EE 553 Term Project Report
Particle Swarm Optimization (PSO) and PSO with Cross-over

Emre Uğur

February 5, 2007

Abstract

In this work, Particle Swarm Optimization (PSO) method is implemented and applied to various mathematical functional optimization and engineering problems. Although, implemented exactly as described in [2], we could not obtain similar results, especially with increasing problem space. Thus, we modified the original algorithm, and obtained better results. Then, we extended the original PSO method, and added the cross-over operator of the evolutionary computation techniques. Instead of applying the cross-over operator in each iteration, we evolved completely different swarms using PSO, and then later cross-overed particles from these different swarms. As a result, we obtained much better results with PSO-Cross, especially when the problem size is increased in the pure mathematical optimization problems.

As an engineering problem, the welded beam design problem is studied. In the experiments, we obtain approximately same results (a little better), than the algorithm provided in [2].

1 Introduction and Motivation

Swarm Intelligence approach to the optimization problems might be loosely divided into two families, namely Ant Colony Optimization (ACO) methods, and Swarm Particle Optimization (SPO) methods. Both methods are inspired from the problem solving capabilities of the social animals that lives in swarms. In principle, the individual capabilities (and intelligence) of these animals are not sufficient for most of the tasks, however, through interaction, the emergent behavior results in solution of the very complex tasks. For example, ACO methods are inspired from the ants and bees, where they leave chemical substances, called pheromone, into the floor, and then react according to this pheromone when decision their actions. Thus, ACO approach to network routing problems mimics their behaviors, small packets are modelled as ants, which leave messages on the nodes of the networks. As a result, based on these messages (pheromones), the network traffic is adjusted. Since the aim of this report is not to discuss

Swarm Intelligence or ACO, after this introduction, I will continue with PSO.

2 Particle Swarm Optimization Method

As mentioned above, PSO is another family of Swarm Intelligence approaches which is inspired from the flocking behavior of the birds and fishes. Although, PSO is not solely invented for optimization problems, but first studied for modelling the animal behaviors, they are lately found to be successful in many optimization problems.

I will try to describe the motivation behind PSO with references to the flocking swarms. In PSO, a number of *"particles"*, which represent the candidate solution, form the *"swarm"*. These particles fly in the problem space, and try to find the optimum solution in this space. Thus each particle could be represented by its *"position"* (variable vector) and *"velocity"* (in which direction to move in that instant). Each particle re-orient itself in each instant based on three main motivations:

- It records the best solution, it has ever seen, and try to go there.
- It is aware of the best solution that has been found so far (the swarm's solution), thus it will try to go to the best solution of the particle in the swarm.
- It has a current velocity, and current acceleration.

Although, the first two motivations seems to be sufficient for the decision of next move, in order to not to trap in local minima, and *explore* the problem space , the third motivation is included. Thus, based on above three motivations, the particle's new velocity (direction of move) is determined, and it's position is updated based on the new velocity vector and current position. As a result the position ($\boldsymbol{x}$) and velocity ($\boldsymbol{v}$) vectors are updated as follows:

$$\begin{aligned}\boldsymbol{x}_{t+1} &= \boldsymbol{x}_t + \boldsymbol{v}_{t+1} \\ \boldsymbol{v}_{t+1} &= c_1.r_1.(\boldsymbol{x}_t^{best} - \boldsymbol{x}_t) + c_2.r_2.(\boldsymbol{x}_t^{globalbest} - \boldsymbol{x}_t) + w.\boldsymbol{v}_t\end{aligned}$$

where the terms on the right of the second equation corresponds to the motivations described below. c_1 and c_2 are fixed pre-defined constants, r_1 and r_2 are random two numbers between the range $[0 - 1]$. $\boldsymbol{x}_t^{best}$ and $\boldsymbol{x}_t^{globalbest}$ represents the positions of the particle's and swarm's best solutions respectively.

As mentioned above, the inertia term, represented by w, is responsible from the exploration capability of the algorithm. For larger values of w, the particle is more likely to explore the solution space, and for smaller values of w, the particle will exploit the local neighborhood of the its current position. As a result, as will be detailed in the next section, w value is decreased in order to allow exploration first and exploitation later.

Both position and velocities are bounded from below and above, and they are set to boundary values whenever they go beyond the maxima.

Additionally, maximum velocity value is decreased during iterations, in order to control the exploration capability of the method.

If the positions and velocities of the particles are updated solely based on the above rules, whole swarm would most likely diverge to the same solution. Thus, in [1], in order to maintain the diversity of the swarm, the *"craziness"* operator is proposed and used, where the velocities of some of the particles randomly selected, are set to random values. As a result, the particles would not be trapped in local minima, and swarm would maintain its divergence.

2.1 PSO Algorithm and Implementation Details

In this section, the algorithm will be presented step by step, and the implementation details (taken from the MATLAB code that is submitted) will be given.

Step 0: Initialize Swarm In this step, the swarm is initialized with randomly setting the positions and velocities between the ranges previously defined. The limits of the position vector is determined with the constraints on the $\boldsymbol{x}$ vector. The maximum and minimum values of the velocities are also defined by the range of $\boldsymbol{x}$ vector, ie. it is selected as a fraction of the problem domain: $\boldsymbol{v}_{max} = \gamma.(\boldsymbol{x}_{upperbound} - \boldsymbol{x}_{lowerbound})$. In the following code, `i` and `j` correspond to the indexes of particles (individuals) and variables respectively. As shown in line 11, whenever a function evaluation is done the variable **`nFuncEval`** is increased.

`nSwarms` stands for the number of swarms. For each swarm, best values are separately computed, and in later steps, the particles move only based on their own swarm. Since in the original algorithm, only one swarm exists, **`nSwarms=1`**, and there is only one best particle in the swarm.

```

% STEP 0. Initialize swarm and statistics
for(i=1:nIndividuals)
  for(j=1:nFeatures)
    particleCurrentPosition(i,j) = minX+(maxX-minX)*rand(1)
        ;
    particleBestPosition(i,j) = particleCurrentPosition(i,j
        );

    particleVelocity(i,j) = -1*maxV+(maxV+maxV)*rand(1);
  end
  particleCurrentFitness(i) = feval(funcName,
      particleCurrentPosition(i,:));
  nFuncEval = nFuncEval + 1;
  particleBestFitness(i) = particleCurrentFitness(i);

  if (particleCurrentFitness(i) < momentaryBestFitness)
    swarmBestFitness(ceil(i/(nIndividuals/nSwarms))) =
        particleCurrentFitness(i);
    swarmBestParticle(ceil(i/(nIndividuals/nSwarms))) = i;
  end
end
```

Step 1: Compute Fitness values This step could be viewed as the starting point of the iteration. Before the particle position and velocities are updated based on their fitness values, first, the fitness values are required to be computed. `startInd` and `endInd` values represents starting and ending individual (particle) indexes in the current swarm. Since in the original version, there is only one swarm, you can assume `startInd=1` and `endInd=nIndividuals`.

```

% STEP 1. Compute fitness values of each particle
for (i=startInd:endInd)
  particleCurrentFitness(i) = feval(funcName,
      particleCurrentPosition(i,:));
  nFuncEval = nFuncEval+1;
end
```

Step 2: Find best particle and swarm positions As described in the previous section, the updates are based on different motivations, including particle's own best position and swarm's best position. Thus, these positions and the corresponding fitness values should be computed and stored.

`swarm` represents, which swarm is currently handled. Since in the original method, only one swarm exists, you can think of `swarm=1` and only one best particle exists.

```

% STEP 2. Fitness Statistics
momentaryBestFitness  = 10000000000;
momentaryBestParticle = -1;
for(i=startInd:endInd)
  if (particleCurrentFitness(i) < momentaryBestFitness)
    momentaryBestFitness = particleCurrentFitness(i);
    momentaryBestParticle = i;
  end
  if (particleCurrentFitness(i) < particleBestFitness(i))
    particleBestFitness(i) = particleCurrentFitness(i);
    for (j=1:nFeatures)
     particleBestPosition(i,j)=particleCurrentPosition(i,j)
         ;
    end
    if (particleBestFitness(i) < swarmBestFitness(swarm))
      swarmBestFitness(swarm) = particleBestFitness(i);
      swarmBestParticle(swarm) = i;
    end
  end
end
```

Step 3: Update inertia and maximum velocity Since the inertia (w) and maximum velocity adjusts the exploration, and the exploration should be decreased during the iterations, as described in the previous section, they will be decreased. The α and β parameters are the constant variables, that accomplish this task. As shown, the inertia could not decrease beyond a minimal value (which is also constant).

```
% STEP 3. Parameter updates
w = alpha * w;
```

```
if (w < minW)                                                      3
  w = minW;                                                        4
end                                                                5
maxV = beta * maxV;                                                6
notBestUpdated = 0;                                                7
```

Step 4: Update velocities of particles In this step, the velocities of the particles are updated, with the rule based on three motivations. I'd like to remind you, in the current (original algorithm), there is only one swarm. However, in the extended version, there would be more than one swarms `nSwarm > 0`. Thus it is apparent that, the velocity updates are based on only particles own swarm's best position.

```
% STEP 4. Particle Velocity Update                                 1
for(i=startInd:endInd)                                             2
  for(j=1:nFeatures)                                               3
   r1 = rand(1);                                                   4
   r2 = rand(1);                                                   5
   tmp=w * particleVelocity(i,j) +                                 6
     c1*r1*(particleBestPosition(i,j)-                             7
         particleCurrentPosition(i,j)) +
     c2*r2*(particleBestPosition(swarmBestParticle(swarm),j        8
         )-particleCurrentPosition(i,j));
   if (tmp > maxV)                                                 9
     particleVelocity(i,j) = maxV;                                 10
   elseif (tmp < -1*maxV)                                          11
     particleVelocity(i,j) = -1*maxV;                              12
   else                                                            13
     particleVelocity(i,j) = tmp;                                  14
   end                                                             15
 end                                                               16
end                                                                17
```

Step 5: Craziness Operator As described, craziness operator randomly changes the velocity vectors of some of the particles. Based on a probability `p_cr`, the craziness operator is applied to `N_cr` particles (individuals):

```
% STEP 5. Apply craziness operator                                 1
if (rand(1)<p_cr)                                                  2
  crazyList = startInd+ceil(rand(nCrazyIndividuals,1)*(            3
      nIndividuals/nSwarms));
  for(i=1:nCrazyIndividuals)                                       4
    crazyIndividual = crazyList(i);                                5
    for(j=1:nFeatures)                                             6
      particleVelocity(crazyIndividual,j) = -1*maxV + (maxV        7
          +maxV)*rand(1);
    end                                                            8
  end                                                              9
end                                                                10
```

Step 6: Move particle Particle position is updated based on the rule defined in the previous section (based on its current position and new velocity vector).

```
                                                                   1
```

```
% STEP 6. Particle position update
for(i=startInd:endInd)
  for(j=1:nFeatures)
    tmp=particleCurrentPosition(i,j)+particleVelocity(i,j);
    if (tmp < minX)
      particleCurrentPosition(i,j) = minX;
    elseif (tmp > maxX)
      particleCurrentPosition(i,j) = maxX;
    else
      particleCurrentPosition(i,j) = tmp;
    end
  end
end
```

Stopping Criteria Steps 1-6 are repeated until the stopping criteria is met. In our case, the number of function evaluations are calculated and when the maximum number is reached, the algorithm stops:

2.2 Particle Swarm Optimization with Cross-over (PSO-Cross)

As shown in the experiments section, when standard PSO algorithm is applied to function minimization problems, its performance catastrophically degrades with increasing the problem space. During our experiments, it is seen that, after some point, for large variable numbers, the algorithm is most likely trapped in local minimas. When inspected in more detail, some components of the position vector approach to global minimum values, and some are trapped in local minima. Moreover, sometimes one component is trapped in the local minima, sometimes the other. For example, for the rastRigin function, which will be described later, in different runs, following local minimas can be obtained, where $x_i = 0$ is the global minimum, and $x_i = 1$ are the trapped local minimas.

$$\texttt{Run1}:100101$$
$$\texttt{Run2}:001000$$
$$\texttt{Run3}:010110$$

As shown, although neither of them reached the global minimum, when best components are "somehow" combined, the resulting position vector would be global minimum. The cross-over operator is a very successful operator well-fits for this problem. It is possible to apply the cross-over operator during the iterations in each step with some probability. However, in the experiments, I saw that the swarm diverges in spite of the craziness operator, and cross-over does not increase the performance. It even prevent the convergence of the algorithm.

What we need is to apply the cross-over after the above solutions are found. Thus i) **we evolve/iterate n swarms,** ii) **obtain convergence for each swarm which corresponds to particles are reached to local minimum points , only after then** iii) **we applied the cross-over operator between these swarms**.

```
1   % STEP 7. Apply cross-over
2   if (isCrossOver)
3    for(i=1:10)
4     selectSwarm1 = ceil(rand(1)*nSwarms);
5     selectSwarm2 = ceil(rand(1)*nSwarms);
6     while(selectSwarm1 == selectSwarm2)
7      selectSwarm2 = ceil(rand(1)*nSwarms);
8     end
9     if (rand(1)<p_cross) % cross-over
10     for(j=1:5)
11      r = ceil(rand(1)*nFeatures);
12      startInd1 = 1 + floor((selectSwarm1-1) * (nIndividuals/
            nSwarms));
13      startInd2 = 1 + floor((selectSwarm2-1) * (nIndividuals/
            nSwarms));
14      particle1 = startInd1 + floor(rand(1) * (nIndividuals/
            nSwarms));
15      particle2 = startInd2 + floor(rand(1) * (nIndividuals/
            nSwarms));
16      tmp1=particleBestPosition(swarmBestParticle(
            selectSwarm1),r);
17      tmp2=particleBestPosition(swarmBestParticle(
            selectSwarm1),r);
18      particleCurrentPosition(particle1,r) = tmp2;
19      particleCurrentPosition(particle2,r) = tmp1;
20     end
21   end
22  end
```

As given in the above code, first the two swarms whose particles will be crossed over are selected in lines 4 and 5. Then, based on a cross-over probability, the cross-over is applied to the selected particles (lines 14,15) from the two different swarms. In this work, *one point cross-over* is applied for 10 times in each iteration. The `isCrossOver` variable is enabled when all the swarms converge to their local or global minimum points. In the current implementation, instead of checking whether the swarms converge or not, we allow cross-over after a pre-determined number of iteration steps (number of function evaluations more correctly).

3 Experiments

The above two algorithms are evaluated using different benchmark optimization problems, which are the purely mathematical ones, and engineering problems. We are more concentrated on the mathematical problems, thus systematical experiments are done over them. Later, we applied the PSO and PSO-Cross in engineering problems. Before giving the results, we will first give the parameters of PSO and PSO-Cross, which are taken from [2]. Since the results in [2] and our work are not found to be consistent, we slightly changed the parameters.

3.1 Mathematical Optimization Functions

Two mathematical functions are tested in this work, namely *Rastrigin* and *Schwefel* functions.

Parameter	Parameter Description	Value
c_1	Weight of the particle's best position in determining particle's new velocity	3.0
c_2	Weight of the swarm's best position in determining particle's new velocity	3.0
w_{min}	Minimum value that inertia can get	0.34
w_{max}	The value inertia is initialized (for exploration)	1.4
α	The decreasing factor of intertia (decrease exploration, increase exploitation)	0.99
β	The decreasing factor of maximum velocity (decrease exploration, increase exploitation)	0.995
γ	The coefficient to determine initial maximum velocity based on range of $\boldsymbol{x}$	0.4
p_{cr}	The probability to apply craziness operator	0.22
N_{cr}	Number of particles to apply craziness operator	$N/5$
p_{cross}	The probability to apply cross-over operator after the swarms are converged (only for PSO-Cross)	0.2
N_{cross}	Number of times that *one-point crossover* is applied (only for PSO-Cross)	10

Table 1: Parameter set for the optimization algorithms

Rastrigin function is characterized by the "existence of many local optima evenly spaced in the problem domain" as stated in [2]. Figure 1 [1] demonstrates this function in two-dimensional case. The optimization problem can be stated as:

$$minf(\boldsymbol{x}) = 3.n + \sum_{i=1}^{n}[{x_i}^2 - 3.cos(2.\pi.x_i)]$$

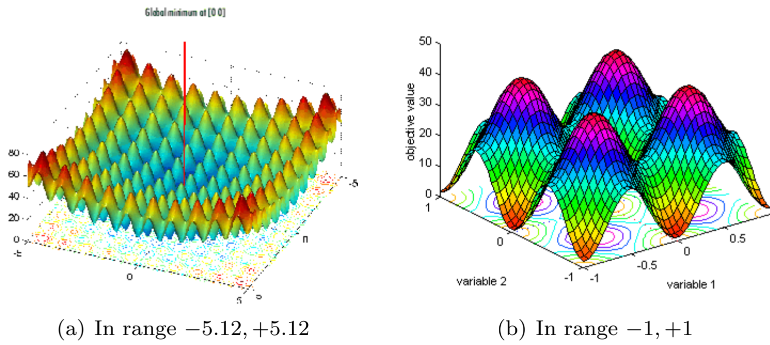
where

$$-5.12 \leq x_i \leq 5.12$$

The function gets its global minimum at $\boldsymbol{x}^* = (0, 0, .., 0)$, where $f(\boldsymbol{x}^*) = 0$ at this point.

Schwefel function has also many local optimum points. Additionally, it is interesting since "the second best optimum solution is far away from the global optimum". Thus it is very difficult to escape from this local

[1] The figure is taken from the web page `http://www.karnig.co.uk/ga/manual/image017.gif`



(a) In range $-5.12, +5.12$ (b) In range $-1, +1$

Figure 1: Rastrigin function is demonstrated. As shown, the global optima is obtained at $\boldsymbol{x} = [00]$.

optimum by making small moves, especially for large problem spaces. Figure 2 [2] demonstrates the plot of the function in two-dimensional space. The optimization problem is defined as:

$$minf(\boldsymbol{x}) = 418.9829.n + \sum_{i=1}^{n}[x_i.sin(\sqrt{|x_i|})]$$

where

$$-500 \leq x_i \leq 500$$

The function gets its global minimum at $\boldsymbol{x}^* = (420.9687, 420.9687, .., 420.9687)$, where $f(\boldsymbol{x}^*) = 0$ at this point.

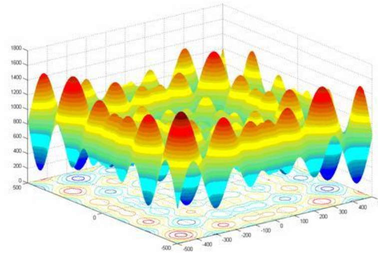


Figure 2: Schwefel function is demonstrated.

[2] Figure is taken from the web page: `http://www-optima.amp.i.kyoto-u.ac.jp/member/student/hedar/Hedar_files/TestGO_files/image12721.jpg`

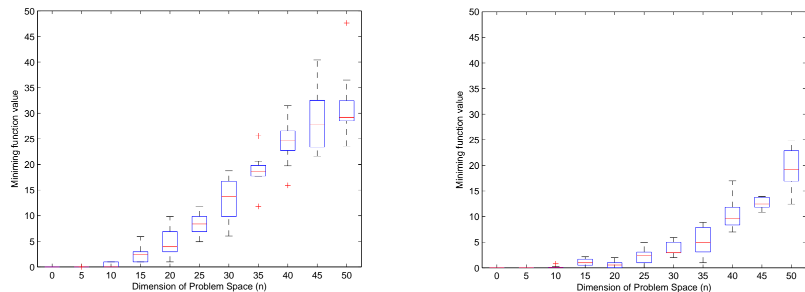
3.2 Results on mathematical optimization problems

Using the above parameters with PSO and PSO-Cross, we obtained different results for the two mathematical functions for different problem dimensions. The algorithm is limited with a maximum of $100,000$ function evaluations unless otherwise stated.

3.2.1 Comparison of PSO and PSO-Cross in Rastrigin function

We systematically increased the size of the problem space, by increasing the size of the position vector, n, from 1 to 30. For each n value, 10 different runs are performed with a swarm size of 80. Figure 3(a) shows the performance of PSO for increasing n, where the box plot corresponds to the average of the 10 independent runs. As shown, as with increasing n, the performance of PSO degrades, and for $n > 8$, it cannot find the optimum for $100,000$ function evaluations.

When PSO-Cross is applied for the very same problem, with same parameters and number of function evaluations, the performance of the algorithm is considerably increased as shown in Figure 3(b). The number of function evaluations is same $(100,000)$, and we used 4 different swarms, that are initially optimized by the original PSO algorithm, and later particles from different swarms are cross-overed as described in Section 2.2.



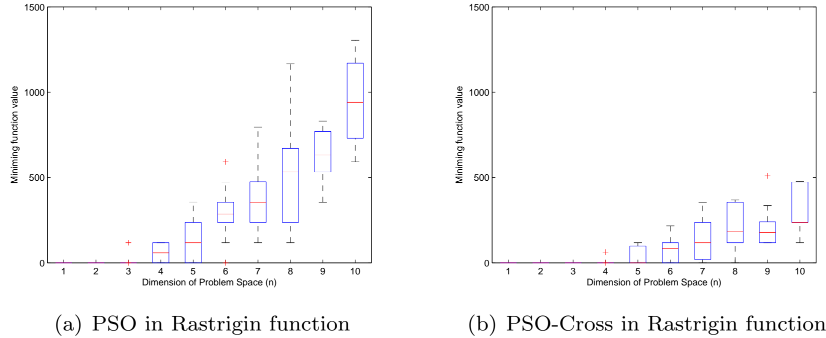
(a) PSO in Rastrigin function

(b) PSO-Cross in Rastrigin function

Figure 3: Comparison of PSO and PSO-Cross for increasing problem space in rastrigin function. Since for each problem size, 10 experiments are conducted, instead of giving one function value per n, we showed the distribution. The distribution of the resulting function after $100,000$ function evaluations are shown in the box-plot, where the red plus sign corresponds to outliers. The rectangular box has lines at the lower quartile, median, and upper quartile values. As shown, PSO-Cross gives better results than PSO with increasing problem space.

3.2.2 Comparison of PSO and PSO-Cross in Schwefel function

As in the previous section, PSO and PSO-Cross are compared, but now in Schwefel function. Schwefel function is a more difficult optimization function, thus both give worse results when compared to rastrigin function. However, as shown in Figure 4, PSO-Cross gets much better results for increasing problem space, when compared to PSO, for the same number of function evaluations.



(a) PSO in Rastrigin function

(b) PSO-Cross in Rastrigin function

Figure 4: Comparison of PSO and PSO-Cross for increasing problem space in schwefel function. More description of the figure can be found in Figure 3. As shown, PSO-Cross gives better results than PSO with increasing problem space.

3.3 Welded Beam Design Problem

To test the algorithms in real engineering problems, one of the benchmark problem, that is used in mechanical engineering, namely welded beam design is studied. The welded beam is demonstrated in Figure 5 [3]. In this problem, fabrication is tried to be minimized based on some constraints. These constraints include

- shear stress (represented by τ),
- bending stress (represented by σ),
- end deflection (represented by δ),
- buckling load on the bar (represented by P_c), and
- side constraints

This problem has four variables to be optimized, namely:

- x_1, thickness of the weld (represented by h in Figure)
- x_2, length of the welded joint (represented by l in Figure)
- x_3, width of the beam (represented by t in Figure), and
- x_4, thickness of the beam (represented by b in Figure)

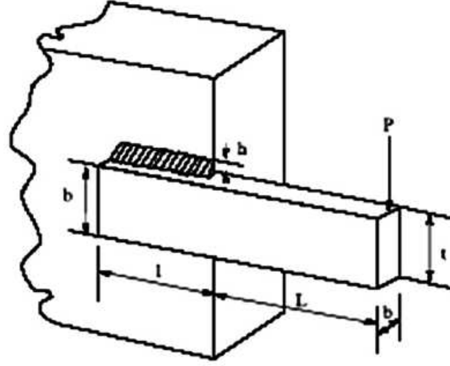


Figure 5: The welded beam design.

This problem could be formally defined as follows:

$$\min f(\boldsymbol{x}) = 1.10471 \cdot {x_1}^2 \cdot x_2 + 0.004811 \cdot x_3 \cdot x_4 \cdot (14 + x_2)$$

subject to

$$\begin{aligned}
g_1(\boldsymbol{x}) &= \tau(\boldsymbol{x}) - \tau_{max} \leq 0 \\
g_2(\boldsymbol{x}) &= \sigma(\boldsymbol{x}) - \sigma_{max} \leq 0 \\
g_3(\boldsymbol{x}) &= x_1 - x_4 \leq 0 \\
g_4(\boldsymbol{x}) &= 1.10471 \cdot {x_1}^2 \cdot x_2 + 0.004811 \cdot x_3 \cdot x_4 \cdot (14 + x_2) - 5 \leq 0 \\
g_5(\boldsymbol{x}) &= 0.125 - x_1 \leq 0 \\
g_6(\boldsymbol{x}) &= \delta(\boldsymbol{x}) - \delta_{max} \leq 0 \\
g_6(\boldsymbol{x}) &= P - P_c(\boldsymbol{x}) \leq 0
\end{aligned}$$

where

$$\begin{aligned}
\tau(\boldsymbol{x}) &= \mathrm{sqrt}(\tau' \cdot \tau' + 2 \cdot \tau' \cdot \tau'' \cdot \frac{x_2}{2 \cdot R} + \tau'' \cdot \tau'') \\
\tau' &= \frac{P}{\sqrt{2} \cdot x_1 \cdot x_2} \\
\tau'' &= \frac{M \cdot R}{J} \\
M &= P \cdot (L + x_2/2) \\
J &= 2 \cdot \left\{ \sqrt{2} \cdot x_1 \cdot x_2 \left(\frac{{x_2}^2}{12} + (\frac{x_1 + x_3}{2})^2 \right) \right\}
\end{aligned}$$

[3]Figure is taken from [2]

$$
\begin{aligned}
R &= \sqrt{0.25 \cdot ({x_2}^2 + (x1 + x3)^2)} \\
\sigma(\boldsymbol{x}) &= \frac{6 \cdot P \cdot L}{x_4 \cdot {x_3}^2} \\
\delta(\boldsymbol{x}) &= \frac{4 \cdot P \cdot L^3}{E \cdot {x_3}^3 \cdot x_4} \\
P_c(\boldsymbol{x}) &= \frac{4.013 \cdot E \cdot \mathrm{sqrt}({x_3}^2 \cdot {x_4}^6/36)}{L^2} \cdot \left(1 - \frac{x_3}{2 \cdot L} \cdot \sqrt{\frac{E}{4 \cdot G}}\right)
\end{aligned}
$$

with

$$P = 6{\times}10^3, L = 14, \delta_{max} = 0.25, E = 30{\times}10^6, G = 12{\times}10^6, \tau_{max} = 13600, \sigma_{max} = 30000$$

There are additional constraints on the optimized variables :

$$0.1 \le x_1 \le 2, \qquad 0.1 \le x_2 \le 10, \qquad 0.1 \le x_3 \le 10, \qquad 0.1 \le x_4 \le 2.$$

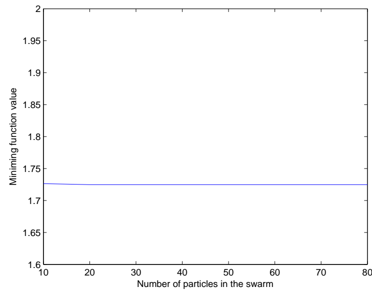
This is the formulation of a standard non-linear optimization problem with non-equality constraints. We will apply penalty methods to embed the constraints into the function and optimize one augmented function f_{aug}:

$$f_{\text{aug}}(\boldsymbol{x}) = f(\boldsymbol{x}) + r \cdot \sum_{i=1}^{7} \big(\max(0, g_i(\boldsymbol{x}))\big)^2$$

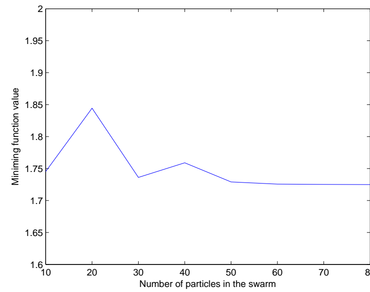
where $r = 10^6$ as suggested in [2].

3.4 Results in Welded Beam Design Problem

Using the above parameters with PSO and PSO-Cross, we obtained similar results in this engineering problem. Furthermore, the function that is obtained at the end of the PSO algorithm is found to be same in [2] (Table 10). As shown in Figure 6(a), for different swarm sizes, we found $f_{\text{aug}} = 1.72485$ in all different random runs. This is better than PSOA and PSOStr results that are given in [2]. Figure 6 shows the results of PSO and PSO-Cross for different swarm sizes. As shown, PSO is more successful than PSO-Cross. I think that, the unsuccess in PSO-Cross is due to the relatively small number of function evaluations. Since the number of function evaluations is small, the cross-over operator is applied much earlier than the occurance of convergence and fine-tuning in different swarms.



(a) PSO in Welded beam design



(b) PSO-Cross in Welded beam design

Figure 6: Comparison of PSO and PSO-Cross for increasing swarm sizes. As shown, while PSO is not affected by the swarm size, PSO-Cross seems to be affected from it. We think that, PSO-Cross is not really affected from the size of the swarm, but due to the relatively small number of function evaluations and iterations, it cannot fine-tune before the cross-over operator is applied.

References

[1] Kennedy, J. and Eberhart, R. C. “Particle swarm optimization”, *Proc. IEEE int’l conf. on neural networks* Vol. IV, pp. 1942-1948. IEEE service center, Piscataway, NJ, 1995.

[2] Dimonopoulos G. G. “Mixed-variable engineering optimization based on evolutionary and social metaphors”, *Computer methods in applied mechanics and engineering*, No: 196, pp. 803-817, 2007.