**EVALUATION OF COMMON ALGORITHMS IN SOLVING OBJECTIVE FUNCTION OF TRANSMISSION LOSS**

Đỗ Đức Hoàng, Lê Minh Khôi, Huỳnh Bảo Thanh

# Introduction

## **1.1.Genetic Algorithm (GA):**

**\*Overview**

-The genetic algorithm is a method for solving both constrained and unconstrained optimization problems that is inspired by evolutionary biology such as inheritance, mutation, selection, and crossover (also called recombination).

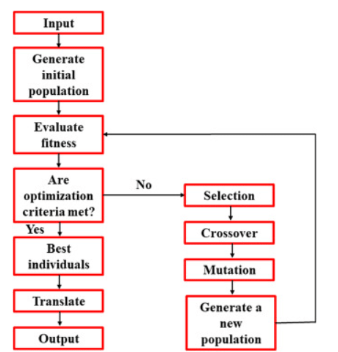
-The genetic algorithm uses three main types of rules at each step to create the next generation:

+ Selection rules select the individuals, called parents, that contribute to the population at the next generation. The selection is generally stochastic, and can depend on the individuals' scores.

+ Crossover rules combine two parents to form children for the next generation.

+ Mutation rules apply random changes to individual parents to form children.

**\*Operation of GA**



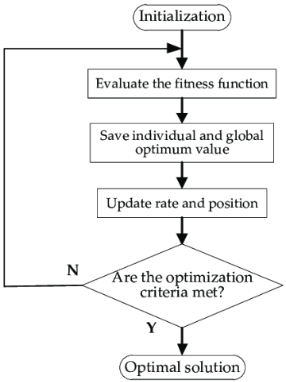
## 1.2.Particle Swarm Optimization (PSO)

**\*Overview**

Particle swarm is a population-based algorithm. A collection of individuals called particles move in steps throughout a region. At each step, the algorithm evaluates the objective function at each particle. After this evaluation, the algorithm decides on the new velocity of each particle. The particles move, then the algorithm reevaluates.

-The inspiration for the algorithm is flocks of birds or insects swarming. Each particle is attracted to some degree to the best location it has found so far, and also to the best location any member of the swarm has found. After some steps, the population can coalesce around one location, or can coalesce around a few locations, or can continue to move.

**\*Operation of PSO**

****

## 1.3.ABC

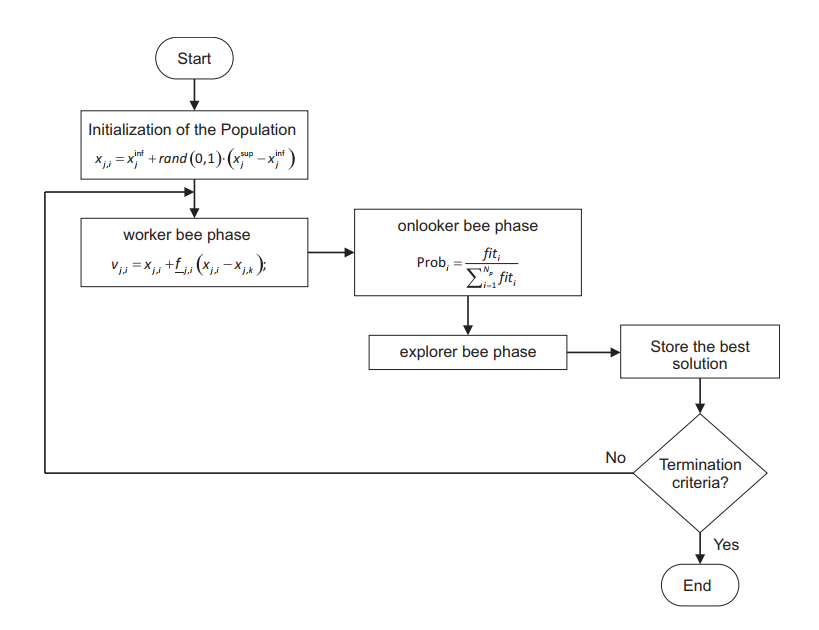
The Artificial Bee Colony (ABC), proposed by Dervis Karaboga in 2005, is among the most commonly used algorithms for optimization. The algorithm functions like what its name implies, a bee colony. In detail, the method utilizes 3 essential components of a bee colony which are food source locations, amount of nectar, and different types of bees (worker, onlooker and explorer).

1. **Food source locations:** possible solutions computed by the algorithm.
2. **Amount of nectar:** how accurate those solutions are compared to the target result.
3. **Difference types of bees:** these bees are just representations of each phase in the computation process.
   1. **Worker bees:** locate themself at a food source, calculate the nectar amount of their current and nearby food sources.
   2. **Onlooker bees:** take the information provided by the worker bees and pick the best food locations.
   3. **Explorer bees:** these are ex-worker bees that have exhausted their food source and they now carry the role of finding new food sources.

The diagram below shows the computation process of the ABC algorithm.

* **Step 1:** Initializes a random set of food source locations inside the specified range of the problem. Calculate the nectar (fitness value) of the set of food sources.
* **Step 2:** Worker bee phase: randomly select neighboring food sources around each current one and assess their nectar.
* **Step 3:** Onlooker bee phase: from those newly discovered food sources, evaluate and select the best one of each worker bee as the new official food source.
* **Step 4:** Explorer bee phase: randomly select a number of new food sources that equal the remaining amount of worker bees after the selection.

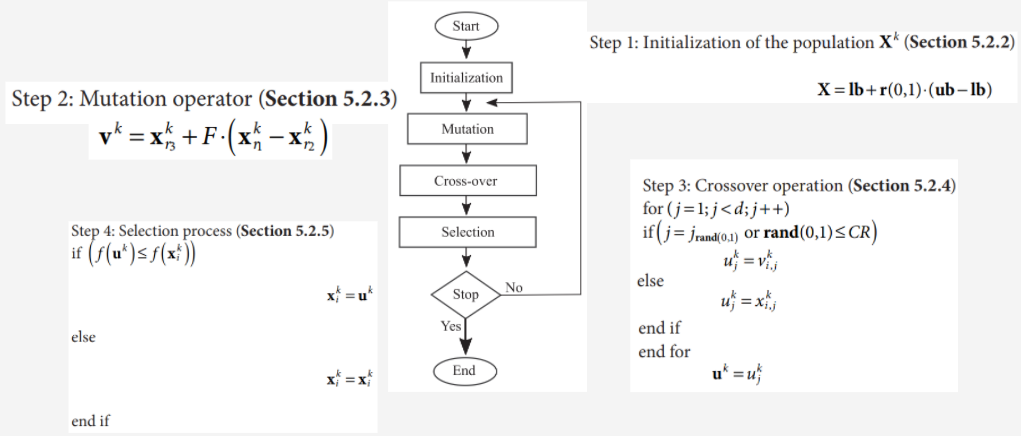
The loop continues until a termination condition is reached.



## 1.4.DE

The DE algorithm have ability to find the optimal global solution of multimodal, nondifferentiable and nonlinear functions following 4 steps:

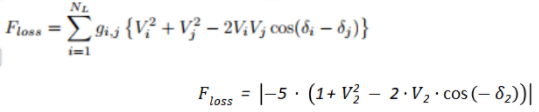
1. Initialize the population.
2. Apply mutation to explore.
3. Increase the diversity by crossover operators.
4. Select the individuals which have the best function value.



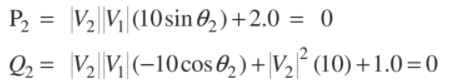
# Code

## Objective Function

### Objective Function



### Constraints



## Main Code

### GA

| %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% % Real-parameter GA implementation % Erik Cuevas, Alma Rodríguez %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% clear all % Two-dimensional objective function funstr=' abs(-5\*(1+x ^2 - 2\*x\*cos(-y)))'; % Search space range range=[0.9 1.05 -pi/2 pi/2]; f=vectorize(inline(funstr)); % Parameter initialization sample = 1; n\_sample = 30; Ma = zeros(n\_sample, 4); % Matrix storing [evaluation fitnessXbest] for each k tim = zeros(n\_sample,1); n=30; %Population size N MaxGen=1500; % Maximal number of generations pc=0.95; % Crossover probability pm=0.05; % Mutation probability %% Start exp while(sample <= n\_sample) rg=1000000; cg=20; tic % Generation of the first population (Section 1) xrange=range(2)-range(1); yrange=range(4)-range(3); xn=rand(1,n)\*xrange+range(1); yn=rand(1,n)\*yrange+range(3); % The optimization process begins for i=1:MaxGen, % Fitness evaluation  for i=1:n zn(i)=f(xn(i),yn(i)); g1 = abs(xn(i))\*(10\*sin(yn(i)))+2; g2 = abs(xn(i))\*(-10\*cos(yn(i)))+abs(xn(i))^2\*10+1; pelnaty = rg\*g1^2 + rg\*g2^2; zn(i) = zn(i)+pelnaty; end % A new population is generated applying the ranking selection for z=1:n e = selectionRM(zn); % Section 2 MP(z,1)=xn(e); MP(z,2)=yn(e); end for z1=1:2:n %The parents p1 and p2 are selected p1=floor(n\*rand)+1; p2=floor(n\*rand)+1; %Crossover is applied (simulated binary crossover SBX) if pc<rand, % Section 3 [NP(z1,:),NP(z1+1,:)]=crossoverRE(MP(p1,:),MP(p2,:)) ; else %Otherwise the parents remain NP(z1,:)=MP(p1,:); NP(z1+1,:)=MP(p2,:); end % Mutation is executed (Normal mutation) if pm<rand, mu1=NP(z1,:); mu2=NP(z1+1,:); % Section 4 NP(z1,:)=mutateRE(mu1); NP(z1+1,:)=mutateRE(mu2); end end % The new population is integrated xn=NP(:,1); yn=NP(:,2); % The final results are shown end [fit index] = min(zn); Ma(sample,1) = sample; Ma(sample,2) = f(xn(index), yn(index)); Ma(sample,3:4) = [xn(index) yn(index)]; t = toc; tim(sample) = t; sample = sample +1; end %% Deviation calculate B = mean(Ma(:,1)); Deviation = sqrt((1/n\_sample)\*sum(Ma(1) - B)); figure(1) hold on plot(Ma(:,1),Ma(:,2),'MarkerSize',10); xlabel('evaluations');ylabel('Best fitness'); hold off filename = 'GA\_result.xlsx'; writematrix(Ma,filename,'Sheet',1); writematrix(tim,filename,'Sheet',2); writematrix([Deviation min(Ma(:,2)) max(Ma(:,2)) min(tim) max(tim)],filename,'Sheet',3) |
| --- |

| function [c,d]=crossoverRE(a,b) % Dispersion factor nc=1; % Random number U[0,1] r=rand; % SBX crossover if r<=0.5 mu=(2\*r)^(1/(nc+1)); else |
| --- |

| mu=(1/(2\*(1-r)))^(1/(nc+1)); end % Generation of individuals for each dimension c1x=((1+mu)\*a(1)+(1-mu)\*b(1))\*0.5; c1y=((1+mu)\*a(2)+(1-mu)\*b(2))\*0.5; c2x=((1-mu)\*a(1)+(1+mu)\*b(1))\*0.5; c2y=((1-mu)\*a(2)+(1+mu)\*b(2))\*0.5; c(1)=c1x; c(2)=c1y; d(1)=c2x; d(2)=c2y; end  function anew=mutateRE(a) % Perturbation size sigma=0.1; % Two-dimensional mutation mx=a(1)+randn\*sigma; my=a(2)+randn\*sigma; anew(1)=mx; anew(2)=my; end |
| --- |

| function [iE] = selectionRM(fP) % It is assigned a rank to each solution % The worst has rank 1 % the best the rank N Ps=length(fP); [D I] = sort(fP,'descend'); % Configured for maximization r = 1:length(D); r(I) = r; fP=r; suma=0; % The roulette-wheel process for k=1:Ps P(k)=fP(k)/(sum(fP)); suma=suma+P(k); A(k)=suma; % cumulative probability end R=rand; for u=1:Ps if (A(u)>=R) break end end % iE is the selected element iE=u; end |
| --- |

### PSO

| % PSO clear all % Clear memory close all % close MATLAB Windows  %% Problem to optimize (minimize), the definition of the objective function funObj = @(x) abs(-5\*(1+x(1).^2 - 2\*x(1)\*cos(-x(2)))); %% Parameter configuration N = 30; % The number of particles d = 2; % Dimensions lb = [0.9 -pi/2]; ub = [1.05 pi/2]; % Upper limit of search the space k = 1; % Current iteration kmax = 10000; % Maximum number of iterations  c1 = 100; % Cognitive constant c2 = 2; % Social constant %penalty parameters rg = 10000000; cg = 20; sample = 1; n\_sample = 30; Ma = zeros(n\_sample, 4); % Matrix storing [evaluation fitnessXbest] for each k tim = zeros(n\_sample,1); while(sample <= n\_sample) tic %% Initialization of particles and velocity for i = 1:N x(i,:) = rand(1,d).\*(ub-lb)+lb; % Initialization of particles v(i,:) = zeros(1,d); % Initialization of velocities end  %% Evaluation of the initial particles in the objective function for i = 1:N xi=x(i,:); % Extraction of the particle xi fx(i,:) = funObj(xi); % Evaluation of the particle xi g1 = abs(xi(1))\*(10\*sin(xi(2)))+2;  g2 = abs(xi(1))\*(-10\*cos(xi(2)))+abs(xi(1))^2\*10+1;  pelnaty = rg\*g1^2 + rg\*g2^2;  rg = rg\*cg;  fx(i,:)=fx(i,:)+pelnaty; end %% Record of the best global particle and the best local particles [gfit, ind] = min(fx); % Fitness of the best global particle g = x(ind,:); % Position of the best global particle fp = fx; % Fitness of the best local particles p = x; % Position of the best local particles no iterations until now  % Computation of the meshgrid %% Iterative process while k <= kmax % Stop criterion  % new iteration  %% Computation of the new velocity for each particle  for i= 1:N % Extraction of the particle xi xi = x(i,:); % Extraction of the local particle pi pi = p(i,:); % Determination of the new velocity for each particle vi v(i,:) = v(i,:)+c1\*rand(1,d).\*(pi-xi)+c2\*rand(1,d).\*(g-xi); end %% Determination of the new position of each particle x = x + v; %% Verify that the particles do not leave the limits lb and ub for i = 1:N % For each particle for j=1:d % For each dimension if x(i,j) < lb(j) % Check the lower limit x(i,j) = lb(j); elseif x(i,j) > ub(j) % Check the upper limit x(i,j) = ub(j); end end end %% Evaluation of new particles with the objective function for i = 1:N xi = x(i,:); % Extraction of the particle xi fx(i,:) = funObj(xi); % Evaluation of the particle xi g1 = abs(xi(1))\*(10\*sin(xi(2)))+2;  g2 = abs(xi(1))\*(-10\*cos(xi(2)))+abs(xi(1))^2\*10+1;  pelnaty = rg\*g1^2 + rg\*g2^2;  rg = rg\*cg; end %% Record of the best global particle and the best local particles [gfitkplus1, ind] = min(fx); % If a better solution was found, update the global particle if gfitkplus1 < gfit %Particle Swarm % Update the fitness of the best global particle gfit = gfitkplus1; % Update the position of the best global particle g = x(ind,:); end for i = 1:N % If any particle is a better solution than the previous, % update your best local particle if fx(i,:) < fp(i,:) % Update the fitness of the best local particles fp(i,:) = fx(i,:); % Update the position of the best local particles p(i,:) = x(i,:); end end %% Historical record of the best solutions found in each generation k = k + 1; Evolution(k) = gfit; end Ma(sample,1) = sample-3; Ma(sample,2) = funObj(g); Ma(sample,3:4) = g; t = toc; tim(sample) = t; sample = sample +1; clc end  %% End of the iterative process, display of results figure % Graph of the evolutionary process of the PSO plot(Evolution) %% Export result B = mean(Ma(:,1)); Deviation = sqrt((1/n\_sample)\*sum(Ma(1) - B)); figure(1) hold on plot(Ma(:,1),Ma(:,2),'MarkerSize',10); xlabel('evaluations');ylabel('Best fitness'); hold off filename = 'PSO\_result.xlsx'; writematrix(Ma,filename,'Sheet',1); writematrix(tim,filename,'Sheet',2); writematrix([Deviation min(Ma(:,2)) max(Ma(:,2)) min(tim) max(tim)],filename,'Sheet',3) |
| --- |

### ABC

| % ABC clear all close all func='abs(-5\*(1+x ^2 - 2\*x\*cos(-y)))';  f=vectorize(inline(func)); range=[0.9 1.05 -pi/2 pi/2]; % Initial parameter sample = 1; n\_sample = 30; Ma = zeros(n\_sample, 4); % Matrix storing [evaluation fitnessXbest] for each k tim = zeros(n\_sample,1); d=2; %Dimensions Np=30; %Population size food\_source=round(Np/2); % Food sources gmax=150; % Maximum number of generations limit=15; % Abandonment criterion %penalty parameters rg = 10000000; cg = 20; Range=[range(2)-range(1) range(4)-range(3)];  while(sample <= n\_sample) tic % Random initial population Pop = (rand(food\_source,d) .\* Range) + [range(1) range(3)];  % Evaluate the fitness values for ii=1:food\_source ValFit(ii)=f(Pop(ii,1),Pop(ii,2)); % Compute the relative fitness Fitness(ii)=calculateFitness(ValFit(ii)); end  % The counters are initialized test=zeros(1,food\_source); % The best solution is updated BestInd=find(ValFit==min(ValFit)); BestInd=BestInd(end); GlobalMin=ValFit(BestInd); GlobalParams=Pop(BestInd,:); g=1; % Generation counter while ((g <= gmax)) % Worker bee phase for i=1:(food\_source) % The parameter to modify is randomly selected Param2Change=fix(rand\*d)+1; % A random solution is used to produce a new % mutant solution, both must be different neighbor=fix(rand\*(food\_source))+1; while(neighbor==i) neighbor=fix(rand\*(food\_source))+1; end solutions=Pop(i,:); % It is applied: v\_{ij}=x\_{ij}+\phi\_{ij}\*(x\_{kj}-x\_ {ij}) solutions(Param2Change)=Pop(i,Param2Change)+(Pop(i,Param2Change)-Pop(neighbor,Param2Change))\*(rand-0.5)\*2; % If the value of the generated parameter is outside the % limits, It is taken to the nearest limit ind=find(solutions<range(3)); solutions(ind)=range(3); ind=find(solutions>range(4)); solutions(ind)=range(4); % The new candidate solution is evaluated ValFitSol=f(solutions(1),solutions(2)); g1 = abs(solutions(1))\*(10\*sin(solutions(2)))+2; g2 = abs(solutions(1))\*(-10\*cos(solutions(2)))+abs(solutions(1))^2\*10+1; pelnaty = rg\*g1^2 + rg\*g2^2; rg = rg\*cg; ValFitSol =ValFitSol + pelnaty; FitnessSol=calculateFitness(ValFitSol); % A Greedy selection criterion is applied between the % current solution and the one produced (mutant), and the % best among them is preserved if (FitnessSol>Fitness(i)) Pop(i,:)=solutions; Fitness(i)=FitnessSol; ValFit(i)=ValFitSol; test(i)=0; else test(i)=test(i)+1; end end % End of the worker bee phase % Probabilities are calculated using normalized fitness values probab=(0.9.\*Fitness./max(Fitness))+0.1; % Onlooker bee phase i=1; t=0; while(t<food\_source) t=t+1; if(rand<probab(i)) % The parameter to be modified is randomly selected Param2Change=fix(rand\*d)+1; % A random solution is used to produce a new % solution. neighbor=fix(rand\*(food\_source))+1; while(neighbor==i) neighbor=fix(rand\*(food\_source))+1; end solutions=Pop(i,:); % applied: v\_{ij}=x\_{ij}+\phi\_{ij}\*(x\_{kj}-x\_ {ij}) solutions(Param2Change)=Pop(i,Param2Change)+(Pop(i,Param2Change)-Pop(neighbor,Param2Change))\*(rand-0.5)\*2; % If the value of the generated parameteris outside % the limits,it is taken to the nearest limit ind=find(solutions<range(1)); solutions(ind)=range(1); ind=find(solutions>range(2)); solutions(ind)=range(2); % The new solution is evaluated ValFitSol=f(solutions(1),solutions(2)); g1 = abs(solutions(1))\*(10\*sin(solutions(2)))+2; g2 = abs(solutions(1))\*(-10\*cos(solutions(2)))+abs(solutions(1))^2\*10+1; pelnaty = rg\*g1^2 + rg\*g2^2; rg = rg\*cg; ValFitSol=ValFitSol+pelnaty; FitnessSol=calculateFitness(ValFitSol); % A Greedy selection criterion is applied between the  % current solutionand the candidate solution if (FitnessSol>Fitness(i)) Pop(i,:)=solutions; Fitness(i)=FitnessSol; ValFit(i)=ValFitSol; test(i)=0; else test(i)=test(i)+1; end end i=i+1; if (i==(food\_source)+1) i=1; end end % The best food source is stored ind=find(ValFit==min(ValFit)); ind=ind(end); if (ValFit(ind)<GlobalMin) GlobalMin=ValFit(ind); GlobalParams=Pop(ind,:); end % End of the onlooker bee phase % Explorer bee phase % Food sources whose "limit" value is reached are determined ind=find(test==max(test)); ind=ind(end); if (test(ind)>limit) test(ind)=0; solutions=(Range).\*rand(1,d)+[range(1) range(3)]; ValFitSol=f(solutions(1),solutions(2)); g1 = abs(solutions(1))\*(10\*sin(solutions(2)))+2; g2 = abs(solutions(1))\*(-10\*cos(solutions(2)))+abs(solutions(1))^2\*10+1; pelnaty = rg\*g1^2 + rg\*g2^2; rg = rg\*cg; ValFitSol=ValFitSol+pelnaty; FitnessSol=calculateFitness(ValFitSol); Pop(ind,:)=solutions; Fitness(ind)=FitnessSol; ValFit(ind)=ValFitSol; end g=g+1; % The iteration is increased % The position and fitness of the best individual are displayed %disp(GlobalMin) %disp(GlobalParams) end Ma(sample,1) = sample; Ma(sample,2) = f(GlobalParams(1),GlobalParams(2)); Ma(sample,3:4) = GlobalParams; ti = toc; tim(sample) = ti;  sample = sample + 1; end %% Deviation calculate B = mean(Ma(:,1)); Deviation = sqrt((1/n\_sample)\*sum(Ma(1) - B)); figure(1) hold on plot(Ma(:,1),Ma(:,2),'MarkerSize',10); xlabel('evaluations');ylabel('Best fitness'); hold off filename = 'ABC\_result.xlsx'; writematrix(Ma,filename,'Sheet',1); writematrix(tim,filename,'Sheet',2); writematrix([Deviation min(Ma(:,2)) max(Ma(:,2)) min(tim) max(tim)],filename,'Sheet',3) |
| --- |

### DE

| %% DE clear all close all %% Function definition %fitness = @(xi) (xi(1)^2) + (xi(2)^2); fitness = @(x) abs(-5\*(1+x(1).^2 - 2\*x(1)\*cos(-x(2)))); %% Parameters % Maximum number of iterations Kmax = 1500; sample = 1; n\_sample = 30; Ma = zeros(n\_sample, 4); % Matrix storing [evaluation fitnessXbest] for each k tim = zeros(n\_sample,1); % Population size N = 30; % Scaling factor F F = 0.2; % Crossover factor CR CR = 0.5; % Number of dimensions dim = 2; % Search space limits lb = [0.9 -pi/2]; ub = [1.05 pi/2]; % Penalization Function definition rg = 10000000; cg = 20; %% Start exp while(sample <= n\_sample) tic %% Initialization of the population if size(ub,2)==1  X= lb + rand(N,dim).\*(ub-lb);% X = lb + r(0,1) \* (ub - lb) end if size(ub,2)>1  for j = 1:N  for i=1:dim  high=ub(i);down=lb(i);  X(j,i)=down + rand(1,1).\*(high-down);  end  end end % The initial population is evaluated for i = 1:N  fitnessX(i,1) = fitness(X(i,:));  g1 = abs(X(i,1))\*(10\*sin(X(i,2)))+2;  g2 = abs(X(i,1))\*(-10\*cos(X(i,2)))+abs(X(i,1))^2\*10+1;  pelnaty = rg\*g1^2 + rg\*g2^2;  rg = rg\*cg;  fitnessX(i,1) = fitnessX(i,1) + pelnaty ; end % The best element is selected [fitnessXbest,ind] = min(fitnessX); xBest = X(ind,:);  evaluations = 1; %% Iterative process while (evaluations <= Kmax)  for i = 1:N  %% Mutation operation  % Three different individuals from the population are selected  index = randperm(N);  a = index(1); % First random element  b = index(2); % Second random element  c = index(3); % Third random element   d = index(4);  e = index(5);  % Mutant vector  %v = X(c,:) + F \* (X(a,:) - X(b,:)); %rand / 1  %v = X(e,:) + F \* (X(a,:) - X(b,:)) + F \* (X(c,:) - X(d,:)); %rand / 2  %v = xBest + F \* (X(a,:) - X(b,:)); %best / 1  %v = xBest + F \* (X(a,:) - X(b,:)) + F \* (X(c,:) - X(d,:)); %best / 2  v = X(i,:) + F \* (xBest - X(i,:)) + F \* (X(a,:) - X(b,:));%cur-bes  % It is evaluated if the mutant vector v is within the search space  % search defined by upper and lower limits  if size(ub,2)==1  for it = 1:dim  if (v(it) < lb)  v(it) = lb;  end  if (v(it) > ub)  v(it) = ub;  end  end  end  if size(ub,2) > 1  for it = 1:dim  high=ub(it);down=lb(it);  if (v(it) < down)  v(it) = down;  end  if (v(it) > high)  v(it) = high;  end  end  end  %% Crossover operation  u=zeros(1,dim);  j0 = randi([1 dim]);  for j=1:dim  if j==j0 || rand<= CR  u(j)=v(j);  else  u(j)=X(i,j);  end  end  % The fitness of the vector i is obtained  fitnessI = fitness(X(i,:));  % the fitness of the test vector u is evaluated  fitnessU = fitness(u);  g1 = abs(u(1))\*(10\*sin(u(2)))+2;  g2 = abs(u(1))\*(-10\*cos(u(2)))+abs(u(1))^2\*10+1;  pelnaty = rg\*g1^2 + rg\*g2^2;  fitnessU = fitnessU+pelnaty;    %% Selection operation  if fitnessU < fitnessI  X(i,:) = u;  if fitnessU < fitnessXbest  fitnessXbest = fitnessU;  xBest = u;  end  end  end  % Graph of the current population  %}  %{  Ma(evaluations,1) = evaluations;  Ma(evaluations,2) = fitnessXbest;  %}  evaluations = evaluations + 1;  %fprintf('Iteration: %d\n',evaluations); end  Ma(sample,1) = sample; Ma(sample,2) = fitness(xBest); Ma(sample,3) = xBest(1); Ma(sample,4) = xBest(2); t = toc; tim(sample) = t; sample = sample +1; clc end %fprintf('Best Solution: (%d,%d)\n',xBest(1),xBest(2)); %fprintf('Best fitness: %d\n',fitnessXbest); disp(Ma); %% Export Result B = mean(Ma(:,1)); Deviation = sqrt((1/n\_sample)\*sum(Ma(1) - B)); disp(t) figure(1) hold on plot(Ma(:,1),Ma(:,2),'MarkerSize',10); xlabel('evaluations');ylabel('Best fitness'); hold off filename = 'DE\_result.xlsx'; writematrix(Ma,filename,'Sheet',1); writematrix(tim,filename,'Sheet',2); writematrix([Deviation min(Ma(:,2)) max(Ma(:,2)) min(tim) max(tim)],filename,'Sheet',3) |
| --- |

# Observation

After experimenting with all 4 algorithms, we have concluded their ranking in terms of accuracy and computational time in the table below. First of all, it is clear that the deviations of each measurement are relatively small which indicates that all algorithms in our setup are consistent and run normally. The first category shown in the table is power transmission loss which is dominated by the ABC algorithm with an average loss of 0.0645. On the other hand, the algorithm with the worst power loss of 4.9841 is GA. In the middle of the list are PSO and DE with 3.8611 and 2.8528 of power loss respectively which are significantly better than GA. In the second category, the 4 algorithms are displayed in descending computation time of one loop. On the top of the list is the GA algorithm which has the longest loop cycle of 4.984 seconds. Further down the ranking are the DE and ABC with 0.2317s and 0.1937s respectively which leaves PSO at the bottom with the fastest computation time of a stunning 0.0371s.

