Modified Scout Optimization Algorithms

By
Namir Clement Shammas

Contents

1/ Introduction

In this paper I present the Modified Scout Optimization Algorithms (MSOA) as modified version of the Scout Optimization Algorithms (SOA) that I had previously developed and published on my web site. The original SOA mehod was inspired by the Fireworks Algorithm (FWA) developed by Y. Tan and Y. Zhu in 2010. The Fireworks Algorithm simulates the explosion of fire crackers to perform nested search for the global optimum. The algorithm uses two separate methods for explosions which are adopted for maintaining global and intensive search processes.

The Scout Optimization Algorithms use a set of pods (or scout masters, if you like) that locate the optimum function values with the help of scouts, associated with each pod. The scout locations help zoom in on better function values (less for minimization an vice versa) around each pod. The SOA method would generate a group of scouts located around a single pod and then the totality of pod locations and scout locations are merged and sorted to extract the best solutions and function values.

The Modified Scout Optimization Algorithms uses a slightly different approach. It generates the sequence of scout groups (each associated with a specific pod) and sorts each two (or more) sequences of scout groups in a separate augmented scout population matrix. Once an iteration has gone through the scouts of all of the pods, the special augmented scout matrix is then merged with the pod's matrix. This combined matrix is then sorted to generate a new pod population.

This study focuses on two versions of MSOA based on the basic SOA method that I present in my SOA study.

2/ Scout Optimization Algorithm Pseudo–Code

This section presents the pseudo code for the baseline version of SOA that I developed. The general steps are:

- Given the following:
    - The *fx* optimized function.
    - An array of ranges of the trust region for each variable.
    - The number of variables *N*.
    - The initial search step size *InitStep*.
    - The final search step size *FinStep*. This value influences the accuracy of the optimized variables.
    - The maximum iterations *MaxIters*.
    - The population size *MaxPop*.
    - The scout population size *MaxScoutPop*.
    - The decay factor flag *bSemilogDecayFlag*. When the value of this variable is true, the algorithm uses a log–linear decay of the step size factor. Otherwise, the algorithm uses a log–log decay of the step size factor.
- Create the population *pop* with *MaxPop* rows and *N+1* columns. The last column stores the function values based on the first *N* columns. Storing the function values in column *N+1* allows me to simply use the *sortrows()* function to sort the population matrix. using the values in the last column.
- Create the scout population *scoutPop* with *MaxScoutPop* rows and *N+1* columns. The last column stores the function values based on the first *N* columns.
- Fill the first *N* columns of matrix *pop* with random values that fall within the trust region.
- Calculate the function values for *MaxPop* rows and store them in column *N+1*.

- Sort the matrix *pop* using the values in column *N+1*. This yields the best guess in row 1.
- Repeat for *iter* from 1 to *MaxIters*
    - If *bSemilogDecayFlag* is true then
        - Calculate *StepSize* in log–linear decay mode using *InitStep*, *FinStep*, *iter*, and *MaxIters*.
    - Else
        - Calculate *StepSize* in log–log decay mode using *InitStep*, *FinStep*, *iter*, and *MaxIters*
    - For *I* = 1 to *MaxPop*
        - Calculate the location and function value for a new candicate for row *I*.
        - If the new candidate has a smaller function value than the member in row *I*, replace that member with the new candidate.
    - Sort the matrix *pop* using the values in column *N+1*. This yields the best guess in row 1.
    - Create an empty ScoutPopMat marix.
    - For *I* = 1 to *MaxPop*
        - Calculate *m2* as half of the scout population count.
        - For *j* = 1 to *m2*
            - For *k*=1 to *N*
                - Calculate *scoutPop(j,k) = pop(i,k) + (XHi(k) – XLow(k)) * StepSize * (2*uniform_rand–1).*
                - Check that *scoutPop(j,k)* is inside the trust region.
        - For *j = m2+1* to *MaxScoutPop*
            - For k=1 to N
                - Calculate *scoutPop(j,k) = normal_rand(pop(i,k), (XHi(k) – XLow(k))/3 * StepSize).*
                - Check that *scoutPop(j,k)* is inside the trust region.
        - Calculate the function values for the scout populations. The calculations store the function values in column *N+1* of matric scoutPop.
        - If *I* = 1 then
            - Copy the *ScoutPop* matrix into the *ScoutPopMat* matrix.
        - Else
            - Append the rows of the *ScoutPop* matrix to the rows of the *ScoutPopMat* matrix.

        - Sort the rows of matrix *ScoutPopMat* using the function values in column *N+1*.
        - Write the first *MaxScoutPop* rows of matrix *ScoutpPopMat* back into that same matrix.
    - Append the rows of matrix *scoutPoMat* to the rows of matrix *pop*.
    - Sort the augmented matrix *pop* using the values in column *N+1*.
    - Extract the first *MaxPop* rows of augmented matrix *pop* and store them as the updated matrix *pop*.
- The best solution is in the first *N* columns of the first row of matrix *pop*. The best optimized function value is in matrix element *pop(1, N+1)*.

The above pseudo-code shows the following main design variations:

1. The calculations of the decay of the step factor. Early implementations showed that a linear decay was not as efficient as the other two options presented in the above pseudo–code—the log–linear and log–log decays. The latter offers slightly better results.
2. The normal RNG used to calculate the elements *scoutPop(j,k)*. The above pseudo–code uses the regular normal RNG. This paper presents versions of SOA that replaces the Gaussian normal RNG with Cauchy RNG, with clipped Cauchy RNG, with the cosine RNG, as well as with other RNG functions.

☛I highly recommend that you normalize your variables so they can have search ranges that are at least of the same magnitude, and better yet, identical. You pass the normalized values to the optimized function. That function performs scaling of any variable, as needed, so that the calculations proceed with the actual values as dictated by the problem.

3/ The Modified Scout Optimization Algorithm Version 1

Listing 3.1 shows the code for the first version of the MSOA algorithm, stored in file *modscout.m*. The code in red fonts highlights the statement related to the mdofification instilled by the MSOA method.

```
function [bestX, bestFx] = modscout(fx, XLow, XHi, InitStep, FinStep, 
MaxIters,...
                                 MaxPop, MaxScoutPop, bSemiLogDecay, ...
                                 bSkipNarrowTrusRegion, bTrace)
```

```
% The MODSCOUT function implements a modified version of the Scout
Optimization
% Algorithm.
% This version uses both uniform and normal RNG to perform perturbation on
% the scout local search.
%
% INPUT
% =====
% fx - handle of optimized function.
% XLow - array of lower limits of trust region.
% XHi - array of upper limits of trust region.
% InitStep - initial search step size.
% FinStep - final search step size.
% MaxIters - maximum number of iterations.
% MaxPop - the population size.
% MaxScoutPop - the size of the local scout population.
% bSemiLogDecay - flag used to use log-linear step decay (when true) or
% log-log step decayscout (when false).
% bSkipNarrowTrusRegion - flag used to narrow the trus range before
% performing the optimization calculation.
% bTrace - optional Boolean flag used to trace intermediate results.
%
% OUTPUT
% ======
% bestX - array of best solution.
% bestFx - best function value.
%
% Example
% =======
%

if nargin < 11, bTrace = false; end
if nargin < 10, bSkipNarrowTrusRegion = false; end
if nargin < 9, bSemiLogDecay = true; end
if nargin < 8, MaxScoutPop = fix(MaxPop/4); end
tpc = fix(MaxIters/10); % calculate then % of total iterations
n = length(XLow);
if length(XHi) < n, XHi = XHi(1) + zeros(1,n); end
m = n + 1;

%---------------------- Narrow Trust Region ----------------
if ~bSkipNarrowTrusRegion
  [XLow, XHi, pop] = RangeZoom(fx, XLow, XHi, MaxIters, MaxPop, 0.1, 3);
  fprintf('New trust region is\n');
  fprintf('Xlow -> ')
  fprintf('%f ', XLow);
  fprintf('\nXHi -> ');
  fprintf('%f ', XHi);
  fprintf('\n');
else
  pop = zeros(MaxPop,m);
  pop(:,m) = 1e+99;
  for i=1:MaxPop
```

```
    pop(i,1:n) = XLow + (XHi - XLow) .* rand(1,n);
    pop(i,m) = fx(pop(i,1:n));
  end
  pop = sortrows(pop,m);
end

%--------------------------------------------------------------
scoutPop = zeros(MaxScoutPop,m);
scoutPop(:,m) = 1e+99;
if bTrace
  fprintf('%f ', pop(1,1:n));
  fprintf('%e\n', pop(1,m));
end

for iter = 1:MaxIters
  % StepFactor = InitStep + (FinStep - InitStep)*(iter-1)/(MaxIters-1);
  if bSemiLogDecay
    StepFactor = exp(log(InitStep) + (iter-1)/(MaxIters-1) *
log(FinStep/InitStep));
  else
     StepFactor = exp(log(InitStep) +
log(FinStep/InitStep)*log(iter)/log(MaxIters));
  end

  % update main population, whenevr possible
  popMember = zeros(1,m);
  for i=2:MaxPop
    if rand > 0.5
      sf = StepFactor;
      if rand > 0.5, sf = 1; end
      popMember(1:n) = pop(i,1:n) + (pop(1,1:n) - pop(i,1:n)) .* rand(1,n) *
sf;
    else
      sf = StepFactor;
      if rand > 0.5, sf = 1; end
      popMember(1:n) = XLow + (XHi - XLow) .* rand(1,n) * sf;
    end
    popMember(m) = fx(popMember(1:n));
    if popMember(m) < pop(i,m)
      pop(i,:) = popMember;
    end
  end
  pop = sortrows(pop,m);

  for i=1:MaxPop
    m2 = fix(MaxScoutPop/2);
    % Use uniform random perturbations
    for j=1:m2
      for k=1:n
        scoutPop(j,k)  = pop(i,k) + (XHi(k) - XLow(k)) * StepFactor *
(2*rand-1);
        if scoutPop(j,k) < XLow(k) || scoutPop(j,k) > XHi(k)
          scoutPop(j,k) = XLow(k) + (XHi(k) - XLow(k)) * rand;
        end

      end
      scoutPop(j,m) = fx(scoutPop(j,1:n));
    end
    % Use normal random perturbations
    for j=m2+1:MaxScoutPop
      for k=1:n
        scoutPop(j,k) = normrand(pop(i,k), (XHi(k) - XLow(k))/3 *
StepFactor);
        if scoutPop(j,k) < XLow(k) || scoutPop(j,k) > XHi(k)
          scoutPop(j,k) = XLow(k) + (XHi(k) - XLow(k)) * rand;
        end

      end
      scoutPop(j,m) = fx(scoutPop(j,1:n));
    end

    if i==1
      scoutPopMat = scoutPop;
    else
      scoutPopMat = [scoutPopMat; scoutPop];
      scoutPopMat = sortrows(scoutPopMat,m);
      scoutPopMat = scoutPopMat(1:MaxScoutPop,:);
    end


  end

  pop = [pop; scoutPopMat];
  pop = sortrows(pop,m);
  pop = pop(1:MaxPop,:);

  if bTrace && mod(iter, tpc) == 0
    fprintf('%f ', pop(1,1:n));
    fprintf('%20.18e, step_factor=%e\n', pop(1,m), StepFactor);
  end
end

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

function y = normrand(mean,sigma)
  y = mean + sigma * randn(1,1);
end
```

Listing 3.1. The source code for file modscout.m.

Listing 3.1 presents the source code for the first function of the MSOA. The code shows the implementation of the steps outlined by the pseudo code presented earlier.

Listing 3.2 shows the three test functions I used for testing the various versions of MSOA.

```
function y = fx1(x)
%FX1 Summary of this function goes here
%   Detailed explanation goes here
  n = length(x);
  y = 0;
  for i=1:n
    y = y + (x(i) - i)^2;
  end
end

function y = fx2(x)
%FX2 Summary of this function goes here
%   Detailed explanation goes here
  n = length(x);
  y = 0;
  for i=1:n
    z = i;
    for j=1:4
      z = z + (i+j)/10^j;
    end
    y = y + (x(i) - z)^2;
  end
end

function y = fx3(x)
%FX3 Summary of this function goes here
%   Detailed explanation goes here
  n = length(x);
  y = 0;
  for i=1:n
    y = y + (x(i) - i^2)^2;
  end
end
```

Listing 3.2. The source code for the test functions fx1, fx2, and fx3.

The optimum x values for function *fx1* are 1, 2, 3, and 4. The optimum x values for function *fx2* are 1.2345, 2.3456, 3.4567, and 4.5678. The optimum x values for function *fx3* are 1, 4, 9, and 16. I will focus on presenting the results of testing functions *fx2* and *fx3*. The optimum values of function *fx2* represent a challenge for multiple–digit accuracy. The optimum values of function *fx3* represent a challenge for finding more dispersed values for the optimum.

Listing 3.1 calls the function *RangeZoom()* to narrow the search range and make the optimization search more efficient since algorithm is searching over a narrower trust region. Listing 3.3 shows the source code for *RangeZoom.m*.

```
function [XLow, XHi] = RangeZoom(fx, XLow, XHi, MaxIters, MaxPop, frac)
% RANGEZOOM narrows the trust region.
```

```
%
% INPUT
% =====
% fx - handle of optimized function.
% XLow - array of lower limits of trust region.
% XHi - array of upper limits of trust region.
% MaxIters - maximum number of iterations.
% MaxPop - the population size.
% frac - the faction of the test population used to calculate
% the mean and standard deviation.
%
% OUTPUT
% ======
% XLow - array of narrowed lower limits of trust region.
% XHi - array of narrowed upper limits of trust region.
%
n = length(XLow);
if length(XHi) < n, XHi = XHi(1) + zeros(1,n); end
m = n + 1;

%---------------------- Narrow Trust Region ----------------
pop = zeros(MaxPop, m);
bestPop = zeros(MaxIters,m);
bestPop(:,m)= 1e+991
for iter =1:MaxIters
  for i=1:MaxPop
    pop(i,1:n) = XLow + (XHi - XLow) .* rand(1,n);
    pop(i,m) = fx(pop(i,1:n));
  end
  pop = sortrows(pop,m);
  bestPop(iter,:) = pop(1,:);
end
bestPop = sortrows(bestPop,m);
bestPop = bestPop(1:MaxIters,:)
m2 = fix(MaxPop*frac);
meanX = mean(bestPop(1:m2,1:n));
sdevX = std(bestPop(1:m2,1:n));
XLow = meanX - 2*sdevX;
XHi = meanX + 2*sdevX;
% check new boundaries with original boundaries
for i=1:n
  if XLow(i) < XLow0(i), XLow(i) = XLow0(i); end
  if XHi(i) > XHi0(i), XHi(i) = XHi0(i); end
end

end
```

Listing 3.3. The source code for file RangeZoom.m.

You have the complete freedom to customize the code for function *RangeZoom()* in the following ways:

- Incorporate the code for the function in one of the scout functions, replacing the call to the function with your custom code.

- Create different versions of *RangeZoom()* that you store in separate MATLAB files.
- Change the constant 2 used to calculate the narrow values for arrays *XLow* and *XHi* with other (possibly higher) values. This step is needed if the function *RangeZoom()* is giving you an inadequate trust region.
- You can change the argument for the parameter *frac* from 0.1 (used in the test files) to a different value that works better for your problem.

☛

The function *ZoomRange()* reduces the average optimized function values by a factor of 100 (i.e. an order of 2 magnitutdes). Using a narrow trust range permitted the optimization functions to do better (than with their counterparts that stuck with the user–given trust region) while using fewer iterations and smaller populations. Another bonus of using *ZoomRange()* is that the test script ran considerably faster.

Listing 3.4 and 3.5 shows test programs *test_scout.m* and *test_scoutb.m, respectively*. These files test functions *fx2* and *fx3*, respectively. The echo of the screen output was written in text files using the MATLAB command *diary*. The test in Listing 3.4 is aimed at a function of four variables (whose optimum variables are 1.2345, 2.3456, 3.4567, and 4.5678), using an initial step factor of 0.1, a final step factor of 0.0001, 100 iterations, a population of 100 members each generating a secondary population of 50 scouts. The first set of results uses a log–linear decay for the step factor. The second set of results uses a log–log decay for the step factor. The MATLAB script performs 40 iterations of each set to obtain results that can be sufficiently regarded as normally–distributed since they exceed 30 iterations—with 30 being the minimum sample size that can be regarded as having normally–distributed statistics.

```
clear
clc
diary 'modscout_fx2.txt'
n = 4;
Iters = 40;
bestFxArr = zeros(Iters,1);
bestXArr = zeros(Iters,n);
fprintf('------------------- Using Semilog Decay of Step Factors ------------
----\n');
for i=1:Iters
```

```
  fprintf('Iter # %i ----------------------------------------------------
\n', i);
  [bestX, bestFx] = modscout(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001,
100, 100, 50, true, false, true);
  fprintf('%f ', bestX);
  fprintf(', bestFx = %20.18e\n', bestFx);
  bestFxArr(i) = bestFx;
  bestXArr(i,:) = bestX;
  fprintf('\n-----------------------------------------------------------
------\n\n');
end

fprintf('-----------------------------------------------------------------
--\n');
fprintf('Min bestFx = %20.18e\n', min(bestFxArr));
fprintf('Max bestFx = %20.18e\n', max(bestFxArr));
fprintf('Mean bestFx = %20.18e\n', mean(bestFxArr));
fprintf('Sdev bestFx = %20.18e\n', std(bestFxArr));
fprintf('Median bestFx = %20.18e\n', median(bestFxArr));
fprintf('Array of bestFx is:\n');
disp(bestFxArr);
for i=1:n
    fprintf('Mean best X(%i) = %f\n', i, mean(bestXArr(:,i)));
end
fprintf('\n------------------- Using Power Decay of Step Factors ------------
----\n');
for i=1:Iters
  fprintf('Iter # %i ----------------------------------------------------
\n', i);
  [bestX, bestFx] = modscout(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001,
100, 100, 50, false, false, true);
  fprintf('%f ', bestX);
  fprintf(', bestFx = %20.18e\n', bestFx);
  bestFxArr(i) = bestFx;
  bestXArr(i,:) = bestX;
  fprintf('\n-----------------------------------------------------------
------\n\n');
end

fprintf('-----------------------------------------------------------------
--\n');
fprintf('Min bestFx = %20.18e\n', min(bestFxArr));
fprintf('Max bestFx = %20.18e\n', max(bestFxArr));
fprintf('Mean bestFx = %20.18e\n', mean(bestFxArr));
fprintf('Sdev bestFx = %20.18e\n', std(bestFxArr));
fprintf('Median bestFx = %20.18e\n', median(bestFxArr));
fprintf('Array of bestFx is:\n');
disp(bestFxArr);
for i=1:n
    fprintf('Mean best X(%i) = %f\n', i, mean(bestXArr(:,i)));
end
diary off
```

Listing 3.4.The test programs test_scout.m

Table 3.1 shows partial results of test file *test_modscout.m* that are stored in text file *modscout_fx2.txt.* I excluded most of the output to make the table short.

```
------------------- Using Semilog Decay of Step Factors ----------------
Iter # 1 ---------------------------------------------------
New trust region is
Xlow -> 0.484332 1.777827 2.795075 4.049301
XHi -> 1.757500 2.731808 4.019590 4.798523
1.029352 2.227028 3.439313 4.895591 1.638943e-01
1.234464 2.345105 3.457144 4.568031 4.96905450985865703e-07,
step_factor=5.336699e-02
1.234464 2.345562 3.456710 4.567831 3.75037948775353713e-09,
step_factor=2.656088e-02
1.234500 2.345600 3.456712 4.567800 1.41258848263175207e-10,
step_factor=1.321941e-02
1.234500 2.345599 3.456700 4.567800 4.65396169070993258e-13,
step_factor=6.579332e-03
1.234500 2.345600 3.456700 4.567800 1.94511780923103098e-13,
step_factor=3.274549e-03
1.234500 2.345600 3.456700 4.567800 4.87622302749599056e-14,
step_factor=1.629751e-03
1.234500 2.345600 3.456700 4.567800 3.23756536546570158e-15,
step_factor=8.111308e-04
1.234500 2.345600 3.456700 4.567800 3.55222804850002801e-16,
step_factor=4.037017e-04
1.234500 2.345600 3.456700 4.567800 1.65773718425058185e-16,
step_factor=2.009233e-04
1.234500 2.345600 3.456700 4.567800 1.36796664542654175e-16,
step_factor=1.000000e-04
1.234500 2.345600 3.456700 4.567800 , bestFx = 1.36796664542654175e-16

----------------------------------------------------------------------

Iter # 2 ---------------------------------------------------
New trust region is
Xlow -> 0.726307 1.820700 3.316493 4.030934
XHi -> 1.737451 2.950721 4.078413 5.000000
1.221959 2.381515 3.859230 4.341581 2.146524e-01
1.234110 2.346458 3.456530 4.566913 1.70289410561141160e-06,
step_factor=5.336699e-02
1.234718 2.345644 3.456829 4.567645 9.00098124744983043e-08,
step_factor=2.656088e-02
1.234490 2.345595 3.456726 4.567790 9.19748123475897361e-10,
step_factor=1.321941e-02
1.234492 2.345597 3.456703 4.567789 1.97044624448740783e-10,
step_factor=6.579332e-03
1.234500 2.345598 3.456705 4.567801 3.27878761204209333e-11,
step_factor=3.274549e-03
```

```
1.234499 2.345598 3.456703 4.567800 1.121965144971890257e-11,
step_factor=1.629751e-03
1.234499 2.345599 3.456703 4.567800 8.605923782069129041e-12,
step_factor=8.111308e-04
1.234499 2.345599 3.456703 4.567800 7.838081995111614615e-12,
step_factor=4.037017e-04
1.234499 2.345599 3.456703 4.567800 7.499270452279587458e-12,
step_factor=2.009233e-04
1.234499 2.345599 3.456703 4.567800 7.456774721227471002e-12,
step_factor=1.000000e-04
1.234499 2.345599 3.456703 4.567800 , bestFx = 7.456774721227471002e-12
```

```
....

----------------------------------------------------------------------

Iter # 39 -----------------------------------------------------
New trust region is
Xlow -> 0.691317 1.809257 2.821901 4.109554
XHi -> 1.746458 2.886066 4.177056 4.928298
1.274534 2.308440 3.505113 4.478294 1.333867e-02
1.235020 2.346250 3.456112 4.567613 1.073577013780695365e-06,
step_factor=5.336699e-02
1.234548 2.345588 3.456689 4.567826 3.229223770993364696e-09,
step_factor=2.656088e-02
1.234500 2.345595 3.456696 4.567799 4.771799976474232458e-11,
step_factor=1.321941e-02
1.234500 2.345598 3.456698 4.567799 8.505675064283571583e-12,
step_factor=6.579332e-03
1.234500 2.345598 3.456700 4.567800 3.894667059061106791e-12,
step_factor=3.274549e-03
1.234500 2.345599 3.456700 4.567800 2.027036684968165258e-12,
step_factor=1.629751e-03
1.234500 2.345600 3.456700 4.567800 8.028457419081070106e-15,
step_factor=8.111308e-04
1.234500 2.345600 3.456700 4.567800 3.145869196480294457e-15,
step_factor=4.037017e-04
1.234500 2.345600 3.456700 4.567800 9.168316080312542911e-16,
step_factor=2.009233e-04
1.234500 2.345600 3.456700 4.567800 6.555146421935908724e-16,
step_factor=1.000000e-04
1.234500 2.345600 3.456700 4.567800 , bestFx = 6.555146421935908724e-16

----------------------------------------------------------------------

Iter # 40 -----------------------------------------------------
New trust region is
Xlow -> 0.735332 1.861856 3.240542 3.943716
XHi -> 1.670755 2.815998 3.898054 5.000000
1.201300 2.437433 3.490924 4.727794 3.630492e-02
```

```
1.235029 2.345683 3.456524 4.567784 3.178079781823887282e-07,
step_factor=5.336699e-02
1.234542 2.345600 3.456657 4.567769 4.568997763433689785e-09,
step_factor=2.656088e-02
1.234516 2.345613 3.456706 4.567783 7.703624648784327138e-10,
step_factor=1.321941e-02
1.234510 2.345609 3.456699 4.567802 1.832140344125608757e-10,
step_factor=6.579332e-03
1.234502 2.345603 3.456700 4.567799 1.446321585910966038e-11,
step_factor=3.274549e-03
1.234502 2.345602 3.456699 4.567799 1.007937627174237219e-11,
step_factor=1.629751e-03
1.234502 2.345602 3.456699 4.567799 8.991894723553437479e-12,
step_factor=8.111308e-04
1.234501 2.345602 3.456699 4.567799 4.933260939462618542e-12,
step_factor=4.037017e-04
1.234500 2.345602 3.456699 4.567799 4.417500232040111411e-12,
step_factor=2.009233e-04
1.234500 2.345602 3.456700 4.567799 3.865373046491161901e-12,
step_factor=1.000000e-04
1.234500 2.345602 3.456700 4.567799 , bestFx = 3.865373046491161901e-12

----------------------------------------------------------------------

----------------------------------------------------------------------
Min bestFx = 8.744361171256913919e-20
Max bestFx = 1.059902660196750036e-11
Mean bestFx = 9.106619809241738513e-13
Sdev bestFx = 2.359320391696244343e-12
Median bestFx = 2.007802027131373065e-14
Array of bestFx is:
   1.0e-10 *

    0.0000
    0.0746
    0.0000
    0.0000
    0.0000
    0.0083
    0.0103
    0.0003
    0.0000
    0.0781
    0.0004
    0.0003
    0.0011
    0.0065
    0.0000
    0.0000
    0.0000
    0.0000
    0.0001
    0.0002
    0.1060
    0.0000
    0.0029
    0.0000
    0.0000
    0.0002
    0.0057
    0.0000
    0.0001
    0.0032
    0.0000
    0.0180
    0.0000
    0.0013
    0.0016
    0.0006
    0.0058
    0.0000
    0.0000
    0.0387

Mean best X(1) = 1.234500
Mean best X(2) = 2.345600
Mean best X(3) = 3.456700
Mean best X(4) = 4.567800

------------------- Using Power Decay of Step Factors ----------------
Iter # 1 -----------------------------------------------------
New trust region is
Xlow -> 0.960892 1.705921 2.872302 4.046542
XHi -> 1.772507 3.052014 4.007313 4.956967
1.345555 2.535376 3.273673 4.721807 1.055653e-01
1.234491 2.345668 3.456684 4.567811 5.139951817455677697e-09,
step_factor=3.162278e-03
1.234492 2.345602 3.456700 4.567801 7.202879556582618176e-11,
step_factor=1.118034e-03
1.234500 2.345602 3.456703 4.567799 1.648672164287294600e-11,
step_factor=6.085806e-04
1.234499 2.345601 3.456700 4.567800 1.672823343249564662e-12,
step_factor=3.952847e-04
1.234500 2.345600 3.456700 4.567800 1.267155298190897322e-14,
step_factor=2.828427e-04
1.234500 2.345600 3.456700 4.567800 4.543240764734911086e-15,
step_factor=2.151657e-04
1.234500 2.345600 3.456700 4.567800 1.867226823680852152e-15,
step_factor=1.707469e-04
1.234500 2.345600 3.456700 4.567800 9.927609530128726022e-16,
step_factor=1.397542e-04
1.234500 2.345600 3.456700 4.567800 3.187496010073828461e-16,
step_factor=1.171214e-04
1.234500 2.345600 3.456700 4.567800 3.459820415202041680e-17,
step_factor=1.000000e-04
1.234500 2.345600 3.456700 4.567800 , bestFx = 3.459820415202041680e-17
```

```
----------------------------------------------------------------------

Iter # 2 ----------------------------------------------------
New trust region is
Xlow -> 0.629087 1.595377 3.096148 4.344138
XHi -> 1.886004 2.908916 3.798728 4.924709
1.224449 2.234525 3.417021 4.532654 1.524843e-02
1.234456 2.345574 3.456734 4.567745 6.849691006289977392e-09,
step_factor=3.162278e-03
1.234490 2.345599 3.456696 4.567790 2.095767699868621470e-10,
step_factor=1.118034e-03
1.234498 2.345600 3.456701 4.567800 4.880111891510732796e-12,
step_factor=6.085806e-04
1.234501 2.345600 3.456699 4.567800 8.063749670786850729e-13,
step_factor=3.952847e-04
1.234500 2.345600 3.456700 4.567800 1.550725310303868977e-14,
step_factor=2.828427e-04
1.234500 2.345600 3.456700 4.567800 1.142382158433512400e-15,
step_factor=2.151657e-04
1.234500 2.345600 3.456700 4.567800 1.421213970339037280e-16,
step_factor=1.707469e-04
1.234500 2.345600 3.456700 4.567800 1.201868828452215616e-17,
step_factor=1.397542e-04
1.234500 2.345600 3.456700 4.567800 6.840680991923764696e-18,
step_factor=1.171214e-04
1.234500 2.345600 3.456700 4.567800 4.078329547726088946e-18,
step_factor=1.000000e-04
1.234500 2.345600 3.456700 4.567800 , bestFx = 4.078329547726088946e-18

----------------------------------------------------------------------

....

----------------------------------------------------------------------

Iter # 39 ----------------------------------------------------
New trust region is
Xlow -> 0.840606 2.081749 3.076767 4.169051
XHi -> 1.862682 2.532921 3.785961 5.000000
1.283493 2.231306 3.348471 4.636384 3.188063e-02
1.234493 2.345640 3.456751 4.567716 1.127338859163125746e-08,
step_factor=3.162278e-03
1.234505 2.345601 3.456701 4.567800 3.173245543269722706e-11,
step_factor=1.118034e-03
1.234500 2.345601 3.456700 4.567800 4.520722582739854492e-13,
step_factor=6.085806e-04
1.234500 2.345600 3.456700 4.567800 9.713944067033708152e-15,
step_factor=3.952847e-04
1.234500 2.345600 3.456700 4.567800 8.318662465894049704e-16,
step_factor=2.828427e-04
```

```
1.234500 2.345600 3.456700 4.567800 6.211176858863220530e-17,
step_factor=2.151657e-04
1.234500 2.345600 3.456700 4.567800 1.007139922819636624e-17,
step_factor=1.707469e-04
1.234500 2.345600 3.456700 4.567800 6.635567539271521973e-19,
step_factor=1.397542e-04
1.234500 2.345600 3.456700 4.567800 5.040517597169701719e-19,
step_factor=1.171214e-04
1.234500 2.345600 3.456700 4.567800 4.828840594263842577e-19,
step_factor=1.000000e-04
1.234500 2.345600 3.456700 4.567800 , bestFx = 4.828840594263842577e-19

----------------------------------------------------------------------

Iter # 40 -----------------------------------------------------
New trust region is
Xlow -> 0.593728 1.886666 3.013627 3.975041
XHi -> 1.854314 2.781387 3.973012 5.000000
1.063565 2.414252 3.560005 4.696102 6.106511e-02
1.234516 2.345622 3.456582 4.567788 1.491361252266437586e-08,
step_factor=3.162278e-03
1.234489 2.345602 3.456697 4.567798 1.426504283505436159e-10,
step_factor=1.118034e-03
1.234498 2.345601 3.456700 4.567801 4.753184055589709404e-12,
step_factor=6.085806e-04
1.234500 2.345600 3.456700 4.567800 2.172119168049934250e-13,
step_factor=3.952847e-04
1.234500 2.345600 3.456700 4.567800 8.412362577781836354e-14,
step_factor=2.828427e-04
1.234500 2.345600 3.456700 4.567800 2.978163103871000114e-14,
step_factor=2.151657e-04
1.234500 2.345600 3.456700 4.567800 1.986110997462655937e-14,
step_factor=1.707469e-04
1.234500 2.345600 3.456700 4.567800 1.921142176845849535e-14,
step_factor=1.397542e-04
1.234500 2.345600 3.456700 4.567800 1.710590850909793340e-14,
step_factor=1.171214e-04
1.234500 2.345600 3.456700 4.567800 1.590787085959413119e-14,
step_factor=1.000000e-04
1.234500 2.345600 3.456700 4.567800 , bestFx = 1.590787085959413119e-14

----------------------------------------------------------------------

----------------------------------------------------------------------
Min bestFx = 3.732544532695463241e-21
Max bestFx = 3.731306391613618907e-14
Mean bestFx = 3.511627942786292024e-15
Sdev bestFx = 8.257977343284162665e-15
Median bestFx = 4.190540321512182549e-17
Array of bestFx is:
   1.0e-13 *

    0.0003
    0.0000
    0.1393
    0.0154
    0.0004
    0.0002
    0.0000
    0.3731
    0.0000
    0.0018
    0.0001
    0.0215
    0.0000
    0.0011
    0.0000
    0.0244
    0.0175
    0.0000
    0.2693
    0.0018
    0.0515
    0.2069
    0.0001
    0.0002
    0.1079
    0.0000
    0.0000
    0.0091
    0.0000
    0.0004
    0.0000
    0.0000
    0.0005
    0.0004
    0.0006
    0.0005
    0.0001
    0.0009
    0.0000
    0.1591

Mean best X(1) = 1.234500
Mean best X(2) = 2.345600
Mean best X(3) = 3.456700
Mean best X(4) = 4.567800
```

Table 3.1. Partial results of test file test_modscout.m store in modscout_fx2.txt.

Statistic/Results	For Log–linear Decay	For Log–log Decay
Min bestFx	8.744361171256913919e-20	3.732544532695463241e-21
Max bestFx	1.059902660196750036e-11	3.731306391613618907e-14
Mean bestFx	9.106619809241738513e-13	3.511627942786292024e-15
Sdev bestFx	2.359320391696244343e-12	8.257977343284162665e-15

Statistic/Results	For Log–linear Decay	For Log–log Decay
Median bestFx	2.007802027131373065e-14	4.190540321512182549e-17
Mean Best X1	1.234500	1.234500
Mean Best X2	2.345600	2.345600
Mean Best X3	3.456700	3.456700
Mean Best X4	4.567800	4.567800

Table 3.2. Summary of results of test file test_modscout.m.

Table 3.2 shows the summary of the results of using file *test_modscout.m*. The table shows that the log–log decay of the step factor yields slightly better results than its log–linear counterpart.

Listing 3.5 uses the test function *fx3* that has the optimum function value at 1, 4, 9, and 16.

```
clear
clc
diary 'modscout_fx3.txt'
n = 4;
Iters = 40;
bestFxArr = zeros(Iters,1);
bestXArr = zeros(Iters,n);
fprintf('------------------- Using Semilog Decay of Walk Factors ------------
----\n');
for i=1:Iters
  fprintf('Iter # %i ----------------------------------------------------
\n', i);
  [bestX, bestFx] = modscout(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001,
100, 100, 50, true, true, true);
  fprintf('%f ', bestX);
  fprintf(', bestFx = %20.18e\n', bestFx);
  bestFxArr(i) = bestFx;
  bestXArr(i,:) = bestX;
  fprintf('\n-------------------------------------------------------------
------\n\n');
end

fprintf('-------------------------------------------------------------------
--\n');
fprintf('Min bestFx = %20.18e\n', min(bestFxArr));
fprintf('Max bestFx = %20.18e\n', max(bestFxArr));
fprintf('Mean bestFx = %20.18e\n', mean(bestFxArr));
fprintf('Sdev bestFx = %20.18e\n', std(bestFxArr));
fprintf('Median bestFx = %20.18e\n', median(bestFxArr));
fprintf('Array of bestFx is:\n');
disp(bestFxArr);
for i=1:n
    fprintf('Mean best X(%i) = %f\n', i, mean(bestXArr(:,i)));
end
```

```
fprintf('\n------------------- Using Power Decay of Walk Factors ------------
----\n');
for i=1:Iters
  fprintf('Iter # %i --------------------------------------------------
\n', i);
  [bestX, bestFx] = modscout(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001,
100, 100, 50,, false, true, true);
  fprintf('%f ', bestX);
  fprintf(‘, bestFx = %20.18e\n’, bestFx);
  bestFxArr(i) = bestFx;
  bestXArr(i,:) = bestX;
  fprintf('\n-------------------------------------------------------------
------\n\n');
end

fprintf('-----------------------------------------------------------------
--\n');
fprintf('Min bestFx = %20.18e\n', min(bestFxArr));
fprintf('Max bestFx = %20.18e\n', max(bestFxArr));
fprintf('Mean bestFx = %20.18e\n', mean(bestFxArr));
fprintf('Sdev bestFx = %20.18e\n', std(bestFxArr));
fprintf('Median bestFx = %20.18e\n', median(bestFxArr));
fprintf('Array of bestFx is:\n');
disp(bestFxArr);
for i=1:n
    fprintf('Mean best X(%i) = %f\n', i, mean(bestXArr(:,i)));
end
diary off
```

Listing 3.5. The source code for test_modscoutb.m.

Table 3.3 shows the summary output generated by Listing 3.5 and extracted from the output diary file *modscout_fx3.txt*. I am not including the details (and you are welcome to browse through the files that contain them) to keep this document short. Again, Table 3.3 shows that the log–log decay of the step factor yields better results than its log–linear counterpart.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	5.105128020140060429e-19	4.210117765191865442e-18
Max bestFx	3.112261161321236217e-10	1.238883274613960241e-12
Mean bestFx	2.196958149961545496e-11	8.847626595055703423e-14
Sdev bestFx	6.316911351096579971e-11	2.662877799390743312e-13
Median bestFx	3.370245728970861809e-13	1.889251774320651683e-15
Mean Best X1	1.000000	1.00000
Mean Best X2	3.999999	4.00000
Mean Best X3	9.000000	9.00000
Mean Best X4	15.999999	16.0000

Table 3.3. Summary of results of test file test_modscoutb.m

4/ The Modified Scout Optimization Algorithm Version 2

The second version of MSOA builds the special augmented scout matrix using two or more scout sets. The maximum number of scout sets grouped together in that augmented matrix is about 10% of the scout population. So a scout population of 50 members maintains up to 5 scout sets the special scout matrix. When an additional scout set is available, the algorithm includes that additional scout set in the special scout matrix, sorts the matrix, and then keeps the augmented matrix rows that correspond to the maximum number of scout sets.

Listing 4.1 shows the source code for *modscout2.m*. The statements in red highlight the new code instilled by the MSOA method.

```
function [bestX, bestFx] = modscout2(fx, XLow, XHi, InitStep, FinStep, ...
MaxIters, MaxPop, MaxScoutPop, bSemiLogDecay, ...
bSkipNarrowTrusRegion, bTrace)

% The MODSCOUT2 function implements a modified version of the Scout
Optimization
% Algorithm.
% This version uses both uniform and normal RNG to perform perturbation on
% the scout local search.
%
% INPUT
% =====
% fx - handle of optimized function.
% XLow - array of lower limits of trust region.
% XHi - array of upper limits of trust region.
% InitStep - initial search step size.
% FinStep - final search step size.
% MaxIters - maximum number of iterations.
% MaxPop - the population size.
% MaxScoutPop - the size of the local scout population.
% bSemiLogDecay - flag used to use log-linear step decay (when true) or
% log-log step decayscout (when false).
% bSkipNarrowTrusRegion - flag used to narrow the trus range before
% performing the optimization calculation.
% bTrace - optional Boolean flag used to trace intermediate results.
%
% OUTPUT
% ======
% bestX - array of best solution.
% bestFx - best function value.
%
% Example
% =======
%

if nargin < 11, bTrace = false; end
if nargin < 10, bSkipNarrowTrusRegion = false; end
if nargin < 9, bSemiLogDecay = true; end
if nargin < 8, MaxScoutPop = fix(MaxPop/4); end
```

```
% number of scout row sets to store in augment matrix scoutPopMat
ScoutSets = fix(MaxScoutPop/10);
if ScoutSets < 2, ScoutSets = 2; end
tpc = fix(MaxIters/10); % calculate then % of total iterations
n = length(XLow);
if length(XHi) < n, XHi = XHi(1) + zeros(1,n); end
m = n + 1;

%---------------------- Narrow Trust Region ----------------
if ~bSkipNarrowTrusRegion
  [XLow, XHi, pop] = RangeZoom(fx, XLow, XHi, MaxIters, MaxPop, 0.1, 3);
  fprintf('New trust region is\n');
  fprintf('Xlow -> ')
  fprintf('%f ', XLow);
  fprintf('\nXHi -> ');
  fprintf('%f ', XHi);
  fprintf('\n');
else
  pop = zeros(MaxPop,m);
  pop(:,m) = 1e+99;
  for i=1:MaxPop
    pop(i,1:n) = XLow + (XHi - XLow) .* rand(1,n);
    pop(i,m) = fx(pop(i,1:n));
  end
  pop = sortrows(pop,m);
end

%------------------------------------------------------------
scoutPop = zeros(MaxScoutPop,m);
scoutPop(:,m) = 1e+99;
if bTrace
  fprintf('%f ', pop(1,1:n));
  fprintf('%e\n', pop(1,m));
end

for iter = 1:MaxIters
  % StepFactor = InitStep + (FinStep - InitStep)*(iter-1)/(MaxIters-1);
  if bSemiLogDecay
    StepFactor = exp(log(InitStep) + (iter-1)/(MaxIters-1) *
log(FinStep/InitStep));
  else
     StepFactor = exp(log(InitStep) +
log(FinStep/InitStep)*log(iter)/log(MaxIters));
  end

  % update main population, whenevr possible
  popMember = zeros(1,m);
  for i=2:MaxPop
    if rand > 0.5
      sf = StepFactor;
      if rand > 0.5, sf = 1; end
      popMember(1:n) = pop(i,1:n) + (pop(1,1:n) - pop(i,1:n)) .* rand(1,n) *
sf;
    else
      sf = StepFactor;
      if rand > 0.5, sf = 1; end
      popMember(1:n) = XLow + (XHi - XLow) .* rand(1,n) * sf;
    end
    popMember(m) = fx(popMember(1:n));
    if popMember(m) < pop(i,m)
      pop(i,:) = popMember;
    end
  end
  pop = sortrows(pop,m);
  scoutPopMat = [];
  for i=1:MaxPop
    m2 = fix(MaxScoutPop/2);
    % Use uniform random perturbations
    for j=1:m2
      for k=1:n
        scoutPop(j,k)  = pop(i,k) + (XHi(k) - XLow(k)) * StepFactor * 
(2*rand-1);
        if scoutPop(j,k) < XLow(k) || scoutPop(j,k) > XHi(k)
          scoutPop(j,k) = XLow(k) + (XHi(k) - XLow(k)) * rand;
        end

      end
      scoutPop(j,m) = fx(scoutPop(j,1:n));
    end
    % Use normal random perturbations
    for j=m2+1:MaxScoutPop
      for k=1:n
        scoutPop(j,k) = normrand(pop(i,k), (XHi(k) - XLow(k))/3 * 
StepFactor);
        if scoutPop(j,k) < XLow(k) || scoutPop(j,k) > XHi(k)
          scoutPop(j,k) = XLow(k) + (XHi(k) - XLow(k)) * rand;
        end

      end
      scoutPop(j,m) = fx(scoutPop(j,1:n));
    end

    if i<ScoutSets
      scoutPopMat = [scoutPopMat; scoutPop];
    else
      scoutPopMat = [scoutPopMat; scoutPop];
      scoutPopMat = sortrows(scoutPopMat,m);
      scoutPopMat = scoutPopMat(1:ScoutSets,:);
    end

  end

  pop = [pop; scoutPopMat];
  pop = sortrows(pop,m);
  pop = pop(1:MaxPop,:);

  if bTrace && mod(iter, tpc) == 0
    fprintf('%f ', pop(1,1:n));
    fprintf('%20.18e, step_factor=%e\n', pop(1,m), StepFactor);
  end
end

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

```
end

function y = normrand(mean,sigma)
  y = mean + sigma * randn(1,1);
end
```

Listing 4.1. The source code for modscout2.m.

Listing 4.2 shows the test program *test_modscout2.m* that calculates the optimum variables for function *fx2*.

```
clear
clc
diary 'modscout2_fx2.txt'
n = 4;
Iters = 40;
bestFxArr = zeros(Iters,1);
bestXArr = zeros(Iters,n);
fprintf('------------------- Using Semilog Decay of Step Factors ------------
----\n');
for i=1:Iters
  fprintf('Iter # %i -----------------------------------------------------
\n', i);
  [bestX, bestFx] = modscout2(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001,
100, 100, 50, true, false, true);
  fprintf('%f ', bestX);
  fprintf(', bestFx = %20.18e\n', bestFx);
  bestFxArr(i) = bestFx;
  bestXArr(i,:) = bestX;
  fprintf('\n-------------------------------------------------------------
------\n\n');
end

fprintf('----------------------------------------------------------------
--\n');
fprintf('Min bestFx = %20.18e\n', min(bestFxArr));
fprintf('Max bestFx = %20.18e\n', max(bestFxArr));
fprintf('Mean bestFx = %20.18e\n', mean(bestFxArr));
fprintf('Sdev bestFx = %20.18e\n', std(bestFxArr));
fprintf('Median bestFx = %20.18e\n', median(bestFxArr));
fprintf('Array of bestFx is:\n');
disp(bestFxArr);
for i=1:n
    fprintf('Mean best X(%i) = %f\n', i, mean(bestXArr(:,i)));
end
fprintf('\n------------------- Using Power Decay of Step Factors ------------
----\n');
for i=1:Iters
  fprintf('Iter # %i -----------------------------------------------------
\n', i);
  [bestX, bestFx] = modscout2(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001,
100, 100, 50, false, false, true);
  fprintf('%f ', bestX);
  fprintf(', bestFx = %20.18e\n', bestFx);
  bestFxArr(i) = bestFx;
  bestXArr(i,:) = bestX;
  fprintf('\n------------------------------------------------------------
------\n\n');
end

fprintf('-------------------------------------------------------------------
--\n');
fprintf('Min bestFx = %20.18e\n', min(bestFxArr));
fprintf('Max bestFx = %20.18e\n', max(bestFxArr));
fprintf('Mean bestFx = %20.18e\n', mean(bestFxArr));
fprintf('Sdev bestFx = %20.18e\n', std(bestFxArr));
fprintf('Median bestFx = %20.18e\n', median(bestFxArr));
fprintf('Array of bestFx is:\n');
disp(bestFxArr);
for i=1:n
    fprintf('Mean best X(%i) = %f\n', i, mean(bestXArr(:,i)));
end
diary off
```

Listing 4.2. The source code for test_modscout2.m.

Table 4.1 shows the summary output generated by Listing 4.2 and extracted from the output diary file *modscout2_fx2.txt*. Table 4.1 shows that the log–log decay of the step factor yields slightly better results than its the log–linear counterpart.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	1.049484764544353129e-17	1.213502952409002092e-21
Max bestFx	8.569746545066849862e-12	7.521857837590060611e-15
Mean bestFx	6.298344956880605697e-13	8.959224544192521618e-16
Sdev bestFx	1.642605714872026592e-12	1.865707801984440054e-15
Median bestFx	5.318104431119190894e-14	4.426682983938272436e-17
Mean Best X1	1.234500	1.234500
Mean Best X2	2.345600	2.345600
Mean Best X3	3.456700	3.456700
Mean Best X4	4.567800	4.567800

Table 4.1. Summary of results of test file test_modscout2.m

Listing 4.3 shows the test program *test_modscout2b.m* that calculates the optimum variables for function *fx3*.

```
clear
clc
diary 'modscout2_fx3.txt'
n = 4;
Iters = 40;
bestFxArr = zeros(Iters,1);
bestXArr = zeros(Iters,n);
fprintf('------------------- Using Semilog Decay of Step Factors ------------
----\n');
for i=1:Iters
```

```
  fprintf('Iter # %i -----------------------------------------------------
\n', i);
  [bestX, bestFx] = modscout2(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001,
100, 100, 50, true, false, true);
  fprintf('%f ', bestX);
  fprintf(', bestFx = %20.18e\n', bestFx);
  bestFxArr(i) = bestFx;
  bestXArr(i,:) = bestX;
  fprintf('\n-------------------------------------------------------------
------\n\n');
end

fprintf('-----------------------------------------------------------------
--\n');
fprintf('Min bestFx = %20.18e\n', min(bestFxArr));
fprintf('Max bestFx = %20.18e\n', max(bestFxArr));
fprintf('Mean bestFx = %20.18e\n', mean(bestFxArr));
fprintf('Sdev bestFx = %20.18e\n', std(bestFxArr));
fprintf('Median bestFx = %20.18e\n', median(bestFxArr));
fprintf('Array of bestFx is:\n');
disp(bestFxArr);
for i=1:n
    fprintf('Mean best X(%i) = %f\n', i, mean(bestXArr(:,i)));
end
fprintf('\n------------------- Using Power Decay of Step Factors ------------
----\n');
for i=1:Iters
  fprintf('Iter # %i -----------------------------------------------------
\n', i);
  [bestX, bestFx] = modscout2(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001,
100, 100, 50, false, false, true);
  fprintf('%f ', bestX);
  fprintf(', bestFx = %20.18e\n', bestFx);
  bestFxArr(i) = bestFx;
  bestXArr(i,:) = bestX;
  fprintf('\n-------------------------------------------------------------
------\n\n');
end

fprintf('-----------------------------------------------------------------
--\n');
fprintf('Min bestFx = %20.18e\n', min(bestFxArr));
fprintf('Max bestFx = %20.18e\n', max(bestFxArr));
fprintf('Mean bestFx = %20.18e\n', mean(bestFxArr));
fprintf('Sdev bestFx = %20.18e\n', std(bestFxArr));
fprintf('Median bestFx = %20.18e\n', median(bestFxArr));
fprintf('Array of bestFx is:\n');
disp(bestFxArr);
for i=1:n
    fprintf('Mean best X(%i) = %f\n', i, mean(bestXArr(:,i)));
end
diary off
```

Listing 4.3. The source code for test_modscout2b.m.

Table 4.2 shows the summary output generated by Listing 4.3 and extracted from the output diary file *modscout2_fx3.txt*. Table 4.2 shows that the log–log decay of the step factor yields slightly better results than its the log–linear counterpart.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	1.995046898163798361e-17	8.903848058520920581e-20
Max bestFx	3.595798095538116375e-10	6.348358984208956560e-13
Mean bestFx	3.037242196816557597e-11	4.386668830953017179e-14
Sdev bestFx	6.907872462496153465e-11	1.252072433798722722e-13
Median bestFx	9.652473011846945035e-13	4.659500001097293768e-16
Mean Best X1	1.000000	1.000000
Mean Best X2	4.000000	4.000000
Mean Best X3	8.999999	9.000000
Mean Best X4	16.000001	16.000000

Table 4.2. Summary of results of test file test_modscout2b.m

5/ Comparing MSOA Versions 1 and 2

Table 5.1 compares the mean best optimized functions while testing function *fx2* using MATLAB functions *modscout()* and *modscout2()*.

Function	For Log–linear Decay	For Log–log Decay
modscout()	2.198023231459746774e-14	3.511627942786292024e-15
modscout2()	6.298344956880605697e-13	8.959224544192521618e-16

Table 5.1. Comparing the mean best optimized functions while testing function fx2 using modscout() and modscout2().

Table 5.1 shows that all of the optimized functions are in the orders of 10^{-13} through 10^{-16}. The function *modscout()* does better using the the log–linear decay factors. The function *modscout2()* does better using the the log–log decay factors.

Table 5.2 compares the mean best optimized functions while testing function *fx3* using MATLAB functions *modscout()* and *modscout2()*.

Function	For Log–linear Decay	For Log–log Decay
modscout()	2.196958149961545496e-11	8.847626595055703423e-14
modscout2()	3.037242196816557597e-11	4.386668830953017179e-14

Table 5.2. Comparing the mean best optimized functions while testing function fx3 using modscout() and modscout2().

Table 5.2 shows that all of the optimized functions are in the orders of 10^{-11} through 10^{-14}. Again, he function *modscout()* does better using the the log–linear decay factors, while function *modscout2()* does better using the the log–log decay factors.

Since the log-log decay mode results in better average minimum function values, the verdict goes in favor of the second version of MSOA.

6/Comparing MSOA and SOA

The question poses itself. Is MSOA better than SOA? Table 6.1 shows the results from using file *test_modscout2_2.m*. This file is similar to file *test_modscout2.m* except the maximum number of iterations, maximum population, maximum scout population are set to 200, 250, and 150, respectively to match the same parameters in *test_scout.m* that I present in the SOA study. Table 6.2 shows the results from using file *test_scout.m* that I presented in the SOA study--Table 6.2 here is Table 3.2 in the SOA study.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	1.972152263052529514e-31	1.972152263052529514e-31
Max bestFx	1.392398962094362988e-13	5.497994261565604955e-20
Mean bestFx	3.563431019356899505e-15	3.692714020621860399e-21
Sdev bestFx	2.200683214755317741e-14	1.223067166552569612e-20
Median bestFx	9.631862874114701402e-21	3.099034174355859008e-24
Mean Best X1	1.234500	1.234500
Mean Best X2	2.345600	2.345600
Mean Best X3	3.456700	3.456700
Mean Best X4	4.567800	4.567800

Table 6.1. Summary of results of test file test_modscout_2.m

Statistic/Results	For Log–linear Decay	For Log–log Decay
Min bestFx	0.000000000000000000e+00	2.465190328815661892e–31
Max bestFx	2.714423369343483748e–16	5.833394895601968535e–19
Mean bestFx	1.292036532307468458e–17	2.910301214191781435e–20
Sdev bestFx	5.004500137324713621e–17	9.890573161407465642e–20
Median bestFx	1.045888199234511120e–21	2.619399079704751543e–23
Mean Best X1	1.234500	1.234500

Statistic/Results	For Log–linear Decay	For Log–log Decay
Mean Best X2	2.345600	2.345600
Mean Best X3	3.456700	3.456700
Mean Best X4	4.567800	4.567800

Table 6.2. Summary of results of test file test_scout.m from the SOA study (same as Table 3.2 in that study).

Comparing the results in Tables 6.1 and 6.2, these results show that SOA does better when using the log-linear decay mode. By contrast, MSOA does better when using the log-log decay mode.

The SOA studies presented many versions of that algorithm. You can adapt the MSOA approach to the other best SOA methods and probably obtain relatively better results, especially when using the log-log decay mode.

A1/ Document History

As you can see in the document, I have numbered the sections and made the listing and table reference numbers use the format *section_number.sequence_number*. This document management style allows me to easily add new listings or tables, sometimes later, during the document development process, without going through the nightmare (and error–prone process) of renumbering subsequent listings and tables in the rest of the document. Moreover, I can add sections in different order and not be forced to add them sequentially. Any required renumbering of listing or tables is limited to those in the one edited section.

Version	***Date***	***Comments***
1.0.0	October 5, 2019	Initial release.