Enhanced African Wild Dog Algorithm

By
Namir C. Shammas

Contents

1-Introduction

Subramanian [1] developed the African Wild Dog Algorithm (AWDA) as part of his doctorate thesis. AWDA belongs to the family of evolutionary metaheuristic optimization methods, like Particle Swarm Optimization (PSO) and Ant Colony Optimization (ACO). AWDA is based on the hunting dynamics of African wild dogs. This paper presents the basic algorithm, as designed by Subramanian, and a few enhanced versions, along with source code in MATLAB. As of the writing of this paper, I have found no source code for AWDA, in Python, Ruby, Excel VBA, and especially MATLAB.

2-The Basic Algorithm

The basic algorithm for AWDA is relatively simple, but has aspects that require some computations and data sorting. Figure 2-1 shows the pseudo-code for AWDA:

Step	***Task***
1	Generate a random matrix Pop() of MaxPop rows. Each row has N column, each for an optimizing variable. To be practical, make the matrix Pop() an augmented matrix by storing the fitness values for the

Step	*Task*
	optimizing function f(**x**) in column N+1. The random values in columns 1 to N are based on trust ranges, preselected for each variable.
2	Evaluate the fitness f(x) for each row of matrix Pop() and store it in columns N+1 of that matrix.
3	Sort the augmented matrix Pop() using column N+1 as the key for an ascending-order sort. This means that the first row of matrix Pop() contains the best/least value of f(x).
4	Calculate the values of the Euclidian distances between each two dogs in the MaxPop heard. These values are stored in matrix DistMat which has MaxPop rows and MaxPop columns. The diagonal values of matrix DistMat are all zeros. The values of DistMat(i,j) and DistMat(j,i) are equal for all valid values of i and j, since the matrix is symmetrical.
5	Calculate the total mean Euclidian distance, meanDist, for the values in matrix DistMat. Use the expression MaxPop^2 – MaxPop as the number of non-zero distances
6	Update the location of African wild dogs using: X(i) = X(i) + (X(j) – X(i)*U(0,1) * C * (meanDist/DistMat(i,j)) Where U(0,1) is a uniform random number. The indices j and i refer to values related to two African wild dogs. The index j is associated with a more fit African wild dog. The variable C is equal to 1 – iteration_number / maximum_iterations. The variable meanDist is the mean of total Euclidian distances. The variable DistMat(i,j) is the Euclidian distance between X(i) and X(j). Calculate the new fitness in Pop(i, N+1)
7	Repeat steps 3 to 6 for a specified number of iterations.

Figure 2-1. The pseudo code for the basic AWDA.

Listing 2-1 shows the MATLAB source code that implements the basic AWDA as outlined in the above pseudo-code:

```
function [bestX,bestFx] = awda(f, XLow, XHi, MaxIters, MaxPop)
% Implements basic African Wild dog Algorithm
%
% INPUT
% =====
%  f - handle to optimized function f(x).
%  XLow - array of low limits.
%  XHi - array of high limits. The size of this array supplies the number
%  of variables to the function.
%  MaxIters - maximum number of iterations.
%  MaxPop - population size.
%
% OUTPUT
```



```
% ======
%
% bestX - array of solutions.
% bestFx - best value of f(x).
%
%
  numVars = length(XHi);
  m = numVars + 1;
  mp1 = fix(MaxPop/2);
  pop = zeros(MaxPop, m);
  distMat = zeros(MaxPop, MaxPop);

  bestFx=1e+99;
  bestX = zeros(1,numVars);
  for i=1:MaxPop
    for j=1:numVars
      pop(i,j) = XLow(j) + (XHi(j) - XLow(j))*rand;
    end
    pop(i,m)=f(pop(i,1:numVars));
  end

  for iter=1:MaxIters

    % sort pop matrix by fitness
    pop = sortrows(pop,m);

    % calculate distance matrix
    sumDist=0;
    for i=1:MaxPop
      for j=1:MaxPop
        if j > i
          sumx = 0;
          for k=1:numVars
            sumx = sumx + (pop(i,k)-pop(j,k))^2;
          end
          sumx = sqrt(sumx);
          distMat(i,j) = sumx;
          sumDist = sumDist + sumx;
        elseif j < i
          distMat(i,j) = distMat(j,i);
          sumDist = sumDist + distMat(i,j);
        end
      end
    end

    % calculate mean Eucidean distance
    meanDist = sumDist / (MaxPop^2 - MaxPop);

    % move dogs
    c = 1 - iter/MaxIters;
    for k=1:mp1
      i1 = mp1 + fix((MaxPop - mp1)*rand);
      i2 = k;
      if pop(i1,m) > pop(i2,m)
        for i=1:numVars
```



```
          pop(i1,i) = pop(i1,i) + (pop(i2,i)-pop(i1,i)) * rand * c * meanDist
/ distMat(i1,i2);
        end
        % update fitness
        pop(i1,m) = f(pop(i1,1:numVars));
      end
    end
  end

  pop = sortrows(pop,m);
  bestFx = pop(1, m);
  bestX = pop(1, 1:numVars);
end
```

Listing 2-1. The MATLAB source code for function awda.

The input parameters of function *awda* are:

- f - handle to the optimized function f(x).
- XLow - array of low limits.
- XHi - array of high limits. The size of this array supplies the number of variables to the function.
- MaxIters - maximum number of iterations.
- MaxPop - population size.

The output parameters of function *awda* are:

- bestX - array of solutions.
- bestFx - best value of f(x).

The above input and output parameters are the same for all MATLAB functions that implement different versions of AWDA.

Listing 2-2 through 2-5 shows four optimization test functions. The *rosenbrock* function is usually difficult to solve. The *rosenborck2* function is easier to solve, because each optimizing variable contributes twice in calculating the function value. This is in contrast with the standard Rosenbrock function where the first and last variables contribute only once to calculating the function value. Thus, the *rosenborck2* function is more symmetrical and is easier to solve.

```
function s= rosenbrock(x)
%ROSENBROCK function
  n=length(x);
  s=0;
  for i=1:n-1
    s = s + (x(i)-1)^2 + 100*(x(i+1)-x(i)^2)^2;
  end
```



```
end
```

Listing 2-2. The MATLAB source code for function rosenbrock.

The optimum values *rosenbrock* function are all 1. for Here are sample sessions of testing function *awda* with function *rosenbrock*.

```
>>> [bestX,bestFx] = awda(@rosenbrock, [0 0], [5 5], 2000, 50)
bestX =
    1.0261    1.0285
bestFx =
    0.0604

>>> [bestX,bestFx] = awda(@rosenbrock, [0 0 0], [5 5 5], 2000, 50)
bestX =
    1.2255    1.4993    2.2463
bestFx =
    0.3010

>> [bestX,bestFx] = awda(@rosenbrock, [0 0 0 0], [5 5 5 5], 2000, 50)
bestX =
    1.0876    1.1984    1.4206    2.0145
bestFx =
    0.2731

>> [bestX,bestFx] = awda(@rosenbrock, [0 0 0 0], [5 5 5 5], 5000, 70)
bestX =
    1.0810    1.1756    1.3735    1.8748
bestFx =
    0.2028
```

Notice that as the *awda* function struggle to get good results with the *rosenbrock* function.

```
function s = rosenbrock2(x)
%ROSENBROCK function enhanced
  n=length(x);
  s=0;
  for i=1:n-1
    s = s + (x(i)-1)^2 + 100*(x(i+1)-x(i)^2)^2;
  end
  s = s + (x(n)-1)^2 + 100*(x(1)-x(n)^2)^2;
end
```

Listing 2-3. The MATLAB source code for function rosenbrock2.

The optimum values *rosenbrock2* function are all 1. Here are sample sessions of testing function *awda* with function *rosenbrock2*.

```
>> [bestX,bestFx] = awda(@rosenbrock2, [0 0], [5 5], 2000, 50)
bestX =
    0.9961    1.0004
bestFx =
    0.0090
```



```
>> [bestX,bestFx] = awda(@rosenbrock2, [0 0 0], [5 5 5], 2000, 50)
bestX =
    1.0007    0.9943    0.9966
bestFx =
    0.0171

>> [bestX,bestFx] = awda(@rosenbrock2, [0 0 0 0], [5 5 5 5], 2000, 50)
bestX =
    0.9751    0.9832    0.9773    0.9916
bestFx =
    0.2573

>> [bestX,bestFx] = awda(@rosenbrock2, [0 0 0 0], [5 5 5 5], 5000, 70)
bestX =
    1.0145    0.9942    1.0133    1.0217
bestFx =
    0.2745
```

The function *awda* does better in finding the optimum values for the easier-to-solve *rosenbrock2* function.

```
function s = fx1(x)
%FX1 function
  n=length(x);
  s=0;
  for i=1:n
    s = s + (x(i) - i)^2;
  end
end
```

Listing 2-4. The MATLAB source code for function fx1.

The optimum values for function *fx1* are [1, 2, 3, 4, …, n]. Here are sample sessions of testing function *awda* with function *fx1*.

```
>> [bestX,bestFx] = awda(@fx1, [0 0 0 0], [5 5 5 5], 2000, 50)
bestX =
    1.0312    2.0022    2.9901    3.9959
bestFx =
    0.0011

>> [bestX,bestFx] = awda(@fx1, [0 0 0 0], [5 5 5 5], 5000, 70)
bestX =
    1.0059    1.9976    2.9997    3.9884
bestFx =
   1.7464e-04
```

The function *awda* does a fair job in finding the optimum values for the *fx1* function. Notice that increasing the maximum number of iterations and the population size helps a bit in getting better results.

```
function s = fx2(x)
%FX1 function
  n=length(x);
```



```
  s=0;
  for i=1:n
    z = i;
    for j=1:3
      z = z + (i+j)/10^j;
    end
    s = s + (x(i) - z)^2;
  end
end
```

Listing 2-5. The MATLAB source code for function fx2.

The optimum values for function *fx2* are [1234, 2.345, 3.456, 4.567, …, n]. Here are sample sessions of testing function *awda* with function *fx2*.

```
>> [bestX,bestFx] = awda(@fx2, [0 0 0 0], [5 5 5 5], 2000, 50)
bestX =
    1.2354    2.3432    3.4651    4.5783
bestFx =
   2.1460e-04

>> [bestX,bestFx] = awda(@fx2, [0 0 0 0], [5 5 5 5], 5000, 70)
bestX =
    1.2303    2.3586    3.4400    4.5645
bestFx =
   4.6055e-04
```

The answers are fairly close to the actual optimum values.

3-Minor Enhancement

The problem with step 5 in the previous pseudo-code is that the sorted order in the matrix Pop, the values in the Euclidian distance matrix, and the mean Euclidian distance become corrupted as step 7 processes the values wild dogs pairs. Figure 3-1 shows the pseudo-code for AWDA which remedies the corruption in the matrices and mean Euclidian distance:

Step	*Task*
1	Generate a random matrix Pop() of MaxPop rows. Each row has N variables (i.e. columns). To be practical, make the matrix Pop() an augmented matrix by storing the fitness values for the optimizing function f(**x**) in column N+1. The random values in columns 1 to N are based on trust ranges, specified for each variable.
2	Evaluate the fitness f(x) for each row of matrix Pop() and store it in columns N+1 of that matrix.
3	Sort the augmented matrix Pop() using column N+1 as the key for an ascending-order sort. This means that the first row of matrix Pop() contains the best/least value of f(x).

Step	*Task*
4	Calculate the values of the Euclidian distances between each two dogs in the MaxPop heard. These values are stored in matrix DistMat which has MaxPop rows and MaxPop columns. The diagonal values of matrix DistMat are all zeros. The values of DistMat(i,j) and DistMat(j,i) are equal for all valid values of i and j, since the matrix is symmetrical.
5	Calculate the total mean Euclidian distance, meanDist, for the values in matrix DistMat. Use the expression MaxPop^2 – MaxPop as the number of non-zero distances.
6	Update the location of African wild dogs using: X(i) = X(i) + (X(j) – X(i)*U(0,1) * C * (meanDist/DistMat(i,j)) Where U(0,1) is a uniform random number. The indices j and i refer to values related to two African wild dogs. The index j is associated with a more fit African wild dog. The variable C is equal to 1 – iteration_number / maximum_iterations. The variable meanDist is the mean of total Euclidian distances. The variable DistMat(i,j) is the Euclidian distance between X(i) and X(j). Calculate the new fitness in Pop(i, N+1).
7	Update the order in the matrix Pop() to reinsert Pop(I,:) in its correct new row.
8	Update the distance matrix and mean Euclidian distance.
9	Repeat steps 6 to 8 for a specified number of iterations.

Figure 3-1. The pseudo-code for the slightly modified AWDA.

The updates in steps 7 and 8 should be efficient and target only the specific values that need updating. This efficiency enables the algorithm's implementations to operate quickly and use less CPU time.

Listing 3-1 shows the MATLAB source code.

```
function [bestX,bestFx] = awda2(f, XLow, XHi, MaxIter, MaxPop)
% Implements African Wild Dog Algorithm. This version performs more
% exhaustive processing to keep the population/fitness matrix and distance
% matrix up to date every time the values of a heard member are updated.
% The payoff from the extra computations is that the distance matrix does
% not become corrupted.
%
% INPUT
% =====
%  f - handle to optimized function f(x).
%  XLow - array of low limits.
%  XHi - array of high limits. The size of this array supplies the number
%  of variables to the function.
```



```
%  MaxIters - maximum number of iterations.
%  MaxPop - population size.
%
% OUTPUT
% ======
%
% bestX - array of solutions.
% bestFx - best value of f(x).
%
%
  numVars = length(XHi);
  m = numVars + 1;
  mp1 = fix(MaxPop/2);
  pop = zeros(MaxPop, m);
  distMat = zeros(MaxPop, MaxPop);
  bestFx=1e+99;
  bestX = zeros(1,numVars);
  for i=1:MaxPop
    for j=1:numVars
      pop(i,j) = XLow(j) + (XHi(j) - XLow(j))*rand;
    end
    pop(i,m)=f(pop(i,1:numVars));
  end

  % sort pop matrix by fitness
  pop = sortrows(pop,m);
  [meanDist,distMat] = SetDistMat(MaxPop, numVars, pop, distMat);

  for iter=1:MaxIter

    % move dogs
    c = 1 - iter/(MaxIter+1);
    for k=1:MaxPop
      while true
        i1 = 1 + fix(MaxPop*rand);
        i2 = 1 + fix(MaxPop*rand);
        if i1 < i2
          buff = i1;
          i1 = i2;
          i2 = buff;
        end
        if i2<i1 && i1<=MaxPop && i2<=MaxPop, break; end
      end
      for i=1:numVars
        pop(i1,i) = pop(i1,i) + (pop(i2,i)-
pop(i1,i))*rand*c*meanDist/distMat(i1,i2);
      end
      % update fitness
      pop(i1,m) = f(pop(i1,1:numVars));
      % sort the pop matrix
      pop = UpdateSortedPop(pop, i1, MaxPop, numVars);
      % update the distance matrix
      [meanDist,distMat] = UpdateDistMat(MaxPop, numVars, pop, i1, meanDist,
distMat);
    end
  end
```



```
  bestFx = pop(1, m);
  bestX = pop(1, 1:numVars);
end

function [meanDist,distMat] = SetDistMat(MaxPop, numVars, pop, distMat)
  % calculate distance matrix
  sumDist=0;
  for i=1:MaxPop
    for j=1:MaxPop
      if j > i
        sumx = 0;
        for k=1:numVars
          sumx = sumx + (pop(i,k)-pop(j,k))^2;
        end
        sumx = sqrt(sumx);
        distMat(i,j) = sumx;
        sumDist = sumDist + sumx;
      elseif j < i
        distMat(i,j) = distMat(j,i);
        sumDist = sumDist + distMat(i,j);
      end
    end
  end
  % calculate mean Euclidean distance
  meanDist = sumDist / (MaxPop^2 - MaxPop);
end

function [meanDist,distMat] = UpdateDistMat(MaxPop, numVars, pop, Idx,
meanDist, distMat)
  % update distance matrix
  sumDist = meanDist * (MaxPop^2 - MaxPop);
  % remove obsolete distances
  for i=1:MaxPop
    sumDist = sumDist - 2 * distMat(Idx,i);
  end
  for i=Idx:Idx
    for j=1:MaxPop
      sumx = 0;
      for k=1:numVars
        sumx = sumx + (pop(i,k)-pop(j,k))^2;
      end
      sumx = sqrt(sumx);
      distMat(i,j) = sumx;
      sumDist = sumDist + sumx;
      distMat(i,j) = distMat(j,i);
      sumDist = sumDist + distMat(i,j);
    end
  end
  % calculate mean Euclidean distance
  meanDist = sumDist / (MaxPop^2 - MaxPop);
end

function pop = UpdateSortedPop(pop, Idx, MaxPop, numVars)
% update oreder in matrix pop by reinserting element at row Idx
  m = numVars + 1;
```



```
  % Handle cases where no actual resorting is needed
  if Idx==1 && pop(Idx,m) < pop(Idx+1,m), return; end
  if Idx==MaxPop && pop(Idx,m) > pop(Idx-1,m), return; end
  if Idx > 1 && Idx < MaxPop
    if pop(Idx,m)>pop(Idx-1,m) && pop(Idx,m) < pop(Idx+1,m), return, end
  end

  apop = pop(Idx,:);
  if Idx==1
    pop(1:MaxPop-1,:) = pop(2:MaxPop,:);
  elseif Idx < MaxPop
    pop(Idx:MaxPop-1,:) = pop(Idx+1:MaxPop,:);
  end
  % search for insertion location
  for i=1:MaxPop
    if apop(m) < pop(i,m)
      Idx = i;
      break;
    end
  end
  % insert pop(idx,:)
  if Idx==MaxPop
    pop(MaxPop,:) = apop;
  elseif Idx==1
    pop(2:MaxPop,:) = pop(1:MaxPop-1,:);
    pop(1,:) = apop;
  else
    pop(Idx+1:MaxPop,:) = pop(Idx:MaxPop-1,:);
    pop(Idx,:) = apop;
  end
end
```

Listing 3-1. The source cod for the awda2 function.

Listing 3-1 shows the MATLAB code implementing the moderate enhancement outlined in the Figure 2-1 pseudo-code. The function *awda2* uses the following local functions:

- Function *SetDistMat* calculates the Euclidean distance matrix and the mean total Euclidean distance.
- Function *UpdateDistMat* updates the Euclidean distance matrix and the mean total Euclidean distance. This update occurs when the main function updates the variables associated with a single wild dog.
- Function *UpdateSortPop* updates the sorted order of matrix *Pop*() when the main function updates the variables associated with a single wild dog.

Using the functions *UpdateSortPop* and *UpdateDistaMat* are efficient and reduce computation time, compared to resorting the entire matrix *Pop*() and recalculating all of the elements in matrix *DistMat*.

Here are sample sessions that use function *awda2* with the four test functions:

```
>> [bestX,bestFx] = awda2(@rosenbrock, [0 0 0 0], [5 5 5 5], 2000, 50)
bestX =
    1.1556    1.3380    1.7849    3.1929
bestFx =
    0.7629

>> [bestX,bestFx] = awda2(@rosenbrock2, [0 0 0 0], [5 5 5 5], 2000, 50)
bestX =
    1.0219    1.0307    1.0238    1.0369
bestFx =
    0.4665

>> [bestX,bestFx] = awda2(@fx1, [0 0 0 0], [5 5 5 5], 2000, 50)
bestX =
    0.9999    1.9999    2.9999    4.0006
bestFx =
   4.1507e-07

>> [bestX,bestFx] = awda2(@fx2, [0 0 0 0], [5 5 5 5], 2000, 50)
bestX =
    1.2340    2.3450    3.4558    4.5669
bestFx =
   5.4232e-08
```

Updating the order of matrix Pop(), the values in the Euclidean distance matrix, and the mean total Euclidean distance seems to generally yield improved results.

4-The Enhanced AWDA Variant 1

After implementing AWDA in MATLAB, I realized that the algorithm can be enhanced. The implementation of the original algorithm required a relatively large number of heard size (MaxPop being in the range 70 to 100, or even more) and a high number of iterations.

Looking at the basic algorithm, one can see that the matrix Pop() is filled with random values in step 1. Any improvement on the optimizing variables depends on the equation that updates X(i). I decided to reinitialize the lowest L rows of matrix Pop() which contains the worst fitness values. Reinitializing the worst L rows on matrix Pop() allows the algorithm to introduce "new blood" that can help against stagnation or very slow convergence. This new infusion empowers AWDA to avoid stagnation. Figure 4-1 shows the pseudo-code for the first variant of the algorithm.

Step	***Task***
1	Generate a random matrix Pop() of MaxPop rows. Each row has N variables (i.e. columns). To be practical, make the matrix Pop() an

Step	*Task*
	augmented matrix by storing the fitness values for the optimizing function f(**x**) in column N+1. The random values in columns 1 to N are based on trust ranges, specified for each variable.
2	Evaluate the fitness f(x) for each row of matrix Pop() and store it in columns N+1 of that matrix.
3	Sort the augmented matrix Pop() using column N+1 as the key for an ascending-order sort. This means that the first row of matrix Pop() contains the best/least value of f(x).
4	Calculate the values of the Euclidian distances between each two dogs in the MaxPop heard. These values are stored in matrix DistMat which has MaxPop rows and MaxPop columns. The diagonal values of matrix DistMat are all zeros. The values of DistMat(i,j) and DistMat(j,i) are equal for all valid values of i and j, since the matrix is symmetrical.
5	Calculate the total mean Euclidian distance, meanDist, for the values in matrix DistMat. Use the expression MaxPop^2 – MaxPop as the number of non-zero distances.
6	Update the location of African wild dogs using: X(i) = X(i) + (X(j) – X(i)*U(0,1) * C * (meanDist/DistMat(i,j)) Where U(0,1) is a uniform random number. The indices j and i refer to values related to two African wild dogs. The index j is associated with a more fit African wild dog. The variable C is equal to 1 – iteration_number / (maximum_iterations + 1). The variable meanDist is the mean of total Euclidian distances. The variable DistMat(i,j) is the Euclidian distance between X(i) and X(j). Calculate the new fitness in Pop(i, N+1).
7	Update the order in matrix Pop() using the values calculated in step 6.
8	Update the Euclidean distance matrix and the mean total Euclidean distance.
9	Reinitialize the last L rows of matrix Pop() and calculate their fitness values.
10	Repeat steps 3 to 9 for a specified number of iterations.

Figure 4-1. The pseudo-code for the first variant of AWDA.

Also notice the slight change in how I calculate variable C. The modified expression avoids assigning zero to C in the last iteration, which seems pointless.

Listing 4-1 shows the MATLAB source code for function *awda3* which implements the first variant of the enhanced AWDA.


```
function [bestX,bestFx] = awda3(f, XLow, XHi, MaxIter, MaxPop)
% Implements African Wild dog Algorithm. This version performs more
% exhaustive processing to keep the population/fitness matrix and distance
% matrix up to date every time the values of a heard member are updated.
% The payoff from the extra computations is that the distance matrix does
% not become corrupted.
%
% INPUT
% =====
%  f - handle to optimized function f(x).
%  XLow - array of low limits.
%  XHi - array of high limits. The size of this array supplies the number
%  of variables to the function.
%  MaxIters - maximum number of iterations.
%  MaxPop - population size.
%
% OUTPUT
% ======
%
% bestX - array of solutions.
% bestFx - best value of f(x).
%
%
  numVars = length(XHi);
  m = numVars + 1;
  mp1 = fix(MaxPop/2);
  pop = zeros(MaxPop, m);
  distMat = zeros(MaxPop, MaxPop);
  bestFx=1e+99;
  bestX = zeros(1,numVars);
  for i=1:MaxPop
    for j=1:numVars
      pop(i,j) = XLow(j) + (XHi(j) - XLow(j))*rand;
    end
    pop(i,m)=f(pop(i,1:numVars));
  end

  for iter=1:MaxIter

    % sort pop matrix by fitness
    pop = sortrows(pop,m);
    % calculate distance matric
    [meanDist,distMat] = SetDistMat(MaxPop, numVars, pop, distMat);

    % move dogs
    c = 1 - iter/(MaxIter+1);
    for k=1:MaxPop
      while true
        i1 = 1 + fix(MaxPop*rand);
        i2 = 1 + fix(MaxPop*rand);
        if i1 < i2
          buff = i1;
          i1 = i2;
          i2 = buff;
        end
```

```
        if i2<i1 && i1<=MaxPop && i2<=MaxPop, break; end
      end
      for i=1:numVars
        pop(i1,i) = pop(i1,i) + (pop(i2,i)-
pop(i1,i))*rand*c*meanDist/distMat(i1,i2);
      end
      % update fitness
      pop(i1,m) = f(pop(i1,1:numVars));
      % sort the pop matrix
      pop = UpdateSortedPop(pop, i1, MaxPop, numVars);
      % update the distance matrix
      [meanDist,distMat] = UpdateDistMat(MaxPop, numVars, pop, i1, meanDist,
distMat);

    end

    for i=fix(MaxPop*3/4):MaxPop
      for j=1:numVars
        pop(i,j) = XLow(j) + (XHi(j) - XLow(j))*rand;
      end
      pop(i,m)=f(pop(i,1:numVars));
    end

  end

  bestFx = pop(1, m);
  bestX = pop(1, 1:numVars);
end

function [meanDist,distMat] = SetDistMat(MaxPop, numVars, pop, distMat)
  % calculate distance matrix
  sumDist=0;
  for i=1:MaxPop
    for j=1:MaxPop
      if j > i
        sumx = 0;
        for k=1:numVars
          sumx = sumx + (pop(i,k)-pop(j,k))^2;
        end
        sumx = sqrt(sumx);
        distMat(i,j) = sumx;
        sumDist = sumDist + sumx;
      elseif j < i
        distMat(i,j) = distMat(j,i);
        sumDist = sumDist + distMat(i,j);
      end
    end
  end
  % calculate mean Eucidean distance
  meanDist = sumDist / (MaxPop^2 - MaxPop);
end

function [sumDist,distMat] = UpdateDistMat(MaxPop, numVars, pop, Idx,
sumDist, distMat)
  % update distance matrix
  sumDist = sumDist * (MaxPop^2 - MaxPop);
```



```
  % remove obsolete distances
  for i=1:MaxPop
    sumDist = sumDist - 2 * distMat(Idx,i);
  end
  for i=Idx:Idx
    for j=1:MaxPop
      sumx = 0;
      for k=1:numVars
        sumx = sumx + (pop(i,k)-pop(j,k))^2;
      end
      sumx = sqrt(sumx);
      distMat(i,j) = sumx;
      sumDist = sumDist + sumx;
      distMat(i,j) = distMat(j,i);
      sumDist = sumDist + distMat(i,j);
    end
  end
  % calculate mean Eucidean distance
  sumDist = sumDist / (MaxPop^2 - MaxPop);
end

function pop = UpdateSortedPop(pop, Idx, MaxPop, numVars)
% update oreder in matrix pop by reinserting element at row Idx
  m = numVars + 1;
  % Handle cases where no actual resorting is needed
  if Idx==1 && pop(Idx,m) < pop(Idx+1,m), return; end
  if Idx==MaxPop && pop(Idx,m) > pop(Idx-1,m), return; end
  if Idx > 1 && Idx < MaxPop
    if pop(Idx,m)>pop(Idx-1,m) && pop(Idx,m) < pop(Idx+1,m), return, end
  end

  apop = pop(Idx,:);
  if Idx==1
    pop(1:MaxPop-1,:) = pop(2:MaxPop,:);
  elseif Idx < MaxPop
    pop(Idx:MaxPop-1,:) = pop(Idx+1:MaxPop,:);
  end
  % search for insertion location
  for i=1:MaxPop
    if apop(m) < pop(i,m)
      Idx = i;
      break;
    end
  end
  % insert pop(idx,:)
  if Idx==MaxPop
    pop(MaxPop,:) = apop;
  elseif Idx==1
    pop(2:MaxPop,:) = pop(1:MaxPop-1,:);
    pop(1,:) = apop;
  else
    pop(Idx+1:MaxPop,:) = pop(Idx:MaxPop-1,:);
    pop(Idx,:) = apop;
  end
end
```

Listing 4-1. The MATLAB source code for the awda3 function.

In listing 4-1, the code following code updates the last quarter of the population with new random values, as show in the snippet below:

```
    for i=fix(MaxPop*3/4):MaxPop
      for j=1:numVars
        pop(i,j) = XLow(j) + (XHi(j) - XLow(j))*rand;
      end
      pop(i,m)=f(pop(i,1:numVars));
    end
```

Here are sample sessions for using function *awda3* to test the optimized functions:

```
>> [bestX,bestFx] = awda3(@rosenbrock, [0 0 0 0], [5 5 5 5], 2000, 50)
bestX =
    1.0028    1.0056    1.0108    1.0216
bestFx =
   1.7303e-04

>> [bestX,bestFx] = awda3(@rosenbrock2, [0 0 0 0], [5 5 5 5], 2000, 50)
bestX =
    1.0004    1.0001    0.9998    1.0002
bestFx =
   1.0988e-04

>> [bestX,bestFx] = awda3(@fx1, [0 0 0 0], [5 5 5 5], 2000, 50)
bestX =
    0.9999    2.0000    2.9998    4.0002
bestFx =
   8.6575e-08

>> [bestX,bestFx] = awda3(@fx2, [0 0 0 0], [5 5 5 5], 2000, 50)
bestX =
    1.2343    2.3450    3.4559    4.5672
bestFx =
   1.3793e-07
```

The above session shows good results.

5-The Enhanced AWDA Variant 2

In this second variant, I replace step (7) above by reinitializing the last L rows using random values close to the best values in Pop(1, 1:N). This step creates new values that are close to the best values obtained so far. Figure 5-1 shows the pseudo-code for the second variant of the algorithm.

Step	*Task*
1	Generate a random matrix Pop() of MaxPop rows. Each row has N variables (i.e. columns). To be practical, make the matrix Pop() an augmented matrix by storing the fitness values for the optimizing function

Step	*Task*
	f(**x**) in column N+1. The random values in columns 1 to N are based on trust ranges, specified for each variable.
2	Evaluate the fitness f(x) for each row of matrix Pop() and store it in columns N+1 of that matrix.
3	Sort the augmented matrix Pop() using column N+1 as the key for an ascending-order sort. This means that the first row of matrix Pop() contains the best/least value of f(x).
4	Calculate the values of the Euclidian distances between each two dogs in the MaxPop heard. These values are stored in matrix DistMat which has MaxPop rows and MaxPop columns. The diagonal values of matrix DistMat are all zeros. The values of DistMat(i,j) and DistMat(j,i) are equal for all valid values of i and j, since the matrix is symmetrical.
5	Calculate the total mean Euclidian distance, meanDist, for the values in matrix DistMat. Use the expression MaxPop^2 – MaxPop as the number of non-zero distances.
6	Update the location of African wild dogs using: X(i) = X(i) + (X(j) – X(i)*U(0,1) * C * (meanDist/DistMat(i,j)) Where U(0,1) is a uniform random number. The indices j and i refer to values related to two African wild dogs. The index j is associated with a more fit African wild dog. The variable C is equal to 1 – iteration_number / (maximum_iterations + 1). The variable meanDist is the mean of total Euclidian distances. The variable DistMat(i,j) is the Euclidian distance between X(i) and X(j). Calculate the new fitness in Pop(i, N+1).
7	Update the order in matrix Pop() using the values calculated in step 6.
8	Update the Euclidean distance matrix and the mean total Euclidean distance.
9	Reinitialize the last L rows of matrix Pop(1:L,:) using random values closely *around* Pop(1,1:N) and calculate their fitness values.
10	Repeat steps 3 to 9 for a specified number of iterations.

Figure 5-1. The pseudo-code for the second variant of AWDA.

Listing 5-1 shows the MATLAB source code for function *awda4* which implements the second variant of the enhanced AWDA.

```
function [bestX,bestFx] = awda4(f, XLow, XHi, MaxIter, MaxPop)
% Implements African Wild dog Algorithm. This version performs more
% exhaustive processing to keep the population/fitness matrix and distance
% matrix up to date every time the values of a heard member are updated.
% The payoff from the extra computations is that the distance matrix does
```



```
% not become corrupted.
%
% INPUT
% =====
%  f - handle to optimized function f(x).
%  XLow - array of low limits.
%  XHi - array of high limits. The size of this array supplies the number
%  of variables to the function.
%  MaxIters - maximum number of iterations.
%  MaxPop - population size.
%
% OUTPUT
% ======
%
% bestX - array of solutions.
% bestFx - best value of f(x).
%
%
  numVars = length(XHi);
  m = numVars + 1;
  mp1 = fix(MaxPop/2);
  pop = zeros(MaxPop, m);
  distMat = zeros(MaxPop, MaxPop);
  bestFx=1e+99;
  bestX = zeros(1,numVars);
  for i=1:MaxPop
    for j=1:numVars
      pop(i,j) = XLow(j) + (XHi(j) - XLow(j))*rand;
    end
    pop(i,m)=f(pop(i,1:numVars));
  end

  for iter=1:MaxIter

    % sort pop matrix by fitness
    pop = sortrows(pop,m);
    % calculate distance matrix
    [meanDist,distMat] = SetDistMat(MaxPop, numVars, pop, distMat);

    % move dogs
    c = 1 - iter/(MaxIter+1);
    for k=1:MaxPop
      while true
        i1 = 1 + fix(MaxPop*rand);
        i2 = 1 + fix(MaxPop*rand);
        if i1 < i2
          buff = i1;
          i1 = i2;
          i2 = buff;
        end
        if i2<i1 && i1<=MaxPop && i2<=MaxPop, break; end
      end
      for i=1:numVars
        pop(i1,i) = pop(i1,i) + (pop(i2,i)-
pop(i1,i))*rand*c*meanDist/distMat(i1,i2);
      end
```



```
      % update fitness
      pop(i1,m) = f(pop(i1,1:numVars));
      % sort the pop matrix
      pop = UpdateSortedPop(pop, i1, MaxPop, numVars);
      % update the distance matrix
      [meanDist,distMat] = UpdateDistMat(MaxPop, numVars, pop, i1, meanDist,
distMat);
    end

    xrange = 0.1;
    for i=fix(MaxPop*3/4):MaxPop
      for j=1:numVars
        delta = (1+xrange*(rand-0.5));
        pop(i,j) = pop(1,j)*delta;
      end
      pop(i,m)=f(pop(i,1:numVars));
    end
   end

  %pop = sortrows(pop,m);
  bestFx = pop(1, m);
  bestX = pop(1, 1:numVars);
end

function [meanDist,distMat] = SetDistMat(MaxPop, numVars, pop, distMat)
  % calculate distance matrix
  sumDist=0;
  for i=1:MaxPop
    for j=1:MaxPop
      if j > i
        sumx = 0;
        for k=1:numVars
          sumx = sumx + (pop(i,k)-pop(j,k))^2;
        end
        sumx = sqrt(sumx);
        distMat(i,j) = sumx;
        sumDist = sumDist + sumx;
      elseif j < i
        distMat(i,j) = distMat(j,i);
        sumDist = sumDist + distMat(i,j);
      end
    end
  end
  % calculate mean Eucidean distance
  meanDist = sumDist / (MaxPop^2 - MaxPop);
end

function [meanDist,distMat] = UpdateDistMat(MaxPop, numVars, pop, Idx,
meanDist, distMat)
  % update distance matrix
  sumDist = meanDist * (MaxPop^2 - MaxPop);
  % remove obsolete distances
  for i=1:MaxPop
    sumDist = sumDist - 2 * distMat(Idx,i);
  end
  for i=Idx:Idx
```



```
    for j=1:MaxPop
      sumx = 0;
      for k=1:numVars
        sumx = sumx + (pop(i,k)-pop(j,k))^2;
      end
      sumx = sqrt(sumx);
      distMat(i,j) = sumx;
      sumDist = sumDist + sumx;
      distMat(i,j) = distMat(j,i);
      sumDist = sumDist + distMat(i,j);
    end
  end
  % calculate mean Eucidean distance
  meanDist = sumDist / (MaxPop^2 - MaxPop);
end

function pop = UpdateSortedPop(pop, Idx, MaxPop, numVars)
% update oreder in matrix pop by reinserting element at row Idx
  m = numVars + 1;
  % Handle cases where no actual resorting is needed
  if Idx==1 && pop(Idx,m) < pop(Idx+1,m), return; end
  if Idx==MaxPop && pop(Idx,m) > pop(Idx-1,m), return; end
  if Idx > 1 && Idx < MaxPop
    if pop(Idx,m)>pop(Idx-1,m) && pop(Idx,m) < pop(Idx+1,m), return, end
  end

  apop = pop(Idx,:);
  if Idx==1
    pop(1:MaxPop-1,:) = pop(2:MaxPop,:);
  elseif Idx < MaxPop
    pop(Idx:MaxPop-1,:) = pop(Idx+1:MaxPop,:);
  end
  % search for insertion location
  for i=1:MaxPop
    if apop(m) < pop(i,m)
      Idx = i;
      break;
    end
  end
  % insert pop(idx,:)
  if Idx==MaxPop
    pop(MaxPop,:) = apop;
  elseif Idx==1
    pop(2:MaxPop,:) = pop(1:MaxPop-1,:);
    pop(1,:) = apop;
  else
    pop(Idx+1:MaxPop,:) = pop(Idx:MaxPop-1,:);
    pop(Idx,:) = apop;
  end
end
```

Listing 5-1. The MATLAB source code for the awda4 function.

In listing 5-1 you have the following code snippet:

```
    xrange = 0.1;
```



```
    for i=fix(MaxPop*3/4):MaxPop
      for j=1:numVars
        delta = (1+xrange*(rand-0.5));
        pop(i,j) = pop(1,j)*delta;
      end
      pop(i,m)=f(pop(i,1:numVars));
    end
```

The variable *xrange* represents the total of 10% variation around the best values. The code updates the last quarter of the populations with random values in the range of -5% to 5% around the best values. You can experiment with changing the value of variable *xrange* and the index of the first population member to update.

Here are sample sessions for using function *awda4* to test the optimized functions:

```
>> [bestX,bestFx] = awda4(@rosenbrock, [0 0 0 0], [5 5 5 5], 2000, 50)
bestX =
    1.0006    1.0012    1.0023    1.0047
bestFx =
   7.1165e-06

>> [bestX,bestFx] = awda4(@rosenbrock2, [0 0 0 0], [5 5 5 5], 2000, 50)
bestX =
    1.0000    1.0000    1.0000    1.0000
bestFx =
   1.0936e-08

>> [bestX,bestFx] = awda4(@fx1, [0 0 0 0], [5 5 5 5], 2000, 50)
bestX =
    1.0000    2.0000    3.0000    4.0000
bestFx =
   1.0537e-10

>> [bestX,bestFx] = awda4(@fx2, [0 0 0 0], [5 5 5 5], 2000, 50)
bestX =
    1.2340    2.3450    3.4560    4.5670
bestFx =
   8.4077e-11
```

The results are very good and show that the second variant of AWDA performs better than the other three versions presented in this study.

6-Conclusion

The AWDA offers an efficient optimization tool, especially when using the second variant to the original algorithm. Replacing less fit members of the optimizing population with random values around the best member of the population avoid stagnation.

7-References

1) C. Subramanian, A. S. S. Sekar, K. Subramanian. "A New Engineering Optimization Method: African Wild Dog Algorithm", International Journal of Soft Computing, 8(3), pp 163-170, 2013.
2) Omid Bozorg-Haddad, “Advanced Optimization by Nature-Inspired Algorithms”, 2018, Springer.
3) Fei Chao, Steven Schockaert, and Qingfu Zhang (Editors), "Advances in Computational Intelligence Systems Contributions Presented at the 17th UK Workshop on Computational Intelligence, September 6 – 8, 2017, Cardiff, UK", 2018, Springr.
4) Patrick Siarry (Editor), "Metaheuristics", 2016, Springer.
5) Eisuke Kita (Editor), “Evolutionary Algorithms”, 2011, InTech.

