2

Artificial Bee Colony Algorithm

Bahriye Akay

Department of Computer Engineering
Erciyes University, Melikgazi, Kayseri, Turkey

Dervis Karaboga

Department of Computer Engineering
Erciyes University, Melikgazi, Kayseri, Turkey

CONTENTS

2.1 Introduction

Swarm intelligence models the collective behaviour of social creatures. The collective intelligence arises from task division and self-organization which is adaptation to new conditions without a global supervision. Self-organization is achieved by repeating the rewarding actions (positive feedback), abandoning repetitive behaviour patterns (negative feedback), communicating to neighbouring agents (multiple interactions) and exploring undiscovered patterns to avoid stagnation states (fluctuation).

Honey bees are social creatures that exhibit swarm intelligence in some of their activities such as nest site selection, mating, foraging etc. As well as the other activities, there is a task division among the bees in the foraging performed to find profitable sources to maximize the nectar amount transferred to the hive. To achieve this crucial task efficiently, the bees are divided into three categories: employed bees, onlooker bees and scouts. The employed bees are responsible for bringing the nectar from discovered flowers to the hive and they dance to give profit and location information about the source to waiting bees in the hive which are called onlookers. The onlooker bees watch the dances and fly to potentially good flowers chosen according to the information gathered from dances. A potentially good solution has high probability to be chosen by an onlooker bee. This positive feedback in foraging and dancing creates interaction between the bees. When a flower is selected by a bee, its nectar decreases by each exploitation and finally exhausts. The exhausted source is abandoned (negative feedback) and a search to find an undiscovered source (fluctuation) is performed by a scout bee. The positive and negative feedback, interaction and fluctuation properties show that a bee swarm is self-organizing and adaptable to internal and environmental conditions without a higher level guidance.

In 2005, Karaboga was inspired by the foraging behaviour of honey bees and proposed the Artificial Bee Colony (ABC) algorithm which simulates the task division and self-organization in a bee colony [1]. The algorithm has three phases: employed bees phase, onlooker bees phase and scout bee phase. The food sources (solutions) in an environment (search space) are searched by the bees to maximize the nectar amount (fitness of the solutions). As in real bees, the employed bees phase searches the vicinity of the sources discovered so far while the onlooker bees phase recruits the waiting bees to quality sources based on the information gathered from the employed bees. The scout bee phase tries to find new undiscovered flowers. Performance of the algorithm has been investigated in single-objective unconstrained [2, 3], constrained optimization [4], multi-objective optimization [5] and the algorithm have been used in many research areas successfully [6, 7]. According to the studies, the ABC algorithm is a simple and powerful meta-heuristic that can be used efficiently especially on high-dimensional and multi-modal problems. On hybrid and composite problems, the algorithm was modified to improve its local search capability and convergence rate. On constrained problems, its selection strategy was modified based on Deb's rules and on multi-objective problems, its greedy selection strategy was replaced with non-dominated sorting to rank solutions and build a population of dominated and non-dominated solutions. It is a good alternative in swarm intelligence algorithms with the advantage of having a smaller number of control parameters and balanced exploration/exploitation capability.

This chapter is organized as follows: in Section 2.2 the pseudo-code for the ABC algorithm is provided and each phase of the algorithm is explained in detail. Control parameters of the algorithm are introduced and their effects are discussed. In Sections 2.3 and 2.4, the ABC algorithm implementations using Matlab and C++ programming languages are presented, respectively. In Section 2.5 a step-by-step procedure is given to show the behaviour of the algorithm and finally it is concluded in Section 2.6.

2.2 The original ABC algorithm

The Artificial Bee Colony [1] algorithm is an optimization tool that exhibits the swarm intelligence of honey bees in the foraging task. In the algorithm, each solution represents a food source position and the algorithm tries to find the source with the maximum nectar amount. The pseudo-code of the ABC algorithm is given in Alg. 1.

Algorithm 1 Main steps of ABC algorithm.

```
1: Set values for the control parameters
2: $SN$: Number of Food Sources,
3: $MCN$: Termination Criteria, Maximum Number of Cycles,
4: $limit$: Maximum number of exploitations for a solution
                                                         ▷ //Initialization
5: for $i = 1$ to $SN$ do
6:     $x(i) \longleftarrow$ a random food source location by Eq. 2.1
7:     $trial(i) = 0$
8: end for
9: $cyc = 1$
10: while $cyc < MCN$ do                                 ▷ //Employed Bees' Phase
11:     for $i = 1$ to $SN$ do
12:         $\hat{x} \longleftarrow$ a neighbour food source location generated by Eq. 2.2
13:         if $f(\hat{x}) < f_i$ then
14:             $x(i) = \hat{x}$
15:             $trial(i) = 0$
16:         else
17:             $trial(i) = trial(i) + 1$
18:         end if
19:     end for
20:     $p \longleftarrow$ assign probability by Eq. 2.3
                                                         ▷ //Onlooker Bees' Phase
21:     $i = 0$
22:     $t = 0$
23:     while $t < SN$ do
24:         if $rand(0,1) < p(i)$ then
25:             $t = t + 1$
26:             $\hat{x} \longleftarrow$ a neighbour food source location generated by Eq. 2.2
27:             if $f(\hat{x}) < f_i$ then
28:                 $x(i) = \hat{x}$
29:                 $trial(i) = 0$
30:             else
31:                 $trial(i) = trial(i) + 1$
32:             end if
33:         end if
34:         $i = (i + 1) \mod (SN - 1)$
35:     end while
36:     Memorize the best solution                       ▷ //Scout bee phase
37:     $si = \{i : trial(i) = max(trial)\}$
38:     if $trial(si) > limit$ then
39:         $x(si) \longleftarrow$ a random food source location by Eq. 2.1
40:         $trial(si) = 0$
41:     end if
42:     $cyc{+}{+}$
43: end while
```

In the initialization phase of ABC, SN number of solutions (food sources) are generated by Eq. 2.1.

$$x_{ij} = x_j^{lb} + rand(0,1)(x_j^{ub} - x_j^{lb}) \tag{2.1}$$

where $i \in \{1, \ldots, SN\}$, $j \in \{1, \ldots, D\}$, D is the problem dimension, x_j^{lb} and x_j^{ub} are the lower and upper bounds in jth dimension of design parameters, respectively.

The food source population is evolved by the employed bees, onlooker bees and scout bee phases based on swarm intelligence characteristics. In the employed bees phase, the nectar exploitation behaviour is simulated by a local search around each source i by Eq. 2.2.

$$\hat{x}_{ij} = x_{ij} + \phi_{ij}(x_{ij} - x_{kj}) \tag{2.2}$$

where k is a randomly chosen neighbour of ith solution and $k \in \{1, 2, \ldots CS\}$: $k \neq i$ and ϕ_{ij} is a real random number drawn from uniform distribution within range [-1,1]. j is a random dimension which is going to be perturbed.

Choosing profitable sources based on the information gained from the employed bees is simulated by a selection strategy in which each solution is assigned a probability proportional to its quality. The quality is measured by the solution fitness (Eq. 2.4) and the probability can be defined by (Eq. 2.3):

$$p_i = 0.1 + 0.9 * \frac{fitness_i}{max(\vec{fitness})} \tag{2.3}$$

where $fitness_i$ (Eq. 2.4) is inversely proportional to the cost function for minimization purposes. When an onlooker bee selects a food source, the bee conducts a local search around the source by Eq. 2.2.

$$fitness_i = \begin{cases} \frac{1}{1+f_i}, if\ f_i \geq 0 \\ 1 + |f_i|, if\ f_i < 0 \end{cases} \tag{2.4}$$

In both the employed bees and onlooker bees phases, if the current source cannot be improved by a new source produced by the local search, its counter corresponding the number of exploitations is incremented by 1. If the counter exceeds a predefined number (control parameter, limit), the solution is discarded from the population by simulating food source exhaustion. As in real scout bees, the bee abandons the exhausted source and searches for a new source by Eq. 2.1.

2.3 Source-code of ABC algorithm in Matlab

```
%/* Control Parameters of ABC algorithm*/
FoodNumber=20; %/*The number of food sources*/
limit=100; %/*used to decide whether a food source will be abandoned*/
maxCycle=1000; %/*The maximum number of cycles */
%/* Problem specific variables*/
```

```
objfun='rastrigin'; %cost function to be optimized
D=20; %/*The number of parameters of the problem to be optimized*/
ub=ones(1,D)*5.12; %/*lower bounds of the parameters. */
lb=ones(1,D)*(-5.12);%/*upper bound of the parameters.*/
%Foods [FoodNumber][D]; /*population of food sources.*/
%ObjVal[FoodNumber];  /*objective function values associated with food
      sources */
%Fitness[FoodNumber]; /*fitness vector*/
%trial[FoodNumber]; /*the number of exploitations of each source*/
%prob[FoodNumber]; /* probability of each source */
%solution [D]; /*New solution (neighbour)*/
%ObjValSol; /*Objective function value of new solution*/
%FitnessSol; /*Fitness value of new solution*/
%GlobalParams[D]; /*the optimum solution vector*/
%GlobalMin; /*the optimum function value*/
%%Initialization
Range = repmat((ub-lb),[FoodNumber 1]);
Lower = repmat(lb, [FoodNumber 1]);
Foods = rand(FoodNumber,D) .* Range + Lower;
ObjVal=feval(objfun,Foods);
Fitness=calculateFitness(ObjVal);
trial=zeros(1,FoodNumber); %reset trial counters
[GlobalMin,GlobalParams]=MemorizeBestSolution(Foods,ObjVal,ObjVal(1),
     Foods(1,:));
cycle=1;
while ((cycle <= maxCycle))
%%%% EMPLOYED BEE PHASE %
     for i=1:(FoodNumber)
         [sol,ObjValSol,FitnessSol]=GenerateNewSolution(Foods,i,objfun,
     lb,ub);
         % /*greedy selection*/
         if (FitnessSol>Fitness(i))
                Foods(i,:)=sol;
                Fitness(i)=FitnessSol;
                ObjVal(i)=ObjValSol;
                trial(i)=0;
          else
                trial(i)=trial(i)+1;
         end
     end
%%%%CalculateProbabilities %
prob=(0.9.*Fitness./max(Fitness))+0.1;
%%%% ONLOOKER BEE PHASE %
i=1;
t=0;
while(t<FoodNumber)
     if(rand<prob(i))
          t=t+1;
         [sol,ObjValSol,FitnessSol]=GenerateNewSolution(Foods,i,objfun,
     lb,ub);
         % /*greedy selection*/
         if (FitnessSol>Fitness(i))
                Foods(i,:)=sol;
                Fitness(i)=FitnessSol;
                ObjVal(i)=ObjValSol;
                trial(i)=0;
          else
                trial(i)=trial(i)+1;
         end
     end
     i=i+1;
     if (i==(FoodNumber)+1)
          i=1;
     end
end
[GlobalMin,GlobalParams]=MemorizeBestSolution(Foods,ObjVal,GlobalMin,
     GlobalParams);
```

```
%%%% SCOUT BEE PHASE %
ind=find(trial==max(trial));
ind=ind(end);
if (trial(ind)>limit)
    trial(ind)=0;
    Foods(ind,:)=(ub-lb).*rand(1,D)+lb;
    ObjVal(ind)=feval(objfun,sol);
    Fitness(ind)=calculateFitness(ObjValSol);
end
fprintf('Cycle=%d ObjVal=%g\n',cycle,GlobalMin);
cycle=cycle+1;
end % End of ABC
function [sol,ObjValSol,FitnessSol]=GenerateNewSolution(Foods,i,objfun
    ,lb,ub)
[FoodNumber,D]=size(Foods);
%/*The parameter to be changed is determined randomly*/
        Param2Change=fix(rand*D)+1;
        %/*A randomly chosen solution is used in producing a mutant
    solution of the solution i*/
        neighbour=fix(rand*(FoodNumber))+1;
        %/*Randomly selected solution must be different from the
    solution i*/
            while(neighbour==i)
                neighbour=fix(rand*(FoodNumber))+1;
            end
       sol=Foods(i,:);
       % /*v_{ij}=x_{ij}+\phi_{ij}*(x_{kj}-x_{ij}) */
       sol(Param2Change)=Foods(i,Param2Change)+(Foods(i,Param2Change)-
    Foods(neighbour,Param2Change))*(rand-0.5)*2;
       % /*if generated parameter value is out of boundaries, it is
    shifted onto the boundaries*/
        ind=find(sol<lb);
        sol(ind)=lb(ind);
        ind=find(sol>ub);
        sol(ind)=ub(ind);
        %evaluate new solution
        ObjValSol=feval(objfun,sol);
        FitnessSol=calculateFitness(ObjValSol);
end
function [GlobalMin,GlobalParams]=MemorizeBestSolution(Foods,ObjVal,
    GlobalMin,GlobalParams)
%/*The best food source is memorized*/
BestInd=find(ObjVal==min(ObjVal));
BestInd=BestInd(end);
if(GlobalMin>ObjVal(BestInd))
GlobalMin=ObjVal(BestInd);
GlobalParams=Foods(BestInd,:);
end
end
function fFitness=calculateFitness(fObjV)
fFitness=zeros(size(fObjV));
ind=find(fObjV>=0);
fFitness(ind)=1./(fObjV(ind)+1);
ind=find(fObjV<0);
fFitness(ind)=1+abs(fObjV(ind));
end
function ObjVal=Sphere(Colony,xd)
S=Colony.*Colony;
ObjVal=sum(S');
end
```

Listing 2.1
ABC Source-code implemented using Matlab.

2.4 Source-code of ABC algorithm in C++

```
#include <iostream>
using namespace std;
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <conio.h>
#include <time.h>
/* Control Parameters of ABC algorithm*/
#define FoodNumber 20
#define limit 100
#define maxCycle 3000
/* Problem specific variables*/
#define D 50
#define lb -5.12
#define ub 5.12
double Foods[FoodNumber][D];
double f[FoodNumber];
double fitness[FoodNumber];
double trial[FoodNumber];
double prob[FoodNumber];
double solution[D];
double ObjValSol;
double FitnessSol;
int neighbour, param2change;
double GlobalMin;
double GlobalParams[D];
double r; /*a random number in the range [0,1)*/
/*benchmark functions */
double sphere(double sol[D]);
typedef double(*FunctionCallback)(double sol[D]);
FunctionCallback function = &sphere;
double CalculateFitness(double fun){
  double result = 0;
  if (fun >= 0)
    result = 1 / (fun + 1);
  else
    result = 1 + fabs(fun);
  return result;
}
void MemorizeBestSource(){
  int i, j;

  for (i = 0; i<FoodNumber; i++)
    if (f[i]<GlobalMin)
    {
      GlobalMin = f[i];
      for (j = 0; j < D; j++)
        GlobalParams[j] = Foods[i][j];
    }
}
void init(int index){
  int j;
  for (j = 0; j<D; j++){
    r = ((double)rand() / ((double)(RAND_MAX)+(double)(1)));
    Foods[index][j] = r*(ub - lb) + lb;
    solution[j] = Foods[index][j];
  }
  f[index] = function(solution);
  fitness[index] = CalculateFitness(f[index]);
  trial[index] = 0;
}
void initial(){
  int i;
  for (i = 0; i<FoodNumber; i++)
    init(i);

  GlobalMin = f[0];
  for (i = 0; i < D; i++)
    GlobalParams[i] = Foods[0][i];
}
void SendEmployedBees(){
  int i, j;
  for (i = 0; i<FoodNumber; i++){
    r = ((double)rand() / ((double)(RAND_MAX)+(double)(1)));
    param2change = (int)(r*D);
    r = ((double)rand() / ((double)(RAND_MAX)+(double)(1)));
    neighbour = (int)(r*FoodNumber);
    while (neighbour == i){
      r = ((double)rand() / ((double)(RAND_MAX)+(double)(1)));
      neighbour = (int)(r*FoodNumber);
    }
    for (j = 0; j < D; j++)
      solution[j] = Foods[i][j];
    /*v_{ij}=x_{ij}+\phi_{ij}*(x_{kj}-x_{ij}) */
    r = ((double)rand() / ((double)(RAND_MAX)+(double)(1)));
    solution[param2change] = Foods[i][param2change] + (Foods[i][
    param2change] - Foods[neighbour][param2change])*(r - 0.5) * 2;
    if (solution[param2change]<lb)
      solution[param2change] = lb;
    if (solution[param2change]>ub)
      solution[param2change] = ub;
    ObjValSol = function(solution);
    FitnessSol = CalculateFitness(ObjValSol);
    /*greedy selection*/
    if (FitnessSol>fitness[i]){
      trial[i] = 0;
      for (j = 0; j < D; j++)
        Foods[i][j] = solution[j];
      f[i] = ObjValSol;
      fitness[i] = FitnessSol;
    }
    else
      trial[i] = trial[i] + 1;
  }
}
void CalculateProbabilities(){
  int i;
  double maxfit;
  maxfit = fitness[0];
  for (i = 1; i<FoodNumber; i++)
    if (fitness[i]>maxfit)
      maxfit = fitness[i];
  for (i = 0; i<FoodNumber; i++)
    prob[i] = (0.9*(fitness[i] / maxfit)) + 0.1;
}
void SendOnlookerBees(){
  int i, j, t;
  i = 0;
  t = 0;
  while (t<FoodNumber){
    r = ((double)rand() / ((double)(RAND_MAX)+(double)(1)));
    if (r<prob[i]) /*choose a food source depending on its probability
     to be chosen*/
    {
      t++;
      r = ((double)rand() / ((double)(RAND_MAX)+(double)(1)));
      param2change = (int)(r*D);

      r = ((double)rand() / ((double)(RAND_MAX)+(double)(1)));
      neighbour = (int)(r*FoodNumber);

      while (neighbour == i){
        r = ((double)rand() / ((double)(RAND_MAX)+(double)(1)));
        neighbour = (int)(r*FoodNumber);
      }
      for (j = 0; j < D; j++)
        solution[j] = Foods[i][j];

      /*v_{ij}=x_{ij}+\phi_{ij}*(x_{kj}-x_{ij}) */
      r = ((double)rand() / ((double)(RAND_MAX)+(double)(1)));
      solution[param2change] = Foods[i][param2change] + (Foods[i][
    param2change] - Foods[neighbour][param2change])*(r - 0.5) * 2;
      if (solution[param2change]<lb)
        solution[param2change] = lb;
      if (solution[param2change]>ub)
        solution[param2change] = ub;
      ObjValSol = function(solution);
      FitnessSol = CalculateFitness(ObjValSol);
      /*greedy selection*/
      if (FitnessSol>fitness[i]){
        trial[i] = 0;
        for (j = 0; j < D; j++)
          Foods[i][j] = solution[j];
        f[i] = ObjValSol;
        fitness[i] = FitnessSol;
      }
      else
        trial[i] = trial[i] + 1;
    }
    i++;
    if (i == FoodNumber)
      i = 0;
  }/*while*/
}
void SendScoutBees(){
  int maxtrialindex, i;
  maxtrialindex = 0;
  for (i = 1; i<FoodNumber; i++)
    if (trial[i]>trial[maxtrialindex])
      maxtrialindex = i;
  if (trial[maxtrialindex] >= limit)
    init(maxtrialindex);
}
int main(){
  int cycle,  j;
  double mean;
  mean = 0;
  srand(time(NULL));
    initial();
    MemorizeBestSource();
    for (cycle = 0; cycle<maxCycle; cycle++){
      SendEmployedBees();
      CalculateProbabilities();
      SendOnlookerBees();
      MemorizeBestSource();
      SendScoutBees();
    }
    for (j = 0; j < D; j++)
      cout << "GlobalParam[" << j + 1 << "]: " << GlobalParams[j] <<
    endl;
    cout << "GlobalMin =" << GlobalMin << endl;
return 0;
}
double sphere(double sol[D]){
  int j;
  double top = 0;
  for (j = 0; j<D; j++)
    top = top + sol[j] * sol[j];
  return top;
}
```

Listing 2.2
ABC Source-code implemented using C++

2.5 Step-by-step numerical example of the ABC algorithm

In this chapter, a numerical example of how the ABC algorithm works is given step by step to solve the Sphere problem. Assume that the problem dimension (D) is 5 and the search space range is $[-5.12, 5.12]$ in all dimensions. The value of the *limit* parameter is 2 and the number of food sources (SN) is 6. $\vec{X_i}$ $(i = 1, 2, \ldots, SN)$ solutions are initialized by Eq. 2.1 and when each food source location is evaluated in the objective function (Sphere), the following $ObjVal_i$ values are obtained.

$$\vec{X_1} = \{4.1460, 0.97170, -2.0820, 3.0824, -2.6902\},\ ObjVal_1 = 39.2066$$
$$\vec{X_2} = \{4.9126, -2.4350, -1.8557, -4.8208, -0.4214\},\ ObjVal_2 = 56.9242$$
$$\vec{X_3} = \{-0.6260, 1.0531, -0.7765, 4.3914, 4.7420\},\ ObjVal_3 = 43.8748$$
$$\vec{X_4} = \{-3.9821, 2.1628, 0.0805, 2.3585, 0.4792\},\ ObjVal_4 = 26.3335$$
$$\vec{X_5} = \{-2.4774, -2.8493, -4.2443, -0.1166, 0.2164\},\ ObjVal_5 = 32.3305$$
$$\vec{X_6} = \{-0.9347, -3.9176, -2.4321, 0.8040, -2.7484\},\ ObjVal_6 = 30.3365$$

In the initialization phase, the number of trial values is set to 0 for all solutions. $\vec{trial} = \{0, 0, 0, 0, 0, 0\}$. $GlobalMin = 26.3335$ and the corresponding $GlobalParams = \{-3.9821, 2.1628, 0.0805, 2.3585, 0.4792\}$.

After initialization, ABC starts to iterate the phases in each cycle. In the employed bee phase, for each solution, a new solution is generated and a greedy selection is applied. At the first cycle, for $\vec{X_1}$, assume that $k = 4, j = 3$ and $\phi = 0.3582$ values are drawn randomly. It means that a new solution is generated by changing the 3rd parameter of $\vec{X_1}$ by substituting k and ϕ values in Eq. 2.2. $New_{1,3} = x_{1,3} + 0.3582 * (x_{1,3} - x_{4,3}) = -2.8566$. The other parameters of $\vec{New_1}$ but for the 3rd parameter are copied from $\vec{X_1}$. Corresponding objective function and fitness values are calculated as 43.0321 and 0.0227, respectively. The same procedure is applied for all solutions as shown in Table 2.1. Each new solution is compared to its basis and the better one is kept in the population. If the basis solution is retained, its counter is incremented by 1; otherwise it is set to 0. Because $\vec{X_1}$ is better than New_1, $\vec{X_1}$ is selected and its counter is incremented by 1. The food source population and trial counters after selection are shown in Table 2.2.

TABLE 2.1
New Food Source Locations Generated by Employed Bee Phase at cycle=1.

	K	j	ϕ	x1	x2	x3	x4	x5	ObjValSol	FitnessSol
New_1	4	3	0.3582	4.1460	0.9717	-2.8566	3.0824	-2.6902	43.0321	0.0227
New_2	3	2	0.9759	4.9126	-5.1200	-1.8557	-4.8208	-0.4214	77.2094	0.0128
New_3	6	1	0.8265	-0.3709	1.0531	-0.7765	4.3914	4.7420	43.6205	0.0224
New_4	1	4	-0.4762	-3.9821	2.1628	0.0805	2.7032	0.4792	28.0783	0.0344
New_5	1	2	0.4424	-2.4774	-4.5397	-4.2443	-0.1166	0.2164	44.8210	0.0218
New_6	4	1	-0.0116	-0.9700	-3.9176	-2.4321	0.8040	-2.7484	30.4038	0.0318

TABLE 2.2
The Food Source Population and Trial Counters after Selection in the Employed Bee Phase at Cycle=1.

	Selected	x1	x2	x3	x4	x5	ObjVal	Fitness	Trial
$\vec{X}_1$	$\vec{X}_1$	4.1460	0.97170	-2.0820	3.0824	-2.6902	39.2066	0.0249	1
$\vec{X}_2$	$\vec{X}_2$	4.9126	-2.4350	-1.8557	-4.8208	-0.4214	56.9242	0.0173	1
$\vec{X}_3$	New_3	-0.3709	1.0531	-0.7765	4.3914	4.7420	43.6205	0.0224	0
$\vec{X}_4$	$\vec{X}_4$	-3.9821	2.1628	0.0805	2.3585	0.4792	26.3335	0.0366	1
$\vec{X}_5$	$\vec{X}_5$	-2.4774	-2.8493	-4.2443	-0.1166	0.2164	32.3305	0.0300	1
$\vec{X}_6$	$\vec{X}_6$	-0.9347	-3.9176	-2.4321	0.8040	-2.7484	30.3365	0.0319	1

Based on the fitness values calculated by Eq. 2.4, the probability values are assigned by Eq. 2.3 as $\vec{P} = \{0.7118, 0.5247, 0.6513, 1.0000, 0.8381, 0.8850\}$. In the onlooker bees phase, for each source a random real number is drawn and if this number is lower than the probability value, an onlooker bee flies to search the vicinity of that solution by Eq. 2.2. For example, assume that the random number is 0.2548 for $\vec{X}_1$. Since this number is lower than the probability of $\vec{X}_1$ (0.7118), a new solution is generated in the vicinity of $\vec{X}_1$ and a greedy selection is applied between $\vec{X}_1$ and its mutant. For $\vec{X}_2$, assume that 0.5687 is generated randomly. Because this number is higher than the probability of $\vec{X}_2$, an onlooker bee is not recruited to this source. This process is iterated until the number of onlooker bees recruited to the sources is equal to the number of food sources ($SN = 6$ in this case). All the iterations at the first cycle of onlooker bee phase are given step-by-step in Table 2.3. The food

TABLE 2.3
The Search and Selection in the Onlooker Bee Phase at Cycle=1.

i	P_i	random	Fly	K	j	ϕ	x1	x2	x3	x4	x5	OF	Fit	Selected	trial
1	0.7118	0.2548	Yes	3	5	0.4455	4.1460	0.9717	-2.0820	3.0824	-5.1200	58.1838	0.0169	$\vec{X}_1$	2
2	0.5247	0.5687	No												1
3	0.6513	0.9037	No												0
4	1.0000	0.8909	Yes	5	2	-0.6044	-3.9821	-0.8665	0.0805	2.3585	0.4792	22.4066	0.0427	$\vec{X}_4 = New4$	0
5	0.8381	0.3054	Yes	4	4	-0.0402	-2.4774	-2.8493	-4.2443	-0.0171	0.2164	32.3172	0.0300	$\vec{X}_5 = New5$	0
6	0.8850	0.9047	No												1
1	0.7118	0.6099	Yes	6	4	0.6110	4.1460	0.9717	-2.0820	4.4745	-2.6902	49.7266	0.0197	$\vec{X}_1$	3
2	0.5247	0.5767	No												1
3	0.6513	0.1829	Yes	2	2	-0.9427	-0.3709	-2.2351	-0.7765	4.3914	4.7420	47.5073	0.0206	$\vec{X}_3$	1
4	1.0000	0.4899	Yes	6	1	0.4254	-5.1200	-0.8665	0.0805	2.3585	0.4792	32.7639	0.0296	$\vec{X}_4$	1

TABLE 2.4
The Food Source Population and Trial Counters after the Onlooker Bee Phase at Cycle=1.

	x1	x2	x3	x4	x5	ObjVal	Fitness	Trial
$\vec{X}_1$	4.1460	0.97170	-2.0820	3.0824	-2.6902	39.2066	0.0249	3
$\vec{X}_2$	4.9126	-2.4350	-1.8557	-4.8208	-0.4214	56.9242	0.0173	1
$\vec{X}_3$	-0.3709	1.0531	-0.7765	4.3914	4.7420	43.6205	0.0224	1
$\vec{X}_4$	-3.9821	-0.8665	0.0805	2.3585	0.4792	22.4066	0.0427	1
$\vec{X}_5$	-2.4774	-2.8493	-4.2443	-0.0171	0.2164	32.3172	0.0300	0
$\vec{X}_6$	-0.9347	-3.9176	-2.4321	0.8040	-2.7484	30.3365	0.0319	1

TABLE 2.5
The Food Source Population and Trial Counters after the Scout Phase at Cycle=1.

	x1	x2	x3	x4	x5	ObjVal	Fitness	Trial
$\vec{X}_1$	3.2228	4.1553	-3.8197	4.2330	1.3554	32.7639	0.0296	0
$\vec{X}_2$	4.9126	-2.4350	-1.8557	-4.8208	-0.4214	56.9242	0.0173	1
$\vec{X}_3$	-0.3709	1.0531	-0.7765	4.3914	4.7420	43.6205	0.0224	1
$\vec{X}_4$	-3.9821	-0.8665	0.0805	2.3585	0.4792	22.4066	0.0427	1
$\vec{X}_5$	-2.4774	-2.8493	-4.2443	-0.0171	0.2164	32.3172	0.0300	0
$\vec{X}_6$	-0.9347	-3.9176	-2.4321	0.8040	-2.7484	30.3365	0.0319	1

source population and the number of trials after the onlooker bee phase is completed at the first cycle are given in Table 2.4.

Once the onlooker bee phase is completed, it is checked whether $GlobalMin$ and $GlobalParams$ need to be updated. Since the objective function value of $\vec{X}_2$ is lower than current $GlobalMin$, $GlobalMin$ is assigned 22.4066 (objective value of $\vec{X}_2$) and $GlobalParams = \{-3.9821, -0.8665, 0.0805, 2.3585, 0.4792\}$.

The scout bee phase checks whether any trial counter exceeds the *limit* control parameter. Since the value of *limit* is predetermined as 2 and the counter associated with $\vec{X}_1$ is 3, the food source $\vec{X}_1$ is assumed to be exhausted. $\vec{X}_1$ is replaced with a new random food source location generated by Eq. 2.1 and its counter is set to 0. The food source population and the number of trials after the scout bee phase is completed at the first cycle are given in Table 2.5.

When the algorithm was iterated through 50 cycles using the control parameter values mentioned before, the following values were obtained: $GlobalMin = 0.0130748$, $GlobalParams = \{-0.0524,\ 0.0679,\ -0.0620,\ -0.0037,\ -0.0432\}$. The convergence of $GlobalMin$ by the ABC algorithm is shown in Fig. 2.1:

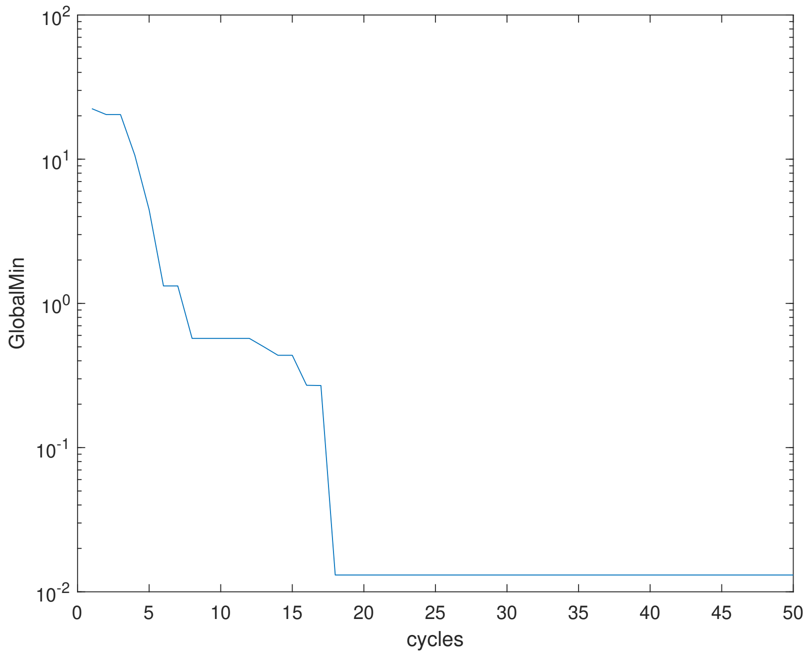

FIGURE 2.1
Convergence of $GlobalMin$ by ABC algorithm, $SN = 6, limit = 2$.

2.6 Conclusions

In this chapter, basic principles of the Artificial Bee Colony Algorithm are provided and the source codes of the ABC algorithm in Matlab and C++ programming languages are also given for the researchers who want to implement ABC and solve their optimization problems. For the readers who need a better understanding on how the ABC algorithm iterates the phases and cycles, a step-by-step procedure on a numerical example is presented. We hope that this chapter may be helpful to the researchers interested in the Artificial Bee Colony algorithm.

References

1. D. Karaboga. An idea based on honey bee swarm for numerical optimization. Technical Report TR06, Erciyes University, Engineering Faculty, Computer Engineering Department, 2005.

2. D. Karaboga, B. Basturk. A powerful and efficient algorithm for numerical function optimization: Artificial bee colony (abc) algorithm. *Journal of Global Optimization*, 39(3):459-471, 2007.
3. D. Karaboga, B. Akay. A comparative study of artificial bee colony algorithm. *Applied Mathematics and Computation*, 214:108-132, 2008.
4. D. Karaboga, B. Akay. A modified artificial bee colony (abc) algorithm for constrained optimization problems. *Applied Soft Computing*, 11(3):3021-3031, 2011.
5. B. Akay. Synchronous and asynchronous pareto-based multi-objective artificial bee colony algorithms. *Journal of Global Optimization*, 57(2):415-445, October 2013.
6. D. Karaboga, B. Akay. A survey: Algorithms simulating bee swarm intelligence. *Artificial Intelligence Review*, 31(1):68-55, 2009.
7. D. Karaboga, B. Gorkemli, C. Ozturk, N. Karaboga. A comprehensive survey: artificial bee colony (abc) algorithm and applications. *Artificial Intelligence Review*, 42(1):21-57, 2014.