Recursive Evolutionary Optimization Strategy

By
Namir Clement Shammas

Contents

Introduction

This study presents a *new strategy* to be used with the rich class of evolutionary optimization algorithms.

Evolutionary optimization algorithms (EOA) are iterative algorithms that search for optimum points using steps that mimic the behavior of genes, animals, insects, and even natural phenomena. The EOAs are notorious for NOT requiring the calculations of the derivatives of the optimized functions and don't require the solution of simultaneous equations. While the EOA use matrices to maintain *swarm* populations (representing rows of possible solutions) they DO NOT perform matrix inversion. As such, coding the EOA algorithm uses simpler calculations.

Typically, EOAs use trust regions that are defined as user-selected ranges for each optimized variable. The trust regions should be wide enough to locate the optimum

values of the variables. These regions can be a double-edged sword. Making them very wide to ensure capturing the optimum also dilutes the search and reduces the efficiency of that search. By contrast, making the trust region narrow may zoom in on local optimum points, while missing the global ones.

This study presents a general approach to handle the trust region searches more efficiently. The new approach should work with most EOAs. The examples presented in this study are coded in MATLAB and use the popular Particle Swarm Optimization (PSO) algorithm. I will not discuss the details of the PSO since the internet has many articles that look into the algorithm steps.

Typical EOA Parameters

Typically, the functions that perform evolutionary optimization require **at least** the following parameters:

1. The name, pointer, or handle to the optimized function.
2. The arrays of the lower and upper bounds for each optimized variable. These two arrays form the initial trust region.
3. The population size used to hold random values for the optimized variables. The optimization code builds a population matrix. The matrix columns represent the different optimization variables. The matrix rows represent different sets of values.
4. The maximum number of iterations used to search for the optimum.

Of course, there may well be additional parameters that are needed to operate or fine tune the tasks of the optimizing function.

The output of EOA functions must report the array of the optimum point. Other optional output parameters are the best function value and the number of iterations (if the implementation performs an early exit).

Zooming in on the Trust Region

Over several years of studying various EOAs I have come to refine the trust search regions using the following approaches (listed chronologically):

- Taking the best optimized point from one set of calculations and using it to create a smaller trust region to use in rerunning the optimization calculations. I did this task manually.

- Coding the optimization function to make two runs. The first run uses the initial trust region. The second run (with the same number of iterations as the first run) uses a narrower trust region based on the optimum values obtained in the first run. I typically use a trust region based on adding and subtracting 15% of the optimum values from the first run. Based on the optimized function, I suggest a range reduction factor between 5% and 20%.
- The recursion schemes. In these schemes we have the EOA function, that your code calls, performing its search in the initial trust region. The function branches off by making a *single-level recursive call* to search in a narrower trust region around the current best candidate for the optimum point. The main function performs the recursive call every time a better candidate for the optimum point is found. Alternatively, the main function can make that recursive call in each iteration. I prefer the first option since it uses a reasonable number of *total* iterations.
    a. Using recursion to get a single candidate for a better optimum value. This task may occur in each iteration or only when a better candidate for the optimum point is obtained. This scheme is easier to code since the single-level recursion function call returns a single set of values for the variables and its related function value. I will call this approach the *simple recursion*.
    b. Using *enhanced recursive* and *quasi-recursive calls* to the optimization function. This task may occur in each iteration or only when a better candidate for the optimum point is obtained. In both recursion types, the single-level recursive call returns an additional population that is merged with the main population. The code then sorts the extended population using their function values and then clips the extended population to its original size. This approach allows the introduction of *multiple better* sets of values to the general population.

In the case of recursive calls, the function calls itself. The code for the function must distinguish between the first call and the recursive call. The example of Particle Swarm Optimization will demonstrate how recursion works. Enhanced recursive calls work with programming languages that do not enforce data typing like MATLAB and Python.

Making quasi-recursive calls uses *paired twin* functions. The main function calls a similar server function to simulate recursion. The server function has similar code

but may have different input and output parameters. Quasi-recursion works with languages that support strong typing of parameters like C++, C#, Java, and Visual Basic.

I will demonstrate the enhanced recursion and quasi-recursion schemes using MATLAB examples applied to the Particle Swarm Optimization algorithm. You can implement the same schemes to other EOAs.

Simple Recursion

This section presents code for implementing a simple recursion where the recursive function returns the same kind of results each time it is called. Such functions don't use programming tricks.

Here is the code for function recPSO2() which implements a simple recursion:

```
function [zbestX,zbestFx] = recPSO2(fx,Lb,Ub,MaxPop,MaxIters, ...
                          nestedMaxIters,breedingProbab,...
                          bNestedCallInEachIteration,bRecursiveCall)
% recPSO2 implements recursive particle swarm optimization. Includes
% breeding feature.
%
% Note: Performs very well.
%
% Reference
% =========
%
% An Analysis of Particle Swarm Optimizers by Frans van den Bergh
%
%
% INPUT
% ======
% fx - handle of optimized function.
% Lb - array of low bound values.
% Ub - array of upper bound values.
% MaxPop - maximum population of swarm.
% MaxIters - maximum number of iterations
% nestedMaxIters - maximum iterations in nested call to PSO function
% breedingProbab - breeding probability
% bNestedCallInEachIteration - flag to call nested to PSO function in each
% iteration.
% bRecursiveCall - recursive call flag.
%
% OUTPUT
% ======
% zbestX - array of best solutions.
% zbestFx - best optimized function value.
%
% Example
% =======
```



```
%
% >> [bestX,bestFx] = recPSO2(@fx3,[0 0 0 0],5*[5 5 5
5],100,100,100,0.4,false)
%
  global bestX bestFx
  if nargin < 9, bRecursiveCall = false; end
  if nargin < 8, bNestedCallInEachIteration = false; end
  if nargin < 7, breedingProbab = 0.4; end
  if nargin < 6, nestedMaxIters = fix(MaxIters/2); end
  if breedingProbab > 1, breedingProbab = breedingProbab/100; end
  n = length(Lb);
  m = n + 1;
  pop = 1e+99+zeros(MaxPop,m);
  pop2 = pop;
  aPop = zeros(1,n);
  vel = zeros(MaxPop,n);

  % Initizialize population
  for i=1:MaxPop
    pop(i,1:n) = Lb + (Ub - Lb) .* rand(1,n);
    vel(i,1:n) = (Ub - Lb) / 10 .* (2*rand(1,n)-1);
    pop(i,m) = fx(pop(i,1:n));
    pop2(i,:) = pop(i,:);
    aPop(1:n) = Lb + (Ub - Lb) .* rand(1,n);
    f0 = fx(aPop);
    if f0 < pop2(i,m)
      pop2(i,1:n) = aPop(1:n);
      pop2(i,m) = f0;
    end
  end

  pop = sortrows(pop,m);
  pop2 = pop;

  fprintf('Best Fx = %e, ', pop(1,m));
  fprintf('Best X = [');
  fprintf(' %f,', pop(1,1:n));
  fprintf("]\n");
  if ~bRecursiveCall
    bestFx = pop(1,m);
    bestX = pop(1,1:n);
  end

  % pso loop
  for iter = 1:MaxIters
    IterFactor = sqrt((iter - 1)/(MaxIters - 1));
    w = 1 - 0.3 * IterFactor;
    c1 = 2 - 1.9 * IterFactor;
    c2 = 2 - 1.9 * IterFactor;

    for i=2:MaxPop
      for j=1:n
        vel(i,j) = w*vel(i,j) + c1*rand*(pop(1,j) - pop(i,j)) + ...
          c2*rand*(pop2(i,j) - pop(i,j));
        p = pop(i,j) + vel(i,j);

        if p < Lb(j) || p > Ub(j)
```



```
      pop(i,j) = Lb(j) + (Ub(j) - Lb(j))*rand;
    else
      pop(i,j) = p;
    end
  end

  pop(i,m) = fx(pop(i,1:n));

  % find new global best?
  if pop(1,m) > pop(i,m)
    pop(1,:) = pop(i,:);
    % find new local best?
  elseif pop(i,m) < pop2(i,m)
    pop2(i,:) = pop(i,:);
  end

  % breeding step here
  if breedingProbab > rand
    k = i + fix((MaxPop-i)*rand);
    r = rand;
    for j=1:n
      xa = pop(i,j);
      xb = pop(k,j);
      va = vel(i,j);
      vb = vel(k,j);
      pop(i,j) = r*xa + (1-r)*xb;
      pop(k,j) = r*xb + (1-r)*xa;
      vel(i,j) = (va+vb)/abs(va+vb)*abs(va);
      vel(k,j) = (va+vb)/abs(va+vb)*abs(vb);
    end
    pop(i,m) = fx(pop(i,1:n));
    pop(k,m) = fx(pop(k,1:n));
  end

end

[pop,Idx] = sortrows(pop,m);
pop2 = sortrows(pop2,m);
vel = vel(Idx,:);

bFoundBetter = false;
if bestFx > pop(1,m)
  fprintf('Best Fx = %e, Best X = [', pop(1,m));
  fprintf(' %f,', pop(1,1:n));
  fprintf("]\n");
  bestFx = pop(1,m);
  bestX = pop(1,1:n);
  bFoundBetter = true;
end

if bFoundBetter || bNestedCallInEachIteration
  Llb = Lb;
  Uub = Ub;
  for i=1:length(bestX)
    Llb(i) = narrowLowerLimit(bestX(i),0.15);
    Uub(i) = narrowUpperLimit(bestX(i),0.15);
  end
```



```
      [bestX2,bestFx2] =
recPSO2(fx,Llb,Uub,MaxPop,nestedMaxIters,0,breedingProbab,false,true);
      aPop = [bestX2 bestFx2];
      pop = [pop; aPop];
      pop = sortrows(pop, m);
      pop = pop(1:MaxPop,:); % discard entry at row index MaxPop+1
      if bestFx > pop(1,m)
        bestFx = pop(1,m);
        bestX = pop(1,1:n);
      end
    end
  end

   zbestFx= pop(1,m);
   zbestX = pop(1,1:n);
end

function x = narrowUpperLimit(x,factor)
  if nargin < 2, factor = 0.15; end
  if x > 0
    x = (1+factor)*x;
  else
    x = (1-factor)*x;
  end
end

function x = narrowLowerLimit(x,factor)
  if nargin < 2, factor = 0.15; end
  if x < 0
    x = (1+factor)*x;
  else
    x = (1-factor)*x;
  end
end
```

The list of input parameters is:

- fx is the handle of optimized function.
- Lb is the array of low bound values.
- Ub is the array of upper bound values.
- MaxPop is the maximum population of swarm.
- MaxIters is the maximum number of iterations
- nestedMaxIters is the maximum iterations in nested call to PSO function
- breedingProbab is the breeding probability.
- bNestedCallInEachIteration is the flag to call nested to PSO function in each iteration.
- bRecursiveCall is the recursive call flag. Your call to function recPSO2() must supply a false Boolean value to this parameter.

The list of output parameters is:

- zbestX is the array of best solutions.
- zbestFx is the best optimized function value.

The function recPSO2() declares the global variables bestX and bestFx to allow the recursive function call to access the best solution found so far. The function recPSO2() uses an augmented matrix to combine storing the values of the optimized variables and the function values for each row. This storage scheme makes sorting the population matrix easier.

The following if statement performs the recursion:

```
    if bFoundBetter || bNestedCallInEachIteration
      Llb = Lb;
      Uub = Ub;
      for i=1:length(bestX)
        Llb(i) = narrowLowerLimit(bestX(i),0.15);
        Uub(i) = narrowUpperLimit(bestX(i),0.15);
      end
      [bestX2,bestFx2]                                                    =
recPSO2(fx,Llb,Uub,MaxPop,nestedMaxIters,0,breedingProbab,false,true);
      aPop = [bestX2 bestFx2];
      pop = [pop; aPop];
      pop = sortrows(pop, m);
      pop = pop(1:MaxPop,:); % discard entry at row index MaxPop+1
      if bestFx > pop(1,m)
        bestFx = pop(1,m);
        bestX = pop(1,1:n);
      end
```

The first few statements in the if clause calculate a local narrow trust region. The code then recursively calls the function. This call uses arguments for the narrow trust region. It also passes the value of nestedMaxIters and 0 to the parameters MaxIters and nestedMaxIters, respectively. The recursive call returns the array bestX2 and the scalar bestFx2. The code combines these values to form an augmented population row. Next, the code merges that row with the population matrix, sorts the matrix (using the function values in the last column), and then retains the first MaxPop rows (discarding the last row). The nested if statement checks whether the first augmented population row has a better solution for the optimum. If so, the code updates the values of bestX and bestFx. Keep in mind that if the recursive call returns a good (but not the best) result, that result is stored in the augmented population matrix and NOT discarded outright.

Here is a simple session to get the optimum point of function fx1(). The lower and upper trust region ranges are [0 0 0 0] and [5 5 5 5], respectively. The function call specifies a population of 100, 100 maximum iterations, 100 iterations for the recursive calls, a breeding probability of 0.4, and a false value for the recursion flag.

```
>> [bestX,bestFx] = recPSO2(@fx1,[0 0 0 0],[5 5 5 5],100,100,100,0.4,false)
Best Fx = 9.811605e-01, Best X = [ 1.111017, 2.360093, 2.125812, 3.726204,]
Best Fx = 1.718592e-01, Best X = [ 1.152478, 1.867598, 3.279981, 3.770457,]
Best Fx = 3.252907e-02, Best X = [ 1.114738, 2.093046, 3.034145, 4.097677,]
Best Fx = 1.457430e-02, Best X = [ 1.063033, 1.989369, 2.920961, 4.065122,]
Best Fx = 2.694269e-02, Best X = [ 0.940596, 1.996285, 2.872746, 3.915109,]
Best Fx = 1.365695e-02, Best X = [ 1.003310, 1.885361, 2.981102, 3.987885,]
Best Fx = 1.312178e-02, Best X = [ 0.928562, 1.941056, 3.047821, 3.952491,]
Best Fx = 7.403889e-03, Best X = [ 1.023199, 2.052501, 2.993456, 4.063769,]
Best Fx = 1.445686e-02, Best X = [ 0.946430, 1.981148, 2.944690, 3.909598,]
Best Fx = 4.967772e-03, Best X = [ 1.055125, 1.966282, 2.987049, 4.024988,]
Best Fx = 1.448494e-02, Best X = [ 1.056113, 1.977581, 2.995271, 3.896023,]
Best Fx = 4.678212e-03, Best X = [ 1.036978, 1.949450, 2.979672, 4.018501,]
Best Fx = 3.978620e-02, Best X = [ 0.985595, 2.091527, 3.159149, 4.076637,]
Best Fx = 1.279093e-03, Best X = [ 0.990805, 1.993228, 3.015394, 3.969806,]
Best Fx = 2.019708e-02, Best X = [ 1.041155, 1.963613, 2.923828, 3.893337,]
Best Fx = 6.601702e-04, Best X = [ 1.007770, 2.016582, 2.988799, 3.985881,]
Best Fx = 1.860988e-02, Best X = [ 0.911325, 1.995672, 3.005006, 3.896545,]
Best Fx = 6.549821e-04, Best X = [ 1.004146, 1.998429, 2.994093, 3.975496,]
Best Fx = 1.778701e-02, Best X = [ 0.884307, 1.972711, 2.957195, 4.042721,]
Best Fx = 1.873597e-04, Best X = [ 1.001048, 2.008670, 3.009210, 3.994875,]
Best Fx = 1.278379e-02, Best X = [ 0.975075, 1.989958, 3.089836, 3.936825,]
Best Fx = 1.829876e-04, Best X = [ 1.010887, 1.999180, 3.005647, 4.005648,]
Best Fx = 2.941136e-02, Best X = [ 1.019497, 1.931705, 3.152842, 4.031721,]
Best Fx = 1.634967e-04, Best X = [ 0.994330, 2.007044, 2.993137, 3.994116,]
Best Fx = 1.861451e-02, Best X = [ 0.934971, 2.065416, 3.098045, 4.022215,]
Best Fx = 6.398526e-05, Best X = [ 1.004630, 1.997658, 2.994640, 4.002887,]
Best Fx = 1.644240e-02, Best X = [ 0.957592, 2.083743, 3.058153, 4.065186,]
Best Fx = 4.017997e-05, Best X = [ 1.001523, 2.000281, 2.996156, 3.995203,]
Best Fx = 2.152210e-05, Best X = [ 0.998866, 2.000210, 2.996843, 3.996802,]
Best Fx = 2.764565e-02, Best X = [ 1.091186, 1.937191, 2.995939, 3.876027,]
Best Fx = 1.297611e-05, Best X = [ 1.001198, 2.002096, 2.997404, 3.999360,]
Best Fx = 2.170713e-02, Best X = [ 0.980008, 2.043606, 3.000821, 4.139303,]
Best Fx = 6.295846e-06, Best X = [ 1.001943, 1.999882, 2.998745, 3.999034,]
Best Fx = 2.868705e-02, Best X = [ 1.013109, 2.068730, 3.103325, 3.885478,]

bestX =

    0.9993    1.9985    3.0040    3.9995


bestFx =

   1.9062e-05
```

With 100 maximum iterations and 100 nested iterations, the above results are close to the correct answer. Let's see if we can do better with 1000 maximum iterations.


```
>> [bestX,bestFx] = recPSO2(@fx1,[0 0 0 0],[5 5 5 5],100,1000,100,0.4,false)
Best Fx = 9.635043e-01, Best X = [ 0.593191, 2.273267, 2.158966, 3.873520,]
Best Fx = 9.441884e-02, Best X = [ 0.998887, 1.846702, 2.818696, 4.195054,]
Best Fx = 4.752950e-03, Best X = [ 1.025422, 2.016730, 2.985442, 3.939877,]
Best Fx = 4.752950e-03, Best X = [ 1.025422, 2.016730, 2.985442, 3.939877,]
Best Fx = 1.407968e-02, Best X = [ 0.917231, 2.054782, 2.940605, 3.973541,]
Best Fx = 3.365945e-03, Best X = [ 0.979200, 2.018389, 3.045357, 4.023193,]
Best Fx = 9.009928e-03, Best X = [ 0.974504, 2.067060, 3.060254, 3.984760,]
Best Fx = 2.465077e-03, Best X = [ 0.969971, 2.030305, 2.976994, 4.010755,]
Best Fx = 3.033980e-02, Best X = [ 1.047388, 1.980228, 3.161385, 3.959279,]
Best Fx = 2.131279e-03, Best X = [ 0.997576, 2.023907, 3.033875, 3.979842,]
Best Fx = 6.845861e-03, Best X = [ 1.042023, 2.071199, 3.002317, 3.997697,]
Best Fx = 1.978269e-03, Best X = [ 1.028232, 2.009838, 3.023373, 3.976802,]
Best Fx = 4.824348e-03, Best X = [ 0.986312, 1.933295, 2.992000, 3.988888,]
Best Fx = 1.148278e-03, Best X = [ 1.009824, 1.983621, 2.973530, 3.990899,]
Best Fx = 2.798356e-02, Best X = [ 1.066380, 1.937437, 2.860367, 4.012874,]
Best Fx = 4.621404e-04, Best X = [ 1.004865, 2.015382, 2.993051, 3.987608,]
Best Fx = 8.621671e-03, Best X = [ 0.931544, 1.986966, 2.948776, 4.033789,]
Best Fx = 4.481457e-04, Best X = [ 1.014181, 2.013373, 2.993403, 4.004968,]
Best Fx = 1.485156e-02, Best X = [ 0.901591, 2.068876, 3.019895, 4.005257,]
Best Fx = 1.048541e-04, Best X = [ 0.997566, 2.000867, 3.008679, 3.995219,]
Best Fx = 5.474271e-03, Best X = [ 1.037120, 2.036753, 2.956919, 4.029825,]
Best Fx = 1.857628e-05, Best X = [ 0.999661, 1.998600, 3.002042, 4.003512,]
Best Fx = 4.536142e-03, Best X = [ 1.001321, 2.021063, 3.055725, 4.031392,]
Best Fx = 1.735725e-05, Best X = [ 1.000601, 2.000133, 3.003373, 4.002366,]
Best Fx = 1.179851e-02, Best X = [ 0.964782, 2.007952, 2.933603, 3.921984,]
Best Fx = 1.362275e-05, Best X = [ 0.999418, 2.001470, 3.002925, 3.998398,]
Best Fx = 1.918151e-02, Best X = [ 1.033611, 1.955536, 2.886777, 4.057056,]
Best Fx = 1.170927e-05, Best X = [ 0.999967, 2.001022, 2.999353, 4.003201,]
Best Fx = 3.749524e-02, Best X = [ 0.910673, 2.039080, 2.833639, 4.017681,]
Best Fx = 2.291554e-06, Best X = [ 1.001406, 2.000323, 3.000170, 3.999573,]
Best Fx = 1.796457e-02, Best X = [ 1.077562, 1.922181, 3.076110, 4.010014,]
Best Fx = 1.383631e-06, Best X = [ 0.999475, 1.999545, 2.999061, 3.999860,]
Best Fx = 1.059110e-02, Best X = [ 1.006009, 2.053616, 3.039110, 4.078426,]
Best Fx = 9.563070e-07, Best X = [ 1.000636, 1.999696, 3.000511, 3.999554,]
Best Fx = 1.842789e-02, Best X = [ 1.096913, 1.987526, 3.080589, 3.951157,]
Best Fx = 7.119332e-07, Best X = [ 1.000131, 2.000395, 2.999619, 3.999373,]
Best Fx = 1.489478e-02, Best X = [ 1.002699, 2.058593, 3.085912, 4.063824,]
Best Fx = 3.975404e-07, Best X = [ 1.000048, 1.999785, 3.000581, 3.999896,]
Best Fx = 6.833265e-03, Best X = [ 0.973007, 2.013556, 2.925928, 4.020839,]
Best Fx = 1.793322e-07, Best X = [ 0.999773, 1.999696, 3.000184, 3.999962,]
Best Fx = 1.743265e-02, Best X = [ 1.091735, 2.028307, 2.947744, 4.074064,]
Best Fx = 6.940801e-08, Best X = [ 0.999763, 2.000060, 2.999973, 4.000095,]
Best Fx = 3.371793e-02, Best X = [ 1.057244, 1.889478, 2.891219, 3.920045,]
Best Fx = 5.155273e-08, Best X = [ 1.000144, 2.000015, 3.000008, 3.999825,]
Best Fx = 1.407712e-02, Best X = [ 1.110250, 1.999607, 3.019242, 4.039391,]
Best Fx = 2.663989e-08, Best X = [ 0.999977, 1.999870, 3.000060, 4.000074,]
Best Fx = 1.389541e-02, Best X = [ 0.984885, 1.925602, 2.981173, 4.088190,]
Best Fx = 1.048096e-08, Best X = [ 0.999941, 2.000062, 2.999943, 4.000003,]
Best Fx = 1.392033e-02, Best X = [ 0.998398, 1.934350, 3.097125, 4.013216,]
Best Fx = 3.607433e-09, Best X = [ 1.000056, 1.999997, 2.999979, 3.999993,]
Best Fx = 1.931769e-02, Best X = [ 0.959017, 2.120935, 2.980826, 3.948567,]
Best Fx = 2.803026e-09, Best X = [ 0.999981, 1.999963, 2.999989, 4.000031,]
Best Fx = 2.101225e-02, Best X = [ 1.042942, 1.955473, 2.920829, 3.895513,]
Best Fx = 9.030495e-10, Best X = [ 0.999990, 1.999992, 3.000025, 4.000011,]
Best Fx = 5.157382e-03, Best X = [ 0.948930, 1.991608, 3.036792, 3.966456,]
Best Fx = 2.823119e-10, Best X = [ 0.999987, 2.000001, 2.999990, 3.999998,]
```

```
Best Fx = 6.636904e-03, Best X = [ 0.977669, 1.991832, 2.923530, 4.014964,]
Best Fx = 1.822469e-10, Best X = [ 1.000004, 2.000000, 3.000011, 3.999994,]
Best Fx = 2.654641e-02, Best X = [ 1.105149, 1.892294, 2.976030, 4.057575,]
Best Fx = 1.036536e-10, Best X = [ 1.000001, 1.999994, 2.999997, 3.999993,]
Best Fx = 1.614271e-02, Best X = [ 0.909854, 1.939143, 3.061667, 4.022584,]
Best Fx = 4.432544e-11, Best X = [ 0.999999, 2.000001, 3.000004, 4.000005,]
Best Fx = 2.373744e-02, Best X = [ 1.050356, 2.093848, 3.026587, 4.108109,]
Best Fx = 3.309644e-11, Best X = [ 0.999996, 2.000003, 3.000003, 3.999998,]
Best Fx = 2.571785e-02, Best X = [ 1.063191, 1.877592, 2.990709, 4.081577,]
Best Fx = 1.631686e-11, Best X = [ 1.000001, 2.000001, 2.999998, 4.000003,]
Best Fx = 1.271924e-02, Best X = [ 1.031985, 1.993030, 3.017029, 3.893428,]
Best Fx = 1.277572e-11, Best X = [ 0.999997, 2.000002, 3.000002, 4.000000,]
Best Fx = 1.976654e-02, Best X = [ 1.071873, 1.995661, 3.026099, 4.117902,]
Best Fx = 4.144325e-12, Best X = [ 1.000000, 1.999999, 2.999999, 3.999999,]
Best Fx = 1.852808e-02, Best X = [ 0.884590, 2.051034, 2.949607, 4.008045,]
Best Fx = 1.891707e-12, Best X = [ 1.000000, 2.000001, 3.000000, 4.000001,]
Best Fx = 3.609088e-02, Best X = [ 1.023564, 1.927039, 2.874792, 3.879438,]
Best Fx = 9.296118e-13, Best X = [ 1.000000, 1.999999, 2.999999, 4.000000,]
Best Fx = 1.502449e-02, Best X = [ 1.060911, 1.970277, 2.961508, 4.094600,]
Best Fx = 1.944805e-13, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.157395e-02, Best X = [ 0.946724, 2.016906, 2.918619, 4.042742,]
Best Fx = 8.066589e-14, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.998150e-02, Best X = [ 0.975409, 1.920332, 2.994548, 3.885982,]
Best Fx = 7.706452e-14, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.540387e-02, Best X = [ 0.959984, 1.997897, 2.882543, 4.001389,]
Best Fx = 2.128512e-14, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 4.082747e-02, Best X = [ 0.977388, 2.157329, 2.930648, 3.896298,]
Best Fx = 1.748554e-14, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.640080e-02, Best X = [ 1.015530, 2.001755, 3.056479, 4.113871,]
Best Fx = 5.186184e-15, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.947113e-02, Best X = [ 0.990023, 1.961463, 3.133738, 4.000831,]
Best Fx = 2.134975e-15, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.576604e-02, Best X = [ 1.078372, 1.981070, 3.095869, 4.008637,]
Best Fx = 1.999442e-15, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 4.036522e-02, Best X = [ 1.006248, 2.011578, 2.803635, 3.959591,]
Best Fx = 1.677990e-15, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 2.108949e-02, Best X = [ 0.988307, 2.080737, 3.012011, 3.880459,]
Best Fx = 2.932564e-16, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.328781e-02, Best X = [ 0.994290, 1.957106, 2.955781, 3.902738,]
Best Fx = 1.567227e-16, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 3.273797e-02, Best X = [ 1.148870, 1.924103, 3.062626, 3.970109,]
Best Fx = 1.042075e-16, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.445468e-02, Best X = [ 1.008261, 1.953572, 2.948739, 3.902004,]
Best Fx = 6.159364e-17, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.171386e-02, Best X = [ 1.003341, 1.935733, 3.086933, 3.996107,]
Best Fx = 3.085870e-17, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 6.044459e-03, Best X = [ 0.959706, 1.948620, 3.029779, 3.970098,]
Best Fx = 2.050574e-17, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 5.169621e-03, Best X = [ 0.982180, 2.043725, 3.017079, 3.948536,]
Best Fx = 1.118426e-17, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 2.442309e-02, Best X = [ 1.001844, 1.880275, 2.919028, 3.940592,]
Best Fx = 6.874890e-18, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.839255e-02, Best X = [ 1.030750, 1.946208, 3.077677, 4.092302,]
Best Fx = 4.719868e-18, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 2.692293e-02, Best X = [ 0.912121, 2.041987, 3.131955, 4.005022,]
Best Fx = 3.911095e-18, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 2.309725e-02, Best X = [ 0.945593, 2.099889, 2.954786, 4.090083,]
```



```
Best Fx = 2.148406e-18, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.335266e-02, Best X = [ 1.018451, 2.040178, 3.105542, 4.016091,]
Best Fx = 2.080961e-18, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 2.474791e-02, Best X = [ 1.050891, 1.908592, 3.088301, 3.922505,]
Best Fx = 8.682302e-19, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.410616e-02, Best X = [ 1.031156, 1.945267, 3.063546, 4.078113,]
Best Fx = 8.269917e-19, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.768939e-02, Best X = [ 1.064222, 2.036122, 3.107839, 4.025115,]
Best Fx = 4.594895e-19, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 8.075740e-03, Best X = [ 1.064287, 1.958649, 2.954853, 3.986045,]
Best Fx = 2.722602e-19, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.404485e-02, Best X = [ 0.958791, 1.997718, 2.944305, 3.903878,]
Best Fx = 1.170651e-19, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 2.472079e-02, Best X = [ 1.039340, 2.033519, 3.075485, 3.872127,]
Best Fx = 9.262421e-20, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.976364e-02, Best X = [ 1.021858, 1.936567, 2.957405, 3.884036,]
Best Fx = 8.614041e-20, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.713256e-02, Best X = [ 1.065279, 2.103096, 2.963457, 3.969883,]
Best Fx = 2.059310e-20, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.035582e-02, Best X = [ 1.068698, 2.052439, 2.946497, 4.004894,]
Best Fx = 1.076233e-20, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 9.406958e-03, Best X = [ 1.024628, 1.922699, 3.052983, 4.004222,]
Best Fx = 9.496918e-21, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 2.917439e-02, Best X = [ 1.032167, 1.847247, 3.060409, 4.034015,]
Best Fx = 3.178424e-21, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.789425e-02, Best X = [ 0.915406, 2.037046, 3.008507, 4.096402,]
Best Fx = 2.512474e-21, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.997655e-02, Best X = [ 0.935606, 1.959769, 2.944272, 3.894616,]
Best Fx = 1.132755e-21, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.947771e-02, Best X = [ 1.068906, 2.024117, 2.911400, 3.920639,]
Best Fx = 6.522768e-22, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 2.452327e-02, Best X = [ 1.138039, 1.968216, 2.937302, 3.977037,]
Best Fx = 5.446059e-22, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.581845e-02, Best X = [ 1.081234, 1.963594, 2.919375, 4.037332,]
Best Fx = 1.164094e-22, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.772322e-02, Best X = [ 0.931390, 2.104425, 2.954706, 4.007736,]
Best Fx = 5.479150e-23, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 3.690769e-03, Best X = [ 0.975992, 1.970207, 3.041327, 4.022779,]
Best Fx = 1.292567e-23, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 3.361211e-02, Best X = [ 1.077351, 2.122315, 3.112052, 3.989400,]
Best Fx = 6.799299e-24, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 2.706583e-02, Best X = [ 0.941156, 1.869063, 3.019604, 4.077939,]
Best Fx = 4.013880e-24, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 3.027094e-02, Best X = [ 0.880231, 2.089633, 3.046687, 3.924418,]
Best Fx = 1.493899e-24, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.783540e-02, Best X = [ 1.070162, 1.905119, 2.997462, 3.937519,]
Best Fx = 1.353908e-24, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.860556e-02, Best X = [ 1.077633, 2.047699, 2.901137, 4.023014,]
Best Fx = 1.265252e-24, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.914643e-02, Best X = [ 0.955328, 1.888762, 2.934697, 3.977359,]
Best Fx = 5.750142e-25, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.397737e-02, Best X = [ 1.037758, 2.018201, 2.900431, 4.048026,]
Best Fx = 1.809132e-25, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 2.887453e-02, Best X = [ 0.859660, 1.923631, 2.949886, 4.028904,]
Best Fx = 8.765882e-26, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 4.371172e-03, Best X = [ 0.990250, 1.947569, 3.035649, 4.016009,]
Best Fx = 3.267808e-26, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
```



```
Best Fx = 1.823464e-02, Best X = [ 0.897891, 1.963052, 3.029816, 3.925474,]
Best Fx = 9.657334e-27, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 8.883122e-03, Best X = [ 1.033445, 1.923015, 3.015865, 3.960172,]
Best Fx = 4.513036e-27, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 4.448698e-02, Best X = [ 0.899187, 2.090083, 3.161891, 4.000419,]
Best Fx = 2.891570e-27, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.334481e-02, Best X = [ 0.999291, 2.000655, 2.890210, 3.964084,]
Best Fx = 8.854964e-28, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.341084e-02, Best X = [ 0.939571, 1.998668, 2.960897, 4.090710,]
Best Fx = 7.654416e-28, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.923817e-02, Best X = [ 0.927334, 1.902503, 2.945177, 3.961964,]
Best Fx = 3.629253e-28, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.826145e-02, Best X = [ 1.030793, 1.988859, 3.091215, 3.905825,]
Best Fx = 3.758183e-29, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 9.369962e-03, Best X = [ 1.056924, 1.935001, 3.029353, 4.032298,]
Best Fx = 2.081853e-29, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.422996e-02, Best X = [ 1.003709, 2.027968, 3.100089, 3.941552,]
Best Fx = 1.143848e-29, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 2.613451e-02, Best X = [ 0.993427, 1.868172, 3.055776, 3.925154,]
Best Fx = 1.084684e-30, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 2.491624e-02, Best X = [ 1.064144, 2.072350, 3.076960, 4.098206,]
Best Fx = 1.035380e-30, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.854679e-02, Best X = [ 0.905117, 1.998575, 2.911222, 4.040748,]
Best Fx = 7.888609e-31, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 2.921814e-02, Best X = [ 1.030095, 1.946607, 3.099039, 3.874888,]
Best Fx = 5.916457e-31, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 2.271977e-03, Best X = [ 1.009017, 1.992984, 3.036131, 4.028914,]
Best Fx = 5.053640e-31, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 2.094267e-02, Best X = [ 1.001332, 2.018664, 3.082310, 3.882451,]
Best Fx = 2.465190e-31, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.357699e-02, Best X = [ 0.918077, 2.032416, 2.924220, 3.991498,]
Best Fx = 1.232595e-32, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 2.475175e-02, Best X = [ 0.992557, 2.117112, 3.104652, 3.994603,]
Best Fx = 0.000000e+00, Best X = [ 1.000000, 2.000000, 3.000000, 4.000000,]
Best Fx = 1.119386e-02, Best X = [ 0.910125, 1.972535, 2.995117, 4.048355,]

bestX =

     1     2     3     4


bestFx =

     0
```

The above results match the actual optimum point. As Shakespeare said, “All is well that ends well”.

The Enhanced Recursive Calls

The last section presented a simple recursive call that returns a single candidate for the best optimum point. In this section I present a version that returns a secondary population of candidates for better optimum points. I present the function recPSO()

that performs recursive random search optimization for the PSO algorithm. Here is the function listing.

```
function [zBestX,zBestFx] = recPSO(fx,Lb,Ub,MaxPop,MaxIters, ...
                          nestedMaxIters,breedingProbab,...
                          bNestedCallInEachIteration,bRecursiveCall)
% recPSO implements recursive particle swarm optimization. Includes
% breeding feature.
%
% Note: Performs very well.
%
% Reference
% =========
%
% An Analysis of Particle Swarm Optimizers by Frans van den Bergh
%
%
% INPUT
% ======
% fx - handle of optimized function.
% Lb - array of low bound values.
% Ub - array of upper bound values.
% MaxPop - maximum population of swarm.
% MaxIters - maxumum number of iterations
% nestedMaxIters - maximum iterations in nested call to PSO function
% breedingProbab - breeding probability
% bNestedCallInEachIteration - flag to call nested to PSO function in each
% iteration.
% bRecursiveCall - recursive call flag.
%
% OUTPUT
% ======
% zBestX - array of best solutions.
% zBestFx - best optimized function value.
%
% Example
% =======
%
% >> [bestX,bestFx] = recPSO(@fx3,[0 0 0 0],5*[5 5 5
5],100,100,100,0.4,false)
%
  global bestX bestFx
  if nargin < 9, bRecursiveCall = false; end
  if nargin < 8, bNestedCallInEachIteration = false; end
  if nargin < 7, breedingProbab = 0.4; end
  if nargin < 6, nestedMaxIters = fix(MaxIters/2); end
  if breedingProbab > 1, breedingProbab = breedingProbab/100; end
  n = length(Lb);
  m = n + 1;
  pop = 1e+99+zeros(MaxPop,m);
  pop2 = pop;
  aPop = zeros(1,n);
  vel = zeros(MaxPop,n);

  % Initizialize population
  for i=1:MaxPop
```



```
    pop(i,1:n) = Lb + (Ub - Lb) .* rand(1,n);
    vel(i,1:n) = (Ub - Lb) / 10 .* (2*rand(1,n)-1);
    pop(i,m) = fx(pop(i,1:n));
    pop2(i,:) = pop(i,:);
    aPop(1:n) = Lb + (Ub - Lb) .* rand(1,n);
    f0 = fx(aPop);
    if f0 < pop2(i,m)
      pop2(i,1:n) = aPop(1:n);
      pop2(i,m) = f0;
    end
  end

  pop = sortrows(pop,m);
  pop2 = pop;

  if ~bRecursiveCall
    fprintf('Best Fx = %e, ', pop(1,m));
    fprintf('Best X = [');
    fprintf(' %f,', pop(1,1:n));
    fprintf("]\n");

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

  % pso loop
  for iter = 1:MaxIters
    IterFactor = sqrt((iter - 1)/(MaxIters - 1));
    w = 1 - 0.3 * IterFactor;
    c1 = 2 - 1.9 * IterFactor;
    c2 = 2 - 1.9 * IterFactor;

    for i=2:MaxPop
      for j=1:n
        vel(i,j) = w*vel(i,j) + c1*rand*(pop(1,j) - pop(i,j)) + ...
          c2*rand*(pop2(i,j) - pop(i,j));
        p = pop(i,j) + vel(i,j);

        if p < Lb(j) || p > Ub(j)
          pop(i,j) = Lb(j) + (Ub(j) - Lb(j))*rand;
        else
          pop(i,j) = p;
        end
      end

      pop(i,m) = fx(pop(i,1:n));

      % find new global best?
      if pop(1,m) > pop(i,m)
        pop(1,:) = pop(i,:);
        % find new local best?
      elseif pop(i,m) < pop2(i,m)
        pop2(i,:) = pop(i,:);
      end

      % breeding step here
      if breedingProbab > rand
```



```
        k = i + fix((MaxPop-i)*rand);
        r = rand;
        for j=1:n
          xa = pop(i,j);
          xb = pop(k,j);
          va = vel(i,j);
          vb = vel(k,j);
          pop(i,j) = r*xa + (1-r)*xb;
          pop(k,j) = r*xb + (1-r)*xa;
          vel(i,j) = (va+vb)/abs(va+vb)*abs(va);
          vel(k,j) = (va+vb)/abs(va+vb)*abs(vb);
        end
        pop(i,m) = fx(pop(i,1:n));
        pop(k,m) = fx(pop(k,1:n));
      end

    end

    [pop,Idx] = sortrows(pop,m);
    pop2 = sortrows(pop2,m);
    vel = vel(Idx,:);

    bFoundBetter = false;
    if bestFx > pop(1,m)
      fprintf('Best Fx = %e, Best X = [', pop(1,m));
      fprintf(' %f,', pop(1,1:n));
      fprintf("]\n");
      bestFx = pop(1,m);
      bestX = pop(1,1:n);
      bFoundBetter = true;
    end

    if bFoundBetter || bNestedCallInEachIteration
      Llb = Lb;
      Uub = Ub;
      for i=1:length(bestX)
        Llb(i) = narrowLowerLimit(bestX(i),0.15);
        Uub(i) = narrowUpperLimit(bestX(i),0.15);
      end
      [pop3,bfx] =
recPSO(fx,Llb,Uub,MaxPop,nestedMaxIters,0,breedingProbab,false,true);
      pop3 = [pop3 bfx];
      pop = [pop; pop3];
      pop = sortrows(pop,m);
      pop = pop(1:MaxPop,:);
      if bestFx > pop(1,m)
        bestFx = pop(1,m);
        bestX = pop(1,1:n);
      end
    end
  end

  if bRecursiveCall
    zBestX = pop(:,1:n);
    zBestFx = pop(:,m);
  else
    zBestFx = pop(1,m);
```



```
    zBestX = pop(1,1:n);
  end
end

function x = narrowUpperLimit(x,factor)
  if nargin < 2, factor = 0.15; end
  if x > 0
    x = (1+factor)*x;
  else
    x = (1-factor)*x;
  end
end

function x = narrowLowerLimit(x,factor)
  if nargin < 2, factor = 0.15; end
  if x < 0
    x = (1+factor)*x;
  else
    x = (1-factor)*x;
  end
end
```

The list of input parameters is:

- fx is the handle of optimized function.
- Lb is the array of low bound values.
- Ub is the array of upper bound values.
- MaxPop is the maximum population of swarm.
- MaxIters is the maximum number of iterations
- nestedMaxIters is the maximum iterations in nested call to PSO function
- breedingProbab is the breeding probability.
- bNestedCallInEachIteration is the flag to call nested to PSO function in each iteration.
- bRecursiveCall is the recursive call flag. Your call to function recPOS() must supply an argument of false to this parameter.

The list of output parameters is:

- zBestX is the array of best solutions.
- zBestFx is the best optimized function value.

The function declares two global arrays, bestX and bestFx. These global arrays allow the recursive call to access the best point and best function value obtained so far.

The function recPSO() uses an augmented matrix to combine storing the values of the optimized variables and the function values for each row. This type of consolidation makes it easier to sort the population matrix by the function values. These values occupy the last column of the augmented population matrix.

The function uses some programming tricks to support the recursive call. The main trick revolves around what the output parameters return. The non recursive call returns the row-wise array of optimum values in zBestX, and the optimum function value in zBestFx. By contrast, the recursive call returns a population matrix in zBestX and a column of best function values in zBestFx. By column—merging the values in zBestX and zBestFx the code obtains an augmented population matrix from the recursive call.

There are three code segments that manage the recursion. The first one is a simple if statement that preserves the values of bestX and bestFx from being overwritten during a recursive function call:

```
  if ~bRecursiveCall
    bestFx = pop(1,m);
    bestX = pop(1,1:n);
  end
```

The second code segment is an if statement that performs the recursion:

```
    if bFoundBetter || bNestedCallInEachIteration
      Llb = Lb;
      Uub = Ub;
      for i=1:length(bestX)
        Llb(i) = narrowLowerLimit(bestX(i),0.15);
        Uub(i) = narrowUpperLimit(bestX(i),0.15);
      end
      [pop3,bfx] =
recPSO(fx,Llb,Uub,MaxPop,nestedMaxIters,0,breedingProbab,false,true);
      pop3 = [pop3 bfx];
      pop = [pop; pop3];
      pop = sortrows(pop,m);
      pop = pop(1:MaxPop,:);
      if bestFx > pop(1,m)
        bestFx = pop(1,m);
        bestX = pop(1,1:n);
      end
    end
```

The first statements in the if clause calculate the narrow trust region range and store them in the local variables Llb and Uub. The for loop uses the values of array bestX to obtain the new narrow trust region. The recursive call to recPSO uses the variables

Llb and Uub to supply the narrow trust region. The call also uses the value in variable nestedMaxIters to specify the maximum number of iterations in the recursive call. The call passes the argument of zero to the parameter nestedMaxIters. The recursive call returns the population matrix pop3 and the column vector bfx. The code merges these two variables to build an augmented population matrix. Next, the code merges the matrices pop and pop3 in a row-wise manner. A call to function sortrows() allows the function to sort the augmented population matrix using the function values, located in the last column m. After sorting the population matrix, the code retains the first MaxPop rows in the augmented population matrix. A nested if statement checks if the first row of the updated augmented population matrix has a better function value. If so, the code updates the values for bestX and bestFx.

The third code segment is the following if statement that appears at the end of function recPSO():

```
  if bRecursiveCall
    zBestX = pop(:,1:n);
    zBestFx = pop(:,m);
  else
    zBestFx = pop(1,m);
    zBestX = pop(1,1:n);
  end
```

The if statement checks the Boolean value of flag bRecursiveCall. If it is true, the if clause copies the population matrix to variable zBestX and copies the function values column to variable zBestFx. By contrast, when the tested condition is false, the else clause copies the row of best values into zBestX and copies the scalar value of the best function value into zBestFx.

The second and third code segments show the programming tricks that a non-typing programming language like MATLAB supports. I don't think it is easy to translate such code to a strong typing language like C++ for example.

Here is a simple test function to optimize:

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


Here is a sample session for using recPSO(). The lower and upper trust ranges are [0 0 0 0] and [5 5 5 5], respectively. The call uses a population of 100 members. The function uses 100 main iterations with 100 recursive iteration that are triggered only when a better optimum point is located:

```
>> [bestX,bestFx] = recPSO(@fx1,[0 0 0 0],[5 5 5 5],100,100,100,0.4,false)
Best Fx = 8.810020e-01, Best X = [ 0.340914, 1.589126, 3.018582, 3.473269,]
Best Fx = 2.845339e-01, Best X = [ 1.006125, 2.070924, 2.686267, 3.574515,]
Best Fx = 9.936698e-03, Best X = [ 1.062673, 1.964632, 3.065439, 4.021808,]
Best Fx = 9.690675e-03, Best X = [ 1.033148, 1.994708, 2.924767, 4.053887,]
Best Fx = 5.418736e-03, Best X = [ 0.980458, 1.997647, 2.934004, 4.025997,]
Best Fx = 5.107848e-03, Best X = [ 1.038942, 2.029491, 3.050219, 4.014131,]
Best Fx = 4.220400e-03, Best X = [ 0.962813, 1.979771, 2.976589, 4.043362,]
Best Fx = 3.389168e-03, Best X = [ 0.981396, 1.983640, 2.949731, 4.015762,]
Best Fx = 3.119437e-03, Best X = [ 0.979718, 2.021341, 3.007141, 3.953078,]
Best Fx = 2.650310e-03, Best X = [ 0.995141, 1.984833, 2.971820, 3.959968,]
Best Fx = 2.117238e-03, Best X = [ 1.008283, 2.032246, 2.974475, 3.981097,]
Best Fx = 1.234244e-03, Best X = [ 0.988326, 2.006235, 2.975936, 4.021909,]
Best Fx = 5.824413e-04, Best X = [ 1.006393, 1.997407, 3.005308, 4.022509,]
Best Fx = 3.822062e-04, Best X = [ 1.006963, 1.997683, 2.995292, 3.982502,]
Best Fx = 1.675075e-04, Best X = [ 1.004448, 1.989527, 3.005814, 4.002056,]
Best Fx = 1.347351e-04, Best X = [ 1.005694, 2.005042, 3.002407, 4.008432,]
Best Fx = 1.228728e-04, Best X = [ 1.002944, 2.007624, 2.995951, 3.993700,]
Best Fx = 1.111265e-04, Best X = [ 1.007133, 2.003585, 2.993117, 4.000119,]
Best Fx = 9.452956e-06, Best X = [ 0.998170, 2.002072, 2.999083, 4.000986,]
Best Fx = 8.256884e-06, Best X = [ 1.002734, 1.999551, 3.000718, 3.999746,]
Best Fx = 7.728530e-06, Best X = [ 1.000342, 2.000619, 3.002121, 3.998348,]
Best Fx = 4.983990e-06, Best X = [ 1.000762, 1.999374, 3.000968, 3.998247,]
Best Fx = 3.441659e-07, Best X = [ 1.000050, 2.000204, 3.000368, 4.000406,]
Best Fx = 3.027121e-07, Best X = [ 0.999671, 2.000337, 3.000170, 3.999772,]
Best Fx = 7.329304e-09, Best X = [ 1.000016, 1.999966, 2.999926, 4.000022,]
Best Fx = 4.404578e-09, Best X = [ 1.000021, 2.000015, 3.000006, 3.999939,]
Best Fx = 2.926935e-09, Best X = [ 0.999955, 2.000012, 3.000023, 4.000016,]
Best Fx = 9.616046e-10, Best X = [ 1.000009, 1.999980, 3.000004, 4.000021,]
Best Fx = 3.801720e-10, Best X = [ 0.999983, 1.999992, 2.999997, 4.000005,]
Best Fx = 2.372613e-10, Best X = [ 0.999992, 1.999999, 2.999994, 3.999989,]
Best Fx = 2.038394e-10, Best X = [ 0.999996, 2.000009, 2.999990, 3.999998,]
Best Fx = 1.311256e-10, Best X = [ 0.999992, 1.999998, 2.999999, 4.000008,]
Best Fx = 7.321562e-11, Best X = [ 0.999997, 1.999994, 2.999996, 3.999996,]

bestX =

    1.0000    2.0000    3.0000    4.0000


bestFx =

   7.3216e-11
```

The above calculations yield the correct answer.

The Quasi-Recursive Call

In presenting the code for recursive function calls you learned about programming tricks that work with programming languages with no data type declaration. This section shows you how to implement a quasi-recursive call using twin functions. The first function is the main one that your code calls. This function calls a variant version of it to perform the logical recursive call. As such, the two functions can have distinct input and output parameters. Here is an example of using a quasi-recursive call for a function that implements the PSO algorithm. I start by presenting the code for the main function recPSO1():

```
function [bestX,bestFx] = recPSO1(fx,Lb,Ub,MaxPop,MaxIters, ...

nestedMaxIters,breedingProbab,bNestedCallInEachIteration)
% recPSO1 implements recursive particle swarm optimization. Includes
% breeding feature.
%
% Note: Performs very well.
%
% Reference
% =========
%
% An Analysis of Particle Swarm Optimizers by Frans van den Bergh
%
%
% INPUT
% ======
% fx - handle of optimized function.
% Lb - array of low bound values.
% Ub - array of upper bound values.
% MaxPop - maximum population of swarm.
% MaxIters - maxumum number of iterations
% nestedMaxIters - maximum iterations in nested call to PSO function
% breedingProbab - breeding probability
% bNestedCallInEachIteration - flag to call nested to PSO function in each
% iteration.
%
% OUTPUT
% ======
% bestX - array of best solutions.
% bestFx - best optimized function value.
%
% Example
% =======
%
% >> [bestX,bestFx] = recPSO1(@fx3,[0 0 0 0],5*[5 5 5
5],100,100,100,0.4,false)
%
  if nargin < 8, bNestedCallInEachIteration = false; end
  if nargin < 7, breedingProbab = 0.4; end
  if nargin < 6, nestedMaxIters = fix(MaxIters/2); end
  if breedingProbab > 1, breedingProbab = breedingProbab/100; end
  n = length(Lb);
```



```
m = n + 1;
pop = 1e+99+zeros(MaxPop,m);
pop2 = pop;
aPop = zeros(1,n);
vel = zeros(MaxPop,n);

% Initizialize population
for i=1:MaxPop
  pop(i,1:n) = Lb + (Ub - Lb) .* rand(1,n);
  vel(i,1:n) = (Ub - Lb) / 10 .* (2*rand(1,n)-1);
  pop(i,m) = fx(pop(i,1:n));
  pop2(i,:) = pop(i,:);
  aPop(1:n) = Lb + (Ub - Lb) .* rand(1,n);
  f0 = fx(aPop);
  if f0 < pop2(i,m)
    pop2(i,1:n) = aPop(1:n);
    pop2(i,m) = f0;
  end
end

pop = sortrows(pop,m);
pop2 = pop;

fprintf('Best Fx = %e, ', pop(1,m));
fprintf('Best X = [');
fprintf(' %f,', pop(1,1:n));
fprintf("]\n");
bestFx = pop(1,m);
bestX = pop(1,1:n);

% pso loop
for iter = 1:MaxIters
  IterFactor = sqrt((iter - 1)/(MaxIters - 1));
  w = 1 - 0.3 * IterFactor;
  c1 = 2 - 1.9 * IterFactor;
  c2 = 2 - 1.9 * IterFactor;

  for i=2:MaxPop
    for j=1:n
      vel(i,j) = w*vel(i,j) + c1*rand*(pop(1,j) - pop(i,j)) + ...
        c2*rand*(pop2(i,j) - pop(i,j));
      p = pop(i,j) + vel(i,j);

      if p < Lb(j) || p > Ub(j)
        pop(i,j) = Lb(j) + (Ub(j) - Lb(j))*rand;
      else
        pop(i,j) = p;
      end
    end

    pop(i,m) = fx(pop(i,1:n));

    % find new global best?
    if pop(1,m) > pop(i,m)
      pop(1,:) = pop(i,:);
      % find new local best?
    elseif pop(i,m) < pop2(i,m)
```

```
        pop2(i,:) = pop(i,:);
      end

      % breeding step here
      if breedingProbab > rand
        k = i + fix((MaxPop-i)*rand);
        r = rand;
        for j=1:n
          xa = pop(i,j);
          xb = pop(k,j);
          va = vel(i,j);
          vb = vel(k,j);
          pop(i,j) = r*xa + (1-r)*xb;
          pop(k,j) = r*xb + (1-r)*xa;
          vel(i,j) = (va+vb)/abs(va+vb)*abs(va);
          vel(k,j) = (va+vb)/abs(va+vb)*abs(vb);
        end
        pop(i,m) = fx(pop(i,1:n));
        pop(k,m) = fx(pop(k,1:n));
      end

    end

    [pop,Idx] = sortrows(pop,m);
    pop2 = sortrows(pop2,m);
    vel = vel(Idx,:);

    bFoundBetter = false;
    if bestFx > pop(1,m)
      fprintf('%i: Best Fx = %e,Best X = [', iter, pop(1,m));
      fprintf(' %f,', pop(1,1:n));
      fprintf("]\n");
      bestFx = pop(1,m);
      bestX = pop(1,1:n);
      bFoundBetter = true;
    elseif mod(iter,1000)==0
      fprintf('%i of %i: Best Fx = %e\n', iter, MaxIters, pop(1,m));
    end

    if bFoundBetter || bNestedCallInEachIteration
      pop3 =
recPSO1b(fx,bestX,bestFx,MaxPop,nestedMaxIters,breedingProbab,iter);
      pop = [pop; pop3];
      pop = sortrows(pop,m);
      pop = pop(1:MaxPop,:);
      if bestFx > pop(1,m)
        bestFx = pop(1,m);
        bestX = pop(1,1:n);
      end
    end
  end

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

end
```


The list of input parameters is:

- fx is the handle of optimized function.
- Lb is the array of low bound values.
- Ub is the array of upper bound values.
- MaxPop is the maximum population of swarm.
- MaxIters is the maximum number of iterations.
- nestedMaxIters is the maximum iterations in nested call to PSO function.
- breedingProbab is the breeding probability.
- bNestedCallInEachIteration is the flag to call nested to PSO function in each iteration.

The list of output parameters is:

- bestX is the array of best solutions.
- bestFx is the best optimized function value.

Unlike function recPSO(), the function recPSO1() does not use global variables. Function resPSO1b() has the following code:

```
function pop = recPSO1b(fx,bestX,bestFx,MaxPop,MaxIters,breedingProbab,gIter)
% recPSO1b implements recursive particle swarm optimization. Includes
% breeding feature. This functin is called by recPSO1().
% Note: Performs very well.
%
% Reference
% =========
%
% An Analysis of Particle Swarm Optimizers by Frans van den Bergh
%
%
% INPUT
% ======
% fx - handle of optimized function.
% bestX - array of best population values supplied by the caller.
% bestFx - best function value supplied by the caller.
% MaxPop - maximum population of swarm.
% MaxIters - maximum number of iterations
% breedingProbab - breeding probability
% gIter - iteration number of the calling function recPSO1().
%
% OUTPUT
% ======
% pop - matrix of population and function values (in last column).

%
% Example
```

```
% =======
%
% >>
%
  if nargin < 7, minFx = 0; end
  if nargin < 6, breedingProbab = 0.4; end
  if breedingProbab > 1, breedingProbab = breedingProbab/100; end
  n = length(bestX);
  Lb = zeros(1,n);
  Ub = zeros(1,n);
  for i=1:length(bestX)
    Lb(i) = narrowLowerLimit(bestX(i),0.15);
    Ub(i) = narrowUpperLimit(bestX(i),0.15);
  end
  m = n + 1;
  pop = 1e+99+zeros(MaxPop,m);
  pop2 = pop;
  aPop = zeros(1,n);
  vel = zeros(MaxPop,n);

  % Initizialize population
  for i=1:MaxPop
    pop(i,1:n) = Lb + (Ub - Lb) .* rand(1,n);
    vel(i,1:n) = (Ub - Lb) / 10 .* (2*rand(1,n)-1);
    pop(i,m) = fx(pop(i,1:n));
    pop2(i,:) = pop(i,:);
    aPop(1:n) = Lb + (Ub - Lb) .* rand(1,n);
    f0 = fx(aPop);
    if f0 < pop2(i,m)
      pop2(i,1:n) = aPop(1:n);
      pop2(i,m) = f0;
    end
  end

  pop = sortrows(pop,m);
  pop2 = pop;

  % pso loop
  for iter = 1:MaxIters
    IterFactor = sqrt((iter - 1)/(MaxIters - 1));
    w = 1 - 0.3 * IterFactor;
    c1 = 2 - 1.9 * IterFactor;
    c2 = 2 - 1.9 * IterFactor;

    for i=2:MaxPop
      for j=1:n
        vel(i,j) = w*vel(i,j) + c1*rand*(pop(1,j) - pop(i,j)) + ...
          c2*rand*(pop2(i,j) - pop(i,j));
        p = pop(i,j) + vel(i,j);

        if p < Lb(j) || p > Ub(j)
          pop(i,j) = Lb(j) + (Ub(j) - Lb(j))*rand;
        else
          pop(i,j) = p;
        end
      end
```



```
      pop(i,m) = fx(pop(i,1:n));

      % find new global best?
      if pop(1,m) > pop(i,m)
        pop(1,:) = pop(i,:);
        % find new local best?
      elseif pop(i,m) < pop2(i,m)
        pop2(i,:) = pop(i,:);
      end

      % breeding step here
      if breedingProbab > rand
        k = i + fix((MaxPop-i)*rand);
        r = rand;
        for j=1:n
          xa = pop(i,j);
          xb = pop(k,j);
          va = vel(i,j);
          vb = vel(k,j);
          pop(i,j) = r*xa + (1-r)*xb;
          pop(k,j) = r*xb + (1-r)*xa;
          vel(i,j) = (va+vb)/abs(va+vb)*abs(va);
          vel(k,j) = (va+vb)/abs(va+vb)*abs(vb);
        end
        pop(i,m) = fx(pop(i,1:n));
        pop(k,m) = fx(pop(k,1:n));
      end

    end

    [pop,Idx] = sortrows(pop,m);
    pop2 = sortrows(pop2,m);
    vel = vel(Idx,:);

    if bestFx > pop(1,m)
      fprintf('%i/%i: Best Fx = %e,Best X = [', gIter, iter, pop(1,m));
      fprintf(' %f,', pop(1,1:n));
      fprintf("]\n");
      bestFx = pop(1,m);
      bestX = pop(1,1:n);
     end
  end

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

end

function x = narrowUpperLimit(x,factor)
  if nargin < 2, factor = 0.15; end
  if x > 0
    x = (1+factor)*x;
  else
    x = (1-factor)*x;
  end
end
```



```
function x = narrowLowerLimit(x,factor)
  if nargin < 2, factor = 0.15; end
  if x < 0
    x = (1+factor)*x;
  else
    x = (1-factor)*x;
  end
end
```

The input parameter list for function recPSO1b() is:

- fx is the handle of optimized function.
- bestX is the array of best population values supplied by the caller.
- bestFx is the best function value supplied by the caller.
- MaxPop is the maximum swarm population.
- MaxIters is the maximum number of iterations.
- breedingProbab is the breeding probability.
- gIter is the iteration number of the calling function recPSO1().

The output parameter pop is the augmented matrix of population and function values (in last column).

The function recPSO1() uses the following if statement to make the quasi-recursive call:

```
    if bFoundBetter || bNestedCallInEachIteration
      pop3 =
recPSO1b(fx,bestX,bestFx,MaxPop,nestedMaxIters,breedingProbab,iter);
      pop = [pop; pop3];
      pop = sortrows(pop,m);
      pop = pop(1:MaxPop,:);
      if bestFx > pop(1,m)
        bestFx = pop(1,m);
        bestX = pop(1,1:n);
      end
    end
  end
```

The first statement in the if clause calls function recPSO1b(). Notice that the call passes the values in array bestX and scalar bestFx to the parameters Lb and Ub, respectively. The function yields a single result which is the augmented population matrix pop3. Next, the code merges the matrices pop and pop3 in a row-wise manner. A call to function sortrows() allows the function to sort the augmented population matrix using the function values in the last column m. After sorting the population matrix, the code retains the first MaxPop rows in the augmented population matrix.

The nested if statement updates the values of bestFx and bestX if the element pop(1,m) is less than the value in bestFx.

Here is a sample session for using recPSO1() to find the optimum value for function fx1():

```
>> [bestX,bestFx] = recPSO1(@fx1,[0 0 0 0],[5 5 5 5],100,100,100,0.4,false)
Best Fx = 1.241046e+00, Best X = [ 0.344030, 1.597999, 2.654321, 3.272229,]
1: Best Fx = 2.449421e-01,Best X = [ 0.866988, 2.061002, 2.792543, 3.575158,]
1/1: Best Fx = 2.207166e-03,Best X = [ 0.979466, 1.972881, 3.026104, 3.980799,]
1/69: Best Fx = 1.785947e-03,Best X = [ 0.974023, 1.985471, 3.024945, 4.016667,]
1/71: Best Fx = 4.080162e-04,Best X = [ 0.983715, 1.988403, 2.997664, 4.001694,]
1/87: Best Fx = 2.961073e-04,Best X = [ 0.991369, 1.988736, 2.994423, 3.992023,]
1/88: Best Fx = 2.777032e-04,Best X = [ 0.988523, 1.995572, 3.008154, 3.992262,]
1/95: Best Fx = 1.147997e-04,Best X = [ 0.995193, 1.996055, 3.004547, 4.007446,]
1/97: Best Fx = 7.033521e-05,Best X = [ 0.995156, 1.997866, 3.001346, 4.006364,]
1/100: Best Fx = 5.509556e-05,Best X = [ 0.992883, 1.998693, 2.999588, 4.001603,]
97: Best Fx = 5.035984e-05,Best X = [ 1.003096, 2.004441, 2.999005, 4.004479,]
97/46: Best Fx = 1.270529e-05,Best X = [ 0.998835, 2.000017, 3.003288, 3.999268,]
97/60: Best Fx = 6.625264e-06,Best X = [ 0.999744, 2.002435, 2.999544, 3.999352,]
97/64: Best Fx = 3.331766e-06,Best X = [ 1.001609, 1.999943, 2.999147, 4.000106,]
97/65: Best Fx = 1.393602e-06,Best X = [ 1.000883, 2.000238, 3.000654, 4.000360,]
97/68: Best Fx = 1.312902e-06,Best X = [ 0.999110, 1.999646, 3.000112, 3.999382,]
97/70: Best Fx = 9.355226e-07,Best X = [ 0.999820, 2.000095, 2.999067, 4.000153,]
97/73: Best Fx = 3.798891e-07,Best X = [ 1.000245, 2.000109, 3.000518, 3.999800,]
97/74: Best Fx = 3.320398e-07,Best X = [ 0.999598, 2.000051, 2.999776, 3.999657,]
97/75: Best Fx = 2.090322e-07,Best X = [ 0.999980, 2.000271, 3.000245, 4.000274,]
97/78: Best Fx = 6.932934e-08,Best X = [ 1.000147, 2.000215, 3.000027, 4.000027,]
97/80: Best Fx = 3.032595e-08,Best X = [ 0.999911, 2.000084, 2.999936, 3.999894,]
97/81: Best Fx = 2.466180e-08,Best X = [ 0.999930, 2.000092, 3.000046, 3.999905,]
97/82: Best Fx = 1.190410e-08,Best X = [ 0.999968, 1.999896, 3.000009, 4.000005,]
97/85: Best Fx = 1.288499e-09,Best X = [ 0.999965, 2.000002, 2.999996, 3.999992,]
97/91: Best Fx = 8.089447e-10,Best X = [ 0.999978, 2.000018, 3.000000, 3.999998,]
97/95: Best Fx = 5.176984e-10,Best X = [ 1.000006, 2.000008, 3.000016, 4.000013,]
97/97: Best Fx = 8.143561e-11,Best X = [ 1.000001, 1.999992, 3.000000, 4.000005,]
97/99: Best Fx = 3.916004e-11,Best X = [ 0.999999, 2.000004, 2.999997, 4.000003,]
97/100: Best Fx = 2.955596e-11,Best X = [ 1.000003, 2.000004, 2.999997, 4.000000,]

bestX =

    1.0000    2.0000    3.0000    4.0000


bestFx =

   2.9556e-11
```

Again, function recPSO1() returns accurate results!

Using augmented population matrices in functions recPSO(), recPSO1(), recPSO1b(), and recPSO2() made it easy to manage and sort these matrices. If you use programming languages like C++, C#, Java, or Visual Basic then you need to use more statements, loops, and routine calls to manage and sort the augmented

population matrices. You may also need to replace functions with subroutines that return results by using reference-type parameters.

Concerns About Missing Other Optimum Points?

You may criticize the recursive call strategy as one that will quickly converge to one solution while missing other better optimum points. To give you some assurance about that not happening I present function recPSO3(). I have taken the code of function recPSO() and added a pure random search step that is executed in every non-recursive iteration. Here is the code for function recPSO3():

```
function [zBestX,zBestFx] = recPSO3(fx,Lb,Ub,MaxPop,MaxIters, ...
                          nestedMaxIters,breedingProbab,...
                          bNestedCallInEachIteration,bRecursiveCall)
% recPSO3 implements recursive particle swarm optimization. Includes
% breeding feature. Has added ranom search step.
%
% Note: Performs very well.
%
% Reference
% =========
%
% An Analysis of Particle Swarm Optimizers by Frans van den Bergh
%
%
% INPUT
% ======
% fx - handle of optimized function.
% Lb - array of low bound values.
% Ub - array of upper bound values.
% MaxPop - maximum population of swarm.
% MaxIters - maximum number of iterations
% nestedMaxIters - maximum iterations in nested call to PSO function
% breedingProbab - breeding probability
% bNestedCallInEachIteration - flag to call nested to PSO function in each
% iteration.
% bRecursiveCall - recursive call flag.
%
% OUTPUT
% ======
% zBestX - array of best solutions.
% zBestFx - best optimized function value.
%
% Example
% =======
%
% >> [bestX,bestFx] = recPSO3(@fx1,[0 0 0 0],[5 5 5 5],100,100,100,0.4,false)
%
  global bestX bestFx
  if nargin < 9, bRecursiveCall = false; end
  if nargin < 8, bNestedCallInEachIteration = false; end
  if nargin < 7, breedingProbab = 0.4; end
  if nargin < 6, nestedMaxIters = fix(MaxIters/2); end
```



```
if breedingProbab > 1, breedingProbab = breedingProbab/100; end
n = length(Lb);
m = n + 1;
pop = 1e+99+zeros(MaxPop,m);
pop2 = pop;
aPop = zeros(1,n);
vel = zeros(MaxPop,n);

% Initizialize population
for i=1:MaxPop
  pop(i,1:n) = Lb + (Ub - Lb) .* rand(1,n);
  vel(i,1:n) = (Ub - Lb) / 10 .* (2*rand(1,n)-1);
  pop(i,m) = fx(pop(i,1:n));
  pop2(i,:) = pop(i,:);
  aPop(1:n) = Lb + (Ub - Lb) .* rand(1,n);
  f0 = fx(aPop);
  if f0 < pop2(i,m)
    pop2(i,1:n) = aPop(1:n);
    pop2(i,m) = f0;
  end
end

pop = sortrows(pop,m);
pop2 = pop;

if ~bRecursiveCall
  fprintf('Best Fx = %e, ', pop(1,m));
  fprintf('Best X = [');
  fprintf(' %f,', pop(1,1:n));
  fprintf("]\n");

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

% pso loop
for iter = 1:MaxIters
  IterFactor = sqrt((iter - 1)/(MaxIters - 1));
  w = 1 - 0.3 * IterFactor;
  c1 = 2 - 1.9 * IterFactor;
  c2 = 2 - 1.9 * IterFactor;

  for i=2:MaxPop
    for j=1:n
      vel(i,j) = w*vel(i,j) + c1*rand*(pop(1,j) - pop(i,j)) + ...
        c2*rand*(pop2(i,j) - pop(i,j));
      p = pop(i,j) + vel(i,j);

      if p < Lb(j) || p > Ub(j)
        pop(i,j) = Lb(j) + (Ub(j) - Lb(j))*rand;
      else
        pop(i,j) = p;
      end
    end

    pop(i,m) = fx(pop(i,1:n));
```



```
      % find new global best?
      if pop(1,m) > pop(i,m)
        pop(1,:) = pop(i,:);
        % find new local best?
      elseif pop(i,m) < pop2(i,m)
        pop2(i,:) = pop(i,:);
      end

      % breeding step here
      if breedingProbab > rand
        k = i + fix((MaxPop-i)*rand);
        r = rand;
        for j=1:n
          xa = pop(i,j);
          xb = pop(k,j);
          va = vel(i,j);
          vb = vel(k,j);
          pop(i,j) = r*xa + (1-r)*xb;
          pop(k,j) = r*xb + (1-r)*xa;
          vel(i,j) = (va+vb)/abs(va+vb)*abs(va);
          vel(k,j) = (va+vb)/abs(va+vb)*abs(vb);
        end
        pop(i,m) = fx(pop(i,1:n));
        pop(k,m) = fx(pop(k,1:n));
      end

    end

    [pop,Idx] = sortrows(pop,m);
    pop2 = sortrows(pop2,m);
    vel = vel(Idx,:);

    bFoundBetter = false;
    if bestFx > pop(1,m)
      fprintf('Best Fx = %e, Best X = [', pop(1,m));
      fprintf(' %f,', pop(1,1:n));
      fprintf("]\n");
      bestFx = pop(1,m);
      bestX = pop(1,1:n);
      bFoundBetter = true;
    end

    if bFoundBetter || bNestedCallInEachIteration
      Llb = Lb;
      Uub = Ub;
      for i=1:length(bestX)
        Llb(i) = narrowLowerLimit(bestX(i),0.15);
        Uub(i) = narrowUpperLimit(bestX(i),0.15);
      end
      [pop3,bfx] =
recPSO3(fx,Llb,Uub,MaxPop,nestedMaxIters,0,breedingProbab,false,true);
      pop3 = [pop3 bfx];
      pop = [pop; pop3];
      pop = sortrows(pop,m);
      pop = pop(1:MaxPop,:);
      if bestFx > pop(1,m)
        bestFx = pop(1,m);
```



```
        bestX = pop(1,1:n);
      end
    end

    % perform random search to ensure that we don't
    % miss other optimum points
    if ~bRecursiveCall
      pop3 = zeros(MaxPop,m);
      for i=1:MaxPop
        pop3(i,1:n) = Lb + (Ub - Lb) .* rand(1,n);
        pop3(i,m) = fx(pop3(i,1:n));
      end
      pop = [pop; pop3];
      pop = sortrows(pop,m);
      pop = pop(1:MaxPop,:);
      if pop(1,m) < bestFx
        bestFx = pop(1,m);
        bestX = pop(1,1:n);
      end
    end

  end

  if bRecursiveCall
    zBestX = pop(:,1:n);
    zBestFx = pop(:,m);
  else
    zBestFx = pop(1,m);
    zBestX = pop(1,1:n);
  end
end

function x = narrowUpperLimit(x,factor)
  if nargin < 2, factor = 0.15; end
  if x > 0
    x = (1+factor)*x;
  else
    x = (1-factor)*x;
  end
end

function x = narrowLowerLimit(x,factor)
  if nargin < 2, factor = 0.15; end
  if x < 0
    x = (1+factor)*x;
  else
    x = (1-factor)*x;
  end
end
```

The extra code that performs the pure random search step is:

```
    % perform random search to ensure that we don't
    % miss other optimum points
    if ~bRecursiveCall
      pop3 = zeros(MaxPop,m);
```



```
    for i=1:MaxPop
      pop3(i,1:n) = Lb + (Ub - Lb) .* rand(1,n);
      pop3(i,m) = fx(pop3(i,1:n));
    end
    pop = [pop; pop3];
    pop = sortrows(pop,m);
    pop = pop(1:MaxPop,:);
    if pop(1,m) < bestFx
      bestFx = pop(1,m);
      bestX = pop(1,1:n);
    end
  end
```

The above if statement creates the augmented population pop3 to store random values in the initial trust region. The code then merges the augmented populations pop and pop3, sorted the extended population by function value, and then clips the extra matrix rows, keeping MaxPop rows. The nested if statement checks if the first row of the augmented population matrix pop has a better optimum value. If so, the if clause updates the values in array bestX and scalar bestFx.

Here is a sample session with function recPSO3():

```
>> [bestX,bestFx] = recPSO3(@fx1,[0 0 0 0],[5 5 5 5],100,100,100,0.4,false)
Best Fx = 1.777176e-01, Best X = [ 1.293102, 2.239454, 2.855361, 3.883594,]
Best Fx = 1.510077e-01, Best X = [ 1.149876, 1.996847, 2.646064, 3.942870,]
Best Fx = 7.493909e-03, Best X = [ 1.031807, 2.058206, 3.003399, 4.055522,]
Best Fx = 3.984933e-03, Best X = [ 0.982189, 1.963437, 2.981890, 3.955246,]
Best Fx = 1.750588e-03, Best X = [ 1.002809, 2.008538, 2.970891, 3.971321,]
Best Fx = 1.151543e-03, Best X = [ 0.982237, 1.999481, 3.021760, 3.980966,]
Best Fx = 7.009209e-04, Best X = [ 1.001147, 2.012055, 2.999976, 4.023543,]
Best Fx = 6.249369e-04, Best X = [ 0.999775, 2.021185, 2.993715, 4.011686,]
Best Fx = 5.533444e-04, Best X = [ 0.981068, 2.006661, 2.992534, 3.990263,]
Best Fx = 3.054012e-04, Best X = [ 0.998876, 2.000811, 3.017402, 3.999189,]
Best Fx = 9.427163e-05, Best X = [ 0.999820, 2.000489, 2.990333, 4.000735,]
Best Fx = 8.043628e-05, Best X = [ 1.005366, 1.994955, 3.004497, 3.997558,]
Best Fx = 2.676747e-05, Best X = [ 0.997243, 2.003170, 2.998511, 3.997373,]
Best Fx = 1.581111e-05, Best X = [ 0.997599, 2.001910, 2.997471, 3.999991,]
Best Fx = 9.088086e-06, Best X = [ 0.999969, 2.000816, 2.998097, 4.002191,]
Best Fx = 5.128788e-06, Best X = [ 1.001488, 1.998727, 2.999410, 3.999027,]
Best Fx = 4.975023e-06, Best X = [ 0.998062, 2.000261, 3.000339, 4.001017,]
Best Fx = 3.749820e-06, Best X = [ 0.998500, 1.999359, 2.999872, 3.998964,]
Best Fx = 3.689929e-06, Best X = [ 0.998287, 1.999815, 3.000407, 3.999255,]
Best Fx = 2.632660e-06, Best X = [ 0.999838, 1.999906, 3.001177, 4.001101,]
Best Fx = 2.086285e-06, Best X = [ 1.000868, 2.000854, 3.000760, 3.999845,]
Best Fx = 1.517432e-06, Best X = [ 1.000596, 2.000700, 3.000471, 3.999329,]
Best Fx = 1.057827e-06, Best X = [ 1.000023, 2.000879, 2.999538, 4.000267,]
Best Fx = 3.729932e-07, Best X = [ 0.999733, 2.000418, 3.000209, 3.999711,]
Best Fx = 2.834985e-07, Best X = [ 1.000320, 2.000377, 2.999920, 3.999819,]
Best Fx = 2.831194e-07, Best X = [ 0.999664, 1.999656, 3.000215, 4.000078,]
Best Fx = 2.680965e-07, Best X = [ 0.999700, 2.000349, 2.999827, 4.000162,]
Best Fx = 6.290033e-08, Best X = [ 0.999892, 1.999865, 2.999911, 3.999841,]
Best Fx = 5.799033e-08, Best X = [ 0.999844, 1.999980, 3.000023, 3.999819,]
```



```
Best Fx = 4.890878e-08, Best X = [ 0.999934, 2.000163, 3.000086, 3.999897,]
Best Fx = 3.614825e-08, Best X = [ 1.000059, 1.999885, 2.999924, 4.000116,]
Best Fx = 2.122902e-08, Best X = [ 0.999952, 1.999871, 3.000008, 4.000048,]
Best Fx = 2.118988e-08, Best X = [ 0.999918, 1.999893, 3.000054, 3.999991,]
Best Fx = 2.039095e-08, Best X = [ 0.999888, 1.999955, 3.000064, 3.999959,]
Best Fx = 1.000470e-08, Best X = [ 1.000036, 1.999939, 3.000059, 4.000039,]
Best Fx = 3.806957e-09, Best X = [ 1.000029, 2.000042, 3.000001, 4.000035,]
Best Fx = 2.544894e-09, Best X = [ 0.999991, 2.000011, 3.000043, 3.999978,]
Best Fx = 8.583234e-10, Best X = [ 1.000014, 2.000021, 3.000008, 4.000012,]
Best Fx = 2.483278e-10, Best X = [ 1.000003, 2.000014, 3.000007, 3.999998,]
Best Fx = 1.080420e-10, Best X = [ 1.000003, 2.000004, 3.000001, 4.000009,]
Best Fx = 1.048314e-10, Best X = [ 0.999999, 2.000008, 2.999999, 4.000006,]
Best Fx = 5.124338e-11, Best X = [ 1.000004, 2.000001, 2.999998, 4.000005,]
Best Fx = 4.470784e-11, Best X = [ 0.999999, 2.000003, 2.999994, 4.000002,]

bestX =

    1.0000    2.0000    3.0000    4.0000


bestFx =

   4.4708e-11
```

As expected, I got the correct answers and without any noticeable additional calculation time!

Bonus Case-Recursive Random Search

The recursive call strategy for EOA also works for the simpler (albeit less efficient) random search method. This section presents the function recRandSearch() to perform recursive random search.

```
function [bestX,bestFx] =
recRandSearch(fx,Lb,Ub,MaxIters,nestedMaxIters,factor)
% recRandSearch implements recursive random search optimization.
%
% Note: Performs very well.
%

% INPUT
% ======
% fx - handle of optimized function.
% Lb - array of low bound values.
% Ub - array of upper bound values.
% MaxIters - maximum number of iterations
% nestedMaxIters - maximum iterations in recursive call to function

% factor - range reduction factor
%
% OUTPUT
% ======
% bestX - array of best solutions.
% bestFx - best optimized function value.
```



```
%
% Example
% =======
%
% >> [bestX,bestFx] = recRandSearch(@fx1,[0 0 0 0],[5 5 5 5],1000,100,0.15)
%
  n = length(Lb);
  if nestedMaxIters > 0
    bestX = Lb + (Ub - Lb).*rand(1,n);
    bestFx = fx(bestX);
  else
    % here Lb is bestX and Ub is bestFx
    bestX = Lb;
    bestFx = Ub;
    for i=1:n
      Ub(i) = narrowUpperLimit(bestX(i), factor);
      Lb(i) = narrowLowerLimit(bestX(i), factor);
    end
  end

  if nestedMaxIters == 0, fprintf("*"); end
  fprintf("Fx =%e, X= [", bestFx);
  fprintf("%f ,", bestX);
  fprintf("]\n");

  for i=1:MaxIters
    X = Lb + (Ub - Lb).*rand(1,n);
    f = fx(X);
    if f < bestFx
      bestX = X;
      bestFx = f;
      [bestX2,bestFx2] =
recRandSearch(fx,bestX,bestFx,nestedMaxIters,0,factor);
      if bestFx2 < bestFx
        bestFx = bestFx2;
        bestX = bestX2;
      end
      if nestedMaxIters == 0, fprintf("*"); end
      fprintf("%d: Fx =%e, X= [", i, bestFx);
      fprintf("%f ,", bestX);
      fprintf("]\n");
    end
  end

end

function x = narrowUpperLimit(x,factor)
  if nargin < 2, factor = 0.15; end
  if x > 0
    x = (1+factor)*x;
  else
    x = (1-factor)*x;
  end
end

function x = narrowLowerLimit(x,factor)
  if nargin < 2, factor = 0.15; end
```



```
  if x < 0
    x = (1+factor)*x;
  else
    x = (1-factor)*x;
  end
end
```

The recRandSearch() function has an if statement at the beginning of the code to examine the value in parameter nestedMaxIters. If that value is positive, then the call to the function is non-recursive. The if clause calculates the initial random values for array bestX and its associated function value bestFx. By contrast, the else clause first copies the values of parameters Lb and Ub to bestX and bestFx, respectively. A for loop then calculates the actual working values for the arrays Lb and Ub based on the values of array bestX.

The main search loop has an if statement to check if a newly calculated function value is better than the value of bestFx. If yes, the if clause updates the values of bestX and bestFx and then makes a recursive function call. Notice that the arguments for parameters Lb, Ub, MaxIters, and nestedMaxIters are bestX, bestFx, nestedMaxIters, and 0 respectively. Passing the values of arrays bestX and scalar bestFx as arguments for parameters Lb and Ub is a programming trick. The code checks if the values returned by the recursive call are better than the current ones. If so, these values are assigned to array bestX and scalar bestFx.

Here is a sample session with function recRandomSearch(). The call finds the optimum of function fx1(). The lowr and upper ranges of the trust region are [0 0 0 0] and [5 5 5 5], respectively. The arguments for MaxIters and nestMaxIters are 100000 and 1000, respectively. The argument for the range reduction factor is 0.15.

```
> [bestX,bestFx] = recRandSearch(@fx1,[0 0 0 0],[5 5 5 5],100000,1000,0.15)
Fx =8.898156e+00, X= [0.855864 ,1.366399 ,0.131787 ,4.499282 ,]
*Fx =7.534099e+00, X= [2.221018 ,0.562057 ,4.645571 ,2.874110 ,]
*Fx =6.237037e+00, X= [2.069805 ,0.478784 ,4.430137 ,3.143751 ,]
*1: Fx =6.237037e+00, X= [2.069805 ,0.478784 ,4.430137 ,3.143751 ,]
*Fx =4.971463e+00, X= [1.972948 ,0.552316 ,3.953027 ,2.989661 ,]
*9: Fx =4.971463e+00, X= [1.972948 ,0.552316 ,3.953027 ,2.989661 ,]
*Fx =4.949774e+00, X= [1.936634 ,0.556659 ,4.003338 ,3.008754 ,]
*90: Fx =4.949774e+00, X= [1.936634 ,0.556659 ,4.003338 ,3.008754 ,]
*Fx =4.910381e+00, X= [1.949853 ,0.544062 ,4.043322 ,3.105637 ,]
*98: Fx =4.910381e+00, X= [1.949853 ,0.544062 ,4.043322 ,3.105637 ,]
*Fx =4.900888e+00, X= [1.963964 ,0.499782 ,4.063029 ,3.231249 ,]
*170: Fx =4.900888e+00, X= [1.963964 ,0.499782 ,4.063029 ,3.231249 ,]
*Fx =4.844036e+00, X= [1.948199 ,0.510256 ,4.025557 ,3.179116 ,]
*174: Fx =4.844036e+00, X= [1.948199 ,0.510256 ,4.025557 ,3.179116 ,]
*Fx =4.833194e+00, X= [2.130492 ,0.604496 ,3.978840 ,3.194007 ,]
```



```
*257: Fx =4.833194e+00, X= [2.130492 ,0.604496 ,3.978840 ,3.194007 ,]
*Fx =4.349402e+00, X= [1.960419 ,0.640252 ,4.045221 ,3.303154 ,]
*432: Fx =4.349402e+00, X= [1.960419 ,0.640252 ,4.045221 ,3.303154 ,]
3: Fx =4.349402e+00, X= [1.960419 ,0.640252 ,4.045221 ,3.303154 ,]
*Fx =2.952595e+00, X= [1.650029 ,1.206721 ,4.255240 ,4.570209 ,]
*Fx =2.736015e+00, X= [1.590017 ,1.163835 ,4.286576 ,4.182879 ,]
*1: Fx =2.736015e+00, X= [1.590017 ,1.163835 ,4.286576 ,4.182879 ,]
*Fx =2.211849e+00, X= [1.609381 ,1.144199 ,3.978774 ,4.387442 ,]
*3: Fx =2.211849e+00, X= [1.609381 ,1.144199 ,3.978774 ,4.387442 ,]
*Fx =2.175059e+00, X= [1.486106 ,1.090005 ,4.038607 ,4.178789 ,]
*8: Fx =2.175059e+00, X= [1.486106 ,1.090005 ,4.038607 ,4.178789 ,]
*Fx =1.927987e+00, X= [1.579467 ,1.208994 ,3.737860 ,4.649675 ,]
*10: Fx =1.927987e+00, X= [1.579467 ,1.208994 ,3.737860 ,4.649675 ,]
*Fx =1.811295e+00, X= [1.822193 ,1.177851 ,3.674981 ,3.938625 ,]
*26: Fx =1.811295e+00, X= [1.822193 ,1.177851 ,3.674981 ,3.938625 ,]
*Fx =1.560403e+00, X= [1.436901 ,1.217131 ,3.855449 ,4.157619 ,]
*33: Fx =1.560403e+00, X= [1.436901 ,1.217131 ,3.855449 ,4.157619 ,]
*Fx =1.396102e+00, X= [1.672093 ,1.287169 ,3.653403 ,3.903407 ,]
*70: Fx =1.396102e+00, X= [1.672093 ,1.287169 ,3.653403 ,3.903407 ,]
*Fx =1.386420e+00, X= [1.421581 ,1.167708 ,3.632776 ,4.339962 ,]
*192: Fx =1.386420e+00, X= [1.421581 ,1.167708 ,3.632776 ,4.339962 ,]
*Fx =1.385073e+00, X= [1.681415 ,1.299185 ,3.655410 ,3.993455 ,]
*204: Fx =1.385073e+00, X= [1.681415 ,1.299185 ,3.655410 ,3.993455 ,]
*Fx =1.036130e+00, X= [1.455255 ,1.364535 ,3.630981 ,4.164075 ,]
*208: Fx =1.036130e+00, X= [1.455255 ,1.364535 ,3.630981 ,4.164075 ,]
8: Fx =1.036130e+00, X= [1.455255 ,1.364535 ,3.630981 ,4.164075 ,]
*Fx =5.021597e-01, X= [0.902379 ,2.390841 ,2.418820 ,3.954145 ,]
*Fx =4.676738e-01, X= [0.869797 ,2.283425 ,2.576197 ,3.563214 ,]
*1: Fx =4.676738e-01, X= [0.869797 ,2.283425 ,2.576197 ,3.563214 ,]
*Fx =4.198814e-01, X= [0.814387 ,2.221827 ,2.772371 ,3.466702 ,]
*2: Fx =4.198814e-01, X= [0.814387 ,2.221827 ,2.772371 ,3.466702 ,]
*Fx =2.283805e-01, X= [0.863788 ,2.194400 ,2.625199 ,3.822350 ,]
*6: Fx =2.283805e-01, X= [0.863788 ,2.194400 ,2.625199 ,3.822350 ,]
*Fx =1.364000e-01, X= [0.930337 ,2.142624 ,2.668847 ,4.039277 ,]
*33: Fx =1.364000e-01, X= [0.930337 ,2.142624 ,2.668847 ,4.039277 ,]
*Fx =1.175484e-01, X= [0.939298 ,2.066648 ,2.716307 ,3.829883 ,]
*176: Fx =1.175484e-01, X= [0.939298 ,2.066648 ,2.716307 ,3.829883 ,]
*Fx =1.148108e-01, X= [0.860550 ,2.057386 ,2.752138 ,4.175031 ,]
*355: Fx =1.148108e-01, X= [0.860550 ,2.057386 ,2.752138 ,4.175031 ,]
*Fx =7.676388e-02, X= [1.020901 ,2.122636 ,2.758605 ,3.945084 ,]
*516: Fx =7.676388e-02, X= [1.020901 ,2.122636 ,2.758605 ,3.945084 ,]
*Fx =6.959014e-02, X= [0.965531 ,2.066701 ,2.755645 ,4.065142 ,]
*815: Fx =6.959014e-02, X= [0.965531 ,2.066701 ,2.755645 ,4.065142 ,]
54: Fx =6.959014e-02, X= [0.965531 ,2.066701 ,2.755645 ,4.065142 ,]
*Fx =3.369267e-02, X= [1.088219 ,1.963254 ,3.130303 ,4.087069 ,]
*Fx =1.038156e-02, X= [1.036458 ,2.085229 ,3.018757 ,4.037902 ,]
*130: Fx =1.038156e-02, X= [1.036458 ,2.085229 ,3.018757 ,4.037902 ,]
*Fx =8.632022e-03, X= [0.980953 ,2.063507 ,3.060834 ,3.976862 ,]
*418: Fx =8.632022e-03, X= [0.980953 ,2.063507 ,3.060834 ,3.976862 ,]
41949: Fx =8.632022e-03, X= [0.980953 ,2.063507 ,3.060834 ,3.976862 ,]

bestX =

    0.9810    2.0635    3.0608    3.9769


bestFx =
```



```
0.0086
```

As expected, the results are close to the actual answers, but at the cost of high number of iterations.

Conclusion

The suggested scheme of recursive calls to EOA functions can help speed up convergence since we are not waiting for the end of the iterations to narrow the trust region and start all over again! The simple recursion approach seems to lag a bit behind the more advanced schemes of recursion and quasi-recursion. The latter scheme seems to inject multiple better points into the main population of *swarms*.

Document History

Version	***Date***	***Comment***
1.0.0	4/27/2023	Initial Release

