate in a counterclockwise direction, a propeller effect is created and a bacterium will swim in a more-or-less straight direction.

Bacteria tend to swim when they're improving in some sense, such as when finding an increasing nutrient gradient, for example. When all flagella rotate in a clockwise direction, a bacterium will tumble quickly and point in a new direction. Bacteria tend to tumble when they encounter a noxious substance or when they're in a gradient that's not improving. Bacteria reproduce about every 20 minutes or so by asexually dividing into two identical daughters. Healthier bacteria tend to reproduce more than less-healthy bacteria.

## Overall Program Structure

The overall program structure for the BFO demo is listed in **Figure 3**.

**Figure 3 Overall BFO Program Structure**

```
using System;
namespace BacterialForagingOptimization
{
    class BacterialForagingOptimizationProgram
    {
        static Random random = null;
        static void Main(string[] args)
        {
            try
            {
                int dim = 2;
                double minValue = -5.12;
                double maxValue = 5.12;
                int S = 100;
                int Nc = 20;
                int Ns = 5;
                int Nre = 8;
                int Ned = 4;
                double Ped = 0.25;
                double Ci = 0.05;
                random = new Random(0);
                // Initialize bacteria colony
                // Find best initial cost and position
                int t = 0;
                for (int l = 0; l < Ned; ++l) // Eliminate-disperse loop
                {
                    for (int k = 0; k < Nre; ++k) // Reproduce-eliminate loop
                    {
                        for (int j = 0; j < Nc; ++j) // Chemotactic loop
                        {
                            // Reset the health of each bacterium to 0.0
                            for (int i = 0; i < S; ++i)
                            {
                                // Process each bacterium
                            }
                            ++t;
                        }
                        // Reproduce healthiest bacteria, eliminate other half
                    }
                    // Eliminate-disperse
                }
                Console.WriteLine("\nBest cost found = " + bestCost.ToString("F4"));
                Console.WriteLine("Best position/solution = ");
                ShowVector(bestPosition);
            }
            catch (Exception ex)
            {
                Console.WriteLine("Fatal: " + ex.Message);
            }
        }
    }
} // Main
static double Cost(double[] position) { ... }
```

```

        static double Cost(double[] position) { ... }
    }
    public class Colony // Collection of Bacterium
    {
        // ...
        public class Bacterium : IComparable<Bacterium>
        {
            // ...
        }
    }
} // ns

```

I used Visual Studio to create a C# console application named BacterialForagingOptimization. I renamed the file Program.cs to BacterialForagingOptimizationProgram.cs and deleted all template-generated using statements except for the reference to the System namespace.

I declared a class-scope Random object named random; BFO is a probabilistic algorithm, as you'll see shortly. Inside the Main method I declared several key variables. Variable dim is the number of dimensions in the problem. Because the goal in this example is to find x and y for the Rastrigin function, I set dim to 2. The minValue and maxValue variables establish arbitrary limits for both x and y. Variable S is the number of bacteria. I used slightly nondescriptive variable names such as S and Nc because these are the names used in the research article, so you can more easily use that article as a reference.

Variable Nc is the number of so-called chemotactic steps. You can think of this as a counter that represents the lifespan of each bacterium. Variable Ns is the maximum number of times a bacterium will swim in the same direction. Variable Nre is the number of reproduction steps. You can think of this as the number of generations of bacteria. Variable Ned is the number of dispersal steps. Every now and then the BFO algorithm randomly disperses some bacteria to new positions, modeling the effects of the external environment on real bacteria. Variable Ped is the probability of a particular bacterium being dispersed. Variable Ci is the basic swim length for each bacterium. When swimming, bacteria will move no more than Ci in any single step. Variable t is a time counter to track BFO progress. Because BFO is relatively new, very little is known about the effects of using different values for BFO parameters.

The main BFO algorithm processing consists of several nested loops. Unlike most AI algorithms such as genetic algorithms that are controlled by a single time counter, BFO is controlled by multiple loop counters.

The program uses a static Cost function. This is the function that you're trying to minimize or maximize. In this example the Cost function is just the Rastrigin function. The input is an array of double that represents a bacterium's position, which in turn represents a possible solution. In this example the Cost function is defined as:

```

double result = 0.0;
for (int i = 0; i < position.Length; ++i) {
    double xi = position[i];
    result += (xi * xi) - (10 * Math.Cos(2 * Math.PI * xi)) + 10;
}
return result;

```

You can find more information about the Rastrigin function by doing an Internet search, but the point is that the Cost function accepts a bacterium's position and returns the value you're trying to minimize or maximize.

## The Bacterium and Colony Classes

The BFO program defines a Colony class, which represents a collection of bacteria, with a nested Bacterium class that defines an individual bacterium. The nested Bacterium class is listed in **Figure 4**.

**Figure 4 The Bacterium Class**

```

public class Bacterium : IComparable<Bacterium>
{
    public double[] position;
    public double cost;
    public double prevCost;
    public double health;
    static Random random = new Random(0);
    public Bacterium(int dim, double minValue, double maxValue)
    {
        this.position = new double[dim];
        for (int p = 0; p < dim; ++p) {
            double x = (maxValue - minValue) * random.NextDouble() + minValue;
            this.position[p] = x;
        }
        this.health = 0.0;
    }
    public override string ToString()
    {
        // See code download
    }
    public int CompareTo(Bacterium other)
    {
        if (this.health < other.health)
            return -1;
        else if (this.health > other.health)
            return 1;
        else
            return 0;
    }
}

```

Class `Bacterium` derives from the `Comparable` interface so that two `Bacterium` objects can be sorted by their health when determining which bacteria will survive to the next generation.

The position field represents a solution. The cost field is the cost associated with the position. Field `prevCost` is the cost associated with a bacterium's previous position; this allows a bacterium to know if it's improving or not, and therefore whether it should swim or tumble. The health field is the sum of the accumulated costs of a bacterium during the bacterium's life span. Because the goal is to minimize cost, small values of health are better than large values.

The `Bacterium` constructor initializes a `Bacterium` object to a random position. The cost field isn't explicitly set by the constructor. The `CompareTo` method orders `Bacterium` objects from smallest health to largest health.

**Figure 5** shows the simple `Colony` class.

**Figure 5 The Colony Class**

```
public class Colony
{
    public Bacterium[] bacteria;
    public Colony(int S, int dim, double minValue, double maxValue)
    {
        this.bacteria = new Bacterium[S];
        for (int i = 0; i < S; ++i)
            bacteria[i] = new Bacterium(dim, minValue, maxValue);
    }
    public override string ToString() { // See code download }
    public class Bacterium : Comparable<Bacterium>
    {
        // ...
    }
}
```

The `Colony` class is essentially a collection of `Bacterium` objects. The `Colony` constructor creates a collection of `Bacterium` where each `Bacterium` is assigned a random position by calling the `Bacterium` constructor.

## The Algorithm

After setting up the BFO variables such as `S` and `Nc`, the BFO algorithm initializes the bacteria colony like so:

```
Console.WriteLine("\nInitializing bacteria colony");
Colony colony = new Colony(S, dim, minValue, maxValue);
for (int i = 0; i < S; ++i) {
    double cost = Cost(colony.bacteria[i].position);
    colony.bacteria[i].cost = cost;
    colony.bacteria[i].prevCost = cost;
}
...
```

Because the `Cost` function is external to the colony, the cost for each `Bacterium` object is set outside the `Colony` constructor using the `Cost` function.

After initialization, the best bacterium in the colony is determined, keeping in mind that lower costs are better than higher costs in the case of minimizing the Rastrigin function:

```
double bestCost = colony.bacteria[0].cost;
int indexOfBest = 0;
for (int i = 0; i < S; ++i) {
    if (colony.bacteria[i].cost < bestCost) {
        bestCost = colony.bacteria[i].cost;
        indexOfBest = i;
    }
}
double[] bestPosition = new double[dim];
colony.bacteria[indexOfBest].position.CopyTo(bestPosition, 0);
Console.WriteLine("\nBest initial cost = " + bestCost.ToString("F4"));
...
```

Next, the multiple loops of the main BFO processing are set up:

```
Console.WriteLine("\nEntering main BFO algorithm loop\n");
int t = 0;
for (int l = 0; l < Ned; ++l)
{
    for (int k = 0; k < Nre; ++k)
    {
        for (int j = 0; j < Nc; ++j)
        {
            for (int i = 0; i < S; ++i) { colony.bacteria[i].health = 0.0; }
            for (int i = 0; i < S; ++i) // Each bacterium
            {
                ...
            }
        }
    }
}
```

The outermost loop with index `l` handles the dispersal steps. The next loop with index `k` handles the reproduction steps. The third loop with index `j` handles the chemotactic steps that represent the lifespan of each bacterium. Inside the chemotactic loop, a new generation of bacteria has just been generated, so the health of each bacterium is reset to 0. After resetting bacteria health values inside the chemotactic loop, each bacterium tumbles to determine a new direction and then moves in the new direction, like so:

```

double[] tumble = new double[dim];
for (int p = 0; p < dim; ++p) {
    tumble[p] = 2.0 * random.NextDouble() - 1.0;
}
double rootProduct = 0.0;
for (int p = 0; p < dim; ++p) {
    rootProduct += (tumble[p] * tumble[p]);
}
for (int p = 0; p < dim; ++p) {
    colony.bacteria[i].position[p] += (Ci * tumble[p]) / rootProduct;
}
...

```

First, for each component of the current bacterium's position, a random value between -1.0 and +1.0 is generated. Then the root product of the resulting vector is computed. And then the new position of the bacterium is calculated by taking the old position and moving some fraction of the value of the  $C_i$  variable.

After tumbling, the current bacterium is updated and then the bacterium is checked to see if it found a new global best solution:

```

colony.bacteria[i].prevCost = colony.bacteria[i].cost;
colony.bacteria[i].cost = Cost(colony.bacteria[i].position);
colony.bacteria[i].health += colony.bacteria[i].cost;
if (colony.bacteria[i].cost < bestCost) {
    Console.WriteLine("New best solution found by bacteria " + i.ToString()
        + " at time = " + t);
    bestCost = colony.bacteria[i].cost;
    colony.bacteria[i].position.CopyTo(bestPosition, 0);
}
...

```

Next, the bacterium enters a swim loop where it will swim in the same direction as long as it's improving by finding a better position:

```

int m = 0;
while (m < Ns && colony.bacteria[i].cost < colony.bacteria[i].prevCost) {
    ++m;
    for (int p = 0; p < dim; ++p) {
        colony.bacteria[i].position[p] += (Ci * tumble[p]) / rootProduct;
    }
    colony.bacteria[i].prevCost = colony.bacteria[i].cost;
    colony.bacteria[i].cost = Cost(colony.bacteria[i].position);
    if (colony.bacteria[i].cost < bestCost) {
        Console.WriteLine("New best solution found by bacteria " +
            i.ToString() + " at time = " + t);
        bestCost = colony.bacteria[i].cost;
        colony.bacteria[i].position.CopyTo(bestPosition, 0);
    }
} // while improving
} // i, each bacterium
++t; // increment the time counter
} // j, chemotactic loop
...

```

Variable  $m$  is a swim counter to limit the maximum number of consecutive swims in the same direction to the value in variable  $N_s$ . After swimming, the time counter is incremented and the chemotactic loop terminates.

At this point, all bacteria have lived their lifespan given by  $N_c$  and the healthiest half of the colony will live and the least-healthy half will die:

```

Array.Sort(colony.bacteria);
for (int left = 0; left < S / 2; ++left) {
    int right = left + S / 2;
    colony.bacteria[left].position.CopyTo(colony.bacteria[right].position, 0);
    colony.bacteria[right].cost = colony.bacteria[left].cost;
    colony.bacteria[right].prevCost = colony.bacteria[left].prevCost;
    colony.bacteria[right].health = colony.bacteria[left].health;
}
} // k, reproduction loop
...

```

Because a `Bacterium` object derives from `IComparable`, the `Array.Sort` method will automatically sort from smallest health (smaller is better) to largest health so the best bacteria are in the left  $S/2$  cells of the colony array. The weaker half of bacteria in the right cells of the colony are effectively killed by copying over the data of the better half of the bacteria array into the weaker right half. Notice this implies that the total number of bacteria,  $S$ , should be a number evenly divisible by 2.

At this point the chemotactic and reproduction loops have finished, so the BFO algorithm enters the dispersal phase:

```

for (int i = 0; i < S; ++i) {
    double prob = random.NextDouble();
    if (prob < Ped) {
        for (int p = 0; p < dim; ++p) {
            double x = (maxValue - minValue) *
                random.NextDouble() + minValue;
            colony.bacteria[i].position[p] = x;
        }
        double cost = Cost(colony.bacteria[i].position);
    }
}

```

```

colony.bacteria[i].cost = cost;
colony.bacteria[i].prevCost = cost;
colony.bacteria[i].health = 0.0;
if (colony.bacteria[i].cost < bestCost) {
    Console.WriteLine("New best solution found by bacteria " +
        i.ToString() + " at time = " + t);
    bestCost = colony.bacteria[i].cost;
    colony.bacteria[i].position.CopyTo(bestPosition, 0);
}
} // if (prob < Ped)
} // for
} // l, elimination-dispersal loop
...

```

Each bacterium is examined. A random value is generated and compared against variable Ped to determine if the current bacterium will be moved to a random location. If a bacterium is in fact dispersed, it's checked to see if it found a new global best position by pure chance.

At this point all loops have been executed and the BFO algorithm displays the best solution found using a program-defined helper method named ShowVector:

```

Console.WriteLine("\n\nAll BFO processing complete");
Console.WriteLine("\nBest cost (minimum function value) found = " +
    bestCost.ToString("F4"));
Console.WriteLine("Best position/solution = ");
ShowVector(bestPosition);
Console.WriteLine("\nEnd BFO demo\n");
}
catch (Exception ex)
{
    Console.WriteLine("Fatal: " + ex.Message);
}
} // Main
...

```

## What's the Point?

BFO is a relatively new approach to finding approximate solutions to numerical optimization problems that can't be handled using traditional mathematical techniques. In my opinion, BFO is an alternative to genetic algorithms and particle swarm optimization. There's little research evidence to answer the question of just how effective BFO is or isn't.

How might BFO be used? There are many possibilities related to software testing and also to optimization in general. For example, imagine you're trying to predict something very difficult, such as short-term changes in the price of some stock on the New York Stock Exchange. You gather some historical data and come up with some complex math equation that relates stock price to your input data, but you need to determine the parameters of your equation. You could potentially use BFO to estimate the values of your parameters where the cost function is a percentage of incorrect predictions made by your equation.

BFO is a meta-heuristic, meaning it's just a conceptual framework that can be used to design a specific algorithm. The version of BFO I've presented here is just one of many possibilities and should be considered a starting point for experimentation rather than the final word on the topic. In fact, in order to keep the size of this article manageable, I removed a bacteria swarming feature that's presented in the original BFO research article.

---

**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](mailto:jammc@microsoft.com).

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

Rate: ★★★★★

Share this content     

## Comments (4)



[Leave a Comment](#)



**yaofengdb2:** Sunday, May 25, 2014 12:21 AM

when update the position,I think the position should not exceed boundary set by maxValue and minValue,



**Yazan ahmad:** Wednesday, April 30, 2014 4:59 AM

link for C# demo is not working please fix it



**Yazan ahmad:** Wednesday, April 30, 2014 4:59 AM

link for C# demo is not working please fix it



**Melisa Sakura:** Thursday, April 24, 2014 10:38 AM

hello mr. james, may i know how do you determine the parameter initialization for bfoa?  
 int S = 100;  
 int Nc = 20;




```
int Ns = 5;  
int Nre = 8;  
int Ned = 4;  
double Ped = 0.25;  
double Ci = 0.05;  
how do you get those numbers? thanks :)
```

[Sign in to Leave a Comment](#)

[Report Abuse](#)

msdn<sup>®</sup>  
magazine

 Visual Studio | Microsoft Azure

Developers! Reach any user on any device  
with the new Visual Studio + Azure.

Connect();

Join us Nov. 12–13 for a look at the future NOW.

Attend the Virtual Event →

