Scout Optimization Algorithms

By
Namir Clement Shammas

Contents

 Version 1.1.1

1/ Introduction

In this paper I present the Scout Optimization Algorithms (SOA) as a set of similar algorithms 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 similar, albeit simpler, approach of nested search for the global optimum as does the Fireworks Algorithm. The basic mechanism of SOA resembles launching search N pods, each with M scouts. Each pod lands at specific set of coordinates. From there, the scouts search the vicinity of the pod's landing locations. At the end of each iteration, the locations of the pods (also regarded as the main population) and the scouts are merged and resorted. The process keeps the best N members as the new main population and discards the rest. This process is repeated for a fixed umber of iterations. As the iterations progress, the search conducted by the scout focuses closer to the best location so far near the pod (which in turn get closer to the optimum). This important closing–in process enhances the accuracy of the results. This paper presents a set of similar algorithms (fifteen in all) that share most of the steps but differ in the random process perturbation used to zoom in on the optimum point. This step is very important in

zooming in on accurate solutions for optimizing the target function. That is why I will present several SOA versions with different random number generating functions.

This study also includes a few extended versions of SOA. The first version of the extended SOA has the scouts themselves employ secondary scouts. This nested scout approach improves the accuracy of the search for the optimum values. This improvement is due to the fact that the secondary scouts search even closer to the location of their lead primary scout.

The various versions of SOA offer you a variety of flavors. This variety allows you to test the different versions and select the one that works best with your optimization problems. Remember that each such problem has a different combination of multi–dimensional curvatures defined by the mathematical contributions of the optimized variables.

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.
    - 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.
    - Append the rows of matrix *scoutPop* to 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 Scout Optimization Algorithm Version 1

Listing 3.1 shows the code for the first version of the SOA algorithm, stored in file *scout.m*:

```
function [bestX, bestFx] = scout(fx, XLow, XHi, InitStep, FinStep,
MaxIters,...
                                 MaxPop, MaxScoutPop, bSemiLogDecay, ...
                                 bSkipNarrowTrusRegion, bTrace)
% The SCOUT function implements a simpler version of the Fireworks
% 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
    pop = [pop; scoutPop];
    pop = sortrows(pop,m);
    pop = pop(1:MaxPop,:);
  end

  if bTrace && mod(iter, tpc) == 0
    fprintf('%f ', pop(1,1:n));
    fprintf('%e, 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);
endend
```

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

Listing 3.1 presents the source code for the first function of the SOA. 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 SOA.

```
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+99l
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_scout().m* 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, 200 iterations, a population of 250 members each generating a secondary population of 150 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 'scout_fx2.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] = scout(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001, 200,
250, 150, true, true, true);
  fprintf(', bestFx = %20.18e\n', bestFx);
  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] = scout(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001, 200,
250, 150, false, true, true);
  fprintf(', bestFx = %20.18e\n', bestFx);
  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 source code for test_scout.m.

Table 3.1 shows a partial view of the output generated by Listing 3.3 and stored in the output diary file *scout_fx2.tx*t. The output shows the initial, intermediate, and final results in increments of 20 iterations. The first row shows the initial best results in matrix *pop*. The first four numbers in each row are the values of the optimized variables. The fifth number represents the optimized function value. The last number is the step size factor. The table gives you a good idea of the rate of convergence of the optimized variables at increments of 20 iterations (which is 10% of the maximum number of iterations of 200).

```
------------------- Using Semilog Decay of Step Factors ----------------
```

```
Iter # 1 ------------------------------------------------------
New trust region is
Xlow -> 0.670492 1.892522 3.084838 4.238682
XHi -> 1.734556 2.854887 3.945409 5.000000
1.234976 2.443886 3.454568 4.319884 7.112729e-02
1.234495 2.345599 3.456702 4.567807 8.495873994705718977e-11,
step_factor=5.170920e-02
1.234500 2.345600 3.456700 4.567800 1.267923609563013936e-14,
step_factor=2.582619e-02
1.234500 2.345600 3.456700 4.567800 8.18668977915103869 4e-19,
step_factor=1.289890e-02
1.234500 2.345600 3.456700 4.567800 1.140856757386162200e-20,
step_factor=6.442364e-03
1.234500 2.345600 3.456700 4.567800 1.084535305708431926e-23,
step_factor=3.217642e-03
1.234500 2.345600 3.456700 4.567800 3.419019798688754739e-25,
step_factor=1.607053e-03
1.234500 2.345600 3.456700 4.567800 3.318146182585880907e-29,
step_factor=8.026434e-04
1.234500 2.345600 3.456700 4.567800 2.465190328815661892e-31,
step_factor=4.008806e-04
1.234500 2.345600 3.456700 4.567800 0.000000000000000000e+00,
step_factor=2.002200e-04
1.234500 2.345600 3.456700 4.567800 0.000000000000000000e+00,
step_factor=1.000000e-04
1.234500 2.345600 3.456700 4.567800 , bestFx = 0.000000000000000000e+00

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

Iter # 2 ------------------------------------------------------
New trust region is
Xlow -> 0.850767 1.900496 3.111291 4.293411
XHi -> 1.636301 2.671081 3.875719 4.974085
1.100604 2.411922 3.376906 4.403564 5.566750e-02
1.234495 2.345621 3.456713 4.567810 7.113379255869153605e-10,
step_factor=5.170920e-02
1.234499 2.345601 3.456702 4.567800 6.539763823887087638e-12,
step_factor=2.582619e-02
1.234500 2.345600 3.456700 4.567800 1.843791802204353235e-14,
step_factor=1.289890e-02
1.234500 2.345600 3.456700 4.567800 4.885933870184693430e-16,
step_factor=6.442364e-03
1.234500 2.345600 3.456700 4.567800 1.118475887471289145e-17,
step_factor=3.217642e-03
1.234500 2.345600 3.456700 4.567800 4.343538623359170721e-18,
step_factor=1.607053e-03
1.234500 2.345600 3.456700 4.567800 4.087688855055471688e-18,
step_factor=8.026434e-04
1.234500 2.345600 3.456700 4.567800 4.076648846482689216e-18,
step_factor=4.008806e-04
1.234500 2.345600 3.456700 4.567800 4.075031024599381413e-18,
step_factor=2.002200e-04
```



```
1.234500 2.345600 3.456700 4.567800 4.075021360688444287e-18,
step_factor=1.000000e-04
1.234500 2.345600 3.456700 4.567800 , bestFx = 4.075021360688444287e-18

....

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

Iter # 39 ------------------------------------------------------
New trust region is
Xlow -> 0.861923 1.984850 3.041476 4.144476
XHi -> 1.502992 2.849215 3.877023 5.000000
1.123288 2.265975 3.590677 4.491321 4.250703e-02
1.234501 2.345594 3.456704 4.567805 7.894757839436951790e-11,
step_factor=5.170920e-02
1.234500 2.345599 3.456700 4.567800 3.540891904395976904e-13,
step_factor=2.582619e-02
1.234500 2.345600 3.456700 4.567800 1.361965767492907247e-16,
step_factor=1.289890e-02
1.234500 2.345600 3.456700 4.567800 7.315596831558704515e-19,
step_factor=6.442364e-03
1.234500 2.345600 3.456700 4.567800 1.556779180596725327e-20,
step_factor=3.217642e-03
1.234500 2.345600 3.456700 4.567800 8.468409026661021638e-22,
step_factor=1.607053e-03
1.234500 2.345600 3.456700 4.567800 2.063095749849478154e-22,
step_factor=8.026434e-04
1.234500 2.345600 3.456700 4.567800 7.302630890233732325e-23,
step_factor=4.008806e-04
1.234500 2.345600 3.456700 4.567800 5.545813194688096526e-23,
step_factor=2.002200e-04
1.234500 2.345600 3.456700 4.567800 1.597344671225209046e-23,
step_factor=1.000000e-04
1.234500 2.345600 3.456700 4.567800 , bestFx = 1.597344671225209046e-23

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

Iter # 40 ------------------------------------------------------
New trust region is
Xlow -> 0.820801 1.943486 3.060107 4.062572
XHi -> 1.679908 2.861210 3.780595 5.000000
1.292477 2.480946 3.489915 4.674165 3.409672e-02
1.234500 2.345603 3.456694 4.567808 1.098895788061001570e-10,
step_factor=5.170920e-02
1.234500 2.345600 3.456700 4.567800 2.492148548230594036e-13,
step_factor=2.582619e-02
1.234500 2.345600 3.456700 4.567800 4.634751949256123562e-17,
step_factor=1.289890e-02
1.234500 2.345600 3.456700 4.567800 1.048159539420871710e-18,
step_factor=6.442364e-03
1.234500 2.345600 3.456700 4.567800 3.852166273046431246e-20,
step_factor=3.217642e-03
```

```
1.234500 2.345600 3.456700 4.567800 2.561738037690816429e-20,
step_factor=1.607053e-03
1.234500 2.345600 3.456700 4.567800 5.223028115008939703e-23,
step_factor=8.026434e-04
1.234500 2.345600 3.456700 4.567800 4.779397142366517281e-23,
step_factor=4.008806e-04
1.234500 2.345600 3.456700 4.567800 2.13857613802408 4908e-23,
step_factor=2.002200e-04
1.234500 2.345600 3.456700 4.567800 2.134973114447101089e-23,
step_factor=1.000000e-04
1.234500 2.345600 3.456700 4.567800 , bestFx = 2.134973114447101089e-23

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

------------------------------------------------------------------------
Min bestFx = 0.000000000000000000e+00
Max bestFx = 2.714423369343483748e-16
Mean bestFx = 1.292036532307468458e-17
Sdev bestFx = 5.004500137324713621e-17
Median bestFx = 1.045888199234511120e-21
Array of bestFx is:
   1.0e-15 *

         0
    0.0041
    0.0000
    0.0001
    0.0000
    0.0000
    0.0186
    0.0000
    0.0001
    0.0000
    0.0000
    0.0164
    0.0000
    0.0000
         0
    0.0000
    0.0000
    0.0000
    0.2714
    0.1695
    0.0000
    0.0000
    0.0000
    0.0000
    0.0362
    0.0000
    0.0000
    0.0000
    0.0000
    0.0000
```



```
    0.0000
    0.0000
    0.0000
    0.0000
    0.0000
    0.0000
    0.0001
    0.0001
    0.0000
    0.0000

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 -> 0.745708 1.932033 3.044250 4.219425
XHi -> 1.522022 2.808906 3.744281 4.976522
1.255943 2.144201 3.352874 4.700367 6.937529e-02
1.234501 2.345602 3.456699 4.567800 4.276588478624854466e-12,
step_factor=2.012630e-03
1.234500 2.345600 3.456700 4.567800 3.766298236376315471e-16,
step_factor=8.152521e-04
1.234500 2.345600 3.456700 4.567800 2.131992513868503375e-18,
step_factor=4.805189e-04
1.234500 2.345600 3.456700 4.567800 3.822468252240680798e-20,
step_factor=3.302325e-04
1.234500 2.345600 3.456700 4.567800 8.288996375940032149e-23,
step_factor=2.468721e-04
1.234500 2.345600 3.456700 4.567800 2.766353017044788924e-24,
step_factor=1.946428e-04
1.234500 2.345600 3.456700 4.567800 4.692411397848156073e-25,
step_factor=1.592046e-04
1.234500 2.345600 3.456700 4.567800 2.530727413707226263e-25,
step_factor=1.337666e-04
1.234500 2.345600 3.456700 4.567800 2.241354991263725897e-25,
step_factor=1.147247e-04
1.234500 2.345600 3.456700 4.567800 2.233632043001612192e-25,
step_factor=1.000000e-04
1.234500 2.345600 3.456700 4.567800 , bestFx = 2.233632043001612192e-25

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

Iter # 2 -----------------------------------------------------
New trust region is
Xlow -> 0.805697 2.008368 3.127984 4.182927
XHi -> 1.627282 2.692712 3.844344 4.955633
1.201424 2.142081 3.561004 4.620578 5.617889e-02
1.234502 2.345602 3.456701 4.567801 8.168293069029103661e-12,
step_factor=2.012630e-03
```

```
1.234500 2.345600 3.456700 4.567800 4.70998765360223498e-15,
step_factor=8.152521e-04
1.234500 2.345600 3.456700 4.567800 2.68825083629765044e-17,
step_factor=4.805189e-04
1.234500 2.345600 3.456700 4.567800 3.24913799139206906e-19,
step_factor=3.302325e-04
1.234500 2.345600 3.456700 4.567800 1.76024785043154611e-20,
step_factor=2.468721e-04
1.234500 2.345600 3.456700 4.567800 1.06111349078194157e-21,
step_factor=1.946428e-04
1.234500 2.345600 3.456700 4.567800 1.99188712186969564e-22,
step_factor=1.592046e-04
1.234500 2.345600 3.456700 4.567800 1.38933753881622612e-22,
step_factor=1.337666e-04
1.234500 2.345600 3.456700 4.567800 1.37959406534035173e-22,
step_factor=1.147247e-04
1.234500 2.345600 3.456700 4.567800 1.37935431917137955e-22,
step_factor=1.000000e-04
1.234500 2.345600 3.456700 4.567800 , bestFx = 1.37935431917137955e-22

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

....

```
Iter # 39 ----------------------------------------------------
New trust region is
Xlow -> 0.828151 1.904035 3.078015 4.166508
XHi -> 1.646091 2.677066 3.823351 5.000000
1.223922 2.259446 3.434222 4.597943 8.948344e-03
1.234500 2.345598 3.456698 4.567800 8.45853651987130576e-12,
step_factor=2.012630e-03
1.234500 2.345600 3.456700 4.567800 1.74024992807250479e-15,
step_factor=8.152521e-04
1.234500 2.345600 3.456700 4.567800 9.03347145544939293e-19,
step_factor=4.805189e-04
1.234500 2.345600 3.456700 4.567800 3.14141284036728041e-20,
step_factor=3.302325e-04
1.234500 2.345600 3.456700 4.567800 4.57076013854434191e-22,
step_factor=2.468721e-04
1.234500 2.345600 3.456700 4.567800 1.40814933521793696e-23,
step_factor=1.946428e-04
1.234500 2.345600 3.456700 4.567800 2.36220616466467582e-24,
step_factor=1.592046e-04
1.234500 2.345600 3.456700 4.567800 1.79216674499343499e-24,
step_factor=1.337666e-04
1.234500 2.345600 3.456700 4.567800 1.73576107020644986e-24,
step_factor=1.147247e-04
1.234500 2.345600 3.456700 4.567800 1.31480965630428588e-24,
step_factor=1.000000e-04
1.234500 2.345600 3.456700 4.567800 , bestFx = 1.31480965630428588e-24

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



```
Iter # 40 ----------------------------------------------------
New trust region is
Xlow -> 0.939306 1.800689 3.077136 4.096293
XHi -> 1.638339 2.940119 3.836253 5.000000
1.181470 2.315555 3.259785 4.650602 4.934644e-02
1.234499 2.345602 3.456707 4.567798 5.74896062988905913e-11,
step_factor=2.012630e-03
1.234500 2.345600 3.456700 4.567800 1.93871613279630678e-15,
step_factor=8.152521e-04
1.234500 2.345600 3.456700 4.567800 1.28076176299760656e-17,
step_factor=4.805189e-04
1.234500 2.345600 3.456700 4.567800 9.84290829544024902e-20,
step_factor=3.302325e-04
1.234500 2.345600 3.456700 4.567800 1.45556601083949871e-21,
step_factor=2.468721e-04
1.234500 2.345600 3.456700 4.567800 4.25638315846146266e-22,
step_factor=1.946428e-04
1.234500 2.345600 3.456700 4.567800 3.92351112885880337e-22,
step_factor=1.592046e-04
1.234500 2.345600 3.456700 4.567800 3.77366462198948099e-22,
step_factor=1.337666e-04
1.234500 2.345600 3.456700 4.567800 3.76327788914477644e-22,
step_factor=1.147247e-04
1.234500 2.345600 3.456700 4.567800 3.76270721927733051e-22,
step_factor=1.000000e-04
1.234500 2.345600 3.456700 4.567800 , bestFx = 3.76270721927733051e-22

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

------------------------------------------------------------------------
Min bestFx = 2.46519032881566189e-31
Max bestFx = 5.83339489560196854e-19
Mean bestFx = 2.91030121419178144e-20
Sdev bestFx = 9.89057316140746564e-20
Median bestFx = 2.61939907970475154e-23
Array of bestFx is:
   1.0e-18 *

    0.0000
    0.0001
    0.0712
    0.0004
    0.0000
    0.0003
    0.0000
    0.0000
    0.0000
    0.0000
    0.0000
    0.0000
    0.0668
    0.0000
    0.1985
```

```
    0.1316
    0.0000
    0.0000
    0.0934
    0.0001
    0.0001
    0.0000
    0.0000
    0.0000
    0.0001
    0.0006
    0.5833
    0.0000
    0.0096
    0.0001
    0.0000
    0.0001
    0.0000
    0.0000
    0.0032
    0.0000
    0.0043
    0.0000
    0.0000
    0.0004

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_scout.m store in scout_fx2.txt.

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
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_scout.m.

Table 3.2 shows the summary of the results of using file *test_scout.m*. The table shows that the log–log decay of the step factor (except for the minimum best function value) 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 'scout_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] = scout(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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] = scout(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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_scoutb.m.

Table 3.3 shows the summary output generated by Listing 3.5 and extracted from the output diary file *scout_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	0.00000000000000000e+00	0.00000000000000000e+00
Max bestFx	1.221763650443924701e–14	6.952062984135061409e–16
Mean bestFx	9.732752177442062738e–16	1.812097031994652036e–17
Sdev bestFx	2.531950450791268875e–15	1.098749903302634835e–16
Median bestFx	4.708229015750529925e–20	8.239786407449882862e–23
Mean Best X1	1.00000	1.00000
Mean Best X2	4.00000	4.00000
Mean Best X3	9.00000	9.00000
Mean Best X4	16.0000	16.0000

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

4/ The Scout Optimization Algorithm Version 2

The second version of SOA goes from two segments to calculate the scout population to four. The first two segments use the range of the trust region in calculating a scout location. The last two segments use the differences between the best population locations and the population locations that have the same row number as the scout population being calculated. These four segments give the second version of SOA a good variation in calculating the scout population.

Listing 4.1 shows the source code for *scout2.m*.


```
function [bestX, bestFx] = scout2(fx, XLow, XHi, InitStep, FinStep,
MaxIters,...
                                  MaxPop, MaxScoutPop, bSemiLogDecay, ...
                                  bSkipNarrowTrusRegion, bTrace)
% The SCOUT function implements a simpler version of the Fireworks
% 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
    delta = fix(MaxScoutPop/4);
    n1 = 1;
    n2 = delta;
    % Use uniform random perturbations
    for j=n1:n2
      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
    n1 = n1 + delta;
    n2 = 2 * delta;
    for j=n1:n2
      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
    % Use uniform random perturbations
    n1 = n1 + delta;
    n2 = 3 * delta;
    p = 1;
    for j=n1:n2
      p = p + 1;
      for k=1:n
        scoutPop(j,k)  = pop(i,k) + (pop(1,k) - pop(p,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
    n1 = n1 + delta;
    n2 = 4 * delta;
    p = 1;
    for j=n1:n2
      p = p + 1;
      for k=1:n
        scoutPop(j,k) = normrand(pop(i,k), (pop(1,k) - pop(p,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
    pop = [pop; scoutPop];
    pop = sortrows(pop,m);
    pop = pop(1:MaxPop,:);
  end

  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 scout2.m.

Listing 4.1 has the following four statements that are used to calculate the *scoutPop()* elements in different ways:

```
scoutPop(j,k)  = pop(i,k) + (XHi(k) – XLow(k)) * StepFactor * (2*rand-1);
scoutPop(j,k) = normrand(pop(i,k), (XHi(k) – XLow(k))/3 * StepFactor);
scoutPop(j,k) = pop(i,k) + (pop(1,k) – pop(p,k)) * StepFactor * (2*rand-1);
scoutPop(j,k) = normrand(pop(i,k), (pop(1,k) – pop(p,k))/3 * StepFactor);
```

The second statement above calls the local function *normrand()* to obtain a normally distributed random number. The argument for the mean value is *pop(i, k)*. The argument for the standard deviation is *(XHi(k) – XLow(k))/3 * StepFactor*. I use the expression *(XHi(k) – XLow(k))/3* to represent half of the six sigmas (so that the normal RNG covers 6 sigmas total—3*sigma above the mean and 3*sigma below the mean) typically associated with normally–distributed number. Using normal random numbers with relatively small standard deviation helps increase the accuracy of the solutions for the optimized variables. The term *pop(1, k) – pop(p, k)* used in the last two MATLAB statements shown above is inspired by the popular Particle Swarm Optimization (PSO) algorithm. The same term is also used in the enhanced versions of the Fireworks Algorithms (FWA).

Listing 4.2 shows the test program *test_scout2.m* that calculates the optimum variables for function *fx2*.

```
clear
clc
diary 'scout2_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] = scout2(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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] = scout2(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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_scout2.m.

Table 4.1 shows the summary output generated by Listing 4.2 and extracted from the output diary file *scout2_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	1.4496380930114729 45e–13	8.073913208654659589e–14
Max bestFx	8.987439548690382112e–12	4.607152892999552170e–12

Statistic/Result	For Log–linear Decay	For Log–log Decay
Mean bestFx	2.5063402762683297 64e–12	1.4023920321959240 52e–12
Sdev bestFx	2.2491530189533465 71e–12	1.2466316298907898 26e–12
Median bestFx	1.8709244876161655 96e–12	9.0771739112195137 30e–13
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_scout2.m

Listing 4.3 shows the test program *test_scout2b.m* that calculates the optimum variables for function *fx3*.

```
clear
clc
diary 'scout2_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] = scout2(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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] = scout2(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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_scout2b.m.

Table 4.2 shows the summary output generated by Listing 4.3 and extracted from the output diary file *scout2_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	2.861391692459047083e–12	1.501186917465269386e–12
Max bestFx	2.963735142474835270e–10	1.276546859751047043e–10
Mean bestFx	9.254635542193029242e–11	3.460369189744170226e–11
Sdev bestFx	7.586226793515576171e–11	2.926898532343415457e–11
Median bestFx	6.476147350713136187e–11	2.392148681529799968e–11
Mean Best X1	1.000000	1.000000
Mean Best X2	3.999999	3.999999
Mean Best X3	9.000001	9.000001
Mean Best X4	15.999999	16.000000

Table 4.2. Summary of results of test file test_scout2b.m

5/ Comparing Versions 1 and 2

Table 5.1 compares the mean best optimized functions while testing function *fx2* using MATLAB functions *scout()* and *scout2()*.

Function	For Log–linear Decay	For Log–log Decay
scout()	1.292036532307468458e–17	2.9103012141917814 35e–20
scout2()	2.506340276268329764e–12	1.402392032195924052e–12

Table 5.1. Comparing the mean best optimized functions while testing function fx2 using scout() and scout2().

Table 5.1 shows that all of the optimized functions are in the orders of 10^{-12} through 10^{-20}. The function *scout()* does better using the both the log–linear and log–log step decay factors.

Table 5.2 compares the mean best optimized functions while testing function *fx3* using MATLAB functions s*cout()* and *scout2()*.

Function	For Log–linear Decay	For Log–log Decay
scout()	9.732752177442062738e–16	1.812097031994652036e–17
scout2()	9.254635542193029242e–11	3.460369189744170226e–11

Table 5.2. Comparing the mean best optimized functions while testing function fx3 using scout() and scout2().

Table 5.2 shows that all of the optimized functions are in the orders of 10^{-11} through 10^{-17}. The function *scout()* does better using the both the log–linear and log–log step decay factors.

Tables 5.1 and 5.2 indicate that, for the tested functions, function *scout()* performs better than function *scout2()*.

6/ The Scout Optimization Algorithm Version 3

The third version of SOA replaces the normal RNG with a Cauchy RNG. The Cauchy distribution is also bell–shaped like the Gaussian distribution. The main difference is that the tails of the Cauchy distributions extend its tails pretty much indefinitely. As a consequence, the Cauchy probability distribution has no estimates for the mean and variance! The next version of SOA (giving you a heads up) implements a version of the Cauchy RNG with clipped values beyond plus and minus 3**sigma*. So, stay tuned!

Listing 6.1 shows the source code for *scout3.m*.

```
function [bestX, bestFx] = scout3(fx, XLow, XHi, InitStep, FinStep,
MaxIters,...
                                  MaxPop, MaxScoutPop, bSemiLogDecay, ...
                                  bSkipNarrowTrusRegion, bTrace)
% The SCOUT function implements a version of the Fireworks
% Algorithm. This version uses both uniform and Cauchy 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 non-uniform random perturbations
    for j=1:MaxScoutPop
      for k=m2+1:n
        scoutPop(j,k) = cauchyrand(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
    pop = [pop; scoutPop];
    pop = sortrows(pop,m);
    pop = pop(1:MaxPop,:);
  end

  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 = cauchyrand(mean, sigma)
  y = sigma*tan((rand - rand)) + mean;
  y = (y + 1.5)/3;
  y = 2*y - 1;
end
```

Listing 6.1. The source code for scout3.m.

Listing 6.1 has the following statement used to calculate the *scoutPop()* values using the Cauchy RNG:

```
scoutPop(j,k) = cauchyrand(pop(i,k), (XHi(k) - XLow(k))/3 * StepFactor);
```

The local function *cauchyrand()* is defined as:

```
function y = cauchyrand(mean, sigma)
  y = sigma*tan((rand - rand)) + mean;
  y = (y + 1.5)/3;
  y = 2*y - 1;
end
```

Figure 6.1 shows the distribution of the *cauchyrand()* function in Listing 6.1.

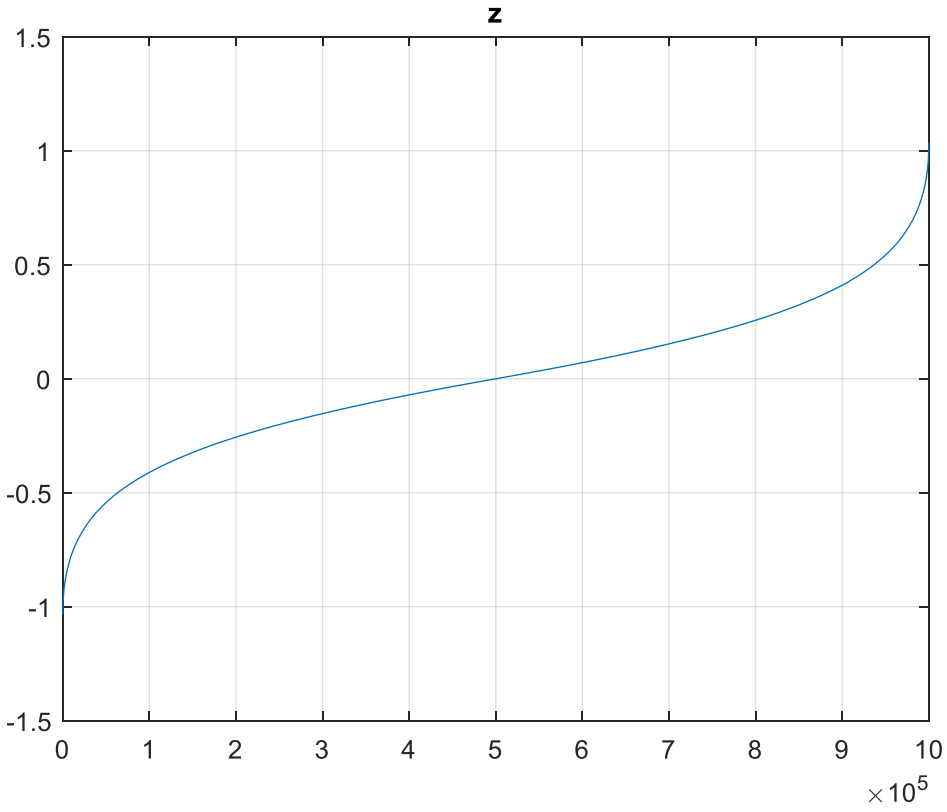

Figure 6.1. The distribution of the cauchyrand() function in Listing 6.1.

I generated the graph in Figure 6.1 using 1 million random numbers with *mean*=0 and *sigma*=1. The mean value of the numbers I obtained was 0.0001 and the standard deviation was 0.3207. The range of the random nmbers was [–1.0336, 1.0356] which is close enough to my target range of [–1, 1].

☛ The range of [–1, 1] is the same target range for empirical distributions (except for the clipped Cauchy distribution) that I use in this study. I aim to make the lower and upper limit close enough to –1 and 1, respectively, based on mean=0 and sigma=1. The SOA function calculations tolerate small variations near these limits, since they check if the calculated population/scout locations are within the trust region. Moreover, a uniform spread of these random numbers is not necessary. The contrary is actually beneficial since it allows the calcultaions to focus on values close to the mean.

Listing 6.2 shows the test program *test_scout3.m* that calculates the optimum variables for function *fx2*.


```
clear
clc
diary 'scout3_fx2.txt'
n = 4;
Iters = 40;
bestFxArr = zeros(Iters,1);
bestXArr = zeros(Iters,n);
fprintf('------------------- Using Semilo Decay of Step factors -------------
---\n');
for i=1:Iters
  fprintf('Iter # %i ------------------------------------------------------
\n', i);
  [bestX, bestFx] = scout3(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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] = scout3(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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 6.2. The source code for test_scout3.m.

Table 6.1 shows the summary output generated by Listing 6.2 and extracted from the output diary file *scout3_fx2.txt*. Table 6.1 shows that the log–log decay of the step factor yields generally better results than its the log–linear counterpart.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	7.888609052210118054e–31	0.000000000000000000e+00
Max bestFx	5.901650799695224197e–15	6.047628330779432365e–16
Mean bestFx	1.715583287202698552e–16	1.661112326866447749e–17
Sdev bestFx	9.327644616993783422e–16	9.549701906409344282e–17
Median bestFx	5.887531178413371384e–20	6.434903275816984488e–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_scout3.m

Listing 6.3 shows the test program *test_scout3b.m* that calculates the optimum variables for function *fx3*.

```
clear
clc
diary 'scout3_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] = scout3(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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] = scout3(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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 6.3. The source code for test_scout3b.m.

Table 6.2 shows the summary output generated by Listing 6.3 and extracted from the output diary file *scout3_fx3.txt*. Table 6.2 shows that the log–log decay of the step factor yields better results than its the log–linear counterpart.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	1.570819277521339758e–27	3.155443620884047222e–30
Max bestFx	3.287532585475314808e–12	1.916343201182821306e–16
Mean bestFx	1.001939813221296812e–13	5.193456574633486015e–18
Sdev bestFx	5.271923995504766835e–13	3.026207670796963614e–17
Median bestFx	4.726110369729839819e–19	6.845535634360838467e–22
Mean Best X1	1.0000	1.0000

Statistic/Result	For Log–linear Decay	For Log–log Decay
Mean Best X2	4.0000	4.0000
Mean Best X3	9.0000	9.0000
Mean Best X4	16.000	16.000

Table 6.2. Summary of results of test file test_scout3b.m

7/ Comparing Versions 1 through 3

Table 7.1 compares the mean best optimized functions while testing function *fx2* using MATLAB functions *scout()* through *scout3()*.

Function	For Log–linear Decay	For Log–log Decay
scout()	1.292036532307468458e–17	2.9103012141917814 35e–20
scout2()	2.506340276268329764e–12	1.402392032195924052e–12
scout3()	1.715583287202698552e–16	1.661112326866447749e–17

Table 7.1. Comparing the mean best optimized functions while testing function fx2 using scout(), scout2(), and scout3().

Table 7.1 shows that all of the optimized functions are in the orders of 10^{-12} through 10^{-20}. The function *scout()* remains the best with using the log–linear and the log–log step decay factors.

Table 7.2 compares the mean best optimized functions while testing function *fx3* using MATLAB functions *scout()*, *scout2()*, and *scout3()*.

Function	For Log–linear Decay	For Log–log Decay
scout()	9.732752177442062738e–16	1.812097031994652036e–17
scout2()	9.254635542193029242e–11	3.460369189744170226e–11
scout3()	1.001939813221296812e–13	5.193456574633486015e–18

Table 7.2. Comparing the mean best optimized functions while testing function fx3 using scout(), scout2(), and scout3().

Table 7.2 shows that all of the optimized functions are in the orders of 10^{-11} through 10^{-18}. The function *scout()* remains the best with using the log–linear step decay factor. The function *scout3()* is the best with using the log–log step decay factor.

8/ The Scout Optimization Algorithm Version 4

The fourh version of SOA uses an empirical *clipped* Cauchy distribution RNG. This might be heretical for the purists, but I decided to be practical and not argue with

results. I experimented with excluding the random function *cauchy(mean, sigma)* RNG values that fall beyond |3**sigma*| values to mimic the 6 sigmas range of Gaussian random numbers. Using the *clipped* Cauchy distribution RNG saves on wasting random numbers that cause the *scoutPop()* elements to fall outside the trust region—resorting to assigning new random values within the trust region.

Listing 8.1 shows the source code for *scout4.m*.

```
function [bestX, bestFx] = scout4(fx, XLow, XHi, InitStep, FinStep,
MaxIters,...
                                  MaxPop, MaxScoutPop, bSemiLogDecay, ...
                                  bSkipNarrowTrusRegion, bTrace)
% The SCOUT function implements a version of the Fireworks
% Algorithm. This version uses both uniform and "clipped" Cauchy 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 non-uniform random perturbations
    for j=1:MaxScoutPop
      for k=m2+1:n
        scoutPop(j,k) = clicppedcauchyrand(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
    pop = [pop; scoutPop];
    pop = sortrows(pop,m);
    pop = pop(1:MaxPop,:);
  end

  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 = clicppedcauchyrand(mean, sigma)

  while 1
    r = rand;
    y = sigma*tan(pi*(r - 0.5)) + mean;
    if abs(y) <= 3*sigma, break; end
  end
end
```

Listing 8.1. The source code for scout4.m.

Listing 8.1 has the following statement used to calculate the *scoutPop()* values using the *clipped* Cauchy RNG:

```
scoutPop(j,k) = cauchyrand(pop(i,k), (XHi(k) - XLow(k))/3 * StepFactor);
```

The local function *clippedcauchyrand()* is defined as:


```
function y = clicppedcauchyrand(mean, sigma)

  while 1
    r = rand;
    y = sigma*tan(pi*(r - 0.5)) + mean;
    if abs(y) <= 3*sigma, break; end
  end
end
```

As you can see from the above definition the open while loop iterates until the absolute value of the generated random number is less than *3*sigma*. Figure 8.1 shows distribution of a sample one million random numbers, with *mean*=0 and *sigma*=1, generated with function *clippedcauchyrand()*.

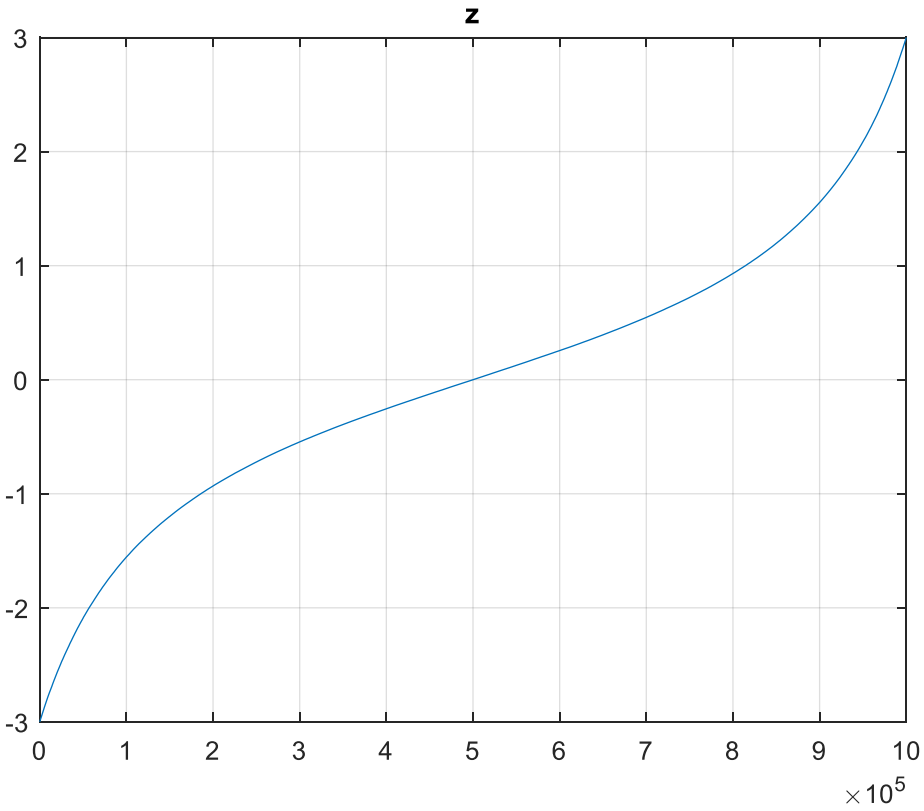

Figure 8.1. The distribution of a sample one million random numbers generated with function clippedcauchyrand().

The range of the sample random numbers is [–3, 3] by design. The mean and standard devation of these random numbers are –0.0004 amd 1.1846, respectively. I delibertatly chose the range [–3, 3] to mimic the six sigmas of the Gaussian distribution. If you wish to change that range to [–1, 1], then replace the tested Boolean expression *abs(y) <= 3*sigma* with *abs(y) <= sigma*.

Listing 8.2 shows the test program *test_scout4.m* that calculates the optimum variables for function *fx2*.

```
clear
clc
diary 'scout4_fx2.txt'
n = 4;
Iters = 40;
bestFxArr = zeros(Iters,1);
bestXArr = zeros(Iters,n);
fprintf('------------------- Using Semilo Decay of Step factors ------------
---\n');
for i=1:Iters
  fprintf('Iter # %i ------------------------------------------------------
\n', i);
  [bestX, bestFx] = scout4(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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] = scout4(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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 8.2. The source code for test_scout4.m.

Table 8.1 shows the summary output generated by Listing 8.2 and extracted from the output diary file *scout4_fx2.txt*. Table 8.1 shows that the log–log decay of the step factor yields better results than its the log–linear counterpart.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	7.099748146989106249e–30	0.000000000000000000e+00
Max bestFx	3.494158800418274944e–12	7.764909452665678826e–18
Mean bestFx	9.076877635186815712e–14	4.710025606657630394e–19
Sdev bestFx	5.523255639613454079e–13	1.507446735588874113e–18
Median bestFx	4.055851128343713579e–21	7.957885820969734469e–23
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 8.1. Summary of results of test file test_scout4.m

Listing 8.3 shows the test program *test_scout4b.m* that calculates the optimum variables for function *fx3*.

```
clear
clc
diary 'scout4_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] = scout4(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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] = scout4(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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 8.3. The source code for test_scout4b.m.

Table 8.2 shows the summary output generated by Listing 8.3 and extracted from the output diary file *scout4_fx3.txt*. Table 8.2 shows that the log–log decay of the step factor yields better results than its the log–linear counterpart.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	1.611248398913916613e–28	0.000000000000000000e+00
Max bestFx	8.413698155776195985e–14	1.087557245756033138e–14
Mean bestFx	7.240256905116018957e–15	3.035806764626491629e–16

Statistic/Result	For Log–linear Decay	For Log–log Decay
Sdev bestFx	2.13648603372471678e–14	1.721951469512161550e–15
Median bestFx	9.983669058850652093e–19	4.3080912692330338 57e–22
Mean Best X1	1.000000	1.000000
Mean Best X2	4.000000	4.000000
Mean Best X3	9.000000	9.000000
Mean Best X4	16.00000	16.00000

Table 8.2. Summary of results of test file test_scout4b.m

9/ Comparing Versions 1 through 4

Table 9.1 compares the mean best optimized functions while testing function *fx2* using MATLAB functions *scout()* through *scout4()*.

Function	For Log–linear Decay	For Log–log Decay
scout()	1.292036532307468458e–17	2.910301214191781435e–20
scout2()	2.506340276268329764e–12	1.402392032195924052e–12
scout3()	1.715583287202698552e–16	1.661112326866447749e–17
scout4()	9.076877635186815712e–14	4.710025606657630394e–19

Table 9.1. Comparing the mean best optimized functions while testing function fx2 using scout() through scout4().

Table 9.1 shows that all of the optimized functions are in the orders of 10^{-12} through 10^{-20}. The function *scout()* remains the best with using the log–linear and log–log step decay factors.

Table 9.2 compares the mean best optimized functions while testing function *fx3* using MATLAB functions *scout()* through *scout4()*.

Function	For Log–linear Decay	For Log–log Decay
scout()	9.732752177442062738e–16	1.812097031994652036e–17
scout2()	9.254635542193029242e–11	3.460369189744170226e–11
scout3()	1.001939813221296812e–13	5.193456574633486015e–18
scout4()	7.240256905116018957e–15	3.035806764626491629e–16

Table 9.2. Comparing the mean best optimized functions while testing function fx3 using scout() through scout4().

Table 9.2 shows that all of the optimized functions are in the orders of 10^{-11} and 10^{-18}. The function *–scout() remains* the best with using the log–linear step decay

factor. The function *scout3()* remains the best with using the log–log step decay factor.

10/ The Scout Optimization Algorithm Version 5

The fifth version of SOA uses an empirical cosine–based RNG. The cosine function is sometimes used to mimic the Gaussian bell distribution. However, since the cosine distribution require values are between 1 and 0 included, I needed to flatten the tails of the distribution when it reaches values close to zero.

The pseudo code for the empirical cosine–based RNG, *cosrand(mean, sigma)* is:

- Let r = U(0,1)
- x = (r – mean)/sigma
- y = exp(–x^2)*cos(x)
- y = mean + sigma * y
- y = (y – 0.2)/0.8;
- return 2*y – 1;

Figure 10.1 shows the cosine distribution generated by one million sample numbers with *mean*=0 and *sigma*=1.

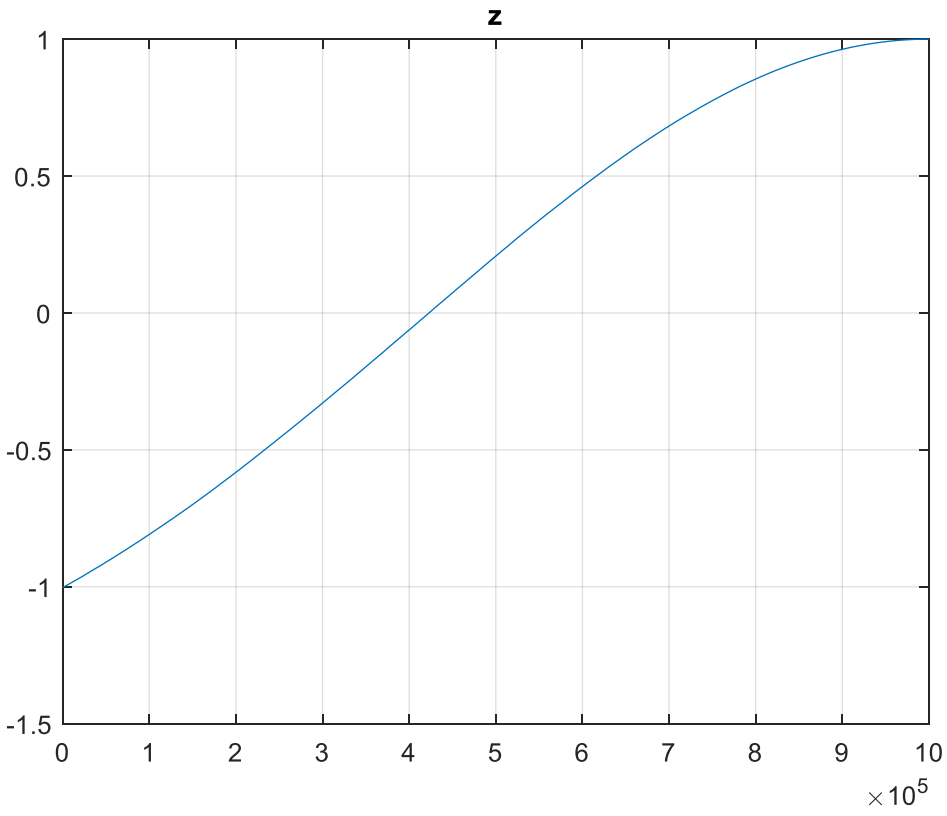

Figure 10.1. The cosine distribution.

Figure 10.1 is based on a sample of one million random numbers with a range of [–1.0031, 1.0000], a mean of 0.1400, and standard deviation of 0.6527. The range of the numbers generated is close enough to the target range of [–1, 1].

Listing 10.1 shows the source code for *scout5.m*.

```
function [bestX, bestFx] = scout5(fx, XLow, XHi, InitStep, FinStep,
MaxIters,...
                                  MaxPop, MaxScoutPop, bSemiLogDecay, ...
                                  bSkipNarrowTrusRegion, bTrace)
% The SCOUT function implements a version of the Fireworks
% Algorithm. This version uses both uniform and cosine 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) = cosrand(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
    pop = [pop; scoutPop];
    pop = sortrows(pop,m);
    pop = pop(1:MaxPop,:);
  end

  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 = cosrand(mean,sigma)

  r = rand;
  x = (r - mean)/sigma;
  y = exp(-x^2)*cos(x);
  y = mean + sigma * y;

  y = (y - 0.2)/0.8;
  y = 2*y - 1;
end
```


Listing 10.1. The source code for scout5.m.

Listing 10.1 has the following statement used to calculate the *scoutPop()* values using the cosine RNG:

```
scoutPop(j,k) = cosrand(pop(i,k), (XHi(k) – XLow(k))/3 * StepFactor);
```

The local function *cosrand()* is defined as:

```
function y = cosrand(mean,sigma)

  r = rand;
  x = (r – mean)/sigma;
  y = exp(–x^2)*cos(x);
  y = mean + sigma * y;


  y = (y – 0.2)/0.8;
  y = 2*y – 1;
end
```

Listing 10.2 shows the test program *test_scout5.m* that calculates the optimum variables for function *fx2*.

```
clear
clc
diary 'scout5_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] = scout5(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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] = scout5(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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 10.2. The source code for test_scout5.m.

Table 10.1 shows the summary output generated by Listing 10.2 and extracted from the output diary file *scout5_fx2.txt*. Table 10.1 shows that the log–log decay of the step factor yields better results than its the log–linear counterpart.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	6.705317694378600346e–30	0.000000000000000000e+00
Max bestFx	2.289318104104485336e–15	5.405974236839752204e–17
Mean bestFx	1.546991795471606629e–16	1.356101162840001786e–18
Sdev bestFx	4.633102785913316512e–16	8.546858461248261695e–18
Median bestFx	1.535390867643421351e–20	1.747431256571159916e–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 10.1. Summary of results of test file test_scout5.m

Listing 10.3 shows the test program *test_scout5b.m* that calculates the optimum variables for function *fx3*.

```
clear
clc
diary 'scout5_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] = scout5(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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] = scout5(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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 10.3. The source code for test_scout5b.m.

Table 10.2 shows the summary output generated by Listing 10.3 and extracted from the output diary file *scout5_fx3.txt*. Table 10.2 shows that the log–log decay of the step factor yields better results than its the log–linear counterpart.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	3.167769572528125531e–30	1.972152263052529514e–31
Max bestFx	3.197325334249145663e–12	2.970788304534754106e–16
Mean bestFx	1.599620770520585648e–13	2.040054922133027985e–17
Sdev bestFx	7.018237258202401128e–13	6.736772760573203844e–17
Median bestFx	8.687173610183244946e–19	5.683920042234242094e–21
Mean Best X1	1.000000	1.000000
Mean Best X2	4.000000	4.000000
Mean Best X3	9.000000	9.000000
Mean Best X4	16.000000	16.000000

Table 10.2. Summary of results of test file test_scout5b.m

11/ Comparing Versions 1 through 5

Table 11.1 compares the mean best optimized functions while testing function *fx2* using MATLAB functions *scout()* through *scout5()*.

Function	For Log–linear Decay	For Log–log Decay
scout()	1.292036532307468458e–17	2.910301214191781435e–20
scout2()	2.506340276268329764e–12	1.402392032195924052e–12
scout3()	1.715583287202698552e–16	1.661112326866447749e–17
scout4()	9.076877635186815712e–14	4.710025606657630394e–19
scout5()	1.546991795471606629e–16	1.356101162840001786e–18

Table 11.1. Comparing the mean best optimized functions while testing function fx2 using scout() through scout5().

Table 11.1 shows that all of the optimized functions are in the orders of 10^{-12} through 10^{-20}. The function *scout()* remains the best with using the log–linear and log–log step decay factors.

Table 11.2 compares the mean best optimized functions while testing function *fx3* using MATLAB functions *scout()* through *scout5()*.

Function	For Log–linear Decay	For Log–log Decay
scout()	9.732752177442062738e–16	1.812097031994652036e–17
scout2()	9.254635542193029242e–11	3.460369189744170226e–11
scout3()	1.001939813221296812e–13	5.193456574633486015e–18
scout4()	7.240256905116018957e–15	3.035806764626491629e–16
scout5()	1.599620770520585648e–13	2.040054922133027985e–17

Table 11.2. Comparing the mean best optimized functions while testing function fx3 using scout() through scout5().

Table 11.2 shows that all of the optimized functions are in the orders of 10^{-11} and 10^{-18}. The function *scout()* remains the best with using the log–linear step decay factor. The function *scout3()* remains the best with using the log–log step decay factor.

12/ The Scout Optimization Algorithm Version 6

The sixth version of SOA uses a second empirical cosine–based RNG. The pseudo code for the this second cosine–based RNG, *cosrand2(mean, sigma)* is:

- Let r = U(0,1)
- x = (r – mean)/sigma
- y = exp(–x^2)*cos(x);
- chs = sign(y)
- chs2 = chs
- if chs is 0 then chs2 = 1
- y = mean + chs * sigma / (chs2 + abs(log10(abs(y))
- y = (y – 0.5877)/(1 – 0.5887)
- return 2*y – 1

Figure 12.1 shows the plot of a sample million random number generated using the second cosine RNG. The plot is based on *mean*=0 and *sigma*=1.

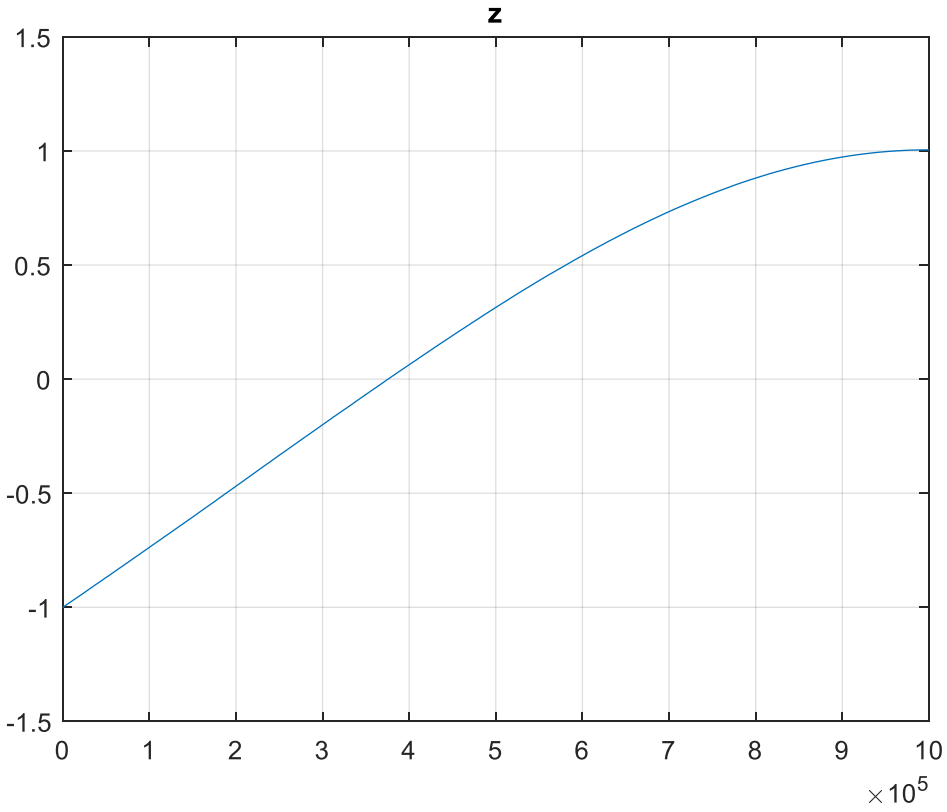

Figure 12.1. The plot of a sample million random number generated using the second cosine RNG.

The sample number generated have a mean and standard deviation of 0.2125 and 0.6303, respectively. The range of the random numbers generated is [–1.0002, 1.0049] which is close enough to the targeted range of [–1, 1].

Listing 12.1 shows the source code for *scout6.m*.

```
function [bestX, bestFx] = scout6(fx, XLow, XHi, InitStep, FinStep,
MaxIters,...
                                  MaxPop, MaxScoutPop, bSemiLogDecay, ...
                                  bSkipNarrowTrusRegion, bTrace)
% The SCOUT function implements a version of the Fireworks
% Algorithm. This version uses both uniform and a second cosine 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, whenever 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 non-uniform random perturbations
    for j=m2+1:MaxScoutPop
      for k=1:n
        scoutPop(j,k) = cosrand2(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
    pop = [pop; scoutPop];
    pop = sortrows(pop,m);
    pop = pop(1:MaxPop,:);
  end

  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 = cosrand2(mean,sigma)

  r = rand;
  x = (r - mean)/sigma;
  y = exp(-x^2)*cos(x);
  chs = sign(y);
  chs2 = chs;
  if chs==0, chs2 = 1; end
  y = mean + chs * sigma / (chs2 + abs(log10(abs(y))));
  y = (y - 0.5877)/(1 - 0.5887);
  y = 2*y - 1;
end
```

Listing 12.1. The source code for scout6.m.

Listing 12.1 has the following statement used to calculate the *scoutPop()* values using the *second cosine* RNG:

```
scoutPop(j,k) = cosrand2(pop(i,k), (XHi(k) - XLow(k))/3 * StepFactor);
```

The local function *cosrand2()* is defined as:

```
function y = cosrand2(mean, sigma)

  r = rand;
  x = (r - mean)/sigma;
  y = exp(-x^2)*cos(x);
  chs = sign(y);
  chs2 = chs;
  if chs==0, chs2 = 1; end
  y = mean + chs * sigma / (chs2 + abs(log10(abs(y))));
  y = (y - 0.5877)/(1 - 0.5887);
  y = 2*y - 1;
end
```


Listing 12.2 shows the test program *test_scout6.m* that calculates the optimum variables for function *fx2*.

```
clear
clc
diary 'scout6_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] = scout6(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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] = scout6(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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 12.2. The source code for test_scout6.m.

Table 12.1 shows the summary output generated by Listing 12.2 and extracted from the output diary file *scout6_fx2.txt*. Table 12.1 shows that the log–log decay of the step factor yields better results than its the log–linear counterpart.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	5.620633949699709114e–29	0.000000000000000000e+00
Max bestFx	3.240931044810125004e–14	2.998012438267159670e–18
Mean bestFx	9.048482271323894618e–16	1.888861594373887854e–19
Sdev bestFx	5.120366330965171994e–15	5.680448658474970445e–19
Median bestFx	1.021979976091847881e–19	1.017168477668730048e–22
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 12.1. Summary of results of test file test_scout6.m

Listing 12.3 shows the test program *test_scout6b.m* that calculates the optimum variables for function *fx3*.

```
clear
clc
diary 'scout6_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] = scout6(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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] = scout6(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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 12.3. The source code for test_scout6b.m.

Table 12.2 shows the summary output generated by Listing 12.3 and extracted from the output diary file *scout6_fx3.txt*. Table 12.2 shows that the log–log decay of the step factor yields better results than its the log–linear counterpart.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	3.155443620884047222e–30	0.000000000000000000e+00
Max bestFx	2.832213348427351861e–13	6.456839548705812054e–16

Statistic/Result	For Log–linear Decay	For Log–log Decay
Mean bestFx	9.426100780722492981e–15	1.822164061425251953e–17
Sdev bestFx	4.508235776587891369e–14	1.023195878276251303e–16
Median bestFx	1.625323791048715099e–19	1.470747717402833343e–21
Mean Best X1	1.000000	1.000000
Mean Best X2	4.000000	4.000000
Mean Best X3	9.000000	9.000000
Mean Best X4	16.000000	16.000000

Table 12.2. Summary of results of test file test_scout6b.m

13/ Comparing Versions 1 through 6

Table 13.1 compares the mean best optimized functions while testing function *fx2* using MATLAB functions *scout()* through *scout6()*.

Function	For Log–linear Decay	For Log–log Decay
scout()	1.2920365323074684 58e–17	2.9103012141917814 35e–20
scout2()	2.5063402762683297 64e–12	1.4023920321959240 52e–12
scout3()	1.7155832872026985 52e–16	1.6611123268664477 49e–17
scout4()	9.0768776351868157 12e–14	4.7100256066576303 94e–19
scout5()	1.5469917954716066 29e–16	1.3561011628400017 86e–18
scout6()	9.0484822713238946 18e–16	1.8888615943738878 54e–19

Table 13.1. Comparing the mean best optimized functions while testing function fx2 using scout() through scout6().

Table 13.1 shows that all of the optimized functions are in the orders of 10^{-12} through 10^{-20}. The function *scout()* remains the best with using the log–linear and log–log step decay factors.

Table 13.2 compares the mean best optimized functions while testing function *fx3* using MATLAB functions *scout()* through *scout6()*.

Function	For Log–linear Decay	For Log–log Decay
scout()	9.732752177442062738e–16	1.812097031994652036e–17
scout2()	9.254635542193029242e–11	3.460369189744170226e–11
scout3()	1.001939813221296812e–13	5.193456574633486015e–18
scout4()	7.240256905116018957e–15	3.035806764626491629e–16
scout5()	1.599620770520585648e–13	2.040054922133027985e–17
scout6()	9.426100780722492981e–15	1.822164061425251953e–17

Table 13.2. Comparing the mean best optimized functions while testing function fx3 using scout() through scout6().

Table 13.2 shows that all of the optimized functions are in the orders of 10^{-11} through $10^{-18.}$ The function *scout() remains* the best with using the log–linear step decay factor. The function *scout3()* remains the best with using the log–log step decay factor.

14/ The Scout Optimization Algorithm Version 7

The seventh version of SOA uses an empirical sigmoid–integral RNG. The pseudo code for this method is:

- Let r = U(0,1)
- x = (r – mean)/sigma
- y = exp(–x)/(1 + exp(–x))^2
- y = mean + sigma * y
- y = (y – 0.1966) / (0.25 – 0.1966)
- return 2*y – 1

Figure 14.1 shows the distribution of a sample one million random numbers generated using the above RNG. The numbers are generated using *mean*=0 and *sigma*=1.

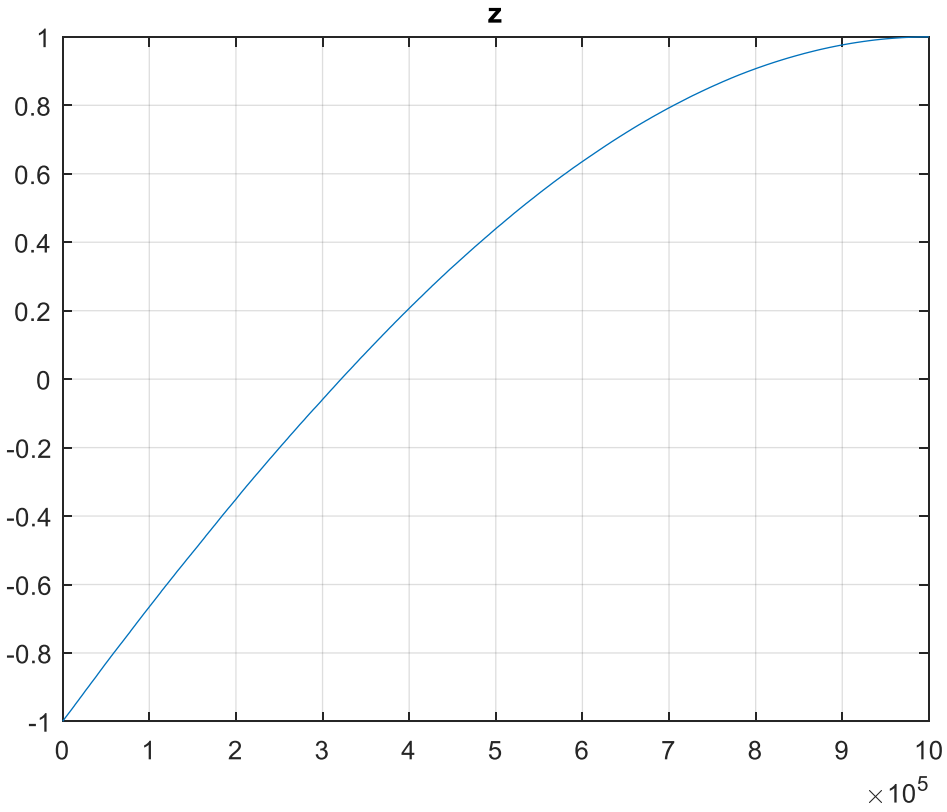

Figure 14.1. The distribution of a sample one million random numbers generated using the empirical sigmoid–integral RNG.

The random numbers generated in Figure 14.1 have a mean and standard deviation of 0.2912 and 0.6090, respevtively. The range of these numbers is [–0.9995, 1.0000] which is close enough to the target range of [–1, 1].

Listing 14.1 shows the source code for *scout7.m*.

```
function [bestX, bestFx] = scout7(fx, XLow, XHi, InitStep, FinStep,
MaxIters,...
                                  MaxPop, MaxScoutPop, bSemiLogDecay, ...
                                  bSkipNarrowTrusRegion, bTrace)
% The SCOUT function implements a version of the Fireworks
% Algorithm. This version uses both uniform and exponential 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 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);
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 non-uniform random perturbations
    for j=m2+1:MaxScoutPop
      for k=1:n
        scoutPop(j,k) = exporand(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
    pop = [pop; scoutPop];
    pop = sortrows(pop,m);
    pop = pop(1:MaxPop,:);
  end

  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 = exporand(mean,sigma)

  r = rand;
  x = (r - mean)/sigma;
  y = mean + sigma * exp(-x)/(1+exp(-x))^2;

  y = (y - .1966) / (0.25 - 0.1966);
  y  = 2*y - 1;
end
```

Listing 14.1. The source code for scout7.m.

Listing 14.1 has the following statement used to calculate the *scoutPop()* values using the exponential RNG:

```
scoutPop(j,k) = exporand(pop(i,k), (XHi(k) - XLow(k))/3 * StepFactor);
```

The local function *exporand()* is defined as:

```
function y = exporand (mean,sigma)

  r = rand;
  x = (r - mean)/sigma;
  y = mean + sigma * exp(-x)/(1+exp(-x))^2;
```



```
  y = (y - .1966) / (0.25 - 0.1966);
  y  = 2*y - 1;
end
```

Listing 14.2 shows the test program *test_scout7.m* that calculates the optimum variables for function *fx2*.

```
clear
clc
diary 'scout7_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] = scout7(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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] = scout7(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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 14.2. The source code for test_scout7.m.

Table 14.1 shows the summary output generated by Listing 14.2 and extracted from the output diary file *scout7_fx2.txt*. Table 14.1 shows that the log–log decay of the step factor yields better results than its the log–linear counterpart.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	3.691869036434335249e–27	0.000000000000000000e+00
Max bestFx	4.133145138164234497e–15	2.610015427440711179e–17
Mean bestFx	2.458827805922748509e–16	7.913666601427630172e–19
Sdev bestFx	8.391562278218991309e–16	4.159227534895520837e–18
Median bestFx	7.424555335051904823e–21	1.256272590284958378e–23
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 14.1. Summary of results of test file test_scout7.m

Listing 14.3 shows the test program *test_scout7b.m* that calculates the optimum variables for function *fx3*.

```
clear
clc
diary 'scout7_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] = scout7(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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] = scout7(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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 14.3. The source code for test_scout7b.m.

Table 14.2 shows the summary output generated by Listing 14.3 and extracted from the output diary file *scout7_fx3.txt*. Table 14.2 shows that the log–log decay of the step factor yields better results than its the log–linear counterpart.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	0.00000000000000000e+00	0.00000000000000000e+00

Statistic/Result	For Log–linear Decay	For Log–log Decay
Max bestFx	5.114756514775285699e–14	9.403397464824959042e–16
Mean bestFx	2.287522184356782288e–15	4.620632725382591975e–17
Sdev bestFx	8.634209973990821623e–15	1.963136271196178700e–16
Median bestFx	3.548266037527064923e–20	1.388680121939456240e–21
Mean Best X1	1.000000	1.000000
Mean Best X2	4.000000	4.000000
Mean Best X3	9.000000	9.000000
Mean Best X4	16.000000	16.000000

Table 14.2. Summary of results of test file test_scout7b.m

15/ Comparing Versions 1 through 7

Table 15.1 compares the mean best optimized functions while testing function *fx2* using MATLAB functions *scout()* through *scout7()*.

Function	For Log–linear Decay	For Log–log Decay
scout()	1.292036532307468458e–17	2.9103012141917814 35e–20
scout2()	2.506340276268329764e–12	1.402392032195924052e–12
scout3()	1.715583287202698552e–16	1.661112326866447749e–17
scout4()	9.076877635186815712e–14	4.710025606657630394e–19
scout5()	1.546991795471606629e–16	1.356101162840001786e–18
scout6()	9.048482271323894618e–16	1.888861594373887854e–19
scout7()	2.458827805922748509e–16	7.913666601427630172e–19

Table 15.1. Comparing the mean best optimized functions while testing function fx2 using scout() through scout7().

Table 15.1 shows that all of the optimized functions are in the orders of 10^{-12} and 10^{-20}. The function *scout()* remains the best with using the log–linear and log–log step decay factors.

Table 15.2 compares the mean best optimized functions while testing function *fx3* using MATLAB functions *scout()* through *scout7()*.

Function	For Log–linear Decay	For Log–log Decay
scout()	9.732752177442062738e–16	1.812097031994652036e–17
scout2()	9.254635542193029242e–11	3.460369189744170226e–11
scout3()	1.001939813221296812e–13	5.193456574633486015e–18
scout4()	7.240256905116018957e–15	3.035806764626491629e–16
scout5()	1.599620770520585648e–13	2.040054922133027985e–17
scout6()	9.426100780722492981e–15	1.822164061425251953e–17

Function	For Log–linear Decay	For Log–log Decay
scout7()	2.287522184356782288e–15	4.620632725382591975e–17

Table 15.2. Comparing the mean best optimized functions while testing function fx3 using scout() through scout7().

Table 15.2 shows that all of the optimized functions are in the orders of 10^{-11} through $10^{-18.}$ The function *scout()* remains the best with using the log–linear step decay factor. The function *scout3()* remains the best with using the log–log step decay factor.

16/ The Scout Optimization Algorithm Version 8

The eight version of SOA uses a Logistic distribution RNG. It uses the CDF function of the PDF function (which is the derivative of the sigmoid function):

PDF = $\frac{e^{-x}}{(1+ e^{-x})^2}$

CDF = $\int_{-\infty}^{r} \frac{e^{-x}}{(1+ e^{-x})^2} dx$ = ln(r / (1–r)) where r = U(0,1)

The pseudo code for function *logirand(mean, sigma)* is:

- Let r = U(0,1)
- x = ln(r / (1 – r))
- y = mean + sigma * x
- y = (y + 14) / 28
- return 2*y – 1

Figure 16.1 shows the plot of a million ranumber numbers generated with the *logirand()* function. The plot is generated using *mean*=0 and *sigma*=1.

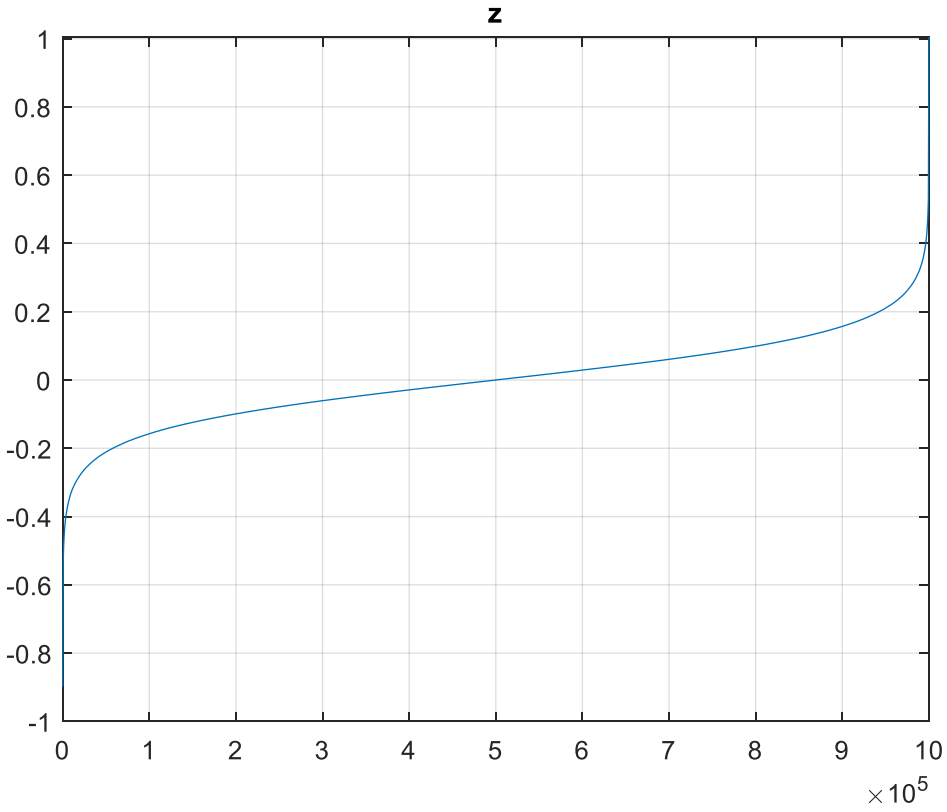

Figure 16.1. The plot of a million ranumber numbers generated with the logirand() function.

The random numbers generated in Figure 16.1 have the mean and standard deviation of –0.0001 and 0.1297, respectively. The range of these numbers is [–0.8977, 1.0050] which is close enough to the target range of [–1, 1].

Listing 16.1 shows the source code for *scout8.m*.

```
function [bestX, bestFx] = scout8(fx, XLow, XHi, InitStep, FinStep,
MaxIters,...
                                  MaxPop, MaxScoutPop, bSemiLogDecay, ...
                                  bSkipNarrowTrusRegion, bTrace)
% The SCOUT function implements a version of the Fireworks
% Algorithm. This version uses both uniform and Logistic distribution 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, whenever 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 non-uniform random perturbations
    for j=m2+1:MaxScoutPop
      for k=1:n
        scoutPop(j,k) = logirand(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
    pop = [pop; scoutPop];
    pop = sortrows(pop,m);
    pop = pop(1:MaxPop,:);
  end

  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 = logirand(mean,sigma)

  r = rand;
  x = log(r/(1 - r));
  y = mean + sigma * x;

  y = (y + 14) / 28;
  y = 2*y - 1;
end
```

Listing 16.1. The source code for scout8.m.

Listing 16.1 has the following statement used to calculate the *scoutPop()* values using the *logistic* RNG:

```
scoutPop(j,k) = logirand (pop(i,k), (XHi(k) - XLow(k))/3 * StepFactor);
```

The local function *logirand()* is defined as:

```
function y = logirand(mean, sigma)

  r = rand;
  x = log(r/(1 - r));
  y = mean + sigma * x;
  y = (y + 14) / 28;
  y = 2*y - 1;
end
```

Listing 16.2 shows the test program *test_scout8.m* that calculates the optimum variables for function *fx2*.

```
clear
clc
diary 'scout8_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] = scout8(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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] = scout8(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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 16.2. The source code for test_scout8.m.

Table 16.1 shows the summary output generated by Listing 16.2 and extracted from the output diary file *scout8_fx2.txt*. Table 16.1 shows that the log–log decay of the step factor yields better results than its the log–linear counterpart.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	1.972152263052529514e–31	0.000000000000000000e+00
Max bestFx	6.683188518673173731e–15	1.838713728836212018e–16
Mean bestFx	3.286421020855316597e–16	5.548496299061203423e–18
Sdev bestFx	1.166879588960126134e–15	2.930981108467543618e–17
Median bestFx	4.932728824142635402e–21	6.676674758881655967e–23
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 16.1. Summary of results of test file test_scout8.m

Listing 16.3 shows the test program *test_scout8b.m* that calculates the optimum variables for function *fx3*.

```
clear
clc
diary 'scout8_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] = scout8(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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] = scout8(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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 16.3. The source code for test_scout8b.m.

Table 16.2 shows the summary output generated by Listing 16.3 and extracted from the output diary file *scout8_fx3.txt*. Table 16.2 shows that the log–log decay of the step factor yields better results than its the log–linear counterpart.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	6.226577732522598807e–28	1.232595164407830946e–32
Max bestFx	1.876406816480206009e–13	1.075120433915251889e–15
Mean bestFx	7.357389825363821327e–15	2.955511532168534031e–17
Sdev bestFx	3.155877278105012862e–14	1.698546509315689145e–16
Median bestFx	1.306982351980128199e–19	8.934521482589232009e–22
Mean Best X1	1.000000	1.000000
Mean Best X2	4.000000	4.000000
Mean Best X3	9.000000	9.000000
Mean Best X4	16.000000	16.000000

Table 16.2. Summary of results of test file test_scout8b.m

17/ Comparing Versions 1 through 8

Table 17.1 compares the mean best optimized functions while testing function *fx2* using MATLAB functions *scout()* through *scout8()*.

Function	For Log–linear Decay	For Log–log Decay
scout()	1.292036532307468458e–17	2.910301214191781435e–20
scout2()	2.506340276268329764e–12	1.402392032195924052e–12
scout3()	1.715583287202698552e–16	1.661112326866447749e–17
scout4()	9.076877635186815712e–14	4.710025606657630394e–19
scout5()	1.546991795471606629e–16	1.356101162840001786e–18
scout6()	9.048482271323894618e–16	1.888861594373887854e–19
scout7()	2.458827805922748509e–16	7.913666601427630172e–19
scout8()	3.286421020855316597e–16	5.548496299061203423e–18

Table 17.1. Comparing the mean best optimized functions while testing function fx2 using scout() through scout8().

Table 17.1 shows that all of the optimized functions are in the orders of 10^{-12} through 10^{-20}. The function *scout()* remains the best with using the log–linear and log–log step decay factors.

Table 17.2 compares the mean best optimized functions while testing function *fx3* using MATLAB functions *scout()* through *scout8()*.

Function	For Log–linear Decay	For Log–log Decay
scout()	9.732752177442062738e–16	1.812097031994652036e–17
scout2()	9.254635542193029242e–11	3.460369189744170226e–11
scout3()	1.001939813221296812e–13	5.193456574633486015e–18
scout4()	7.240256905116018957e–15	3.035806764626491629e–16
scout5()	1.599620770520585648e–13	2.040054922133027985e–17
scout6()	9.426100780722492981e–15	1.822164061425251953e–17
scout7()	2.287522184356782288e–15	4.620632725382591975e–17
scout8()	7.357389825363821327e–15	2.955511532168534031e–17

Table 17.2. Comparing the mean best optimized functions while testing function fx3 using scout() through scout8().

Table 17.2 shows that all of the optimized functions are in the orders of 10^{-11} through $10^{-18.}$ The function *scout()* remains the best with using the log–linear step

decay factor. The function *scout3()* remains the best with using the log–log step decay factor.

18/ The Scout Optimization Algorithm Version 9

The ninth version of SOA uses a *modified* Logistic distribution RNG. The pseudo code for *logirand(mean, sigma)* function is:

- Let r1 = U(0,1)
- Let r2 = U(0, 1)
- if r1 < r2
- x = ln(r1/(1 – r2));
- else
- x = ln(r2/(1 – r1));
- end
- y = mean + sigma * x
- y = (y + 14) / 28
- return 2*y – 1

Figure 18.1 shows a sample one million randon numbers generated using the *logirand()* function. The numbers are generated using *mean*=0 and *sigma*=1.

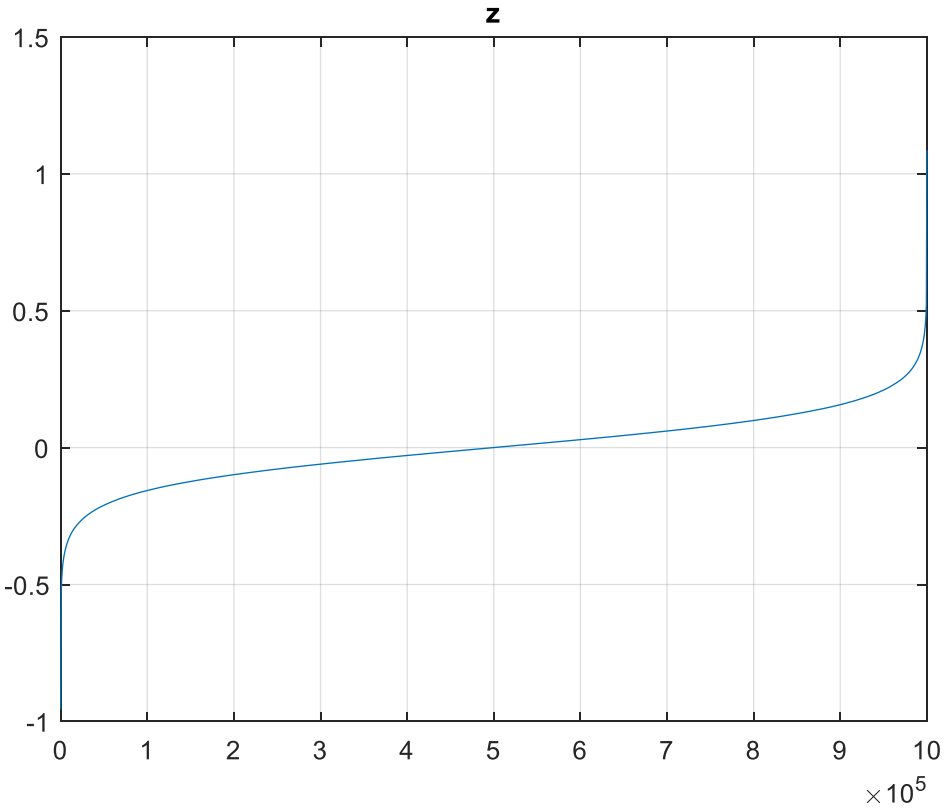

Figure 18.1. A sample one million randon numbers generated using the logirand() function.

The numbers generated in Figure 18.1 have a mean and standard deviation of 0.0001 and 0.1254, respectively. The range of these numbers is [–0.9557, 1.0863] which is close enough to the target range of [–1, 1]. Notice that in Figure 18.1 there is a very steep variation between 0.5 and 1 in the upper range of the random numbers. Likewise, there is a steep variation between –0.5 and –1 in the lower range of random numbers. Most of the random numbers seem to lie in the range [–0.5, 0.5].

Listing 18.1 shows the source code for *scout9.m*.

```
function [bestX, bestFx] = scout9(fx, XLow, XHi, InitStep, FinStep,
MaxIters,...
                                  MaxPop, MaxScoutPop, bSemiLogDecay, ...
                                  bSkipNarrowTrusRegion, bTrace)
% The SCOUT function implements a version of the Fireworks
% Algorithm. This version uses both uniform and modified Logistic
distribution 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 non-uniform random perturbations
    for j=m2+1:MaxScoutPop
      for k=1:n
        scoutPop(j,k) = logirand(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
    pop = [pop; scoutPop];
    pop = sortrows(pop,m);
    pop = pop(1:MaxPop,:);
  end

  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 = logirand(mean,sigma)

  r1 = rand;
  r2 = rand;
  if r1 < r2
    x = log(r1/(1 - r2));
  else
    x = log(r2/(1 - r1));
  end
  y = mean + sigma * x;

  y = (y + 14) / 28;
  y = 2*y - 1;
end
```

Listing 18.1. The source code for scout9.m.

Listing 18.1 has the following statement used to calculate the *scoutPop()* values using the *logistic* RNG:

```
scoutPop(j,k) = logirand (pop(i,k), (XHi(k) - XLow(k))/3 * StepFactor);
```

The local function *logirand()* is defined as:

```
function y = logirand(mean,sigma)

  r1 = rand;
  r2 = rand;
  if r1 < r2
    x = log(r1/(1 - r2));
  else
    x = log(r2/(1 - r1));
  end
  y = mean + sigma * x;
  y = (y + 14) / 28;
  y = 2*y - 1;
end
```


Listing 18.2 shows the test program *test_scout9.m* that calculates the optimum variables for function *fx2*.

```
clear
clc
diary 'scout9_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] = scout9(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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] = scout9(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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 18.2. The source code for test_scout9.m.

Table 18.1 shows the summary output generated by Listing 18.2 and extracted from the output diary file *scout9_fx2.txt*. Table 18.1 shows that the log–log decay of the step factor yields better results than its the log–linear counterpart.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	0.00000000000000000e+00	0.00000000000000000e+00
Max bestFx	9.97223537091588698e–14	4.43706972193389703e–16
Mean bestFx	4.67438261932125417e–15	1.17523322046545868e–17
Sdev bestFx	1.98979065891060144e–14	7.00749090081657150e–17
Median bestFx	1.02648551573239173e–20	2.36581278741953970e–23
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 18.1. Summary of results of test file test_scout9.m

Listing 18.3 shows the test program *test_scout9b.m* that calculates the optimum variables for function *fx3*.

```
clear
clc
diary 'scout9_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] = scout9(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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] = scout9(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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 18.3. The source code for test_scout9b.m.

Table 18.2 shows the summary output generated by Listing 18.3 and extracted from the output diary file *scout9_fx3.txt*. Table 18.2 shows that the log–log decay of the step factor yields better results than its the log–linear counterpart.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	0.00000000000000000e+00	2.524354896707237777e–29
Max bestFx	2.527376462721873790e–13	2.039785325129382372e–16
Mean bestFx	1.788218497877370596e–14	8.537616829009803630e–18

Statistic/Result	For Log–linear Decay	For Log–log Decay
Sdev bestFx	5.051809452740583685e–14	3.679578328530402652e–17
Median bestFx	7.004539001377047040e–19	4.836268760065842493e–22
Mean Best X1	1.000000	1.000000
Mean Best X2	4.000000	4.000000
Mean Best X3	9.000000	9.000000
Mean Best X4	16.000000	16.000000

Table 18.2. Summary of results of test file test_scout9b.m

19/ Comparing Versions 1 through 9

Table 19.1 compares the mean best optimized functions while testing function *fx2* using MATLAB functions *scout()* through *scout9()*.

Function	For Log–linear Decay	For Log–log Decay
scout()	1.2920365323074684 58e–17	2.9103012141917814 35e–20
scout2()	2.5063402762683297 64e–12	1.4023920321959240 52e–12
scout3()	1.7155832872026985 52e–16	1.6611123268664477 49e–17
scout4()	9.0768776351868157 12e–14	4.7100256066576303 94e–19
scout5()	1.5469917954716066 29e–16	1.3561011628400017 86e–18
scout6()	9.0484822713238946 18e–16	1.8888615943738878 54e–19
scout7()	2.4588278059227485 09e–16	7.9136666014276301 72e–19
scout8()	3.2864210208553165 97e–16	5.5484962990612034 23e–18
scout9()	4.6743826193212541 79e–15	1.1752332204654586 80e–17

Table 19.1. Comparing the mean best optimized functions while testing function fx2 using scout() through scout9().

Table 19.1 shows that all of the optimized functions are in the orders of 10^{-12} through 10^{-20}. The function *scout()* remains the best with using the log–linear and log–log step decay factors.

Table 19.2 compares the mean best optimized functions while testing function *fx3* using MATLAB functions *scout()* through *scout9()*.

Function	For Log–linear Decay	For Log–log Decay
scout()	9.732752177442062738e–16	1.812097031994652036e–17
scout2()	9.254635542193029242e–11	3.460369189744170226e–11
scout3()	1.001939813221296812e–13	5.193456574633486015e–18
scout4()	7.240256905116018957e–15	3.035806764626491629e–16
scout5()	1.599620770520585648e–13	2.040054922133027985e–17

Function	For Log–linear Decay	For Log–log Decay
scout6()	9.426100780722492981e–15	1.8221640614252519 53e–17
scout7()	2.287522184356782288e–15	4.620632725382591975e–17
scout8()	7.357389825363821327e–15	2.9555115321685340 31e–17
scout9()	1.788218497877370596e–14	8.537616829009803630e–18

Table 19.2. Comparing the mean best optimized functions while testing function fx3 using scout() through scout9().

Table 19.2 shows that all of the optimized functions are in the orders of 10^{-11} and $10^{-18.}$ The function *scout()* remains the best with using the log–linear step decay factor. The function *scout3()* remains the best with using the log–log step decay factor.

20/ The Scout Optimization Algorithm Version 10

The tenth version of SOA uses an empirical exponential ratio RNG. The pseudo code for function *exporatiorand(mean, sigma, minR, maxR)* is:

- Let r1 = U(0,1)
- Let r2 = U(0, 1)
- x = exp(r1)/exp(r2)
- x = (x – minR)/(maxR – minR)
- y = mean + sigma * x
- y = (y + 0.3674) / (0.9964 + 0.3674);
- return 2*y – 1;

Where *minR* is equal to *exp(0)* and *maxR* is equal to *exp(1)*. These two parameters are calculated once in the main function *scout10()* and passed to the RNG function to speed up calculations. Figure 20.1 shows the distribution of a sample one million random numbers generated using the exponential ratio RNG.

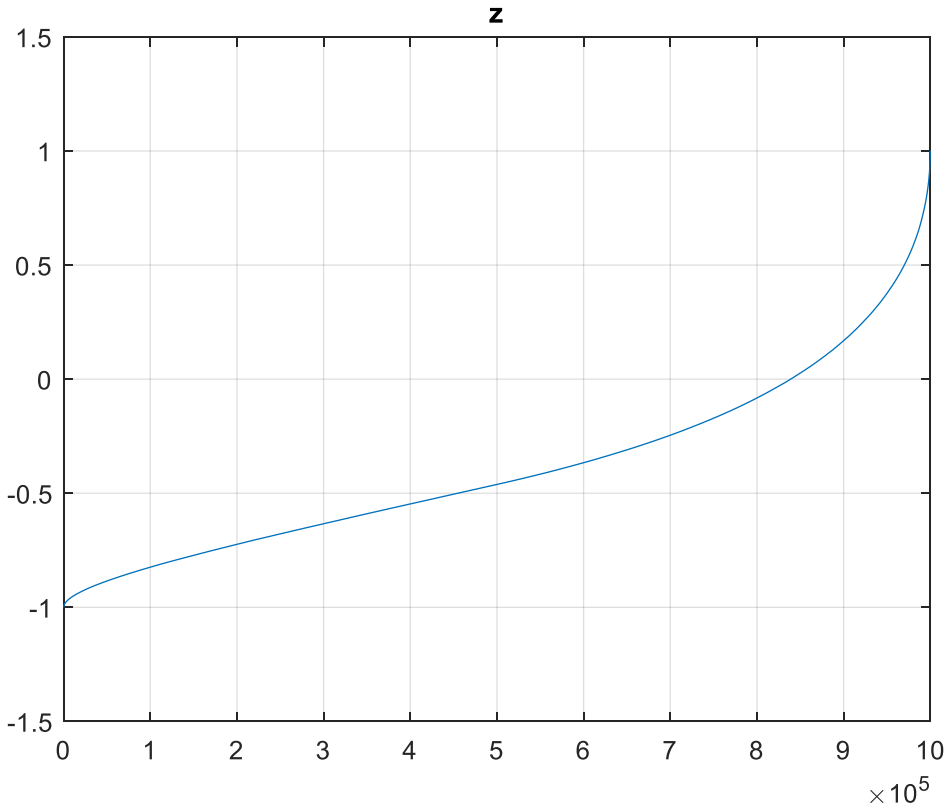

Figure 20.1. Tthe distribution of a sample one million random numbers generated using the exponential ratio RNG.

The mean and standard deviation of the random numbers in Figure 20.1 are –0.3882 amd 0.13828, respectively. The range of these numbers is [–1.0004, 1.0025] which is close enough to the target range of [–1, 1].

Listing 20.1 shows the source code for *scout10.m*.

```
function [bestX, bestFx] = scout10(fx, XLow, XHi, InitStep, FinStep,
MaxIters,...
                                   MaxPop, MaxScoutPop, bSemiLogDecay, ...
                                   bSkipNarrowTrusRegion, bTrace)
% The SCOUT function implements a version of the Fireworks
% Algorithm. This version uses both uniform and exponential ratio
distribution 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;
minR = exp(0);
maxR = exp(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 non-uniform random perturbations
    for j=m2+1:MaxScoutPop
      for k=1:n
        scoutPop(j,k) = exporatiorand(pop(i,k), (XHi(k) - XLow(k))/3 *
StepFactor, minR, maxR);
        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
    pop = [pop; scoutPop];
    pop = sortrows(pop,m);
    pop = pop(1:MaxPop,:);
  end

  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 = exporatiorand(mean, sigma, minR, maxR)

  r1 = rand;
  r2 = rand;
  x = exp(r1)/exp(r2);
  x = (x - minR)/(maxR - minR);

  y = mean + sigma * x;
  y = (y + 0.3674) / (0.9964 + 0.3674);
  y = 2*y - 1;
end
```

Listing 20.1. The source code for scout10.m.

Listing 20.1 has the following statement used to calculate the *scoutPop()* values using the empirical exponential ratio RNG:

```
scoutPop(j,k) = exporatiorand (pop(i,k), (XHi(k) - XLow(k))/3 * StepFactor,
minR, maxR);
```

The local function logirand() is defined as:

```
function y = exporatiorand(mean, sigma, minR, maxR)

  r1 = rand;
  r2 = rand;
  x = exp(r1)/exp(r2);
  x = (x - minR)/(maxR - minR);

  y = mean + sigma * x;
  y = (y + 0.3674) / (0.9964 + 0.3674);
  y = 2*y - 1;
end
```


Listing 20.2 shows the test program test_scout10.m that calculates the optimum variables for function *fx2*.

```
clear
clc
diary 'scout10_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] = scout10(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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] = scout10(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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 20.2. The source code for test_scout10.m.

Table 20.1 shows the summary output generated by Listing 20.2 and extracted from the output diary file *scout10_fx2.txt*. Table 20.1 shows that the log–log decay of the step factor yields better results than its the log–linear counterpart.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	4.8835420413838262 08e–28	0.0000000000000000e+00
Max bestFx	9.685170535140815084e–15	4.416156147896434022e–19
Mean bestFx	7.780733984248558112e–16	1.574815625057527184e–20
Sdev bestFx	1.984377796915277477e–15	7.164932499434228928e–20
Median bestFx	1.053669634777385906e–19	1.439466122148698946e–23
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 20.1. Summary of results of test file test_scout10.m

Listing 20.3 shows the test program *test_scout10b.m* that calculates the optimum variables for function *fx3*.

```
clear
clc
diary 'scout10_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] = scout10(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001,
200, 250, 150, 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] = scout10(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001,
200, 250, 150, 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 20.3. The source code for test_scout10b.m.

Table 20.2 shows the summary output generated by Listing 20.3 and extracted from the output diary file *scout10_fx3.txt*. Table 20.2 shows that the log–log decay of the step factor yields better results than its the log–linear counterpart.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	8.381647117973250432e–31	0.000000000000000000e+00
Max bestFx	2.332119488040334904e–13	3.991913782996626126e–15

Statistic/Result	For Log–linear Decay	For Log–log Decay
Mean bestFx	1.756929092686453132e–14	1.061800201579965287e–16
Sdev bestFx	4.868830711874942891e–14	6.308469919522621561e–16
Median bestFx	1.929623021049735268e–18	1.831797807023672282e–21
Mean Best X1	1.000000	1.000000
Mean Best X2	4.000000	4.000000
Mean Best X3	9.000000	9.000000
Mean Best X4	16.000000	16.000000

Table 20.2. Summary of results of test file test_scout10b.m

21/ Comparing Versions 1 through 10

Table 21.1 compares the mean best optimized functions while testing function *fx2* using MATLAB functions *scout()* through *scout10()*.

Function	For Log–linear Decay	For Log–log Decay
scout()	1.292036532307468458e–17	2.910301214191781435e–20
scout2()	2.506340276268329764e–12	1.402392032195924052e–12
scout3()	1.715583287202698552e–16	1.661112326866447749e–17
scout4()	9.076877635186815712e–14	4.710025606657630394e–19
scout5()	1.546991795471606629e–16	1.356101162840001786e–18
scout6()	9.048482271323894618e–16	1.888861594373887854e–19
scout7()	2.458827805922748509e–16	7.913666601427630172e–19
scout8()	3.286421020855316597e–16	5.548496299061203423e–18
scout9()	4.674382619321254179e–15	1.175233220465458680e–17
scout10()	7.780733984248558112e–16	1.574815625057527184e–20

Table 21.1. Comparing the mean best optimized functions while testing function fx2 using scout() through scout10().

Table 21.1 shows that all of the optimized functions are in the orders of 10^{-12} through 10^{-20}. The function *scout10()* has grabbed the top spot as the best function using the log–log step decay factor. The function *scout()* remains the best for the log–linear step decay factor.

Table 21.2 compares the mean best optimized functions while testing function *fx3* using MATLAB functions scout() through scout10().

Function	For Log–linear Decay	For Log–log Decay
scout()	9.732752177442062738e–16	1.812097031994652036e–17
scout2()	9.254635542193029242e–11	3.460369189744170226e–11

Function	For Log–linear Decay	For Log–log Decay
scout3()	1.001939813221296812e–13	5.193456574633486015e–18
scout4()	7.240256905116018957e–15	3.035806764626491629e–16
scout5()	1.599620770520585648e–13	2.040054922133027985e–17
scout6()	9.426100780722492981e–15	1.822164061425251953e–17
scout7()	2.287522184356782288e–15	4.620632725382591975e–17
scout8()	7.357389825363821327e–15	2.955511532168534031e–17
scout9()	1.788218497877370596e–14	8.537616829009803630e–18
scout10()	1.756929092686453132e–14	1.061800201579965287e–16

Table 21.2. Comparing the mean best optimized functions while testing function fx3 using scout() through scout10().

Table 21.2 shows that all of the optimized functions are in the orders of 10^{-11} through 10^{-18}. The function *scout()* remains the best with using the log–linear step decay factor. The function *scout3()* remains the best with using the log–log step decay factor.

22/ The Scout Optimization Algorithm Version 11

The tenth version of SOA uses a second *modified* Logistic distribution RNG. The pseudo code for function *exporatiorand(mean, sigma, minR, maxR)* is:

- Let r1 = exp(U(0,1))
- Let r2 = exp(U(0, 1))
- x = r1)/ (r1 + r2)
- x = (x – minR)/(maxR – minR)
- y = mean + sigma * x
- y = (y + 0.3674) / (0.9964 + 0.3674)
- return 2*y – 1

Where minR is equal to exp(0) and maxR is equal to exp(1). These two parameters are calculated once in the main function *scout11()* and passed to the RNG function to speed up calculations.

Listing 22.1 shows the source code for *scout11.m*.

```
function [bestX, bestFx] = scout11(fx, XLow, XHi, InitStep, FinStep,
MaxIters,...
                                   MaxPop, MaxScoutPop, bSemiLogDecay, ...
                                   bSkipNarrowTrusRegion, bTrace)
% The SCOUT function implements a version of the Fireworks
% Algorithm. This version uses both uniform and a second empirical
% exponential ratio 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;
e1 = exp(1);
e0 = exp(0);
minR = e0/(e0 + e1);
maxR = e1/(e0 + e1);

%---------------------- 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 non-uniform random perturbations
    for j=m2+1:MaxScoutPop
      for k=1:n
```

```
        scoutPop(j,k) = exporatiorand(pop(i,k), (XHi(k) - XLow(k))/3 *
StepFactor, minR, maxR);
        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
    pop = [pop; scoutPop];
    pop = sortrows(pop,m);
    pop = pop(1:MaxPop,:);
  end

  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 = exporatiorand(mean, sigma, minR, maxR)

  r1 = exp(rand);
  r2 = exp(rand);
  x = r1/(r1+r2);
  x = (x - minR)/(maxR - minR);

  y = mean + sigma * x;
  y = (y + 0.3674) / (0.9964 + 0.3674);
  y = 2*y - 1;
end
```

Listing 22.1. The source code for scout11.m.

Listing 22.1 has the following statement used to calculate the scoutPop() values using the logistic RNG:

```
scoutPop(j,k) = exporatiorand (pop(i,k), (XHi(k) - XLow(k))/3 * StepFactor,
minR, maxR);
```

The local function logirand() is defined as:

```
function y = exporatiorand (mean, sigma, minR, maxR)

  r1 = exp(rand);
  r2 = exp(rand);
  x = r1/(r1+r2);
  x = (x - minR)/(maxR - minR);

  y = mean + sigma * x;
  y = (y + 0.3674) / (0.9964 + 0.3674);
  y = 2*y - 1;
```

```
end
```

Listing 22.2 shows the test program *test_scout11.m* that calculates the optimum variables for function *fx2*.

```
clear
clc
diary 'scout11_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] = scout11(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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] = scout11(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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 22.2. The source code for test_scout11.m.

Table 22.1 shows the summary output generated by Listing 22.2 and extracted from the output diary file scout11_*fx2*.txt. Table 22.1 shows that the log–log decay of the step factor yields better results than its the log–linear counterpart.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	3.102195509781628925e–28	0.0000000000000000e+00
Max bestFx	5.182672665639653166e–13	6.189294618762394775e–18
Mean bestFx	1.303519708820639281e–14	1.818361544640136985e–19
Sdev bestFx	8.193297007315502811e–14	9.798970599552726843e–19
Median bestFx	2.20049013151595830 5e–20	4.731081704548736106e–23
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 22.1. Summary of results of test file test_scout11.m

Listing 22.3 shows the test program *test_scout11b.m* that calculates the optimum variables for function *fx3*.

```
clear
clc
diary 'scout11_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] = scout11(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001,
200, 250, 150, 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] = scout11(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001,
200, 250, 150, 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 22.3. The source code for test_scout11b.m.

Table 22.2 shows the summary output generated by Listing 22.3 and extracted from the output diary file scout11_*fx3*.txt. Table 22.2 shows that the log–log decay of the step factor yields better results than its the log–linear counterpart.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	0.000000000000000000e+00	0.000000000000000000e+00
Max bestFx	1.432491233654116957e–11	4.985947329353123980e–17

Statistic/Result	For Log–linear Decay	For Log–log Decay
Mean bestFx	3.593607453951064561e–13	3.429692250637279689e–18
Sdev bestFx	2.264771944023154953e–12	1.108168009768523048e–17
Median bestFx	1.618372517568264519e–20	2.188989254117347708e–22
Mean Best X1	1.000000	1.000000
Mean Best X2	4.000000	4.000000
Mean Best X3	9.000000	9.000000
Mean Best X4	16.000000	16.000000

Table 22.2. Summary of results of test file test_scout11b.m

23/ Comparing Versions 1 through 11

Table 23.1 compares the mean best optimized functions while testing function *fx2* using MATLAB functions *scout()* through *scout11()*.

Function	For Log–linear Decay	For Log–log Decay
scout()	1.292036532307468458e–17	2.910301214191781435e–20
scout2()	2.506340276268329764e–12	1.402392032195924052e–12
scout3()	1.715583287202698552e–16	1.661112326866447749e–17
scout4()	9.076877635186815712e–14	4.710025606657630394e–19
scout5()	1.546991795471606629e–16	1.356101162840001786e–18
scout6()	9.048482271323894618e–16	1.888861594373887854e–19
scout7()	2.458827805922748509e–16	7.913666601427630172e–19
scout8()	3.286421020855316597e–16	5.548496299061203423e–18
scout9()	4.674382619321254179e–15	1.175233220465458680e–17
scout10()	7.780733984248558112e–16	1.574815625057527184e–20
scout11()	1.303519708820639281e–14	1.818361544640136985e–19

Table 23.1. Comparing the mean best optimized functions while testing function fx2 using scout() through scout11().

Table 23.1 shows that all of the optimized functions are in the orders of 10^{-12} through 10^{-20}. The function *scout()* remains the best with using the log–linear step decay factor. The function *scout10()* remains the best with using the log–log step decay factor.

Table 23.2 compares the mean best optimized functions while testing function *fx3* using MATLAB functions scout() through scout11().

Function	For Log–linear Decay	For Log–log Decay
scout()	9.732752177442062738e–16	1.812097031994652036e–17

Function	For Log–linear Decay	For Log–log Decay
scout2()	9.254635542193029242e–11	3.460369189744170226e–11
scout3()	1.001939813221296812e–13	5.193456574633486015e–18
scout4()	7.240256905116018957e–15	3.035806764626491629e–16
scout5()	1.599620770520585648e–13	2.040054922133027985e–17
scout6()	9.426100780722492981e–15	1.822164061425251953e–17
scout7()	2.287522184356782288e–15	4.620632725382591975e–17
scout8()	7.357389825363821327e–15	2.955511532168534031e–17
scout9()	1.788218497877370596e–14	8.537616829009803630e–18
scout10()	1.756929092686453132e–14	1.061800201579965287e–16
scout11()	3.593607453951064561e–13	3.429692250637279689e–18

Table 23.2. Comparing the mean best optimized functions while testing function fx3 using scout() through scout8().

Table 23.2 shows that all of the optimized functions are in the orders of 10^{-11} through 10^{-18}. The function *scout11()* has grabbed the top spot as the best function using the log–log step decay factor. The function *scout()* is still the best for the log–linear decay factor.

24/ The Extended Scout Optimization Algorithm

The extended version of SOA is based in version 1 of SOA. The basic difference is that the scouts that search for optimum variables spawn their own secondary scouts that search even closer. This double–scout generational approach is aimed to help in increasing accuracy of the solution.

Listing 24.1 shows the source code for ex*scout.m*.

```
function [bestX, bestFx] = exscout(fx, XLow, XHi, InitStep, FinStep,
MaxIters, ...
                                   MaxPop, MaxScoutPop, MaxScoutPop2 , ...
                                   bSkipNarrowTrusRegion, bTrace)
% The SCOUT function implements a version of the Fireworks
% Algorithm.
% This version uses an additional/secondary set of scouts.
%
% 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.
% MaxScoutPop - the size of the local scout population.
% MaxScoutPop2 - the size of the secondary local scout population.
% 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
% =======
% >> [bestX, bestFx] = exscout(@fx2, [0 0 0 0], [5 5 5 5], 0.1, 0.0001, 1000,
100, 100, 50, true)
%

if nargin < 11, bTrace = false; end
if nargin < 10, bSkipNarrowTrusRegion = false; 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;

pop = zeros(MaxPop,m);
scoutPop = zeros(MaxScoutPop,m);
scoutPop2 = zeros(MaxScoutPop2,m);

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

if bTrace
  fprintf('%f ', pop(1,1:n));
  fprintf('%e\n', pop(1,m));
end

for iter = 1:MaxIters
  StepFactor = exp(log(InitStep) +
log(FinStep/InitStep)*log(iter)/log(MaxIters));

  % 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));
      % ---------------------- Seconday scouts
      m3 = fix(MaxScoutPop2/2);
      % Use uniform random perturbations
      for j2=1:m3
        for k2=1:n
          scoutPop2(j2,k2)  = scoutPop(j,k2) + sqrt(XHi(k2) - XLow(k2)) *
StepFactor * (2*rand-1);
          if scoutPop2(j2,k2) < XLow(k2) || scoutPop2(j2,k2) > XHi(k2)
            scoutPop2(j2,k2) = XLow(k2) + (XHi(k2) - XLow(k2)) * rand;
          end
        end
        scoutPop2(j2,m) = fx(scoutPop2(j2,1:n));
      end
      % Use normal random perturbations
      for j2=m3+1:MaxScoutPop2
        for k2=1:n
          scoutPop2(j2,k2) = normrand(scoutPop(j,k2), sqrt((XHi(k2) -
XLow(k2))/3) * StepFactor);
          if scoutPop2(j2,k2) < XLow(k2) || scoutPop2(j2,k2) > XHi(k2)
            scoutPop2(j2,k2) = XLow(k2) + (XHi(k2) - XLow(k2)) * rand;
          end
        end
        scoutPop2(j2,m) = fx(scoutPop2(j2,1:n));
      end

      scoutPop2 = sortrows(scoutPop2,m);
      scoutPop(j,:) =  scoutPop2(1,:);
    end %for j
    % 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));

      % ---------------------- Seconday scouts
      m3 = fix(MaxScoutPop2/2);
      % Use uniform random perturbations
      for j2=1:m3
        for k2=1:n
          scoutPop2(j2,k2)  = scoutPop(j,k2) + sqrt(XHi(k2) - XLow(k2)) *
StepFactor * (2*rand-1);
          if scoutPop2(j2,k2) < XLow(k2) || scoutPop2(j2,k2) > XHi(k2)
            scoutPop2(j2,k2) = XLow(k2) + (XHi(k2) - XLow(k2)) * rand;
          end
        end
        scoutPop2(j2,m) = fx(scoutPop2(j2,1:n));
      end
      % Use non-uniform random perturbations
      for j2=m3+1:MaxScoutPop2
        for k2=1:n
          scoutPop2(j2,k2) = normrand(scoutPop(j,k2), sqrt((XHi(k2) -
XLow(k2))/3) * StepFactor);
          if scoutPop2(j2,k2) < XLow(k2) || scoutPop2(j2,k2) > XHi(k2)
            scoutPop2(j2,k2) = XLow(k2) + (XHi(k2) - XLow(k2)) * rand;
          end
        end
        scoutPop2(j2,m) = fx(scoutPop2(j2,1:n));
      end

      scoutPop2 = sortrows(scoutPop2,m);
      scoutPop(j,:) =  scoutPop2(1,:);
    end % for j =
    pop = [pop; scoutPop];
    pop = sortrows(pop,m);
    pop = pop(1:MaxPop,:);
  end

  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 24.1. The source code for exscout.m.

Listing 24.1 uses the normal random number generators, just like in function *scout()*. Listing 24.2 shows the code for file *test_exscout.m*.

```
clc
diary 'exscout_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] = exscout(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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] = exscout(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001, 200,
250, 150, 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 24.2. The source code for test_exscout.m.

Table 24.1 shows the summary output generated by Listing 24.2 and extracted from the output diary file *exscout_fx2.txt*. Table 24.1 shows that the results are mixed for both types of decay factors.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	0.00000000000000000e+00	0.00000000000000000e+00
Max bestFx	3.95863578634120310 7e–17	7.07421697370711274 4e–17
Mean bestFx	1.30446510039708866 4e–18	2.12212812055615855 0e–18
Sdev bestFx	6.32124555162900958 2e–18	1.12097983593531389 2e–17
Median bestFx	1.80118615311972901 9e–22	7.31605354096015217 4e–23
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 24.1. Summary of results of test file test_exscout.m

Listing 24.3 shows the test program *test_exscoutb.m* that calculates the optimum variables for function *fx3*.

```
clear
clc
diary 'exscout_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] = exscout(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001,
200, 250, 150, 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] = exscout(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001,
200, 250, 150, 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 24.3. The source code for test_exscoutb.m.

Table 24.2 shows the summary output generated by Listing 24.3 and extracted from the output diary file *exscout_fx3.txt*. Table 24.2 shows that the results are somewhat mixed for both typs of step decay factors.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	0.00000000000000000e+00	0.00000000000000000e+00
Max bestFx	1.98454033537646028e–17	2.97637082181190319e–16
Mean bestFx	9.61233352386952996e–19	1.14914905749841858e–17
Sdev bestFx	3.32936117052061503e–18	5.23510714834477687e–17

Statistic/Result	For Log–linear Decay	For Log–log Decay
Median bestFx	3.7033816687873452 02e–23	2.034166153392932939e–24
Mean Best X1	1.000000	1.000000
Mean Best X2	4.000000	4.000000
Mean Best X3	9.000000	9.000000
Mean Best X4	16.000000	16.000000

Table 24.2. Summary of results of test file test_exscoutb.m

25/ The Second Extended SOA Algorithm

The second extended SOA is based on the function *scout()*. This function divides the *MaxScoutPop* iterations into two halves. One set uses the uniform random numbers to calculate the elements of *scoutPop()*. The second uses normally distributed random numbers to calculate the elements of *scoutPop()*. In the second extended SOA there is one loop that calculates the scout locations using both types of random numbers. Each iteration selects the scout location with the smaller function value to populate the matrix *scoutPop()*. This approach doubles the scout locations (by applying two mehods of calculations *MaxScoutPop* times) to pick the best of both worlds, so to speak.

Listing 25.1 shows the source code for the function *exscout2()*.

```
function [bestX, bestFx] = exscout2(fx, XLow, XHi, InitStep, FinStep,
MaxIters,...
                                    MaxPop, MaxScoutPop, bSemiLogDecay, ...
                                    bSkipNarrowTrusRegion, bTrace)
% The SCOUT function implements an extended version of SOA.
% This version uses both uniform and normal RNG to perform perturbation on
% the scout local search. The function uses a single loop to determine the
% best way to calculate a scout location (using uniform and normal random
% numbers)
%
% 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;
scout1 = zeros(1,m);
scout2 = zeros(1,m);
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
    for j=1:MaxScoutPop
      % Use uniform random perturbations
      for k=1:n
        scout1(k)  = pop(i,k) + (XHi(k) - XLow(k)) * StepFactor * (2*rand-1);
        if scout1(k) < XLow(k) || scout1(k) > XHi(k)
          scout1(k) = XLow(k) + (XHi(k) - XLow(k)) * rand;
        end
      end
      scout1(m) = fx(scout1(1:n));

      % Use normal random perturbations
      for k=1:n
        scout2(k) = normrand(pop(i,k), (XHi(k) - XLow(k))/3 * StepFactor);
        if scout2(k) < XLow(k) || scout2(k) > XHi(k)
          scout2(k) = XLow(k) + (XHi(k) - XLow(k)) * rand;
        end
      end
      scout2(m) = fx(scout2(1:n));

      % select the better scout
      if scout1(m) < scout2(m)
        scoutPop(j,:) = scout1;
      else
        scoutPop(j,:) = scout2;
      end
    end
    pop = [pop; scoutPop];
    pop = sortrows(pop,m);
    pop = pop(1:MaxPop,:);
  end

  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 25.1. The source code for the function exscout2().

Listing 25.2 shows the test program *test_exscout2.m* that calculates the optimum variables for function *fx2*.

```
clear
clc
diary 'exscout2_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] = exscout2(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001,
200, 250, 150, 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] = exscout2(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001,
200, 250, 150, 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 25.2. The source code for test_exscout2.m.

Table 25.1 shows the summary output generated by Listing 25.2 and extracted from the output diary file *exscout2_fx2.txt*. Table 25.1 shows that the log–log decay of the step factor yields better results than its the log–linear counterpart.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	0.00000000000000000e+00	0.00000000000000000e+00
Max bestFx	1.26967041187919706 1e–14	7.526050923777682827e–19
Mean bestFx	7.932205351262983089e–16	2.552710681203200861e–20
Sdev bestFx	2.868844163465778672e–15	1.209804983531617949e–19
Median bestFx	4.217578129035457142e–21	1.436859765160893984e–23
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 25.1. Summary of results of test file test_exscout2.m

Listing 25.3 shows the test program *test_exscout2b.m* that calculates the optimum variables for function *fx3*.

```
clear
clc
diary 'exscout2_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] = exscout2(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001,
200, 250, 150, 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] = exscout2(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001,
200, 250, 150, 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 25.3. The source code for test_exscout2b.m.

Table 25.2 shows the summary output generated by Listing 25.3 and extracted from the output diary file *exscout2_fx3.txt*. Table 25.2 shows that the log–log decay of the step factor yields better results than its the log–linear counterpart.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	1.416498362937479323e–28	0.0000000000000000e+00
Max bestFx	9.880510648159797478e–14	5.679832023430901937e–17
Mean bestFx	2.639631438533924882e–15	2.312440186405938175e–18
Sdev bestFx	1.561776562206362755e–14	9.671774102433112462e–18
Median bestFx	1.492468538982974034e–19	2.851598899254401244e–23
Mean Best X1	1.000000	1.000000
Mean Best X2	4.000000	4.000000
Mean Best X3	9.000000	9.000000
Mean Best X4	16.000000	16.000000

Table 25.2. Summary of results of test file test_exscout2b.m

26/ The Third Extended SOA Algorithm

The third extended SOA function is based on the second extended version. The only difference is that this version replaces using normally distributed random numbers with ones generated using the empirical exponential ratio function. The latter function proved to work well in the function *scout11()*. That's all!

Listing 26.1 shows the source code for the function *exscout3()*.

```
function [bestX, bestFx] = exscout3(fx, XLow, XHi, InitStep, FinStep,
MaxIters,...
                                    MaxPop, MaxScoutPop, bSemiLogDecay, ...
                                    bSkipNarrowTrusRegion, bTrace)
% The SCOUT function implements an extended version of SOA.
% This version uses both uniform and exponential ratio RNG to perform
perturbation on
% the scout local search. The function uses a single loop to determine the
% best way to calculate a scout location (using uniform and exponential ratio
random
% numbers)
%
% 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;
minR = exp(0);
maxR = exp(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;
scout1 = zeros(1,m);
scout2 = zeros(1,m);
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
    for j=1:MaxScoutPop
      % Use uniform random perturbations
      for k=1:n
        scout1(k)  = pop(i,k) + (XHi(k) - XLow(k)) * StepFactor * (2*rand-1);
        if scout1(k) < XLow(k) || scout1(k) > XHi(k)
          scout1(k) = XLow(k) + (XHi(k) - XLow(k)) * rand;
        end
      end
      scout1(m) = fx(scout1(1:n));

      % Use non-uniform random perturbations
      for k=1:n
        scout2(k) = exporatiorand(pop(i,k), (XHi(k) - XLow(k))/3 *
StepFactor, minR, maxR);
        if scout2(k) < XLow(k) || scout2(k) > XHi(k)
          scout2(k) = XLow(k) + (XHi(k) - XLow(k)) * rand;
        end
      end
      scout2(m) = fx(scout2(1:n));

      % select the better scout
      if scout1(m) < scout2(m)
        scoutPop(j,:) = scout1;
      else
        scoutPop(j,:) = scout2;
      end
    end
    pop = [pop; scoutPop];
```



```
    pop = sortrows(pop,m);
    pop = pop(1:MaxPop,:);
  end

  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 = exporatiorand(mean, sigma, minR, maxR)

  r1 = rand;
  r2 = rand;
  x = exp(r1)/exp(r2);
  x = (x - minR)/(maxR - minR);

  y = mean + sigma * x;

  y = (y + 0.3674) / (0.9964 + 0.3674);
  y = 2*y - 1;
end
```

Listing 26.1. The source code for the function exscout3().

Listing 26.2 shows the test program *test_exscout3.m* that calculates the optimum variables for function *fx2*.

```
clear
clc
diary 'exscout3_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] = exscout3(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001,
200, 250, 150, 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] = exscout3(@fx2, zeros(1,n), 5+zeros(1,n), 0.1, 0.0001,
200, 250, 150, 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 26.2. The source code for test_exscout3.m.

Table 26.1 shows the summary output generated by Listing 26.2 and extracted from the output diary file *exscout3_fx2.txt*. Table 26.1 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	0.00000000000000000e+00	0.00000000000000000e+00
Max bestFx	4.862388767905535423e–14	9.958615465658420508e–17
Mean bestFx	1.910321595029786951e–15	2.551591766979891007e–18
Sdev bestFx	8.393392720831378144e–15	1.573729782558128042e–17
Median bestFx	9.963940789762719914e–21	7.438681002322149563e–24
Mean Best X1	1.234500	1.234500
Mean Best X2	2.345600	2.345600

Statistic/Result	For Log–linear Decay	For Log–log Decay
Mean Best X3	3.456700	3.456700
Mean Best X4	4.567800	4.567800

Table 26.1. Summary of results of test file test_exscout3.m

Listing 26.3 shows the test program *test_scout3b.m* that calculates the optimum variables for function *fx3*.

```
clear
clc
diary 'exscout3_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] = exscout3(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001,
200, 250, 150, 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] = exscout3(@fx3, zeros(1,n), 25+zeros(1,n), 0.1, 0.0001,
200, 250, 150, 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 26.3. The source code for test_exscout3b.m.

Table 26.2 shows the summary output generated by Listing 26.3 and extracted from the output diary file *exscout3_fx3.txt*. Table 26.2 shows that the log–log decay of the step factor yields better results than its the log–linear counterpart.

Statistic/Result	For Log–linear Decay	For Log–log Decay
Min bestFx	4.848699041276345643e–26	0.000000000000000000e+00
Max bestFx	2.102210254438262950e–13	2.468019342861691875e–15
Mean bestFx	6.676105491922986065e–15	1.6045254024871922 54e–16
Sdev bestFx	3.315680873936561493e–14	5.724535525237439388e–16
Median bestFx	6.170469944580677554e–18	1.799469298432318968e–22
Mean Best X1	1.000000	1.000000
Mean Best X2	4.000000	4.000000
Mean Best X3	9.000000	9.000000
Mean Best X4	16.000000	16.000000

Table 26.2. Summary of results of test file test_exscout3b.m

27/ The Fourth Extended SOA Algorithm

The fourth extended SOA function is based on the function *scout10()*. The main difference is a new function parameter, the *CutOffIterations*. In the first *CutOffIterations* iterations, the *exscout4()* function works just like *scout10()*. Beyond these iterations, the function calculates the scout locations based on the leading 10% of the main population.

Listing 27.1 shows the source code for the function *exscout4()*.


```
function [bestX, bestFx] = exscout4(fx, XLow, XHi, InitStep, FinStep,
MaxIters,...
                                    MaxPop, MaxScoutPop, ...
                                    CutOffIters, bSemiLogDecay, ...
                                    bSkipNarrowTrusRegion, bTrace)
% The SCOUT function implements a simpler version of the Fireworks
% Algorithm.
% This version uses both uniform and exponential ratio 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.
% CutOffIters - the leading number of iterations where the function works
% like function scout10(). Beyond that point, the scout locations are
% calculated using the leading/best 10% of the 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;
minR = exp(0);
maxR = exp(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);

  if iter <= CutOffIters
    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) = exporatiorand(pop(i,k), (XHi(k) - XLow(k))/3 *
StepFactor, minR, maxR);
          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
      pop = [pop; scoutPop];
      pop = sortrows(pop,m);
      pop = pop(1:MaxPop,:);
    end
  else
    % narrow seacrh phase
    offset = 0;
    for i=1:MaxPop
      m2 = fix(MaxScoutPop/2);
      % Use uniform random perturbations
      for j=1:m2
        for k=1:n
          scoutPop(j,k)  = pop(1+offset,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 non-uniform random perturbations
      for j=m2+1:MaxScoutPop
        for k=1:n
          scoutPop(j,k) = exporatiorand(pop(1+offset,k), (XHi(k) - XLow(k))/3
* StepFactor, minR, maxR);
          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
      pop = [pop; scoutPop];
      pop = sortrows(pop,m);
      pop = pop(1:MaxPop,:);
      offset = mod(offset + 1, tpc);
    end
```



```
  end
  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 = exporatiorand(mean, sigma, minR, maxR)

  r1 = rand;
  r2 = rand;
  x = exp(r1)/exp(r2);
  x = (x - minR)/(maxR - minR);

  y = mean + sigma * x;

  y = (y + 0.3674) / (0.9964 + 0.3674);
  y = 2*y - 1;
end
```

Listing 27.1. The source code for the function exscout4().

Notice the following statements that calculate the scout locations when *iter* is greater than *CutOffIter*:

```
scoutPop(j,k)  = pop(1+offset,k) + (XHi(k) - XLow(k)) * StepFactor * (2*rand-
1);

scoutPop(j,k) = exporatiorand(pop(1+offset,k), (XHi(k) - XLow(k))/3 *
StepFactor, minR, maxR);
```

The above statement use the population element *pop(1+offset,k)* with the variable *offset* initialized as zero. The row index *1+offset* accesses the best rows of matrix *pop*. The loop for *i=1:MaxPop* increments the values in *offset* using the modulus function. Thus, the values in ariable *offset* cycle between 0 and *tpc–1* (where *tpc* is 10% of the maximum number of iterations):

```
offset = mod(offset + 1, tpc);
```

Listing 27.2 shows the test program *test_exscout4.m* that calculates the optimum variables for function *fx2*. The function uses the argument 250 (which is half of the population of 500) for parameter *CutOffIters*.

```
clear
clc
diary 'exscout4_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] = exscout4(@fx2, [0 0 0 0], [5 5 5 5], .1, 0.0001, 200,
250, 150, 250, 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] = exscout4(@fx2, [0 0 0 0], [5 5 5 5], .1, 0.0001, 200,
250, 150, 250, 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 27.2. The source code for test_exscout4.m.

Table 27.1 shows the summary output generated by Listing 27.2 and extracted from the output diary file *exscout4_fx2.txt*. Table 27.1 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	2.603240987229338958e–29	0.000000000000000000e+00
Max bestFx	1.764806662649901189e–14	4.550892518833957499e–17
Mean bestFx	4.498897231662258218e–16	1.925253382933299785e–18
Sdev bestFx	2.789232514510854278e–15	8.465696292486173511e–18
Median bestFx	2.219835508025333061e–21	8.059225725358113455e–23
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 27.1. Summary of results of test file test_exscout4.m

Listing 27.3 shows the test program *test_scout4b.m* that calculates the optimum variables for function *fx3*. The function uses the argument 100 (which is half of the population of 200) for parameter *CutOffIters*.

```
clear
clc
diary 'exscout4_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] = exscout4(@fx3, zeros(1,n), 25+zeros(1,n), .1, 0.0001,
200, 250, 150, 100, 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] = exscout4(@fx3, zeros(1,n), 25+zeros(1,n), .1, 0.0001,
200, 250, 150, 1000, 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('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 27.3. The source code for test_exscout4b.m.

Table 27.2 shows the summary output generated by Listing 27.3 and extracted from the output diary file *exscout4_fx3.txt*. Table 27.2 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	0.000000000000000000e+00	0.000000000000000000e+00
Max bestFx	1.007983845140686005e–12	1.147013623066053444e–13
Mean bestFx	2.697518685777862863e–14	3.042744148156201184e–15
Sdev bestFx	1.592702879803461660e–13	1.813898628753289117e–14
Median bestFx	8.647737933155699171e–20	1.150565116880812593e–22
Mean Best X1	1.000000	1.000000
Mean Best X2	4.000000	4.000000
Mean Best X3	9.000000	9.000000
Mean Best X4	16.000000	16.000000

Table 27.2. Summary of results of test file test_exscout4b.m.

28/ Comparing All Versions of the SOA Implementations

Table 28.1 compares the mean best optimized functions while testing function *fx2* using MATLAB functions *scout()* through *exscout4()*.

Mean Best Function	For Log–linear Decay	For Log–log Decay
scout()	1.2920365323074684 58e–17	2.91030121419178143 5e–20
scout2()	2.5063402762683297 64e–12	1.4023920321959240 52e–12
scout3()	1.7155832872026985 52e–16	1.6611123268664477 49e–17
scout4()	9.0768776351868157 12e–14	4.7100256066576303 94e–19
scout5()	1.5469917954716066 29e–16	1.3561011628400017 86e–18
scout6()	9.0484822713238946 18e–16	1.8888615943738878 54e–19
scout7()	2.4588278059227485 09e–16	7.9136666014276301 72e–19
scout8()	3.2864210208553165 97e–16	5.5484962990612034 23e–18
scout9()	4.6743826193212541 79e–15	1.1752332204654586 80e–17
scout10()	7.7807339842485581 12e–16	1.5748156250575271 84e–20
scout11()	1.3035197088206392 81e–14	1.8183615446401369 85e–19
exscout()	1.3044651003970886 64e–18	2.1221281205561585 50e–18
exscout2()	7.9322053512629830 89e–16	2.5527106812032008 61e–20
exscout3()	1.9103215950297869 51e–15	2.5515917669798910 07e–18
exscout4()	4.4988972316622582 18e–16	1.9252533829332997 85e–18

Table 28.1. Comparing the mean best optimized functions while testing function fx2 using scout() through exscout4().

Table 28.1 shows that the function *exscout()* is the new top contendor, followed by functions *scout()* and *scout5()* for the log–linear step decay. The function *scou10()* is the new top contendor, followed by functions *exscout2()* and *scout()* for the case of log–log step decay. These rankings make function *scout()* the best in both step size dcay modes, based on testing function *fx2*.

Table 28.2 compares the mean best optimized functions while testing function *fx3* using MATLAB functions *scout()* through *exscout4()*.

Mean Best Function	For Log–linear Decay	For Log–log Decay
scout()	9.7327521774420627 38e–16	1.8120970319946520 36e–17
scout2()	9.2546355421930292 42e–11	3.4603691897441702 26e–11

Mean Best Function	For Log–linear Decay	For Log–log Decay
scout3()	1.0019398132212968e–13	5.1934565746334860 15e–18
scout4()	7.240256905116018957e–15	3.035806764626491629e–16
scout5()	1.599620770520585648e–13	2.040054922133027985e–17
scout6()	9.426100780722492981e–15	1.822164061425251953e–17
scout7()	2.287522184356782288e–15	4.620632725382591975e–17
scout8()	7.357389825363821327e–15	2.955511532168534031e–17
scout9()	1.788218497877370596e–14	8.537616829009803630e–18
scout10()	1.756929092686453132e–14	1.061800201579965287e–16
scout11()	3.593607453951064561e–13	3.429692250637279689e–18
exscout()	9.612333523869529968e–19	1.149149057498418580e–17
exscout2()	2.639631438533924882e–15	2.312440186405938175e–18
exscout3()	6.676105491922986065e–15	1.604525402487192254e–16
exscout4()	2.697518685777862863e–14	3.042744148156201184e–15

Table 28.2. Comparing the mean best optimized functions while testing function fx3 using scout() through exscout4().

Table 28.2 shows that the function *exscout()* is the top contendor, followed by *scout()* and *scout7()* for the log–linear step decay. The function *exscout2()* is the top contendor, followed by *scout11()* and *scout3()* for the case of both log–log step decay. Thus, there is clear winner that appears in both decay modes.

Table 28.3 compares the median best optimized functions while testing function *fx2* using MATLAB functions *scout()* through *exscout4()*.

Median Best Function	For Log–linear Decay	For Log–log Decay
scout()	1.045888199234511120e–21	2.619399079704751543e–23
scout2()	1.870924487616165596e–12	9.077173911219513730e–13
scout3()	5.887531178413371384e–20	6.434903275816984488e–24
scout4()	4.055851128343713579e–21	7.957885820969734469e–23
scout5()	1.535390867643421351e–20	1.747431256571159916e–24
scout6()	1.021979976091847881e–19	1.017168477668730048e–22
scout7()	7.424555335051904823e–21	1.256272590284958378e–23
scout8()	4.932728824142635402e–21	6.676674758881655967e–23
scout9()	1.026485515732391737e–20	2.365812787419539709e–23
scout10()	1.053669634777385906e–19	1.439466122148698946e–23
scout11()	2.200490131515958305e–20	4.731081704548736106e–23

Median Best Function	For Log–linear Decay	For Log–log Decay
exscout()	1.801186153119729019e–22	7.316053540960152174e–23
exscout2()	4.217578129035457142e–21	1.436859765160893984e–23
exscout3()	9.963940789762719914e–21	7.438681002322149563e–24
exscout4()	2.219835508025333061e–21	8.059225725358113455e–23

Table 28.3. Comparing the median best optimized functions while testing function fx2 using scout() through exscout4().

Table 28.3 shows that the function *exscout()* is the top contendor, followed by *scout ()* and *exscout4()* for the log–linear step decay. The function *scout5()* is the top contendor, followed by *scout3()* and *exscout3()* for the case of both log–log step decay.

Table 28.4 compares the median best optimized functions while testing function *fx3* using MATLAB functions *scout()* through *exscout4()*.

Median Best Function	For Log–linear Decay	For Log–log Decay
scout()	4.708229015750529925e–20	8.239786407449882862e–23
scout2()	6.476147350713136187e–11	2.392148681529799968e–11
scout3()	4.726110369729839819e–19	6.845535634360838467e–22
scout4()	9.983669058850652093e–19	4.308091269233033857e–22
scout5()	8.687173610183244946e–19	5.683920042234242094e–21
scout6()	1.625323791048715099e–19	1.470747717402833343e–21
scout7()	3.548266037527064923e–20	1.388680121939456240e–21
scout8()	1.306982351980128199e–19	8.934521482589232009e–22
scout9()	7.004539001377047040e–19	4.836268760065842493e–22
scout10()	1.929623021049735268e–18	1.831797807023672282e–21
scout11()	1.618372517568264519e–20	2.188989254117347708e–22
exscout()	3.703381668787345202e–23	2.034166153392932939e–24
exscout2()	1.492468538982974034e–19	2.851598899254401244e–23
exscout3()	6.170469944580677554e–18	1.799469298432318968e–22
exscout4()	8.647737933155699171e–20	1.150565116880812593e–22

Table 28.4. Comparing the median best optimized functions while testing function fx3 using scout() through exscout4().

Table 28.4 shows that the function *scout11()* is the top contendor, followed by *scout7()* and *scout()* for the log–linear step decay. The function *exscout()* is the top contendor, followed by *exscout2()* and *scout()* for the case of both log–log step

decay. The function scout() appears in the top rankings for both step size decays for testing function *fx3*.

29/ Conclusions

The family of functions that implement the Scout Optimization Algorithms perform, in general, quite well. The functions *scout()*, *exscout()*, *exscout2()*, *scout11()*, and *scout10()* rank as the top four functions. In general, all of the functions perform between excellent and very good. All the SOA functions returned correct values for the optimization variables. Studying the values of the optimizated function was a way to push the envelope.

The various tables of results show largely, that the log–log step decay scheme is better than the log–linear counterpart. The log–log step decay schemes reduce the step factor quicker and allows the calculations to zoom in on the solution and not overshoot. Using a linear step decay scheme often lead to overshooting and the inability to zoom in closer to the optimization variables' values. Early testing of SOA with linear step decay scheme lead me to avoid using it and replace it with the two other schemes that I used in this study.

☛ It is worth pointing out that the test programs used *conservative* (and somewhat high) arguments for the maximum number of iterations, population size, and scout population size. Typical test values were 200, 250, and 150, respectively. In translating the function *exscout2()* into Excel VBA (see Appendix A3 for the VBA code) I had to resort to using smaller values because the VBA code was slower. The reduction in speed was due to the fact that VBA lacks built–in commands to sort a matrix, like MATLAB's *sortrows()*. VBA also lacks commands to copy arrays, array slices, matrices, and matrix slices in a single statement. Consequently, I had to code a VBA matrix sort subroutine which is relatively slow. I also had to use *For* loops to copy arrays and matrices. The initial test for the VBA code used the same numerical arguments as in the MATLAB test programs. The VBA code took too long to display any intermediate results on an Excel spreadsheet. So, to make sure that the VBA code was working, I reduced the arguments for the maximum number of iterations, population size, and scout population size to 50, 50, and 10, respectively. I was ready to accept any result just to make sure the VBA code worked. To my surprised, the low arguments supplied generated quite good results! I then tried the same arguments in one of the MATLAB test programs and also got good results. When I ran the MATLAB test program the results zipped quickly on

the screen. I did spot a few outliers with best optimized function values in the order of 10^{-3}! This high value was due the failure of function *ZoomRange()* to produce correct narrow trust region ranges. If I turned off the feature to narrow these ranges, I consistently got best function values in the order of 10^{-9}. This value is ten or more orders of magnitude bigger than results appearing in this study. So, the choice is yours when using arguments for the maximum number of iterations, population size, and scout population size:

1. Use the function *ZoomRange()* and reject faulty results that may occur when using smaller arguments for the maximum number of iterations, population size, and scout population size.
2. Avoid using function *ZoomRange()* and obtain results that are still good but may not be as accurate as you like, when using smaller arguments for the maximum number of iterations, population size, and scout population size.
3. Stick with high value arguments and obtain very good results.

A1/ Downloading Files

The file storing this document is also part of a *scout.zip* file that also contains all the MATLAB source code files, the Excel files, and the text diary files that store the output for the various MATLAB test files.

A2/ An Implementation in Excel VBA

This appendix section offers you an Excel VBA implementation for a variant of the function *exscout2()*. Figure A2.1 shows a sample spreadsheet that allows you to input data and obtain output from the Excel VBA code.

A	B	C	D	E	F
InitStep	Initial Range –>	X1	X2	X3	X4
0.1		0	0	0	0
FinalStep		5	5	5	5
0.0001					
MaxIters	Narrow Range –>	0	0.943517	2.928267975	3.674244468
50		2.52413512	4.581947	5	5
MaxPop					
50	Current Best X	1.2344963	2.345605	3.456714604	4.56780829
MaxScoutPop	Current Best Fx	3.21564E–10			

10					
SemiLogDecay?					
No					
SkipNarrowTrusRegion?					
No					

Figure A2.1. Sample Excel spreadsheet fx3 used to support SOA.

The Excel file, which I include in the ZIP distribution file, has three worksheets labeled *fx1*, *fx2*, and *fx3*. Each worksheet has its complete set of VBA code associated with it. Each set of VBA code has a custom subroutine to support an optimized function.

Looking at Figure A2.1 you can see that the input cells are:

- Cell A2 stores the initial step size.
- Cell A4 stores the final step size.
- Cell A6 stores the maximum number of iterations.
- Cell A8 stores the population size.
- Cell A10 stores the scout population size.
- Cell A12 is a Yes/No answer for the step size decay mode.
- Cell A14 is a Yes/No answer for skipping the narrowing of the trust region range.
- Cells C3 to F4 contain the initial trust region range.

The VBA code displays the following:

- The narrow trust region range in cells C5 to F6.
- The updated solution in cells C8 to F8.
- The best function value in cell C9.

Listing A2.1 shows the VBA source code associated with the sheet labeled *fx2*.

```
Option Explicit
Option Base 1

Sub Fx(ByRef X() As Double, ByVal Row As Integer, ByVal N As Integer)
  Dim I As Integer, M As Integer
  Dim Sum As Double
  Static Z(4) As Double

  If Z(1) = 0 Then
    ' The array of optimum values
    Z(1) = 1.2345
```



```
    Z(2) = 2.3456
    Z(3) = 3.4567
    Z(4) = 4.5678
  End If
  M = N + 1
  For I = 1 To N
    Sum = Sum + (X(Row, I) - Z(I)) ^ 2
  Next I
  X(Row, M) = Sum
End Sub

Sub Scout()
 Const SCOUT1 = 1
 Const SCOUT2 = 2
 Const ELEM1 = 1

 Dim I As Integer, J As Integer, K As Integer
 Dim Iter As Integer, MaxIters As Integer, Tpc As Integer
 Dim InitStep As Double, FinalStep As Double
 Dim MaxPop As Integer, MaxScoutPop As Integer
 Dim bSemiLogDecay As Boolean, bSkipNarrowTrusRegion As Boolean
 Dim XLow() As Double, XHi() As Double, e1 As Double, e0 As Double
 Dim Pop() As Double, scoutPop() As Double, popAug() As Double
 Dim Scout() As Double, PopMember() As Double, StepFactor As Double
 Dim N As Integer, M As Integer
 Dim minR As Double, maxR As Double, SF As Double

 On Error GoTo HandleErr

 InitStep = [A2].Value
 FinalStep = [A2].Value
 MaxIters = [A6].Value
 MaxPop = [A8].Value
 MaxScoutPop = [A10].Value
 bSemiLogDecay = UCase(Trim([A12].Value)) = "YES"
 bSkipNarrowTrusRegion = UCase(Trim([A14].Value)) = "YES"
 Range("C5:F9").Clear
 N = 4
 M = N + 1

 ReDim Pop(MaxPop, M), scoutPop(MaxScoutPop, M), XLow(N), XHi(N)
 ReDim Scout(2, M), PopMember(1, M)

 e0 = Exp(0)
 e1 = Exp(1)
 minR = e0 / (e0 + e1)
 maxR = e1 / (e0 + e1)

 Tpc = MaxIters \ 10
 ' read user-defined trust region range
 For I = 1 To N
   XLow(I) = Cells(2, 2 + I)
   XHi(I) = Cells(3, 2 + I)
 Next I

 Randomize Timer
```

```
 If Not bSkipNarrowTrusRegion Then
   Call ZoomRange(N, XLow, XHi, MaxIters, MaxPop, 0.1, 3, Pop)
 Else
   '' initialize function values in eac row of pop
   For I = 1 To MaxPop
     Pop(I, M) = 1E+99
   Next I

   For I = 1 To MaxPop
     For J = 1 To N
       Pop(I, J) = XLow(J) + (XHi(J) - XLow(J)) * Rnd(1)
     Next J
    Call Fx(Pop, I, N)
   Next I
   Call SortPop(Pop, MaxPop, N)
 End If

 For I = 1 To N
   Cells(5, 2 + I) = XLow(I)
   Cells(6, 2 + I) = XHi(I)
 Next I

 For I = 1 To MaxScoutPop
   scoutPop(I, M) = 1E+99
 Next I

 ' ---------------------------------------- MAIN LOOP
 For Iter = 1 To MaxIters
   If bSemiLogDecay Then
    StepFactor = Exp(Log(InitStep) + (Iter - 1) / (MaxIters - 1) *
Log(FinalStep / InitStep))
  Else
    StepFactor = Exp(Log(InitStep) + Log(FinalStep / InitStep) * Log(Iter) /
Log(MaxIters))
  End If

  For I = 2 To MaxPop
    If Rnd(1) > 0.5 Then
      SF = StepFactor
      If Rnd(1) > 0.5 Then SF = 1
      For K = 1 To N
        PopMember(ELEM1, K) = Pop(I, K) + (Pop(1, K) - Pop(I, K)) * Rnd(1) *
SF
      Next K
    Else
      SF = StepFactor
      If Rnd(1) > 0.5 Then SF = 1
      For K = 1 To N
        PopMember(ELEM1, K) = XLow(K) + (XHi(K) - XLow(K)) * Rnd(1) * SF
      Next K
    End If
    Call Fx(PopMember, ELEM1, N)
    If PopMember(ELEM1, M) < Pop(I, M) Then
      For K = 1 To M
        Pop(I, K) = PopMember(ELEM1, K)
      Next K
```

```
    End If
  Next I
  Call SortPop(Pop, MaxPop, N)

  For I = 1 To MaxPop
    DoEvents
    For J = 1 To MaxScoutPop
      DoEvents
      For K = 1 To N
        Scout(SCOUT1, K) = Pop(I, K) + (XHi(K) - XLow(K)) * StepFactor * (2 *
Rnd(1) - 1)
        If Scout(SCOUT1, K) < XLow(K) Or Scout(SCOUT1, K) > XHi(K) Then
          Scout(SCOUT1, K) = XLow(K) + (XHi(K) - XLow(K)) * Rnd(1)
        End If
      Next K
      Call Fx(Scout, SCOUT1, N)

      For K = 1 To N
        Scout(SCOUT2, K) = ExpoRand(Pop(I, K), (XHi(K) - XLow(K)) / 3 *
StepFactor, minR, maxR)
        If Scout(SCOUT2, K) < XLow(K) Or Scout(SCOUT2, K) > XHi(K) Then
          Scout(SCOUT2, K) = XLow(K) + (XHi(K) - XLow(K)) * Rnd(1)
        End If
      Next K
      Call Fx(Scout, SCOUT2, N)

      ' select the better scout location
      If Scout(SCOUT1, M) < Scout(SCOUT2, M) Then
        For K = 1 To M
          scoutPop(J, K) = Scout(SCOUT1, K)
        Next K
      Else
        For K = 1 To M
          scoutPop(J, K) = Scout(SCOUT2, K)
        Next K
      End If
    Next J

    ReDim popAug(MaxPop + MaxScoutPop, M)
    ' augment matrix pop with matrix scoutPop
    For J = 1 To MaxPop
      For K = 1 To M
        popAug(J, K) = Pop(J, K)
      Next K
    Next J
    For J = MaxPop + 1 To MaxPop + MaxScoutPop
      For K = 1 To M
        popAug(J, K) = scoutPop(J - MaxPop, K)
      Next K
    Next J
    Call SortPop(popAug, MaxPop + MaxScoutPop, N)
    ' size matrix pop down
    For J = 1 To MaxPop
      For K = 1 To M
        Pop(J, K) = popAug(J, K)
      Next K
    Next J
```



```
  Next I

  If Iter Mod Tpc = 0 Then
     For I = 1 To N
       Cells(8, 2 + I) = Pop(1, I)
     Next I
     [C9].Value = Pop(1, M)
  End If

 Next Iter
 MsgBox "Done!"
ExitProc:
  Exit Sub

HandleErr:
  MsgBox Err.Description
  Resume Next
End Sub

Sub SortPop(ByRef X() As Double, ByVal MaxPop As Integer, ByVal N As Integer)
' Gonnet & Baeza-Yates (1991) ShellSort implementation
  Dim I As Integer, J As Integer, K As Integer, Offset As Integer
  Dim bInOrder As Boolean, Buffer As Double, iBuff As Integer
  Dim M As Integer
  Dim Idx() As Integer, X2() As Double

  On Error GoTo HandleErr

  M = N + 1
  ReDim Idx(MaxPop), X2(MaxPop, M)
  For I = 1 To MaxPop
    Idx(I) = I
    For K = 1 To M
      X2(I, K) = X(I, K)
    Next K
  Next I

  Offset = MaxPop
  Do While Offset > 1
    Offset = Fix(Offset * 5# / 11#)
    If Offset < 1 Then Offset = 1
    Do
      bInOrder = True
      For I = 1 To MaxPop - Offset
        DoEvents
        J = I + Offset
        If X(Idx(I), M) > X(Idx(J), M) Then
           iBuff = Idx(I)
           Idx(I) = Idx(J)
           Idx(J) = iBuff
           bInOrder = False
        End If
      Next I
    Loop Until bInOrder
  Loop

  For I = 1 To MaxPop
```

```
    For K = 1 To M
      X(I, K) = X2(Idx(I), K)
    Next K
  Next I


ExitProc:
  Exit Sub

HandleErr:
  Resume Next

End Sub

Sub ZoomRange(ByVal N As Integer, ByRef XLow() As Double, ByRef XHi() As
Double, ByVal MaxIters As Integer, _
              ByVal MaxPop As Integer, ByVal Frac As Double, ByVal Groups As
Integer, _
              ByRef Pop() As Double)

  Dim I As Integer, J As Integer, K As Integer, p As Integer, Iter As Integer
  Dim XLow0() As Double, XHi0() As Double, bestPop() As Double
  Dim SumX() As Double, SumX2() As Double, Mean As Double, Sdev As Double
  Dim M2 As Integer, M As Integer

  On Error GoTo HandleErr
  M = N + 1
  ReDim XLow0(N), XHi0(N), SumX(N), SumX2(N)

  For I = 1 To N
    XLow0(I) = XLow(I)
    XHi0(I) = XHi(I)
  Next I
  p = Groups
  ReDim Pop(MaxPop, M), bestPop(p * MaxIters, M)
  For I = 1 To p * MaxIters
    bestPop(I, M) = 1E+99
  Next I
  Randomize Timer
  For Iter = 1 To p * MaxIters Step p
    DoEvents
    For I = 1 To MaxPop
      DoEvents
      For K = 1 To N
        Pop(I, K) = XLow(K) + (XHi(K) - XLow(K)) * Rnd(1)
      Next K
      Call Fx(Pop, I, N)
    Next I
    Call SortPop(Pop, MaxPop, N)
    For I = 1 To p
      For K = 1 To N
        bestPop(Iter + I - 1, K) = Pop(I, K)
      Next K
    Next I
  Next Iter

  Call SortPop(bestPop, UBound(bestPop, 1), N)
```

```
  'ReDim Preserve bestPop(MaxIters, M)
  M2 = Fix(MaxPop * Frac)
  For K = 1 To N
    SumX(K) = 0
    SumX2(K) = 0
  Next K

  For I = 1 To M2
    For K = 1 To N
      SumX(K) = SumX(K) + bestPop(I, K)
      SumX2(K) = SumX2(K) + bestPop(I, K) ^ 2
    Next K
  Next I

  For K = 1 To N
    Mean = SumX(K) / M2
    Sdev = Sqr((SumX2(K) - SumX(K) ^ 2 / M2) / (M2 - 1))
    XLow(K) = Mean - 2 * Sdev
    XHi(K) = Mean + 2 * Sdev

    If XLow(K) < XLow0(K) Then XLow(K) = XLow0(K)
    If XHi(K) > XHi0(K) Then XHi(K) = XHi0(K)
  Next K


ExitProc:
  Exit Sub

HandleErr:
  Resume Next
End Sub

Function ExpoRand(ByVal X0 As Double, ByVal Gamma As Double, _
                  ByVal minR As Double, ByVal maxR As Double) As Double
  Dim r1 As Double, r2 As Double, X As Double

  r1 = Exp(Rnd(1))
  r2 = Exp(Rnd(1))
  X = r1 / (r1 + r2)
  X = (X - minR) / (maxR - minR)

  ExpoRand = X0 + Gamma * X
End Function
```

Listing A2.1. The VBA source code for a SOA version that tests function fx2.

The code in Listing A2.1 has the following functions and subroutines:

- Subroutine fx that calculate the value of the optimized function and stores it in row *Row* and column *N+1* of matrix *X*. Customize the code for this subroutine to optimize different functions.
- Subroutine *Scout()* which is the main subroutine that you need to execute.
- Subroutine *SortPop()* that sorts a population matrix. The subroutine uses a Shell–Metzner algorithm. To speed up the process, the implementation uses

indices that are swapped and a copy of the matrix to help write back the sorted rows of the matrix *X*.

- Subroutine *ZoomRange()* that narrows the trust region range and returns the matrix *Pop*.
- Function *Exporand()* which implements the exponential ratio RNG.

The VBA code in Listing A2.1 implements a matrix sort subroutine and has loops to calculate the mean and standard deviation. The VBA code requires coding commands similar to MATLAB functions, such as *sortrows()*, *mean()*, and *std()*.

The Excel VBA does not support vector or matrix operations that you can perform in a single statement. You have to use loops to work with vectors and nested loops to work with matrices.

I recommend setting a break point at the various *Resume Next* statements in the various subroutines. These breakpoints allow you to trap run–time errors. In debug mode, resume to the next statement. The offending statement would be the one right before the statement where the program resumes execution. Study the offending statement, halt the program, fix the issue, and rerun the *Scout()* subroutine.

Finally, let me present the VBA code that implements the test functions *fx1* and *fx3*. Listing A2.2 shows the VBA source code for a subroutine *Fx* that implements the test function *fx1*.

```
Sub Fx(ByRef X() As Double, ByVal Row As Integer, ByVal N As Integer)
  Dim I As Integer, M As Integer
  Dim Sum As Double

  M = N + 1
  For I = 1 To N
    Sum = Sum + (X(Row, I) - I) ^ 2
  Next I
  X(Row, M) = Sum
End Sub
```

Listing A2.2. The VBA source code for a subroutine Fx that implements the test function fx1.

Listing A2.3 shows the VBA source code for a subroutine *Fx* that implements the test function *fx3*.

```
Sub Fx(ByRef X() As Double, ByVal Row As Integer, ByVal N As Integer)
  Dim I As Integer, M As Integer
```



```
  Dim Sum As Double

  M = N + 1
  For I = 1 To N
    Sum = Sum + (X(Row, I) - I ^ 2) ^ 2
  Next I
  X(Row, M) = Sum
End Sub
```

Listing A2.3. The VBA source code for a subroutine Fx that implements the test function fx3

A3/ Handling Functions with Local and Global Minima

The optimized functions that I used in this study have one optimum value. This appendix examine how, for example, the function *scout()* handles functions with two local minia and one global minimum. Listing A3.1 shows the source code in file mo*fx1*.m. This file has minima at 1, 2.5, and 4 for all the variables.

```
function y = mofx1(x)
 n = length(x);
 s1 = 0;
 for i=1:n
   s1 = s1 + (x(i) - 1)^2;
 end

 s2 = 10;
 for i=1:n
   s2 = s2 + (x(i) - 4)^2;
 end

 s3 =100;
 for i=1:n
   s3 = s3 + (x(i) - 2.5)^2;
 end
 y = s1 * s2 * s3;
end
```

Listing A3.1 The source code in file mofx1.m.

The function in Listing A3.1 impllements:

$$f(x) = \left(\sum_{i=1}^{4}(x_i - 1)^2\right) * \left(10 + \sum_{i=1}^{4}(x_i - 2.5)^2\right) * \left(100 + \sum_{i=1}^{4}(x_i - 4)^2\right)$$

And,

$f(x) \geq 0 \; for \; all \; x_i$

$f(x) = 0 \; for \; all \; x_i = 1$

Here is a sample session with function *scout()*:


```
>> [bestX, bestFx] = scout(@mofx1, [0 0 0 0], [5 5 5 5], 0.1, 0.0001, 200,
250, 150, false, false, true)
New trust region is
Xlow -> 0.705616 0.691215 0.624721 0.579853
XHi -> 1.410948 1.466382 1.418702 1.520181
1.258371 0.866035 1.137281 0.969953 5.063244e+02
1.000000 0.999998 1.000000 1.000004 1.04425526997964921e-07,
step_factor=2.012630e-03
1.000000 1.000000 1.000000 1.000000 5.29414137727003066e-12,
step_factor=8.152521e-04
1.000000 1.000000 1.000000 1.000000 2.09003787634168205e-15,
step_factor=4.805189e-04
1.000000 1.000000 1.000000 1.000000 1.76806135434381320e-18,
step_factor=3.302325e-04
1.000000 1.000000 1.000000 1.000000 1.07135432140414841e-19,
step_factor=2.468721e-04
1.000000 1.000000 1.000000 1.000000 2.25446711083920170e-22,
step_factor=1.946428e-04
1.000000 1.000000 1.000000 1.000000 3.86342380571453126e-23,
step_factor=1.592046e-04
1.000000 1.000000 1.000000 1.000000 2.05832631900323443e-24,
step_factor=1.337666e-04
1.000000 1.000000 1.000000 1.000000 2.00097376461095084e-24,
step_factor=1.147247e-04
1.000000 1.000000 1.000000 1.000000 1.97470777795500180e-24,
step_factor=1.000000e-04

bestX =

    1.0000    1.0000    1.0000    1.0000


bestFx =

   1.9747e-24
```

The initial trust region is [0 0 0 0] and [5 5 5 5]. The function *scout()* is able to zoom in on the global minima at x = 1 for all four variables.

Listing A3.2 shows the source code for *mofx2.m*. The function in this file is similar to the first one, with the minima located much further apart at 1, 10, and 20.

```
function y = mofx2(x)
 n = length(x);
 s1 = 10;
 for i=1:n
   s1 = s1 + (x(i) - 1)^2;
 end

 s2 = 0;
 for i=1:n
   s2 = s2 + (x(i) - 10)^2;
 end
```

```
 s3 =100;
 for i=1:n
   s3 = s3 + (x(i) - 20)^2;
 end
 y = s1 * s2 * s3;
end
```

Listing A3.2 The source code in file mofx2.m.

The function in Listing A3.2 impllements:

$$f(x) = \left(\sum_{i=1}^{4}(x_i - 10)^2\right) * \left(10 + \sum_{i=1}^{4}(x_i - 1)^2\right) * \left(100 + \sum_{i=1}^{4}(x_i - 20)^2\right)$$

And,

$f(x) \geq 0 \; for \; all \; x_i$

$f(x) = 0 \; for \; all \; x_i = 1$

Here is a sample session with function *scout()*:

```
>> [bestX, bestFx] = scout(@mofx2, [0 0 0 0], [35 35 35 35], 0.1, 0.0001,
200, 250, 150, false, false, true)
New trust region is
Xlow -> 7.228352 6.610893 7.450960 7.216103
XHi -> 13.236237 13.186405 12.316169 12.939138
9.987684 10.426776 10.823724 9.887823 1.483749e+05
10.000015 10.000005 10.000008 9.999968 2.193556340189104143e-04,
step_factor=2.012630e-03
10.000000 10.000000 10.000000 10.000000 3.45399301801229403 8e-08,
step_factor=8.152521e-04
10.000000 10.000000 10.000000 10.000000 1.560354659808610745e-10,
step_factor=4.805189e-04
10.000000 10.000000 10.000000 10.000000 9.26397459262954250e-13,
step_factor=3.302325e-04
10.000000 10.000000 10.000000 10.000000 8.483165112284489357e-15,
step_factor=2.468721e-04
10.000000 10.000000 10.000000 10.000000 2.533395780019746942e-16,
step_factor=1.946428e-04
10.000000 10.000000 10.000000 10.000000 4.54397095313494504e-17,
step_factor=1.592046e-04
10.000000 10.000000 10.000000 10.000000 2.72284907448303825 1e-17,
step_factor=1.337666e-04
10.000000 10.000000 10.000000 10.000000 2.07877570453959474 3e-17,
step_factor=1.147247e-04
10.000000 10.000000 10.000000 10.000000 2.071114562582770785e-17,
step_factor=1.000000e-04

bestX =

   10.0000   10.0000   10.0000   10.0000
```



```
bestFx =

   2.0711e-17
```

The initial trust region is [0 0 0 0] and [35 35 35 35]. The function *scout()* is able to zoom in on the global minima at x = 10 for all four variables.

Listing A3.3 shows the source code for *mofx3.m*. The function in this file is similar to the first one, with the minima showing a limited level of overlapping.

```
function y = mofx3(x)
 n = length(x);
 s1 = 10;
 for i=1:n
   s1 = s1 + (x(i) - 2)^2;
 end

 s2 = 0;
 for i=1:n
   z = i;
   for j=1:4
     z = z + (i+j)/10^j;
   end
   s2 = s2 + (x(i) - z)^2;
 end

 s3 =100;
 for i=1:n
   s3 = s3 + (x(i) - 4)^2;
 end
 y = s1 * s2 * s3;
end
```

Listing A3.3 The source code in file mofx3.m.

The function in Listing A3.3 impllements:

$$f(x) = \left(\sum_{i=1}^{4}(x_i - z_i)^2\right) * \left(10 + \sum_{i=1}^{4}(x_i - 2)^2\right) * \left(100 + \sum_{i=1}^{4}(x_i - 4)^2\right)$$

Where:

$$z_i = i + \sum_{j=1}^{4}(i + j)/10^j$$

And,

$f(x) \geq 0 \; for \; all \; x_i$

Here is a sample session with function *scout()*:

```
>> [bestX, bestFx] = scout(@mofx3, [0 0 0 0], [5 5 5 5], 0.1, 0.0001, 200,
250, 150, false, false, true)
New trust region is
Xlow -> 0.815854 1.930265 3.045274 4.055484
XHi -> 1.649914 2.800310 3.821453 4.917229
1.268593 2.329499 3.431272 4.730066 6.356402e+01
1.234500 2.345598 3.456699 4.567798 1.377891023565209486e-08,
step_factor=2.012630e-03
1.234500 2.345600 3.456700 4.567800 1.861750455902988726e-12,
step_factor=8.152521e-04
1.234500 2.345600 3.456700 4.567800 3.19543785361241783le-15,
step_factor=4.805189e-04
1.234500 2.345600 3.456700 4.567800 6.759310953008199084e-18,
step_factor=3.302325e-04
1.234500 2.345600 3.456700 4.567800 2.352895648599663490e-20,
step_factor=2.468721e-04
1.234500 2.345600 3.456700 4.567800 3.574367305259031307e-24,
step_factor=1.946428e-04
1.234500 2.345600 3.456700 4.567800 7.185085189434991307e-26,
step_factor=1.592046e-04
1.234500 2.345600 3.456700 4.567800 1.700611879156208993e-27,
step_factor=1.337666e-04
1.234500 2.345600 3.456700 4.567800 0.000000000000000000e+00,
step_factor=1.147247e-04
1.234500 2.345600 3.456700 4.567800 0.000000000000000000e+00,
step_factor=1.000000e-04

bestX =

    1.2345    2.3456    3.4567    4.5678


bestFx =

     0
```

The initial trust region is [0 0 0 0] and [5 5 5 5]. The function *scout()* is able to zoom in on the global minima at [1.2345 2.3456 3.4567 4.5678].

Listing A3.4 shows the source code for *mofx4.m*. The function in this file is similar to the *mofx3()*, with the all the subfunctions having values greater than zero at the minima.

```
function y = mofx4(x)
 n = length(x);
 s1 = 50;
 for i=1:n
   s1 = s1 + (x(i) - 2)^2;
 end

 s2 = 1;
 for i=1:n
   z = i;
```



```
   for j=1:4
     z = z + (i+j)/10^j;
   end
   s2 = s2 + (x(i) - z)^2;
 end

 s3 =100;
 for i=1:n
   s3 = s3 + (x(i) - 4)^2;
 end
 y = s1 * s2 * s3;
end
```

Listing A3.3 The source code in file mofx3.m.

The function in Listing A3.1 impllements:

$$f(x) = \left(1 + \sum_{i=1}^{4}(x_i - z_i)^2\right) * \left(50 + \sum_{i=1}^{4}(x_i - 2)^2\right) * \left(100 + \sum_{i=1}^{4}(x_i - 4)^2\right)$$

Where:

$$z_i = i + \sum_{j=1}^{4}(i + j)/10^j$$

And,

$$f(x) > 5000 \; for \; all \; x_i$$

Here is a sample session with function *scout()*:

```
>> [bestX, bestFx] = scout(@mofx4, [0 0 0 0], [5 5 5 5], 0.1, 0.0001, 200,
250, 150, false, true, true)
1.218356 2.727668 3.266396 4.276397 8.073385e+03
1.271584 2.354481 3.437394 4.520234 6.568548495293742235e+03,
step_factor=2.012630e-03
1.271619 2.354494 3.437371 4.520244 6.568548481271061974e+03,
step_factor=8.152521e-04
1.271619 2.354495 3.437370 4.520246 6.568548481247506970e+03,
step_factor=4.805189e-04
1.271619 2.354495 3.437370 4.520246 6.568548481247242307e+03,
step_factor=3.302325e-04
1.271619 2.354495 3.437370 4.520246 6.568548481247221389e+03,
step_factor=2.468721e-04
1.271619 2.354495 3.437370 4.520246 6.568548481247218660e+03,
step_factor=1.946428e-04
1.271619 2.354495 3.437370 4.520246 6.568548481247218660e+03,
step_factor=1.592046e-04
1.271619 2.354495 3.437370 4.520246 6.568548481247218660e+03,
step_factor=1.337666e-04
1.271619 2.354495 3.437370 4.520246 6.568548481247218660e+03,
step_factor=1.147247e-04
```



```
1.271619 2.354495 3.437370 4.520246 6.56854848l247218660e+03,
step_factor=1.000000e-04

bestX =

    1.2716    2.3545    3.4374    4.5202


bestFx =

   6.5685e+03
```

The initial trust region is [0 0 0 0] and [5 5 5 5]. However, unlike with the previous versions of the mofx functions the global minima is NOT [1.2345 2.3456 3.4567 4.5678] as one might expect, but close to [1.27 2.35 3.43 4.52]. These results are due to the fact that none of the subfunctions in *mofx4()* go to zero at expected minim values. Moreover at [1.2345 2.3456 3.4567 4.5678] the second and third subfunctions contribute values that are greater than zero.

A4/ Effect of Trust Region Ranges on Optimized Function Values

You might ask the question if the trust region ranges have an impact of the optimized function values? This study has had two major versions. Adding regular upgrades for the population locations is part of version 2. Version 1 lacked this set of calculations and generated mean optimized function values that are significantly bigger than those of version 2. I performed the testing by avoiding the call to function *ZoomRange()*. The results with version 1 of the SOA functions showed a relationship between the optimized function values and the trust region ranges.

With uniform trust region ranges, the version 1 SOA function *scout()* gave the following approximate equation:

Mean Best Fx = $10^{-11} * (Delta\_X)^2$

With four different trust region ranges, the version 1 SOA function *scout()* gave the following approximate equation:

Mean Best Fx = $10^{-11.65059} * [\prod_{i=1}^{4} \sqrt{Delta\_X(i)}\ ]$

Switching from optimizing function *fx2* from 4 to 5 variables gave the following equation:

Mean Best Fx = $10^{-10.92824} * [\prod_{i=1}^{5} Delta\_X(i)^{0.4}\ ]$

What the above three equations have in common is that sum of the powers of the ranges is 2! The exponent of 10 is –11 or close to it!

Switching to testing the same concepts with version 2 of the SOA function *scout()* the correlation between the trust region ranges and optimized function values drops significantly. These results eliminate any credible correlation between the trust region ranges and optimized function values. I guess that reducing the optimized function values is akin to generating noise–type of optimized function values.

A5/ References

1. *Fireworks Algorithm: A Novel Swarm Intelligence Optimization Method*, by Ying Tan (author), published by Springer-Verlag Berlin Heidelberg 2015.
2. *The bare bones Fireworks algorithm: A minimalist global optimizer* by Junzhi Li, Ying Tan, Key Laboratory of Machine Perception (Ministry of Education), Department of Machine Intelligence, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, PR China. Applied Soft Computing 62 (2018) 454–462.
3. *Introduction to Fireworks Algorithm* by Ying Tan, Chao Yu, Shaoqiu Zheng, Ke Ding, School of Electronics Engineering and Computer Science, Peking University, Beijing, China.
4. *A Cooperative Framework for Fireworks Algorithm* by Shaoqiu Zheng, Student Member, IEEE, Junzhi Li, Student Member, IEEE, Andreas Janecek, and Ying Tan, Senior Member, IEEE, May 2015.
5. *Accelerating the Fireworks Algorithm with an Estimated Convergence Point* by Yu, Jun (Graduate School of Design, Kyushu University), Takagi, Hideyuki (Faculty of Design, Kyushu University), Tan, Ying (School of Electronics Engineering and Computer Science, Peking University). Kyushu University Institutional Repository. http://hdl.handle.net/2324/1932638.
6. *Analysis and Improvement of Fireworks Algorithm* by Xi-Guang Li, Shou-Fei Han, and Chang-Qing Gong, School of Computer, Shenyang Aerospace University, Shenyang 110136, China; lixiguang@sau.edu.cn (X.-G.L.); gongchangqing@sau.edu.cn. Correspondence :hanshoufei@gmail.com. Published: 17 February 2017.
7. *Dynamic Search in Fireworks Algorithm* by Shaoqiu Zheng, Andreas Janecek, Junzhi Li and Ying Tan.
8. *Dynamic Search Fireworks Algorithm with Covariance Mutation for Solving the CEC 2015 Learning Based Competition Problems* by Chao Yu[1], Ling

Chen Kelley[2,3], and Ying Tan[1,*]. [1]The Key Laboratory of Machine Perception and Intelligence (Ministry of Education), Department of Machine Intelligence, School of Electronics Engineering and Computer Science, Peking University, Beijing, China, 100871. [2]Department of History, School of Humanities, Tsinghua University. Beijing, China, 100084. [3]Department of World Languages and Cultures, College of Liberal Arts and Sciences, Auburn University at Montgomery, Montgomery, AL, United States 36117. Email: chaoyu@pku.edu.cn, lingchenkelley@gmail.com, ytan@pku.edu.cn.

9. *Exponentially Decreased Dimension Number Strategy Based Dynamic Search Fireworks Algorithm for Solving CEC2015 Competition Problems* by Shaoqiu Zheng, Chao Yu, Junzhi Li and Ying Tan, Department of Machine Intelligence, School of Electronics Engineering and Computer Science, Peking University. Key Laboratory of Machine Perception (Ministry of Education), Peking University, Beijing,100871, P.R. China. Email: f zhengshaoqiu, chaoyu, ljz, ytan g @pku.edu.cn.
10. *Enhanced Fireworks Algorithm* by Shaoqiu Zheng[1], Andreas Janecek[2] and Ying Tan[1]. [1]Department of Machine Intelligence, School of Electronics Engineering and Computer Science, Peking University Key Laboratory of Machine Perception (Ministry of Education), Peking University, Beijing,100871, P.R. China. Email: zhengshaoqiu@pku.edu.cn, ytan@pku.edu.cn. [2]University of Vienna, Research Group Entertainment Computing, 1090 Vienna, Austria Email: andreas.janecek@univie.ac.at. Paper from 2013 IEEE Congress on Evolutionary Computation June 20-23, Cancún, México.
11. *Fireworks Algorithm for Unconstrained Function Optimization Problems* by Evans Baidoo, Kwame Nkrumah University of Science and Technology, Department of Computer Science, PMB, KNUST, Ghana, Email: ebaidoo2.cos@st.knust.edu.gh. Applied Computer Science, vol. 13, no. 1, pp. 61–74.
12. *Fireworks Algorithm with Enhanced Fireworks Interaction* by Bei Zhang, Yu-Jun Zheng, Min-Xia Zhang, and Sheng-Yong Chen. IEEE/ACM Transactions On Computational Biology And Bioinformatics, Vol. 14, No. 1, January/February 2017.
13. *The Effect of Information Utilization: Introducing a Novel Guiding Spark in the Fireworks Algorithm* by Junzhi Li, Shaoqiu Zheng, and Ying Tan, Senior Member, IEEE. IEEE Transactions on Evolutionary Computation, Vol. 21, No. 1, February 2017.

14. *Adaptive Fireworks Algorithm* by Junzhi Li, Shaoqiu Zheng and Ying Tan. 2014 IEEE Congress on Evolutionary Computation (CEC) July 6-11, 2014, Beijing, China. Authors are with the Key Laboratory of Machine Perception (Ministry of Education), Department of Machine Intelligence, School of Electronics Engineering and Computer Science, Peking University, Beijing, 100871, P.R. China. email:{ ljz, zhengshaoqiu, ytan } @pku.edu.cn.
15. *Loser-out Tournament Based Fireworks Algorithm for Multi-modal Function Optimization* by Junzhi Li and Ying Tan, Senior Member, IEEE Junzhi Li and Ying Tan are with the Key Laboratory of Machine Perception (Ministry of Education), Department of Machine Intelligence, School of Electronics Engineering and Computer Science, Peking University, Beijing, 100871, P.R. China. Prof. Y. Tan is the corresponding author. This work was supported by the Natural Science Foundation of China (NSFC) under grant no. 61673025 and 61375119 and supported by Beijing Natural Science Foundation (4162029), and partially supported by National Key Basic Research Development Plan (973 Plan) Project of China under grant no. 2015CB352302. Email:{ljz,ytan}@pku.edu.cn.
16. *Optimal Choice of Parameters for Fireworks Algorithm* by Vijay Kumar[a*], Jitender Kumar Chhabra[b], Dinesh Kumar[c]. [a]Computer Science and Engineering Department, Thapar University, Patiala, Punjab, India. [b]Computer Engineering Department, National Institute of Technology, Kurukshetra, Haryana, India. [c]Computer Science and Engineering Department, GJUS&T, Hisar, Haryana, India. E-mail address: vijaykumarchahar@gmail.com. 4th International Conference on Eco-friendly Computing and Communication Systems, ICECCS 2015.

A6/ 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.

I created this document using Microsoft Word. The autocorrection feature was turned on. However, since this feature encountered too many words in the listings that it considered misspelled, Word stopped flagging misspelled words in the entire text! I have tried to manually detect these spelling errors and typos. I apologize if I missed one, or two, or three, or four hundred! You get my meaning! Despite the misspellings I strived to present accurate information.

Version	***Date***	***Comments***
0.9.0/1.0.0	January 15, 2019	Initial release.
1.1.0	January 20, 2019	1. Added a reference section. 2. Clarified general explanation of the Scout algorithm. 3. Clarified handling difficult multi-objective functions. 4. Added MATLAB source code for file *mc.m* in Listing A3.4.
1.1.1	January 23, 2019	1. Edited the A4 section.

