

Computer Aided Applied Single Objective Optimization
Dr. Prakash Kotecha
Department of Chemical Engineering
Indian Institute of Technology, Guwahati

Lecture - 16
Implementation of Genetic Algorithm on MATLAB

Welcome back. In the session, we will look at the Implementation of Genetic Algorithm. So, we will be directly implementing real coded genetic algorithm. So, for that, we will be using the SBX crossover operator which we have discussed and for mutation, we will be using the polynomial mutation that we have discussed right. So, before we actually implementing genetic algorithm, let us quickly have a recap of the pseudo code of genetic algorithm that will help us to quickly implement the algorithm right.

(Refer Slide Time: 00:58)

Pseudocode

Input: Fitness function, lb, ub, N_p , T, C_r , P_m , η , θ

1. Initialize a random population (P)
2. Evaluate fitness (f) of P

for t = 1 to T

Perform tournament selection of tournament size, k

for i = 1 to $N_p/2$

Randomly choose two parents

if $r \leq p_t$

Generate two offspring using SBX-crossover

Bound the offspring and store them in offspring population

else

Copy the selected parents and their fitness to offspring population

end

end

end

for i = 1 to N_p

if $r \leq p_m$

Perform polynomial mutation of i^{th} solution in offspring population

Bound the mutated offspring

else

No change in i^{th} solution of offspring

end

end

end

Evaluate the fitness

Combine parent and offspring population to perform selection

end

$$\beta = \begin{cases} (2u)^{1/\eta} & \text{if } u < 0.5 \\ \left(\frac{1}{2(1-u)}\right)^{1/\eta} & \text{otherwise} \end{cases}$$

$$O_1 = 0.5[(1+\beta)X_1 + (1-\beta)X_2]$$

$$O_2 = 0.5[(1-\beta)X_1 + (1+\beta)X_2]$$

$$\delta = \begin{cases} (2r)^{1/\theta} & \text{if } r < 0.5 \\ 1 - [2(1-r)]^{1/\theta} & \text{if } r \geq 0.5 \end{cases}$$

$$y = O + (ub - lb)\delta$$

3

Pseudo code, as we have seen we need to provide the fitness function so that the objective function can be evaluated. The lower and upper bounds, so this 3 constitute the details about the problem. We need to give the population size, the number of generations we want to carry out because that would be our termination criteria, the mutation and crossover probability indicated by p_c p_m and for a SBX crossover, we required distribution index for crossover and η_m indicates the distribution index required for the polynomial mutation and k is going to be the number of candidates involved in the tournaments.

So, in our case we will take k is equal to 2. So, the code which we have written is on the assumption that k is equal to 2. So, we will have competition between two solutions and the winner will get into the mating pool right. So, in our case for the code which we are going to discuss, k is fixed to be 2. So, the first step is to initialize a random population within the lower and upper bounds and then, we need to evaluate the fitness of the population right. Once we have done this, we have this iteration loop; exterior iteration loop. The steps given in this loop. We will have to be performed capital T times right.

So, for every iteration, we need to do a tournament selection right. So, in each tournament, the number of candidates competing will be k . So, in our case it is going to be 2. So, based on this tournament selection, we will select N_p population members right. Once we are done with that in this for loop, we will be implementing the crossover operator. So, since there are N_p solutions and we require 2 members per crossover. The number of crossover should 1 to N_p by 2 right.

So, again because of this criteria N_p has to be a even number. So, in this case, we will randomly choose 2 parents. We will generate a random number if that is less than equal to crossover probability, we will perform crossover; else we will directly copy the selected parents and their fitness to offspring population right. If we happen to do cross over then, we will be using this SBX crossover which is given over here right.

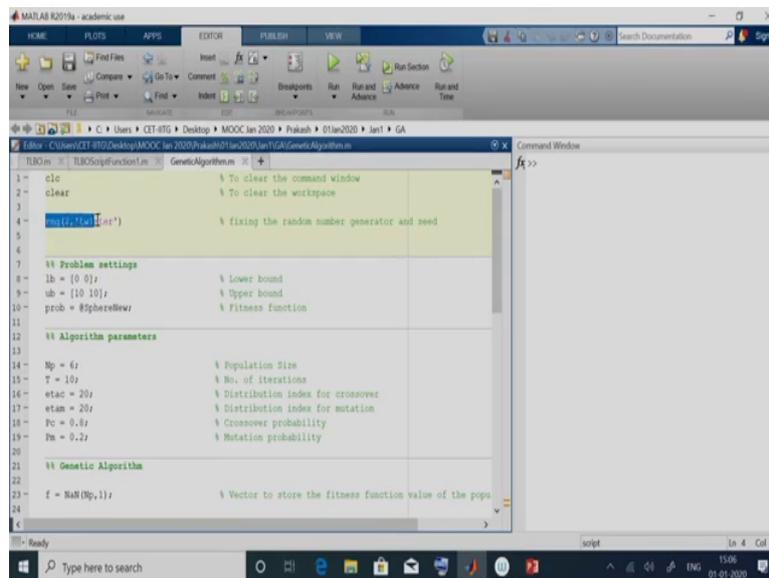
So, for each variable, we will have to generate a random number between 0 and 1 depending upon whether it is less than equal to 0.5 or greater than 0.5, we will have to calculate the beta

value. And using the beta value, we need to calculate the offspring 1 and the offspring 2 ok. So, that is going to be the crossover.

So, once crossover is performed, we need to do mutation for each of the N_p members right on the offspring. So, at the end of crossover, we will have offspring. So, we will be performing mutation on the offerings right. So, again, we will have to generate a random number if that is less than equal to mutation probability, we will be performing the polynomial mutation; else we will not change the solution.

And then, once we complete mutation for all the N_p members, we will evaluate the fitness of all of them and then combine the parent and offspring population to select the best N_p members. So, we will be employing a μ plus λ strategy. μ is our parents; λ indicates the off springs which we were regenerated. So, we will combine all the solutions which we have μ plus λ and we will select the best N_p members. So, this is the pseudo code of real coded genetic algorithm right. So, let us see its implementation.

(Refer Slide Time: 04:23)

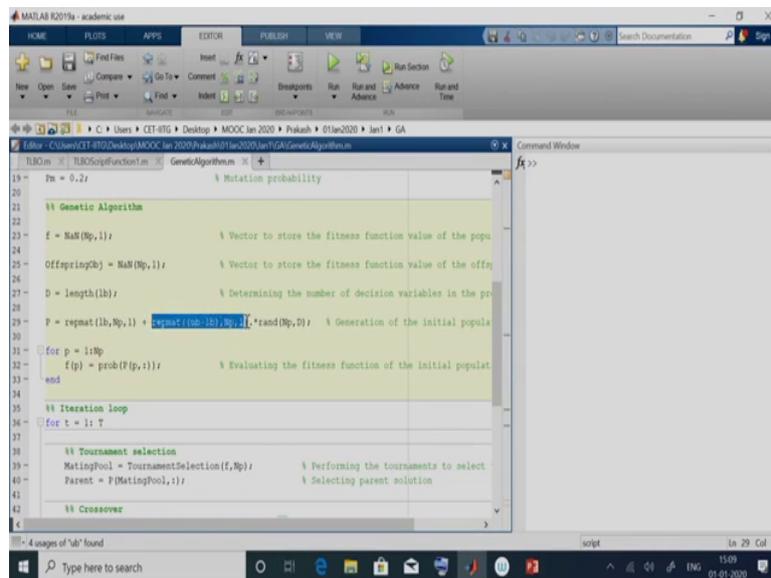


```
1 = clc % To clear the command window
2 = clear % To clear the workspace
3
4 rng('twister',2) % fixing the random number generator and seed
5
6
7 %% Problem settings
8 lb = [0 0]; % Lower bound
9 ub = [10 10]; % Upper bound
10 prob = @sphereNew; % Fitness function
11
12 %% Algorithm parameters
13
14 Np = 6; % Population Size
15 T = 10; % No. of Iterations
16 etac = 20; % Distribution index for crossover
17 etam = 20; % Distribution index for mutation
18 Pc = 0.6; % Crossover probability
19 Pm = 0.2; % Mutation probability
20
21 %% Genetic Algorithm
22
23 f = NaN(Np,1); % Vector to store the fitness function value of the popu
24
```

So, let us first quickly go through the code and then, we will execute it line by line. So, these two statements are to just clear the command window and the workspace. This statement, line 4 ensures that we get the same set of random numbers, every time we execute this code right. So, we are using the rng inbuilt function of MATLAB right.

We have to use this rng function, we need to provide a seed and an algorithm. So, there are various Algorithms to select the random number, we have chosen the twister Algorithm and we have set the seed to be 2. So, every time we run this algorithm, we will get the same values as long as we do not change this right. Despite the fact that it is a stochastic Algorithm this. rng ensures that we get the same set of random numbers every time be executed.

(Refer Slide Time: 05:12)



```
19 - Pm = 0.2; % Mutation probability
20
21 %% Genetic Algorithm
22 - f = NaN(Np,1); % Vector to store the fitness function value of the popula
23
24 - OffspringObj = NaN(Np,1); % Vector to store the fitness function value of the offspr
25
26 - D = length(lb); % Determining the number of decision variables in the problem
27
28 - P = repmat(lb,Np,1) + repmat((ub-lb)/Np,1) * rand(Np,D); % Generation of the initial popula
29
30
31 - for p = 1:Np
32 -     f(p) = prob(F(p,:)); % Evaluating the fitness function of the initial popula
33 - end
34
35 %% Iteration loop
36 - for t = 1:T
37
38     %% Tournament selection
39     MatingPool = TournamentSelection(f,Np); % Performing the tournaments to select
40     Parent = P(MatingPool,:); % Selecting parent solution
41
42     %% Crossover
```

So, these three things are straight forward, we are defining the lower bound of the problem, upper bound of the problem and we are defining the function sphearnew right into the variable prob. So, these are the 6 parameters that are required for executing GA; one is the population size, the number of iterations, the distribution index for crossover, the distribution index for mutation, the crossover probability and the mutation probability.

So, our results are sensitive to these values which we have taken right. So, these values have to be provided. So, we are not providing k over here because the code itself is built for k is equal to 2, as we will see when we are writing the tournament selection function, we will be comparing only 2 solutions. So, that is inherently fix in this code.

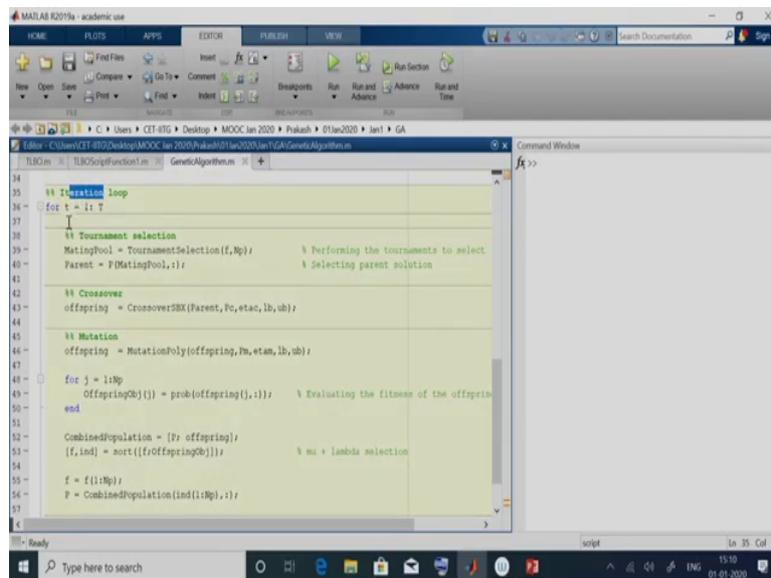
So, then we define some of the variables which will be required right. So, again as usual we will define f right. So, f is to contain the fitness function values of the Np population members.

Initially, we are initializing it with NaN right as and when we will calculate, we will assign the appropriate value right. This offspring Obj will contain the fitness function value of the offsprings. From the parents, we will generate off springs. We will evaluate its fitness function value. So, those should be stored in this variable offspring right Obj. So, again the size of offspring will be the same as the number of population members.

So, the number of decision variables of the problem can be calculated by determining the length of lower bound right. So, lower bound in this case is 2. So, D would take a value of 2. So, as and when we change the number of elements in the lower bound, D would automatically get evaluated and subsequently, it will impact the rest of the program right. So, line 29 is the generation of the initial population right. So, we are replicating the lower bound N_p times plus we are determining the range right that is the range is $ub - lb$. We are replicating that N_p times right.

So, this second part, this highlighted part will have a dimension of N_p cross D right. Similarly, we are generating N_p cross D random numbers between 0 to 1 and then, we are doing a element to element multiplication right. So, this line will help us to determine the population. At the end of line 29, we would have generated the population members.

(Refer Slide Time: 07:36)

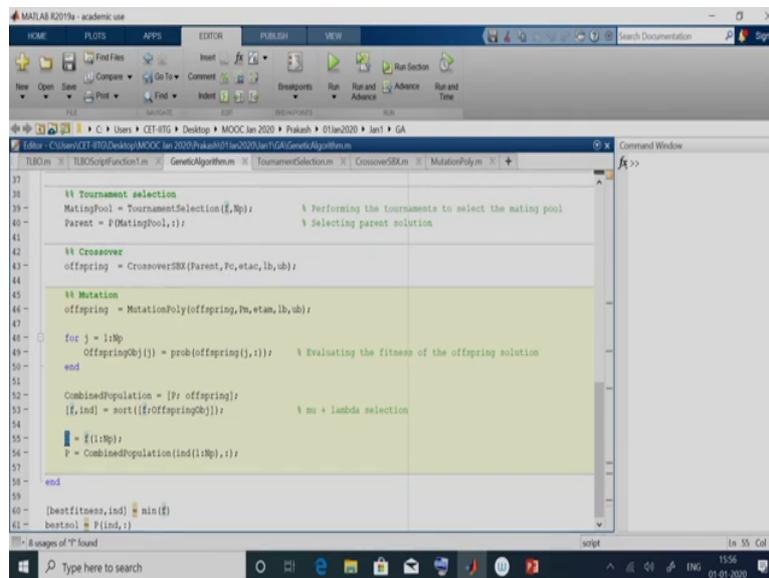


```
34 %% Tournament loop
35 for t = 1:T
36     %
37     %% Tournament selection
38     MatingPool = TournamentSelection(f,Np); % Performing the tournaments to select
39     Parent = F(MatingPool,:); % Selecting parent solution
40
41     %% Crossover
42     offspring = Crossover2BX(Parent,Pc,etac,lb,ub);
43
44     %% Mutation
45     offspring = MutationPoly(offspring,Pm,etam,lb,ub);
46
47     for j = 1:Np
48         OffspringObj(j) = prob(offspring(j,:)); % Evaluating the fitness of the offspring
49     end
50
51     CombinedPopulation = [P; offspring];
52     [f,ind] = sort(f(OffspringObj)); % mu + lambda selection
53
54     f = f(1:Np);
55     P = CombinedPopulation(ind(1:Np),:);
56
57 end
```

So, 31 to 33 in this for loop, we are determining the fitness function value for each member right. So, this loop runs for N_p times right and each time, we are sending the p th row. So, this P of p comma full colon will ensure that we are sending the entire p th row. We are sending it to the function handle `prob` function; handle `prob` nothing but `shpearnew` or whatever function, we define right.

So, that we will be evaluating the fitness of each member right. So, line 31 to 33 will help us to evaluate the fitness of the randomly generated population. Once that fitness is evaluated, we are ready to begin the genetic algorithm right.

(Refer Slide Time: 08:26)



```
37
38 %% Tournament selection
39 MatingPool = TournamentSelection(f,Np); % Performing the tournaments to select the mating pool
40 Parent = P(MatingPool,:); % Selecting parent solution
41
42 %% Crossover
43 offspring = CrossoverSBX(Parent,Pc,etac,lb,ub);
44
45 %% Mutation
46 offspring = MutationPoly(offspring,Pm,etam,lb,ub);
47
48 for j = 1:Np
49     OffspringObj(j) = prob(offspring(j,:)); % Evaluating the fitness of the offspring solution
50 end
51
52 CombinedPopulation = [P; offspring];
53 [f,ind] = sort([f;OffspringObj]); % ms + lambda selection
54
55 P = f(1:Np);
56 P = CombinedPopulation(ind(1:Np),:);
57
58 end
59
60 [bestfitness,ind] = min(f)
61 bestsol = P(ind,:)
```

So, this is the iteration loop; line 36 to 58 whatever is there in between line 36 to 58 will be implemented capital T times because we want to perform T iterations right which is a user defined parameter. So, genetic algorithm if you see broadly there are three operations that we need to perform; one is the tournament selection right. From the tournament selection, we will get the mating pool.

This mating pool will have to be used for crossover right and mutation. So, first will perform tournament selection, we will select the mating pool, this mating pool will be used for crossover and will generate offspring, those offspring will be used for mutation.

So, will use three function files right. Tournament selection, we are going to implement in a separate function file. Crossover; SBX crossover we will be implementing in another function file and polynomial mutation, we will be implementing in another function file right.

So, and we will be just accessing those 3 files. So, here if we see so, the three functions are tournament selection, crossover and mutation poly. So, from tournament selection, we will get the index of the population members which will constitute the mating pool right and then, we are accessing those members right. So, mating pool will be a not a set of solution, but it will merely indicate the index of the solution right.

So, since we know the index, we are accessing those population members and we are giving it to the variable parent right. So, once this parent is fed into this crossover SBX operator right, we will get the offspring. This offspring is intern fed into the mutation poly function which is nothing but polynomial mutation right and then, we will get the final offspring right. So, for the final offspring, we will evaluate the objective function right. The number of off springs will be equal to the population size right and then, we will be combining the population right.

So, this is the initial population P, we are combining with offspring right and that will be the combined population. Similarly, we are combining the objective function like stacking one below the other right and then we are sorting right. Once we sort we will get f and we will also get a track of which variable which value has been sent to which position right.

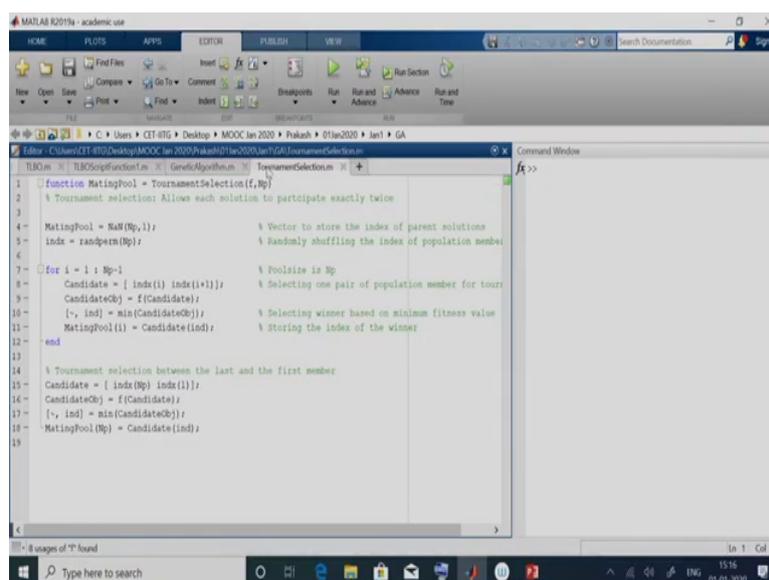
So, that will be stored in this ind. So, once we have this f and ind, so we are in need of only N_p population member. So, the size of f will be $2 N_p$; size of i n d will be $2 N_p$ right, but we require only the first N_p values right. So, what will do is we will say f which is for the subsequent generation, only the first N_p values. So, we are extracting the first N_p values and storing it in f.

Similarly, P is nothing but the combined population which we have determined over here right and then, we are accessing only the first N_p elements of i n d right. So, i n d will have the indexes which are on the top. So, we are selecting the first N_p solutions right. First N_p

solutions indicated by this variable `ind`. So, we are extracting those solutions right and then, we are storing it in the variable `P` right. So, this what is going to happen.

So, now let us look into the tournament selection operator. The input to this tournament selection is the fitness function value of the N_p members and the size of the population N_p right. So, only those two things are to be effect, we do not need to decide which are the solutions. Because, if you remember, when we form the mating pool, we were only determining which solution is better right. We were not looking at the decision vector right. We are only looking at the fitness function value. So, that is why we are sending only the fitness function value over here right.

(Refer Slide Time: 12:01)



```
function MatingPool = TournamentSelection(f,Np)
% Tournament selection: Allows each solution to participate exactly twice
%
4- MatingPool = NaN(Np,1); % Vector to store the index of parent solutions
5- ind = randperm(Np); % Randomly shuffling the index of population member
6
7- for i = 1 : Np-1 % Poolsize is Np
8- Candidate = [ ind(i) ind(i+1)]; % Selecting one pair of population member for tournament
9- CandidateCb = f(Candidate);
10- [~, ind] = min(CandidateCb); % Selecting winner based on minimum fitness value
11- MatingPool(i) = Candidate(ind); % Storing the index of the winner
12- end
13
14 % Tournament selection between the last and the first member
15- Candidate = [ ind(Np) ind(1)];
16- CandidateCb = f(Candidate);
17- [~, ind] = min(CandidateCb);
18- MatingPool(Np) = Candidate(ind);
19
```

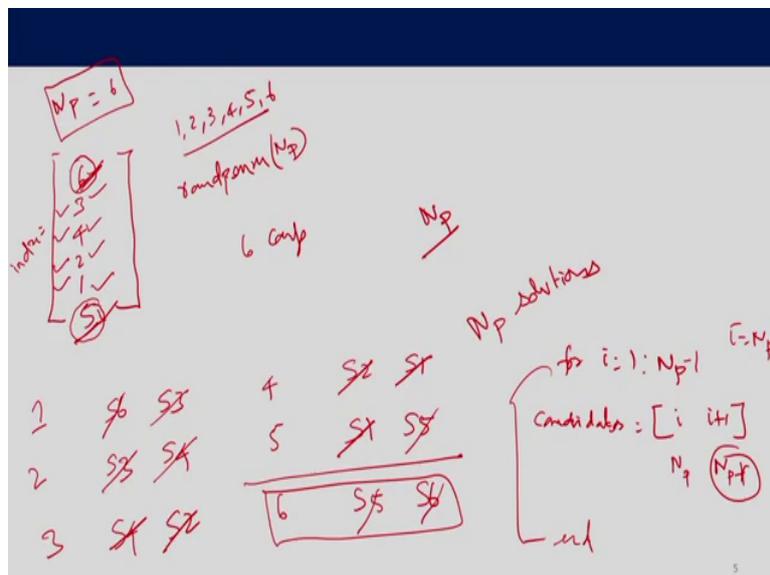
So, let us look into this tournament selection right. In tournament selection what we receive from the main script is the fitness function value and the size of the population right. Here, we

have explicitly given the size of the population, otherwise that can also be determined using the length of `f` right. So, this tournament selection operator ensures that each solution gets to participate in 2 tournaments right.

So, that is how we had looked that the tournament selection. We are implementing a binary tournament selection where in each tournament there are 2 solutions and each solution in the population will get 2 opportunities. So, initially we are defining this mating pool as NaN with the size of `Np` right.

So, we are ultimately going to store in this mating pool as to which of the solutions not exactly the solutions, their locations right which of the solutions would constitute the mating pool right. So, that is what is going to be in this mating pool. In this line, we are using the `randperm` function.

(Refer Slide Time: 13:01)



Let us assume that N_p is equal to let us say 6 right. So, what we will do is we will first shuffle the number 1 to 6 right. So, the numbers which we have is 1, 2, 3, 4, 5 and 6 right. So, we will use this `randperm` function in MATLAB. When we do `randperm` of N_p right. So, it will help us to shuffle these numbers. So, let us say that this shuffled numbers are 6, 3, 4, 2, 1, 5. So, all the numbers would occur once; 1, 2, 3, 4, 5, 6 right. So, this is what will be indicating by the variable index right.

Since, our population sizes 6, we need to conduct 6 competitions right. So, how we are going to conduct the competition is the 2 neighbors are going to compete right. So, the first competition is going to be between solution 6 and solution 3 right; these two. The second competition is going to be between solution 3 and solution 4 right. So, that would be this competition.

The third competition would be between solution 4 and solution 2. This is the third completion. Fourth competition is going to be between solution 2 and solution 1 right. Fifth competition is going to be between solution 1 and solution 5 right. Now, to conduct the 6th competition right, we have solution 5 all the other members except 6 and 5 have already participated twice.

So, for example, S 3 has participated twice; S 4 has participated twice; S 2 also has participated twice; S 1 also has participated twice right. So, only S 5 and S 6 have participated once right it. So, that is because this 6 is located over at the top. So, the last competition is going to be conducted between the last value of this index and the first value of this index right.

So, that would be between S 5 and S 6. This is the way we are going to conduct N_p tournaments right. So, from each tournament, we will get 1 solution. So, at the end of it will get N_p solutions and every solution would have got an opportunity to compete twice. So, with this let say this is S 5 also got twice; S 6 also has got 2 opportunities right. So, this is how we are going to implement the tournament selection ok.

So, when we are implementing tournament selection right. So, we will obviously be running loop from i is equal to 1 to N_p and the members would be; so, members will call it as candidates right. So, the candidates would be i and $i + 1$ right. So, the first time when we do i , i and $i + 1$, 1 2 will be fine.

The second time when i is equal to 2 3 would be fine right and similarly for all the variable will happen and then, when i is equal to N_p right, we will have a problem because we are trying to access the N_p th solution and $N_p + 1$ solution which we do not have right. So, that is why we will run this loop for i is equal to 1 to $N_p - 1$ right.

So, we will be conducting till this 5 competitions and this competition, we will be conducting separately out of this for loop right; whatever happens in the for loop that is there and out of this for loop, we will conduct 1 more competition that competition will be for the last member in this index and the first member in this index. So, that competition will be conducted separately right. So, this index will shuffle the solutions and give them in a random order right of the N_p solutions which we have right. So, what we are doing is we are running 1 to $N_p - 1$ times our population size is N_p . So, our pool size is also to be N_p .

So, in this case what we are doing is first we are selecting $N_p - 1$ solution, then what we are saying is we are selecting the candidates who will compete right. So, this ensures that there are $N_p - 1$ tournaments right. So, from each tournament, we will get 1 winner right. So, when this loop is completed 7 to 12 is completed, we will have selected $N_p - 1$ solution. We are supposed to select N_p solutions; but right now, we are selecting only $N_p - 1$ solutions right.

So, what we are doing here? We are selecting the i th value and the $i + 1$ th value. So, that what I had indicated to you that the first and second to subsequent solutions would be participating in a particular tournament right. So, we are taking the 2 neighbors. So, the neighbor is i and $i + 1$. So, these are the indexes of those are put in candidate and then, over here in line 9, we are accessing their objective function values or the fitness function values.

So, now we know the 2 solutions which are participating and their fitness function value right. So, the fitness function value is stored in candidate Obj, the indexes of the solution are stored in candidate right. So, now, since we need to work with this fitness function value, we are finding out which is the minimum of 2.

So, this candidate Obj its value will always be 2 cross 1; 2 rows 1 column because we have selected 2 candidates. We are determining the minimum value in it right and we are not interested in the value a such right. We are only interested in the winner. So, the winner would be indicated by the position, i n d right and then, we access that particular candidate and put it inside the mating pool right. So, mating pool of i is going to be the winner candidate. This will be able to see with an example right, once we get into the debug mode and when we executed this part of the code will be much clearer.

So, far we have selected in Np minus 1 solutions. So, here what we are doing is we are selecting the candidate, index of Np right; index of Np would be the last value in that in this variable index right and the first value. We are now selecting these 2 solutions. So, that is candidate. So, it will store the index. Once we store the objective function values of those 2 candidates in this candidate Obj, we can find out where is the minimum located right. So, depending upon where this minimum is located, using this variable ind, we access that particular candidate and we dump it into the mating pool right.

So, this is the tournament selection which is implemented. So, it is a binary tournament, every member will be able to participate exactly twice using this code. So, that is why k is not given as a user defined parameter. Inherently, we have assumed k to be 2 when we wrote this piece of code. So, that completes our discussion on tournament selection.

Obviously, for some of you it would be still little bit problematic to understand, but once we run this code, we will be able to actually see what is happening. Its similar to that example that we used to solve; once you solve an example things become a lot more easier to understand. So, the same thing over here, once we execute this code with sample problem, you will be able to better understand this tournament selection.

So, let us go back to the genetic algorithm script right. So, from the tournament selection, we obtain the mating pools. This function will provide the mating pool. When we say mating pool, it will be just a vector. A vector containing integers right. So, let us say the vector contains values 2 4 8 9, it means the 2nd solution, the 4th solution, the 8th solution, the 9th solution comprise the MatingPool right. The entire solution vector would not be there, but only their location or their indexes would be available in the MatingPool.

So, once we know the indexes, we can actually extract the entire solution using the population right. So, that is what we will be doing over here. So, in this line 40, that is what we are doing right. So, parent is equal to P of MatingPool right. So, those particular rows and their corresponding columns all the columns corresponding to that selected rows right.

So, mating pool will be a vector. So, let us say MatingPool is 2 4 8. So, what we are doing is this entire second row, the entire fourth row and the entire eighth row is selected and stored in this variable parent. So, that is the reproduction operator. In the reproduction operator or the selection operator, we have determined the MatingPool; the solutions which are reasonably good right which can go for crossover.

So, coming to this SBX cross over operator. So, there we will have to pass on the values of the parent right. We will have to give the crossover probability whether crossover needs to happen for a particular pair or not that would be determined by the crossover probability. This η_c is the crossover index which be required in the equation for SBX operator and the lower bound and the upper bound right.

So, there is no guarantee that the solutions which are generated by SBX crossover operator will be within the bounds right. So, if it is out of the bound, we will have to bound it right. So, to bound this we are actually passing this lower bound and upper bound right and what we are expecting is the offspring from crossover operator. So, let us look into this crossover SBX; before actually looking into this crossover SBX let us quickly recap SBX crossover.

(Refer Slide Time: 22:37)

Simulated Binary Crossover (SBX)

➤ Compute the β as

$$\beta = \begin{cases} (2u)^{1/\eta_c} & \text{if } u \leq 0.5 \\ \left(\frac{1}{2(1-u)}\right)^{1/\eta_c} & \text{otherwise} \end{cases}$$

η_c is probability distribution mean

➤ Offspring are generated as

$$O_1 = 0.5[(1+\beta)X_1 + (1-\beta)X_2]$$

$$O_2 = 0.5[(1-\beta)X_1 + (1+\beta)X_2]$$

X_1	Parent 1
X_2	Parent 2
O_1	Offspring 1
O_2	Offspring 2

Simulated Binary Crossover for Continuous Search Space, Complex systems, 9(2), 115-148, 1995
Multi Objective Optimization Using Evolutionary Algorithms, John Wiley & Sons, Inc, 2001

So, for SBX crossover first we will have to select 2 solutions and then, we will have to decide whether they are going to go for cross over or not, that will be decided by random number which we select. Random number if we say it 0.2 and for a cross over probability is 0.8. So, 0.2 is less than 0.8. So, that particular pair will undergo crossover right.

So, if that particular pair comes for crossover, what we will have to do is we will have to first generate u. u is a random number between 0 and 1. So, for each variable will have a random number. So, if we have D decision variables, we will how to generate D random numbers between 0 to 1 and for each of them will have to check this condition whether random number is less than or equal to 0.5 or greater than 0.5 right, depending upon that value we will have to calculate beta.

Since, u has to be selected for each variable, we will have a beta value for each variable. Once we calculate beta value, then we know the parents right. So, once the beta values are known using the two parents X_1 and X_2 or let us say P_1 and P_2 right. So, X_1 and X_2 you might be confusing it with the decision variables. So, to distinguish that we can use P_1 and P_2 . So, these are the two parents which are undergoing crossover right.

So, once we know beta we can use these two equations to generate two new off springs right. So, this is how we will be generating to off springs using SBX crossover right and if the random number which we selected happens to be greater than crossover probability, then we will merely copy the parents as there off springs right; then there is no crossover of operator, but the parents are copied as the off springs right.

(Refer Slide Time: 24:19)

```

1 function offspring = CrossoverSBX(Parent,Pc,eta0,ub)
2
3 [Np,D] = size(Parent); % Determining the no. of population and decision variable
4 inds = randperm(Np); % Permutating numbers from 1 to Np
5 Parent = Parent(inds,:); % Randomly shuffling parent solutions
6 offspring = NaN(Np,D); % Matrix to store offspring solutions
7
8 for i = 1:2:Np % Selecting parents in pairs for crossover
9
10     r = rand; % Generating random number to decide if crossover is to
11
12     if r < Pc % Checking for crossover probability
13
14         for j = 1:D
15
16             z = rand; % Generating random number to determine the Beta value
17
18             if z <= 0.5 % Calculating beta value
19                 beta = (2*z)/(eta0+1);
20             else % Calculating beta value
21                 beta = (1/(2*(1-z)))/(eta0+1);
22             end
23
24             offspring(i,j) = 0.5*((1+beta)*Parent(i,j) + (1-beta)*Parent(i+1,j)); % Genera

```

So, for this SBX crossover operator, we will have to provide the parent right. The crossover probability the distribution index η c the lower and upper bound right. So, first what we do is we determine as to how many members are there and what is the size of the problem or how many decision variables are there. So, size of parent gives the number of rows and number of columns.

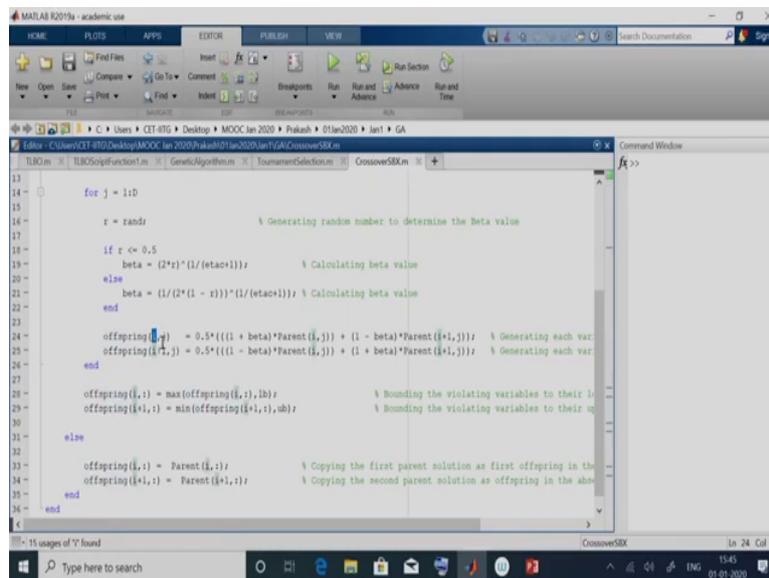
The number of rows will indicate the population size and the number of columns would indicate the number of decision variables right. So, again, we will be using the randperm function right. So, the randperm function as we have seen previously will help us to shuffle the numbers. The numbers from 1 to N_p right. So, we will shuffle the numbers. So, this will only give the location right.

And then, what we are doing is we are arranging the parents right as per the this index which we are obtained in this line 4. So, in line 4, we will merely get the indexes. Now, this parent is randomly shuffled right. This we are doing because at the end of a particular iteration, we will be sorting right. So, since we are sorting the population, the good solutions will be at the top right; whereas, when we are doing crossover, we are supposed to randomly select right.

So, what we are going to do is we are going to select the first solution and second solution and since the fitness function are sorted, we will be pairing 2 good solutions. So, we want to avoid that. So, that is why we are using this randperm function to shuffle the solutions right. Once the solutions are shuffled it might happen that the first solution is good the second solution need not be the second best solution right. So, that way it will help us to create a diverse population right.

So, we create this parent population and then we will be using this variable offspring to store the off springs which we generate right. So, the size of this is non priory that there will be in N_p rows because there are N_p parents and there will be D columns because the number of columns in a offspring will be equal to the number of decision variables right. So, this is just we are pre- allocating the memory.

(Refer Slide Time: 26:19)



```
23
24     for j = 1:D
25
26         r = rand; % Generating random number to determine the beta value
27
28         if r <= 0.5
29             beta = (2*r)^(1/(eta+1)); % Calculating beta value
30         else
31             beta = (1/(2*(1 - r)))^(1/(eta+1)); % Calculating beta value
32         end
33
34         offspring(i,j) = 0.5*((1 + beta)*Parent(i,j)) + (1 - beta)*Parent(i+1,j); % Generating each var
35         offspring(i+1,j) = 0.5*((1 - beta)*Parent(i,j)) + (1 + beta)*Parent(i+1,j); % Generating each var
36     end
37
38     offspring(i,:) = max(offspring(i,:),lb); % Bounding the violating variables to their l
39     offspring(i+1,:) = min(offspring(i+1,:),ub); % Bounding the violating variables to their u
40
41     else
42
43         offspring(i,:) = Parent(i,:); % Copying the first parent solution as first offspring in the
44         offspring(i+1,:) = Parent(i+1,:); % Copying the second parent solution as offspring in the abo
45     end
46 end
```

So, once that is done, we are going to implement the crossover right. So, for crossover we are implementing this loop. So, this for loop, we are implementing from 1 in interval of 2 to Np right. So, the first time the value of i will be 1; second time the value of i would be 3; the third time the value of i would be 5 right.

So, it will jump in steps of 2; 1, 3 and 5 and so on till we get this Np right. So, that we are doing because we will be taking the first solution and second solution. So, second time when we are doing crossover, we will be taking the third solution and the fourth solution; when the third time we are implementing crossover, we will be selecting the fifth solution and the sixth solution right. So, that is why we skip in terms in intervals of two. So, the first thing is to decide whether we need to perform crossover or not for a particular pair.

We generate a random number using this rand function, the value would be stored in r and then, we check over here whether r is less than Pc right. If r is less than Pc, then we need to perform a crossover which will be happening in the section from line 14 to line 26 right in this section crossover would be happening. So, to perform the crossover, we have D variables right.

So, here we have chosen to implement it through a for loop so that its easier to understand for those of you who are not very comfortable working with vectorize code, but this could have been vectorized right. So, what we will be doing here is first will say j is equal to 1, we will generate a random number right. If that random number is less than or equal to 0.5, then we will use this equation to calculate the value of beta, else will use this equation to calculate the value of beta. So, remember there were two equations for generating the value of beta.

So, depending upon the value of random numbers, we had to select one of those equation right. So, this r was to decide whether we need to perform crossover or not. This r helps us to determine which of those two equations would be used to find out the value of beta right. So, once we have generated the value of beta, we can find out the 2 offspring right.

So, the jth variable of the ith offspring would be $0.5 \times (1 + \beta)$ into parent of i comma j. So, beta would be a scalar value right; $1 + \beta$ into parent of i comma j. So, the ith parent right plus $1 - \beta$ into parent of i plus 1. So, the second parent. So, first time when we are doing i will be 1. So, parent of 1 comma 1 plus $1 - \beta$ into parent of 2 comma 1. So, the first variable of the second parent right.

So, similarly we generate the second offspring; just that instead of this plus, we have a minus over here and instead of this minus, we have a plus over here. So, that is how our 2 equations were right. So, this will help us to generate the 2 off springs right. So, once we have generated off springs, we need to make sure that they are bounded. So, here we check for the bounds. So, this for loop is completed for all the D variables. Once this is completed, we check for the newly generated solutions. So, the newly generator solution is the ith solution and the i plus 1 solution.

So, we check whether it is in the lower bound and the upper bound whether it is in between lower and upper bound, if not we employ the usual corner bounding strategy to bound it right. So, this the condition if we decide to perform crossover right, but if we decide not to perform the crossover because of this randomly generated number right, then we will directly copy the parent and since, parent is copied we do not have a bounding procedure over there. So, with this we will be able to generate N_p offerings right.

Why N_p offspring. This loop does not run for N_p times; but each time the loop runs, we generate 2 offspring and the loop will run for N_p by 2 times and each time, we will get 2 offspring. So, at the end of the completion of SBX crossover operator for all the solutions right we will have a N_p solutions; N_p solutions which are known as offspring. So, here one thing that you need to remember is that we are generating 2 off springs right.

So, offspring i as well as offspring i plus 1. So, both of them need to be bounded. So, here if you see in line 28 29 what we are doing is we are bounding for the both the solutions i th offspring and i plus 1th offspring, we are checking for the lower bound right. So, the offspring is indicated by offspring i comma colon. So, this the highlighted portion indicates the offspring. This is the lower bound and we are using the max operator.

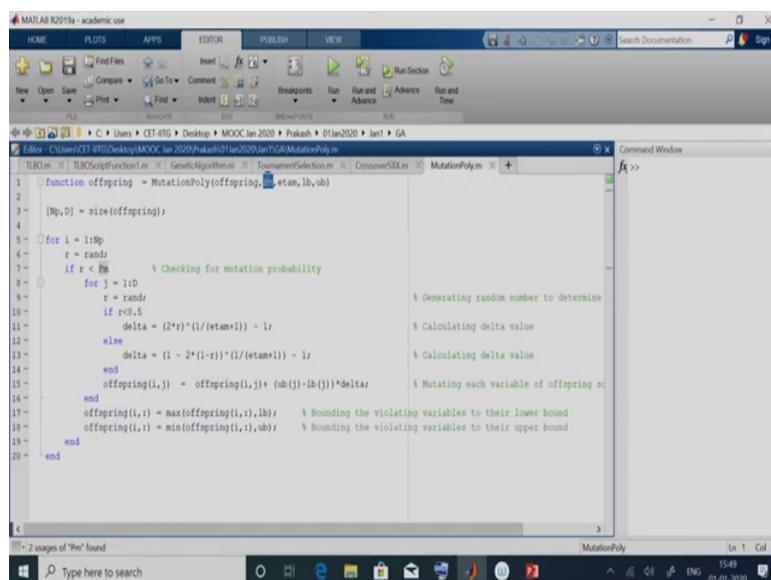
So, what we will get over here when line 28 is that the i th offspring no longer violates the lower bound, after this line is executed. Similarly, this line 29 is for making sure that the i plus 1th offspring does not violate the lower bound right. So, both of this lines only take care about the lower bound and similarly both of these lines line 30 and 31 take care about the upper bound right. So, the i th offspring, we are using the main function to make sure that it does not violate the upper bound and similarly, line 31 we are bounding the i plus 1th offspring with the upper bound.

So, this you need to make sure that both the solutions do not violate the upper and lower bound right. So, that is why we have 4 lines over here. Once, we have calculated the offspring right, we need to go back; then, we need to perform the polynomial mutation. So, again will be providing this offspring to the polynomial mutation function. So, here we have used the

function mutation poly right. So, mutation is done using polynomial mutation that is why it is mutation poly. We need to give the offspring which were generated from crossover.

We need to provide the mutation probability; we need to give the distribution index for mutation right and the lower bound and upper bound similar to our crossover because in mutation also it is not guaranteed that the solution would be between upper and lower bounds right. So, if it is out of bounds will have to bounded right. So, now, let us look into this polynomial mutation function. So, this is our function right.

(Refer Slide Time: 32:43)



```
1 function offspring = MutationPoly(offspring, etam, lb, ub)
2
3 [Np, D] = size(offspring);
4
5 for i = 1:Np
6     r = rand;
7     if r < etam % Checking for mutation probability
8         for j = 1:D
9             r = rand; % Generating random number to determine
10            if r < 0.5 % Calculating delta value
11                delta = (2*r)^(1/(etam+1)) - 1;
12            else % Calculating delta value
13                delta = (1 - 2*(1-r))^(1/(etam+1)) - 1;
14            end
15            offspring(i,j) = offspring(i,j) + (ub(j)-lb(j))*delta; % Mutating each variable of offspring
16        end
17        offspring(i,:) = max(offspring(i,:), lb); % Bounding the violating variables to their lower bound
18        offspring(i,:) = min(offspring(i,:), ub); % Bounding the violating variables to their upper bound
19    end
20 end
```

So, before looking into the function, we can quickly recap polynomial mutation right.

(Refer Slide Time: 32:48)

Polynomial Mutation

➤ Compute δ as

$$\delta = \begin{cases} (2r)^{1/\eta_m} - 1 & \text{if } r < 0.5 \\ 1 - [2(1-r)]^{1/\eta_m} & \text{if } r \geq 0.5 \end{cases}$$

η_m is probability distribution mean

➤ Offspring is generated as

$$y = O + (ub - lb)\delta$$

O	Offspring solution
y	Offspring solution after mutation
ub	upper bound
lb	lower bound

Multi-Objective Optimization Using Evolutionary Algorithms, John Wiley & Sons, Inc, 2001 2

So, again similar to crossover, we will have to generate a random number. If the random number is less than or equal to mutation probability P_m , this is user defined value. So, if this relation satisfies, we will perform mutation; else we will not perform mutation right. So, if you do not perform mutation, the offspring which we generated from crossover will remain the same right.

So, this is the condition that we will have to check. If this condition is satisfied right, then what we will have to do is we will how to determine we will have to generate D random numbers right. So, because we have D decision variable. So, for each of that we need to go and check whether r is less than 0.5 or r is greater than or equal to 0.5. So, if r is less than 0.5, we will use this equation to determine the value of delta if r is greater than or equal to 0.5, we will use this equation to determine the value of delta right.

So, once we determine the value of delta, we can generate the new offspring using this equation. So, the current offspring for which we are doing mutation plus ub minus lb . So, these are the lower and upper bounds of the problem into Δ which we have calculated here right. So, this will give us the new offspring which is called as y over here or you can just call it as O right. So, this will be our new offspring.

So, we will receive the offsprings, the mutation probability, the probability index for mutation η , the lower and upper bound. So, similar to crossover if was determined the number of population and the dimension of the problems. For each solution we will be performing mutation, we generate a random number r . If r is less than the mutation probability, we will be performing the mutation right; else we will we do not need to do anything right, the offspring will be remain the same right.

So, if it happens that we need to perform mutation, then we have to run a loop for j is equal to 1 to D , for all the D decision variables. So, we again generate a random number r is equal to rand and if the random number which we generated happens to be less than 0.5, we determine the delta by this particular equation; else we determine the delta by this particular equation and then, we say the new offspring is the current offspring right plus upper bound minus lower bound into delta.

So, since this we are doing it variable by variable that is why we are having offspring, the j th variable of the i th offspring is equal to the current j th variable of the i th offspring plus the upper bound of the j th variable minus the lower bound of the j th variable into delta. We are not saying delta of j th variable because delta we are individually calculating for every decision variable right.

So, that is going to be just a scalar right. So, we directly multiply it with delta once is for loop is over, we would have completed mutation right. So, again here we are merely bounding it right. So, the i th member if it is violating the lower bound, we use the max operator right to bound it to the lower bound. If it is violating the upper bound you, we use this min operator to bound it to the upper bound. So, this is the polynomial mutation function.

So, it is very easy to implement there are only 4 or 5 steps; first is to generate a random number, check it with mutation probability. Second is to if mutation has to happen we need to generate random numbers for every decision variable and depending upon the value of the decision variable, we will have to use one of those equations to calculate the value of delta right; after determining the value of delta we will have to generate the new offspring a which is the current offspring which is undergoing mutation plus upper bound minus lower bound which are the domains of the decision variable into the delta value which we calculated right.

So, this has to be implemented for all the solutions. So, let us go back to this. So, once we have implemented polynomial mutation again, we will get the mutated off springs right. So, this steps very straight forward that each of the offspring is being sent to the objective function using the function handle prob right and we will be able to determine its fitness function which is stored in the jth location of this variable offspring Obj right.

So, now at the end of line 50, when we have completed this for loop, we have N_p parents and N_p off springs right that we are combining right. Combined population is p semicolon offspring right. So, that is the combined population. So, now, we have combined only the decision variables right. So, the size of combined population would be $2 N_p$ rows right. So, if our population size is 6, then will have 12 rows by the number of decision variables.

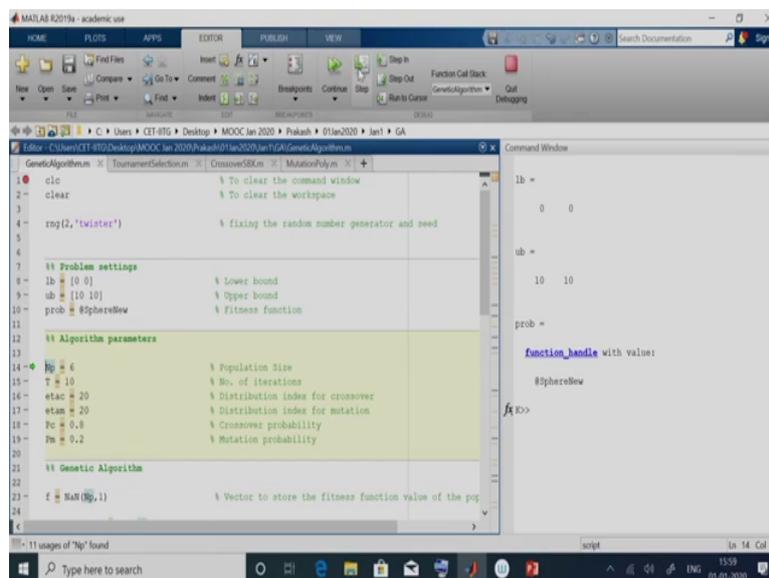
So, if we have 10 decision variables, it will be 12 cross 10 if N_p is 6 right. So, combined population only takes care of the solution the fitness function is actually stored in f and this offspring Obj. So, we need to stack them and then we are sorting right. So, once we sort that, we will get the sorted fitness function value and we will also get this index variables.

So, this variable index variable helps to identify as to which solution has gone into which position right. So, using this i n d variable, we will be able to extract the best N_p population right. So, i n d of 1 to N_p corresponds to the population for which the fitness is f of f 1 to N_p right.

So, that is how we extract the population from the combined population and their fitness from the combined fitness function value that completes the implementation of genetic algorithm. So, these two lines are similar to what we have been implementing in other algorithms right. We find the minimum value of f right. So, that you get the best fitness function value and its location and then we extract the solution using this `ind` variable right.

So, here we have shown it, but the better way of doing it is directly accessing it from `f` right. So, `f` is already sorted. So obviously, since it sorted the first value corresponds to the minimum value and the first value of the variable `P` will correspond to the best solution right. So, it is not necessary to once again find out the minimum and then, call it to be the best fitness function value.

(Refer Slide Time: 39:20)

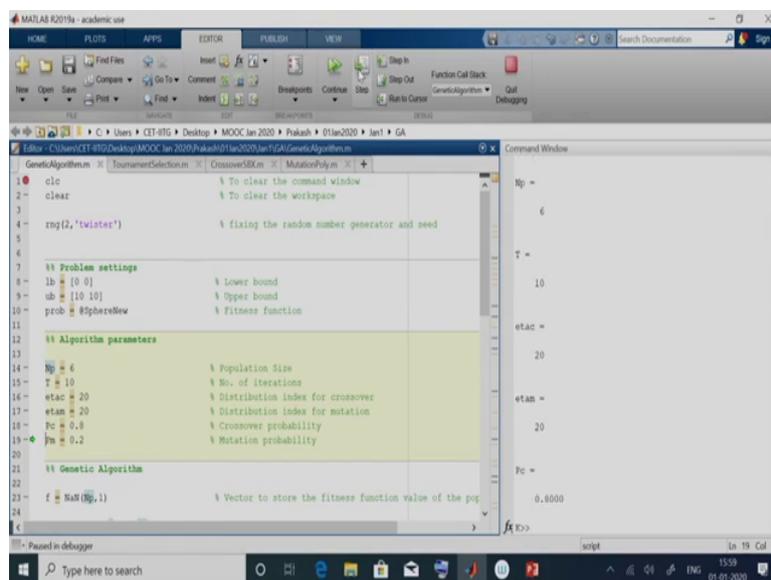


```
GeneticAlgorithm.m
1  clc                                % To clear the command window
2  clear                              % To clear the workspace
3  rng(2, 'twister')                  % fixing the random number generator and seed
4
5
6
7  %% Problem settings
8  lb = [0 0]                          % Lower bound
9  ub = [10 10]                         % Upper bound
10 prob = @SphereEval                  % Fitness function
11
12 %% Algorithm parameters
13
14 Np = 6                               % Population Size
15 T = 10                               % No. of iterations
16 etac = 20                            % Distribution index for crossover
17 etam = 20                            % Distribution index for mutation
18 Pc = 0.8                             % Crossover probability
19 Pm = 0.2                             % Mutation probability
20
21 %% Genetic Algorithm
22
23 f = NaN(Np, 1)                       % Vector to store the fitness function value of the pop
24
```

```
Command Window
lb =
     0     0
ub =
    10    10
prob =
function_handle with value:
    @SphereEval
f(1) >>
```

Now, let us execute this program. So, let us get into this debug mode right and will execute it line by line. So, let me just give this run. So, the first two lines you are familiar with that, it will clear the screen and clear the workspace right. So, this we will fix as the set of random numbers that will be used in this code right. So, that is straight forward. So, these next three lines are defining the lower bound, upper bound and problem which is common for all the other 4 techniques which you have seen.

(Refer Slide Time: 39:51)



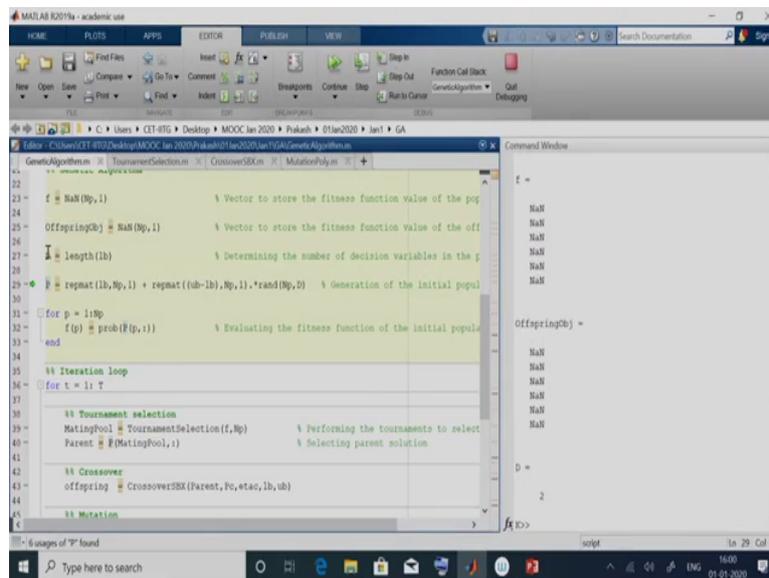
```
1 clc % To clear the command window
2 clear % To clear the workspace
3 rng(2, 'twister') % fixing the random number generator and seed
4
5
6
7 %% Problem settings
8 lb = [0 0] % Lower bound
9 ub = [10 10] % Upper bound
10 prob = @sphereView % Fitness function
11
12 %% Algorithm parameters
13
14 Np = 6 % Population Size
15 T = 10 % No. of iterations
16 etac = 20 % Distribution index for crossover
17 etam = 20 % Distribution index for mutation
18 Pc = 0.6 % Crossover probability
19 Pm = 0.2 % Mutation probability
20
21 %% Genetic Algorithm
22
23 f = NaN(Np, 1) % Vector to store the fitness function value of the pop
24
```

Command Window

```
Np =
     6
T =
    10
etac =
    20
etam =
    20
Pc =
    0.6000
```

So, these six lines will help us to fix the parameters required for executing genetic algorithm right.

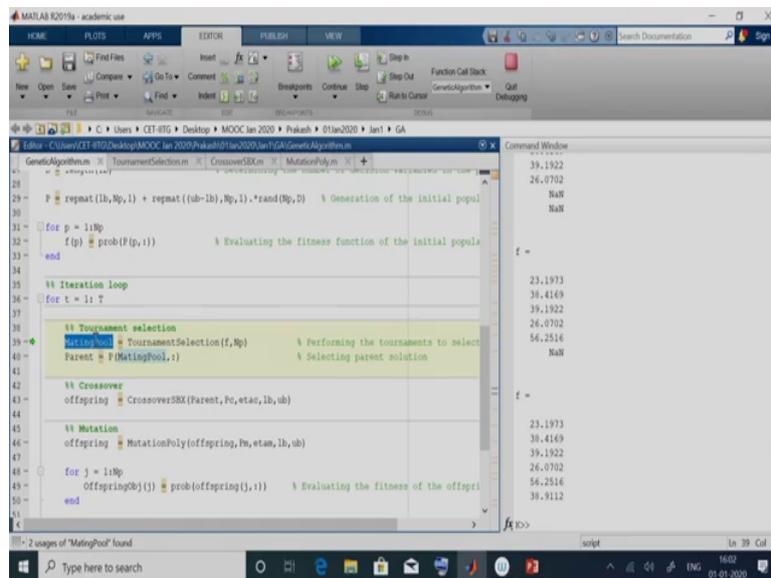
(Refer Slide Time: 40:00)



```
22 f = NaN(Np,1) % Vector to store the fitness function value of the pop
23
24 offspringObj = NaN(Np,1) % Vector to store the fitness function value of the off
25
26 l = length(lb) % Determining the number of decision variables in the p
27
28 % Generation of the initial population
29 p = repmat(lb,Np,1) + repmat((ub-lb),Np,1).*rand(Np,b)
30
31 for p = 1:Np
32 f(p) = prob(f(p,:)) % Evaluating the fitness function of the initial population
33 end
34
35 % Iteration loop
36 for t = 1:T
37
38 % Tournament selection
39 MatPool = TournamentSelection(f,Np) % Performing the tournaments to select
40 Parent = MatPool(:,i) % Selecting parent solution
41
42 % Crossover
43 offspring = CrossoverSBSX(Parent,Pc,etao,lb,ub)
44
45 % Mutation
```

So, then we are defining variable f right which will have NaN values in N_p times. So, N_p in this case our case is 6. So, it will define NaN 6 times right. So, similarly offspring objective right. So, this variable is going to be used to save the fitness function of the offspring right. So, o f offspring this Obj indicates that it is the fitness function value or the objective function value. So, this is offspring Obj right. Line 27 will determine the length of the lower bound or the number of decision variables right.

(Refer Slide Time: 41:02)



The screenshot shows the MATLAB R2019a interface. The Editor window displays a script for a Genetic Algorithm. The script includes the following code:

```
29- repmat(lb,Np,1) + repmat((ub-lb),Np,1).*rand(Np,1) % Generation of the initial popu
30
31- for p = 1:Np
32- f(p) = prob(f,p,1) % Evaluating the fitness function of the initial popula
33- end
34
35- %% Iteration loop
36- for t = 1:T
37
38- %% Tournament selection
39- [idx,fit] = TournamentSelection(f,Np) % Performing the tournaments to select
40- Parent = P(MatingPool,idx) % Selecting parent solution
41
42- %% Crossover
43- offspring = CrossoverSBX(Parent,Pc,etao,lb,ub)
44
45- %% Mutation
46- offspring = MutationPoly(offspring,Pm,etao,lb,ub)
47
48- for j = 1:Np
49- OffspringObj(j) = prob(offspring(j),1) % Evaluating the fitness of the offspr
50- end
51
```

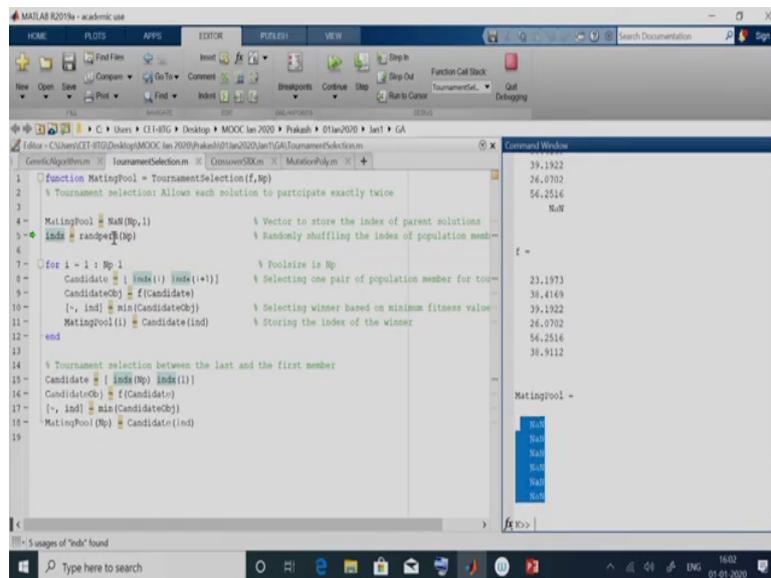
The Command Window shows the output of the fitness function evaluations:

```
f =
    39.1922
    26.0702
         NaN
         NaN
    23.1973
    38.4169
    39.1922
    26.0702
    56.2516
         NaN
    f =
    23.1973
    38.4169
    39.1922
    26.0702
    56.2516
    39.9112
```

So, now we are in the iteration loop right. So, for t is 1 this loop will run for t is equal to 1 for now right and we are only sending the fitness function value. So, f indicates the fitness function value and Np indicates the number of population right. We are not actually sending the variable P which actually contains the population right.

The actual solutions are not required in tournament selection, only the fitness functions are required right. So, that is why when we receive, we will not receive the solutions themselves for mating pool. We will only receive the indexes of the solutions which constitute the mating pool right.

(Refer Slide Time: 41:45)



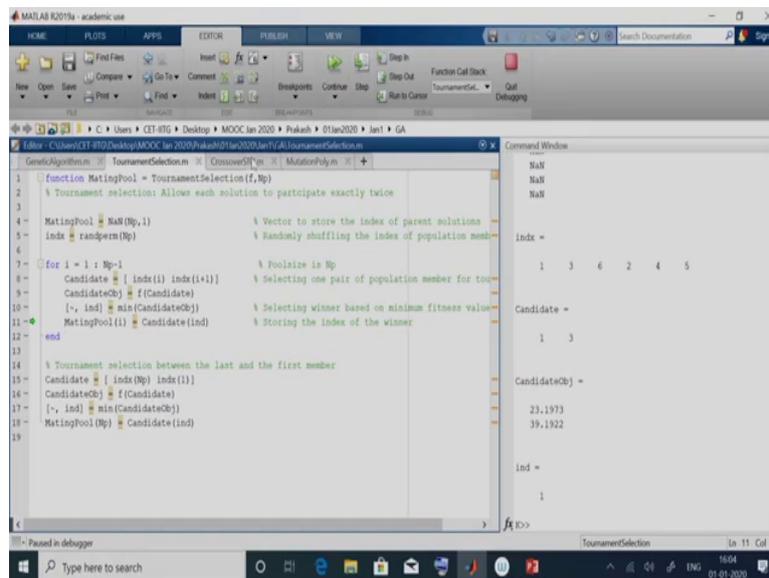
```
1 function MatingPool = TournamentSelection(f,Np)
2 % Tournament selection: Allows each solution to participate exactly twice
3
4 MatingPool = NaN(Np,1) % Vector to store the index of parent solutions
5 % randperm(Np) % Randomly shuffling the index of population members
6
7 for i = 1 : Np/2
8     Candidate = [ randi(Np) randi(Np+1)] % Poolsize is Np
9     CandidateObj = f(Candidate) % Selecting one pair of population member for tournament
10     [~, ind] = min(CandidateObj) % Selecting winner based on minimum fitness value
11     MatingPool(i) = Candidate(ind) % Storing the index of the winner
12 end
13
14 % Tournament selection between the last and the first member
15 Candidate = [ randi(Np) randi(1)]
16 CandidateObj = f(Candidate)
17 [~, ind] = min(CandidateObj)
18 MatingPool(Np) = Candidate(ind)
19
```

Command Window

```
39.1922
26.0702
56.2516
NaN
f =
23.1973
38.4169
39.1922
26.0702
56.2516
39.9112
MatingPool =
NaN
NaN
NaN
NaN
NaN
NaN
```

So, now if we give step in over here. So, over here we have this f those 6 values and Np as 6 right. So, now, initially, we are defining mating pool as a vector right. So, this we require 6 members. So, that is why the size of the mating pool is 6 by 1 right. So, this Np is now 6 right.

(Refer Slide Time: 42:07)



So, `inds` as expected, it is giving a permutation of the 6 values from 1 to 6 right. So, right now it has given 1, 3, 6, 2, 4 and 5 right. So, that is in this variable `inds` right. So, now, first competition is going to be between 1 and 3; the next competition is between 3 and 6; the third competition is between 6 and 2; the fourth competition is between 2 and 4; the fifth competition will be between 4 and 5 and the last competition will be between 5 and 1 right that is how we are going to conduct 6 competitions right.

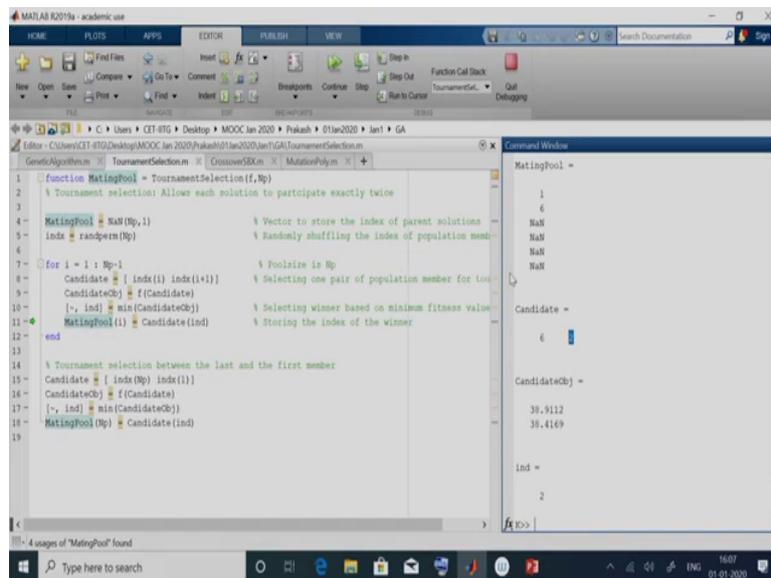
So, here for `i` is equal to 1 to `Np` minus 1 right. So, if we execute this. So, here we are selecting the `i`th value of `inds` and the `i` plus 1th value. So, now, `i` is equal to 1 right. So, basically, we are selecting this 1 and 3 right. So, if we see candidate it has to be 1 and 3. So, these two are the candidates right; the first solution and the third solution. But what we are interested in the fitness of these two solutions. So, in this line 9, will give that right.

solution 1, 1 that is why we have 1 over here right. Next time, we if we go over here. So, index was 1, 3, 6 right. So, now we have done with the competition between 1 and 3.

Now, we will be doing between 3 and 6 right. If we do this, so the Candidates for the tournament is 3 and 6 right. So, now, we need to extract the fitness function. So, the fitness function is given over here. So, what we are expecting Candidate Obj to have is 39.19 and 38.91 right. So, let us see right. So, that those are the two values which we wanted right. So, now, among these to the second one is the minimum right.

So, what we expect ind is to be 2 right. So, ind is 2. So, now, we need to see which solution corresponds to this winner right. So, that can be obtained by Candidate of ind right. So, Candidate was 3 and 6 right. So, the second one. So, 6 is actually the winner remember it is not i n d is not the winner; i n d only indicates the location of the winner. So, at the position 2, it is the Candidate 6 right. So, the winner is 6 in this tournament.

(Refer Slide Time: 45:30)



The screenshot shows the MATLAB R2019a environment. The main editor window displays the following code for the `MatingPool` function:

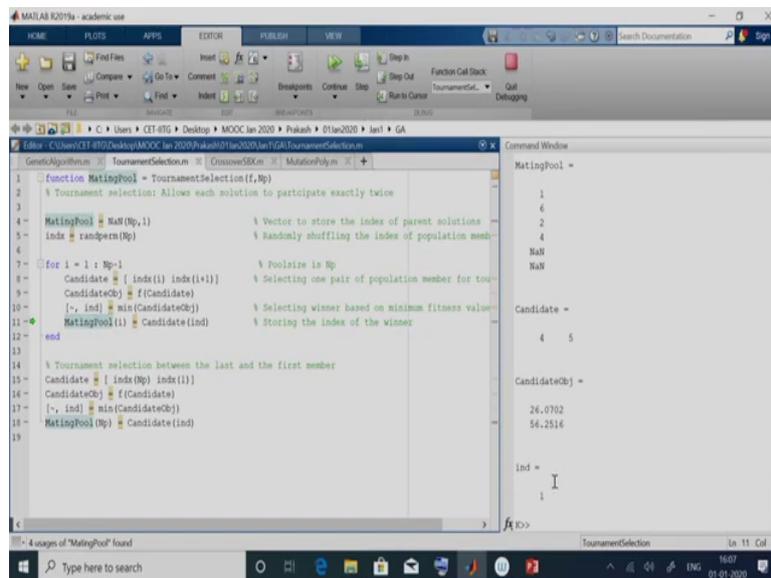
```
1 function MatingPool = TournamentSelection(f,Np)
2     % Tournament selection: Allows each solution to participate exactly twice
3
4     MatingPool = NaN(Np,1) % Vector to store the index of parent solutions
5     indx = randperm(Np) % Randomly shuffling the index of population members
6
7     for i = 1 : Np-1 % Poolsize is Np
8         Candidate = [ indx(i) indx(i+1)] % Selecting one pair of population member for tournament
9         CandidateObj = f(Candidate)
10        [~, ind] = min(CandidateObj) % Selecting winner based on minimum fitness value
11        MatingPool(i) = Candidate(ind) % Storing the index of the winner
12    end
13
14    % Tournament selection between the last and the first member
15    Candidate = [ indx(Np) indx(1)]
16    CandidateObj = f(Candidate)
17    [~, ind] = min(CandidateObj)
18    MatingPool(Np) = Candidate(ind)
19
```

The Command Window on the right shows the execution results:

```
MatingPool =
     1
     6
    NaN
    NaN
    NaN
    NaN
    Candidate =
         6
    CandidateObj =
    39.9112
    39.4169
    ind =
         2
```

So, MatingPool will populate it with 6 right. So, similarly we can conduct the third tournament right. So, it would be straight forward. So, now, the tournament is between 6 and 2 these are there fitness function values right. So, the solutions located at the second position right. In this case it happens to be the second solution itself is the winner right.

(Refer Slide Time: 45:52)



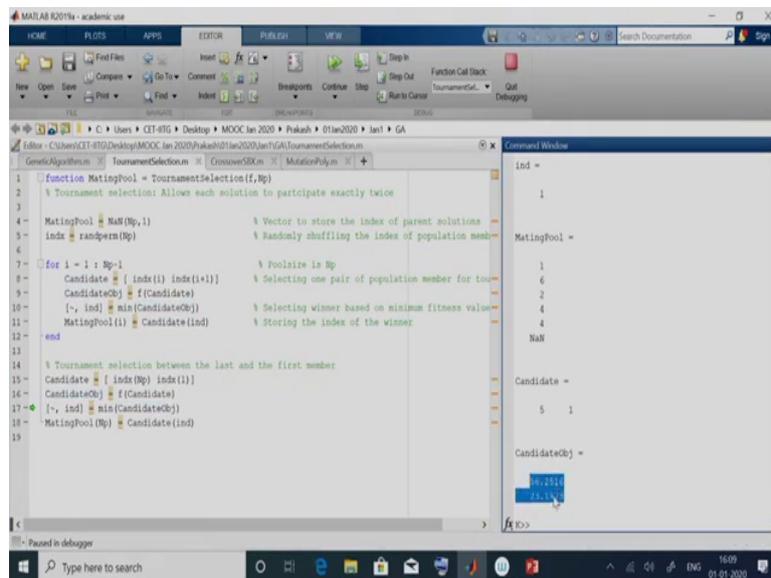
```
function MatingPool = TournamentSelection(f,Np)
% Tournament selection: Allows each solution to participate exactly twice
3
4 MatingPool = NaN(Np,1) % Vector to store the index of parent solutions
5 indx = randperm(Np) % Randomly shuffling the index of population members
6
7 for i = 1 : Np-1 % Poolsize is Np
8     Candidate = [ indx(i) indx(i+1)] % Selecting one pair of population member for tournament
9     CandidateObj = f(Candidate)
10     [~, ind] = min(CandidateObj) % Selecting winner based on minimum fitness value
11     MatingPool(i) = Candidate(ind) % Storing the index of the winner
12 end
13
14 % Tournament selection between the last and the first member
15 Candidate = [ 1 indx(Np) indx(1)]
16 CandidateObj = f(Candidate)
17 [~, ind] = min(CandidateObj)
18 MatingPool(Np) = Candidate(ind)
19
```

Command Window

```
MatingPool =
     1
     6
     2
     4
    NaN
    NaN
Candidate =
     4     5
CandidateObj =
    26.0702
    56.2516
ind =
     1
```

So, in the third competition, the solution 2. So, if we keep doing this in the fourth competition the solution 4 right; in the fifth competition it is again the solution located at fourth position right.

(Refer Slide Time: 46:09)



```
function MatingPool = TournamentSelection(f,Np)
% Tournament selection: Allows each solution to participate exactly twice
3
MatingPool = NaN(Np,1) % Vector to store the index of parent solutions
4
indx = randperm(Np) % Randomly shuffling the index of population members
5
for i = 1 : Np-1 % Poolsize is Np
6
    Candidate = [ indx(i) indx(i+1)] % Selecting one pair of population member for tournament
7
    CandidateObj = f(Candidate) % Fitness values of the pair
8
    [~, ind] = min(CandidateObj) % Selecting winner based on minimum fitness value
9
    MatingPool(i) = Candidate(ind) % Storing the index of the winner
10
end
11
% Tournament selection between the last and the first member
12
Candidate = [ indx(Np) indx(1)]
13
CandidateObj = f(Candidate)
14
[~, ind] = min(CandidateObj)
15
MatingPool(Np) = Candidate(ind)
16
end
```

Command Window

```
indx =
     1
     6
     2
     4
     4
    NaN

Candidate =
     5     1

CandidateObj =
    1.2310
    1.1110
```

So, this thing. So, here if you see 1, 6, 2, 4, 4 right; these are the winners in the 5 competition right and this for loop is now over because this for loop was running between 1 and N_p minus 1. 1 and N_p minus 1 because if we had run till N_p this would have created a problem, we would have been accessing the N_p plus 1th value right. Since we do not have that value, we need to compete with the first one, we have written that exactly this portion is repeated over here except for that the last competition is between last member in $index$ and the first member of $index$ right.

So, remember in the slide which I had written that the last competition is between the last member of the index and the first member of index right. So, in this case that happens to be 5 and 1 right. So, that is because index if we see. So, the first competition was between 1, 3; the second competition is 3, 6; the third competition 6, 2; the fourth competition 2, 4 and the fifth

competition is between 4 and 5; except for solution 1 and 5 everyone has participated twice right.

So, the last competition is between the solution 5 and the solution 1 right. So, that is why we have this candidate 5 and 1. So, if we step in this, so we get this. These are the objective functions. So, the winner will be 23.

(Refer Slide Time: 47:34)

```
function MatingPool = TournamentSelection(f,Np)
% Tournament selection: Allows each solution to participate exactly twice
%
1  MatingPool = NaN(Np,1) % Vector to store the index of parent solutions
2  indx = randperm(Np) % Randomly shuffling the index of population members
3
4  for i = 1 : Np-1 % Poolsize is Np
5      Candidate = [ indx(i) indx(i+1)] % Selecting one pair of population member for tournament
6      CandidateObj = f(Candidate) % Calculating fitness values
7      [~, ind] = min(CandidateObj) % Selecting winner based on minimum fitness value
8      MatingPool(i) = Candidate(ind) % Storing the index of the winner
9  end
10
11 % Tournament selection between the last and the first member
12 Candidate = [ indx(Np) indx(1)]
13 CandidateObj = f(Candidate)
14 [~, ind] = min(CandidateObj)
15 MatingPool(Np) = Candidate(ind)
end
```

Command Window

```
Candidate =
     5     1
CandidateObj =
    56.2516
    23.1973
ind =
     2
MatingPool =
```

So, it is located at the second position, the winner is located at the second position. So, this value 1 has to enter the mating pool right. So, this is our mating pool right. So, here if we look at the fitness function, this is something which we have done while learning genetic algorithm right. So, the best fitness function value here is 23.197 which is the first solution right. So, and as we discussed the best solution will appear twice will definitely appear twice right. So, one is appearing over here; one is appearing over here.

Similarly, the worst solution was if you look at f , the worst solution is located at the fifth position right 56.2516 and if we see fifth solution does not find the place in the mating pool right. This is as expected because it does not matter with whom the word solution completes, it is going to always lose right. If it is going to lose in the tournament, it is not going to find the position in the MatingPool. So, in this case it happens that the fourth solution right which is 26.0702 occurs twice right.

As you can see mating pool is not the solution at itself, it merely tells that the first solution, the sixth solution, the second solution, the fourth solution, the fourth solution and the first solution constitute the mating pool right. So, this only help us to determine the indexes right of the solutions which are better.

(Refer Slide Time: 49:09)

The screenshot shows the MATLAB R2019a environment with a script editor and a Command Window. The script, named 'GeneticAlgorithm.m', is running a Genetic Algorithm. The Command Window displays the state of the 'MatingPool' and 'Parent' variables.

```

MatingPool =
     1
     6
     2
     4
     4
     1

Parent =
     4.3599     2.0448
     3.3033     5.2914
     0.2593     4.1927
     4.3532     2.6683
     4.3532     2.6683
     4.3599     2.0448
  
```

So, now if we go into step it goes into our main script right. So, now, we have finished determining the MatingPool right. Now, we need to access the corresponding solutions right because for a tournament selection, we need only the fitness function value we do not require the solutions as such right; whereas, for crossover and mutation we required the actual solution right. So, now if we execute this. So, this parent if we see, these are the solutions. So, these are the solution which will undergo crossover right.

So, the first solution is repeated twice right. It is over here and here that is why it is at the first and the last location. The 6th solution was 6 solution was 3.3033. So, it is located in the second position right. So, that is how we have formed the parents now right. Now, the parents have been determined, we need to perform the crossover right. So, for crossover we are sending the parents, the crossover probability 0.8, the distribution index 20, the lower bound 0 and the upper bound 10 right.

(Refer Slide Time: 50:16)

```
function offspring = CrossoverSEx(Parent,Po,etao,lb,ub)
2
3 [Np,D] = size(Parent); % Determining the no. of population and decision
4 indx = randperm(Np); % Permutating numbers from 1 to Np
5 Parent = Parent(indx,:); % Randomly shuffling parent solutions
6 offspring = NaN(Np,D); % Matrix to store offspring solutions
7
8 for i = 1:2:Np % Selecting parents in pairs for crossover
9
10     r = rand; % Generating random number to decide if crossover
11
12     if r < Po % Checking for crossover probability
13
14         for j = 1:D
15
16             r = rand; % Generating random number to determine the beta
17
18             if r <= 0.5
19                 beta = (2*r)^(1/(etao+1)); % Calculating beta value
20             else
21                 beta = (1/(2*(1-r)))^(1/(etao+1)); % Calculating beta value
22             end
23
24             offspring(i,j) = 0.5*((1+beta)*Parent(i,j) + (1-beta)*Parent(i+1,j));
end
```

Command Window

```
Parent =
4.3599 2.0465
3.3033 5.2914
0.2593 4.1927
4.3532 2.6683
4.3532 2.6683
4.3599 2.0465

Np =
6

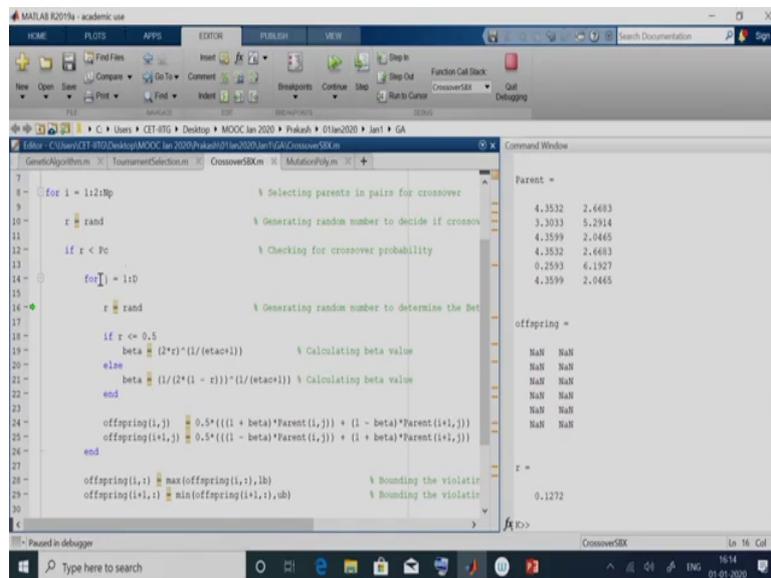
D =
2

indx =
6 5 3 1
```

Let us just step into this. So, first we are determining the number of rows and the number of columns of parent right and then, we are randomly arranging it. So, now the solutions have been shuffled right 4, 2, 6, 5, 3, 1. Since, we have shuffled with randperm the solutions can be considered to be randomly arranged right. So, will perform a crossover between 4 and 2; 6 and 5 and 3 and 1 right.

So, there will be 3 crossovers because we have 6 solutions. So, the number of crossover is N_p by 2 times and each time, we will get 2 offspring. So, the total off springs will be still N_p .

(Refer Slide Time: 50:56)



```
7 for i = 1:2:Sp % Selecting parents in pairs for crossover
8
9
10 r = rand % Generating random number to decide if crossover
11
12 if r < Pc % Checking for crossover probability
13
14 for j = 1:D
15
16 r = rand % Generating random number to determine the beta
17
18 if r <= 0.5
19     beta = (2*r)^(1/(etac+1)) % Calculating beta value
20 else
21     beta = (1/(2*(1-r)))^(1/(etac+1)) % Calculating beta value
22 end
23
24 offspring(i,j) = 0.5*((1+beta)*Parent(i,j)) + (1-beta)*Parent(i+1,j)
25 offspring(i+1,j) = 0.5*((1-beta)*Parent(i,j)) + (1+beta)*Parent(i+1,j)
26 end
27
28 offspring(i,:) = max(offspring(i,:),lb) % Bounding the violation
29 offspring(i+1,:) = min(offspring(i+1,:),ub) % Bounding the violation
30
```

Command Window

```
Parent =
4.3532 2.6683
3.3033 5.2914
4.3599 2.0465
4.3532 2.6683
0.2593 4.1927
4.3599 2.0465

offspring =
NaN NaN
NaN NaN
NaN NaN
NaN NaN
NaN NaN
NaN NaN

r =
0.1272
```

So, now we have arranged the parents. The parent solution as per this index right. Initially, the parents were arranged like this right. Now, would we have shuffle using this index right. The first solution is 4.3599, 2.0465 which is over here right. So, we have randomly shuffled the solutions right. To store a offsprings, we create this 6 by 2 matrix with NaN values right.

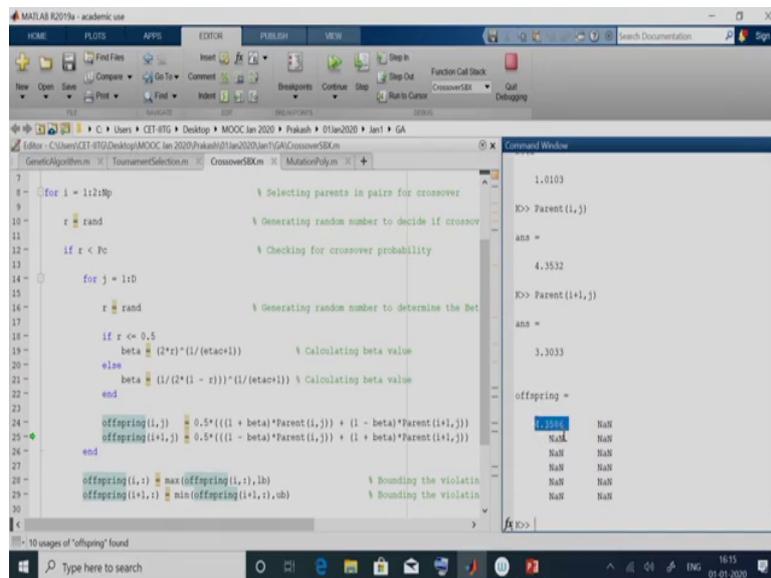
6 by 2 because there are going to be 6 offspring; the number of offspring will be equal to the population size and the number of decision variables will govern the number of columns ok. So, now, let us step into this. So, now, i is equal to 1 right. In this case, we first need to just generate a random number right.

(Refer Slide Time: 51:44)

```
7  
8 for i = 1:2:Np  
9  
10 r = rand  
11  
12 if r < Pc  
13  
14 for j = 1:D  
15  
16 r = rand  
17  
18 if r <= 0.5  
19 beta = (2*r)^(1/(eta*c+1)) % Calculating beta value  
20  
21 else  
22 beta = (1/(2*(1 - r)))^(1/(eta*c+1)) % Calculating beta value  
23  
24 offspring(i,j) = 0.5*((1 + beta)*Parent(i,j)) + (1 - beta)*Parent(i+1,j)  
25 offspring(i+1,j) = 0.5*((1 - beta)*Parent(i,j)) + (1 + beta)*Parent(i+1,j)  
26  
27  
28 offspring(i,:) = max(offspring(i,:),lb) % Bounding the violatio  
29 offspring(i+1,:) = min(offspring(i+1,:),ub) % Bounding the violatio  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110  
111  
112  
113  
114  
115  
116  
117  
118  
119  
120  
121  
122  
123  
124  
125  
126  
127  
128  
129  
130  
131  
132  
133  
134  
135  
136  
137  
138  
139  
140  
141  
142  
143  
144  
145  
146  
147  
148  
149  
150  
151  
152  
153  
154  
155  
156  
157  
158  
159  
160  
161  
162  
163  
164  
165  
166  
167  
168  
169  
170  
171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183  
184  
185  
186  
187  
188  
189  
190  
191  
192  
193  
194  
195  
196  
197  
198  
199  
200  
201  
202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215  
216  
217  
218  
219  
220  
221  
222  
223  
224  
225  
226  
227  
228  
229  
230  
231  
232  
233  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269  
270  
271  
272  
273  
274  
275  
276  
277  
278  
279  
280  
281  
282  
283  
284  
285  
286  
287  
288  
289  
290  
291  
292  
293  
294  
295  
296  
297  
298  
299  
300  
301  
302  
303  
304  
305  
306  
307  
308  
309  
310  
311  
312  
313  
314  
315  
316  
317  
318  
319  
320  
321  
322  
323  
324  
325  
326  
327  
328  
329  
330  
331  
332  
333  
334  
335  
336  
337  
338  
339  
340  
341  
342  
343  
344  
345  
346  
347  
348  
349  
350  
351  
352  
353  
354  
355  
356  
357  
358  
359  
360  
361  
362  
363  
364  
365  
366  
367  
368  
369  
370  
371  
372  
373  
374  
375  
376  
377  
378  
379  
380  
381  
382  
383  
384  
385  
386  
387  
388  
389  
390  
391  
392  
393  
394  
395  
396  
397  
398  
399  
400  
401  
402  
403  
404  
405  
406  
407  
408  
409  
410  
411  
412  
413  
414  
415  
416  
417  
418  
419  
420  
421  
422  
423  
424  
425  
426  
427  
428  
429  
430  
431  
432  
433  
434  
435  
436  
437  
438  
439  
440  
441  
442  
443  
444  
445  
446  
447  
448  
449  
450  
451  
452  
453  
454  
455  
456  
457  
458  
459  
460  
461  
462  
463  
464  
465  
466  
467  
468  
469  
470  
471  
472  
473  
474  
475  
476  
477  
478  
479  
480  
481  
482  
483  
484  
485  
486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500  
501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539  
540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552  
553  
554  
555  
556  
557  
558  
559  
560  
561  
562  
563  
564  
565  
566  
567  
568  
569  
570  
571  
572  
573  
574  
575  
576  
577  
578  
579  
580  
581  
582  
583  
584  
585  
586  
587  
588  
589  
590  
591  
592  
593  
594  
595  
596  
597  
598  
599  
600  
601  
602  
603  
604  
605  
606  
607  
608  
609  
610  
611  
612  
613  
614  
615  
616  
617  
618  
619  
620  
621  
622  
623  
624  
625  
626  
627  
628  
629  
630  
631  
632  
633  
634  
635  
636  
637  
638  
639  
640  
641  
642  
643  
644  
645  
646  
647  
648  
649  
650  
651  
652  
653  
654  
655  
656  
657  
658  
659  
660  
661  
662  
663  
664  
665  
666  
667  
668  
669  
670  
671  
672  
673  
674  
675  
676  
677  
678  
679  
680  
681  
682  
683  
684  
685  
686  
687  
688  
689  
690  
691  
692  
693  
694  
695  
696  
697  
698  
699  
700  
701  
702  
703  
704  
705  
706  
707  
708  
709  
710  
711  
712  
713  
714  
715  
716  
717  
718  
719  
720  
721  
722  
723  
724  
725  
726  
727  
728  
729  
730  
731  
732  
733  
734  
735  
736  
737  
738  
739  
740  
741  
742  
743  
744  
745  
746  
747  
748  
749  
750  
751  
752  
753  
754  
755  
756  
757  
758  
759  
760  
761  
762  
763  
764  
765  
766  
767  
768  
769  
770  
771  
772  
773  
774  
775  
776  
777  
778  
779  
780  
781  
782  
783  
784  
785  
786  
787  
788  
789  
790  
791  
792  
793  
794  
795  
796  
797  
798  
799  
800  
801  
802  
803  
804  
805  
806  
807  
808  
809  
810  
811  
812  
813  
814  
815  
816  
817  
818  
819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
860  
861  
862  
863  
864  
865  
866  
867  
868  
869  
870  
871  
872  
873  
874  
875  
876  
877  
878  
879  
880  
881  
882  
883  
884  
885  
886  
887  
888  
889  
890  
891  
892  
893  
894  
895  
896  
897  
898  
899  
900  
901  
902  
903  
904  
905  
906  
907  
908  
909  
910  
911  
912  
913  
914  
915  
916  
917  
918  
919  
920  
921  
922  
923  
924  
925  
926  
927  
928  
929  
930  
931  
932  
933  
934  
935  
936  
937  
938  
939  
940  
941  
942  
943  
944  
945  
946  
947  
948  
949  
950  
951  
952  
953  
954  
955  
956  
957  
958  
959  
960  
961  
962  
963  
964  
965  
966  
967  
968  
969  
970  
971  
972  
973  
974  
975  
976  
977  
978  
979  
980  
981  
982  
983  
984  
985  
986  
987  
988  
989  
990  
991  
992  
993  
994  
995  
996  
997  
998  
999  
1000
```

So, the random number which has to generated is 0.12 right and the crossover probability is 0.8. So, since r is less than crossover probability, we will perform the crossover over here right. So, if we step into this. So, for j is equal to 1 to D. So, for the first variable, we will generate a random number. So, the random number happens to be 5 9 6 7. So, it will not go into line 19, it will in straight go to line 21 ok. So, the value of beta is calculated to be on 0.0103 right. Once we have calculated beta, we need to calculate the offspring right.

(Refer Slide Time: 52:44)



```
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

```
1.0103
IO> Parent (1, j)
ans =
4.3532
IO> Parent (i+1, j)
ans =
3.3033
offspring =
4.3586 NaN
NaN NaN
NaN NaN
NaN NaN
NaN NaN
NaN NaN
```

So, if we say i is now 1, j is also 1 right. So, we are using the value 4.3532 with 3.033 right. So, that if we access this over here parent of i comma j and parent of $i + 1$ comma j . So, those are the two values which will be participating in this equation to get us the offspring right. So, here if you see this value is found to be 4.3586 right and similarly, for the second offspring also the first variable can be determined using this equation right.

(Refer Slide Time: 53:10)

```
function offspring = CrossoverSRX(Parent,Pc,etaac,lb,ub)
1
2
3 [Np,D] = size(Parent) % Determining the no. of population and decision
4 ind = randperm(Np) % Permutating numbers from 1 to Np
5 Parent = Parent(indx,:,:) % Randomly shuffling parent solutions
6 offspring = NaN(Np,D) % Matrix to store offspring solutions
7
8 for i = 1:2:Np % Selecting parents in pairs for crossover
9
10     r = rand % Generating random number to decide if crossover
11
12     if r < Pc % Checking for crossover probability
13
14         for j = 1:D
15
16             r = rand % Generating random number to determine the beta
17
18             if r <= 0.5
19                 beta = (2*i)^(1/(etaac+1)) % Calculating beta value
20             else
21                 beta = (1/(2*(1 - r)))^(1/(etaac+1)) % Calculating beta value
22             end
23
24             offspring(i,j) = 0.5*((1 + beta)*Parent(i,j) + (1 - beta)*Parent(i+1,j))
25
26 end
27
28 end
```

Command Window

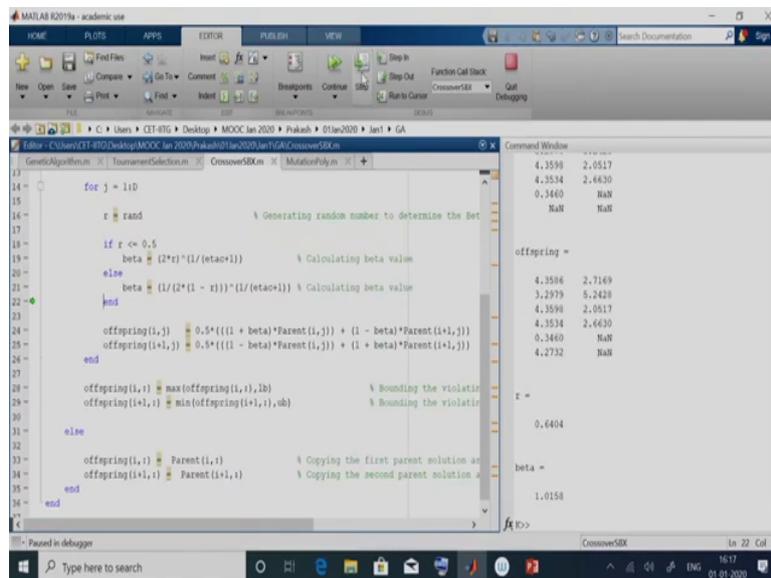
```
offspring =
4.3586 2.7169
3.2979 5.2428
NaN NaN
NaN NaN
NaN NaN

r =
0.1069
```

So, step so, the second offspring the first variable has been determined right. So, this has to be repeated once more so that the second variable is also determined right. So, at the end of this i is equal to 1 right, we have generated 2 off springs. So, now, we need to check whether this is in bounds or not. So, in this case it happens that it is in bounds. So, the next 2 steps will not have any impact on them right and then, we go on to the next crossover right.

So, i is equal to 3; we have skipped 2 because these two solutions participated in the crossover right. So, 4 and 2 have already participated in the crossover. Now, it is these two solutions right. So, that is why we are skipping by 2. So, now i is equal to 3. So, if we generate a random number, the random number happens to be 0.1069. So, again we will have to perform the crossover right.

(Refer Slide Time: 54:08)



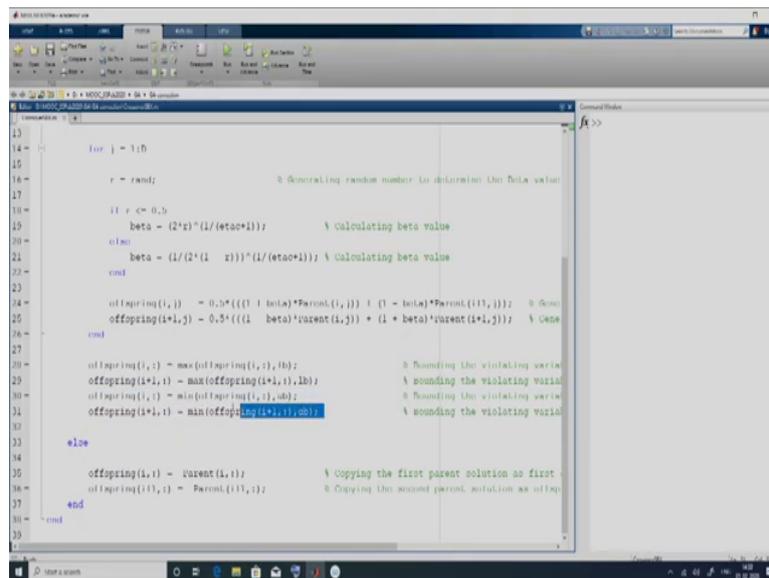
```
13
14 for j = 1:D
15
16     r = rand; % Generating random number to determine the Bet
17
18     if r <= 0.5
19         beta = (2*r)/(1+(2*a-1)); % Calculating beta value
20     else
21         beta = (1/(2*(1-r)))/(1+(2*a-1)); % Calculating beta value
22     end
23
24     offspring(i,j) = 0.5*((1+beta)*Parent(i,j)) + (1-beta)*Parent(i+1,j)
25     offspring(i+1,j) = 0.5*((1-beta)*Parent(i,j)) + (1+beta)*Parent(i+1,j)
26 end
27
28 offspring(i,:) = max(offspring(i,:),lb); % Bounding the violation
29 offspring(i+1,:) = min(offspring(i+1,:),ub); % Bounding the violation
30
31 else
32     offspring(i,:) = Parent(i,:); % Copying the first parent solution as
33     offspring(i+1,:) = Parent(i+1,:); % Copying the second parent solution
34 end
35 end
36 end
37
```

Command Window

4.3590	2.0517
4.3534	2.6430
0.3460	NaN
NaN	NaN
offspring =	
4.3586	2.7169
3.2979	5.2428
4.3590	2.0517
4.3534	2.6430
0.3460	NaN
4.2732	NaN
r =	
0.6404	
beta =	
1.0158	

So, both the values would be generated. So, we have completed 2 crossover and we have got 4 offsprings right. Let us see the last one whether we goes into this crossover or not. So, the random number is 0.4678. So, again it will go into the to this crossover operation right. So, this we can just quickly complete ok.

(Refer Slide Time: 54:35)



```
13
14
15
16     r = rand; % Generating random number to determine the beta value
17
18     if r <= 0.5
19         beta = (2*r)^(1/(eta+1)); % Calculating beta value
20     else
21         beta = (1/(2*(1-r)))^(1/(eta+1)); % Calculating beta value
22     end
23
24     offspring(i,:) = 0.5*((1+beta)*Parent(i,:) + (1-beta)*Parent(i+1,:)); % Gene
25     offspring(i+1,:) = 0.5*((1-beta)*Parent(i,:) + (1+beta)*Parent(i+1,:)); % Gene
26
27
28     offspring(i,:) = max(offspring(i,:), lb); % Bounding the violating variable
29     offspring(i+1,:) = max(offspring(i+1,:), lb); % Bounding the violating variable
30     offspring(i,:) = min(offspring(i,:), ub); % Bounding the violating variable
31     offspring(i+1,:) = min(offspring(i+1,:), ub); % Bounding the violating variable
32
33
34
35     offspring(i,:) = Parent(i,:); % Copying the first parent solution as first
36     offspring(i+1,:) = Parent(i+1,:); % Copying the second parent solution as offspring
37
38 end
39
40
```

So, here one thing that you need to remember is that we are generating two off springs right. So, offspring i as well offspring i plus 1th. So, both of them need to be bounded. So, here if you see in line 28, 29 what we are doing is we are bounding for the both the solution the i th offspring and the i th plus 1 offspring, we are checking for the lower bound right. So, the offspring is indicated by offspring i comma colon.

So, this the highlighted portion indicates the offspring. This is the lower bound and we are using the max operator. So, what we will get over here line 28 is that the i th offspring no longer violates the lower bound, after this line is executed. Similarly, this line line 29 is for making sure that the i plus 1th offspring does not violate the lower bound right. So, both of this lines only take care about lower bound and similarly, both of these lines line 30 and line 31 take care about the upper bound right.

So, the i th offspring we are using the main function to make sure that it does not violate the upper bound and similarly, line 31 we are bounding the i plus 1th offspring with the upper bound. So, this you need to make sure that both the solutions do not violate the upper and lower bound right. So, that is why we have 4 lines over here.

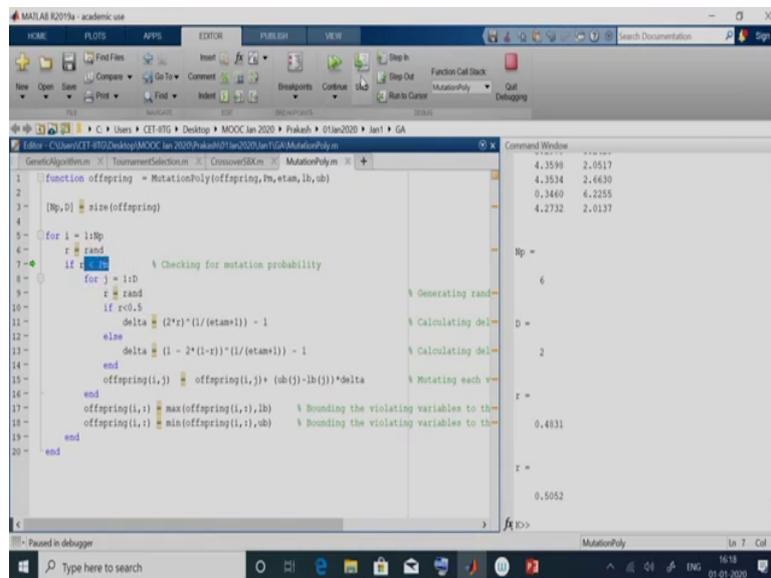
(Refer Slide Time: 55:54)

The screenshot shows the MATLAB R2019a editor with a script named 'GeneticAlgorithm.m'. The script includes several functions: 'TournamentSelection', 'CrossoverSBX', and 'MutationPoly'. The main script contains a loop for $t = 1:T$ iterations. Inside the loop, it performs tournament selection, crossover, and mutation. The Command Window displays the following fitness values:

4.3590	2.0517
4.3534	2.6630
0.3460	6.2255
4.2732	2.0137
offspring =	
4.3586	2.7149
3.2979	5.2428
4.3590	2.0517
4.3534	2.6630
0.3460	6.2255
4.2732	2.0137
offspring =	
4.3586	2.7149
3.2979	5.2428
4.3590	2.0517
4.3534	2.6630
0.3460	6.2255
4.2732	2.0137

So, at the end of crossover, we have this as our offspring right. So, the offspring have been generated. Now, these offspring have to be mutated right. So, this is sent into the function mutation poly right.

(Refer Slide Time: 56:10)



The image shows a MATLAB Editor window with a function named 'MutationPoly' and a Command Window showing its execution results. The function code is as follows:

```
1 function offspring = MutationPoly(offspring, Pn, etam, lb, ub)
2
3 [Np, D] = size(offspring)
4
5 for i = 1:Np
6     r = rand
7     if r <= Pn % Checking for mutation probability
8         for j = 1:D
9             r = rand % Generating rand
10            if r < 0.5 % Calculating del
11                delta = (2*r) * (1/(etam+1)) - 1 % Calculating del
12            else % Calculating del
13                delta = (1 - 2*(1-r)) * (1/(etam+1)) - 1 % Calculating del
14            end
15            offspring(i,j) = offspring(i,j) + (ub(j)-lb(j))*delta % Mutating each v
16        end
17        offspring(i,:) = max(offspring(i,:), lb) % Bounding the violating variables to th
18        offspring(i,:) = min(offspring(i,:), ub) % Bounding the violating variables to th
19    end
20 end
```

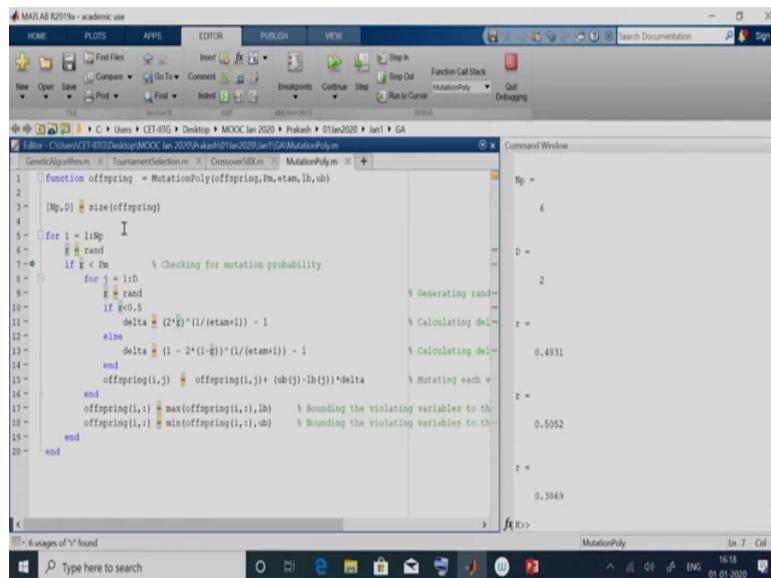
The Command Window shows the following output:

```
4.3590 2.0517
4.3534 2.6430
0.3460 6.2255
4.2732 2.0137
Np =
     6
D =
     2
r =
    0.4831
r =
    0.5052
```

So, if is step in over here again, we determine the number of rows and the number of columns which constitute the population size and the number of decision variable. So, each member has to undergo a mutation. So, we have this 1 to Np. We generate a random number. In this case it happens to be 0.4831 and our mutation probability is 0.2.

So, this condition is not satisfied right. So, it will not change anything right. So, the first offspring will remain as such right. So, this solution will remain as such right. So, the second time, we generate a random number which is again 0.5052. So, it will again not satisfy this condition right. So, no change in the offspring, no mutation is happening.

(Refer Slide Time: 56:52)



```
function offspring = MutationPoly(offspring,Pm,etas,lb,ub)
1
2
3 [Np,D] = size(offspring)
4
5 for i = 1:Np
6     rand
7     if rand < Pm % Checking for mutation probability
8         for j = 1:D
9             rand % Generating rand
10            if rand < 0.5
11                delta = (2*rand - 1)/(etas+1) - 1 % Calculating del
12            else
13                delta = (1 - 2*(1- rand))/(etas+1) - 1 % Calculating del
14            end
15            offspring(i,j) = offspring(i,j) + (ub(j)-lb(j))*delta % Mutating each v
16        end
17        offspring(i,:) = max(offspring(i,:),lb) % Bounding the violating variables to th
18        offspring(i,:) = min(offspring(i,:),ub) % Bounding the violating variables to th
19    end
20 end
```

Command Window

```
Np =
     6
D =
     2
r =
     0.4931
r =
     0.5052
r =
     0.3849
```

So, the third time random number is generated is 0.3, again no mutation right. The fourth time 0.79, so again no mutation is happening right. So, the fifth time, its 0.58. So, no mutation right and this time, it is 0.1623 right. So, since it is less than P_m , we will have a mutation occurring over here right. So, it satisfies this condition. So, it has come into this loop. So, for every variable we need to generate a random number.

(Refer Slide Time: 57:31)

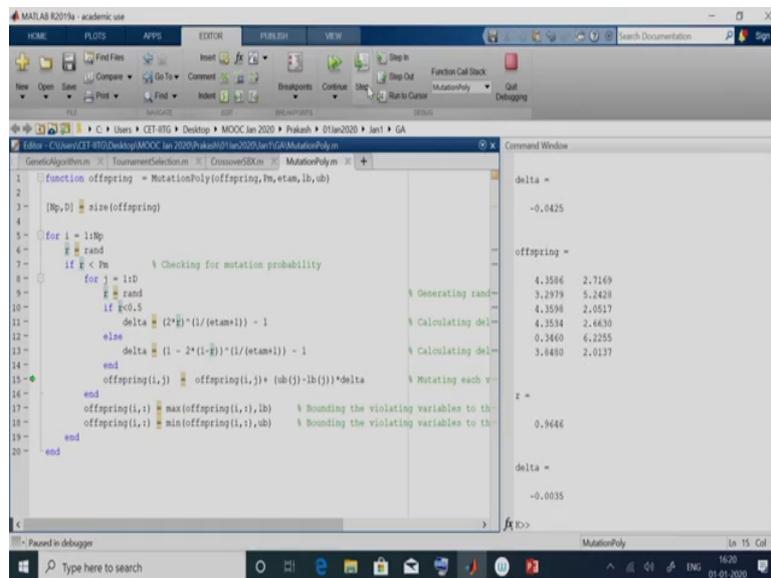
```
function offspring = MutationPoly(offspring,Pm,etas,lb,ub)
2
3 [Np,D] = size(offspring)
4
5 for i = 1:Np
6     r = rand
7     if r < Pm % Checking for mutation probability
8         for j = 1:D
9             r = rand % Generating rand
10            if r<0.5
11                delta = (2*r)^(1/(etas+1)) - 1 % Calculating del
12            else
13                delta = (1 - 2*(1-r))^(1/(etas+1)) - 1 % Calculating del
14            end
15            offspring(i,j) = offspring(i,j) + (ub(j)-lb(j))*delta % Mutating each v
16        end
17        offspring(i,:) = max(offspring(i,:),lb) % Bounding the violating variables to th
18        offspring(i,:) = min(offspring(i,:),ub) % Bounding the violating variables to th
19    end
20 end
```

Command Window

```
r =
0.3649
r =
0.7936
r =
0.5800
r =
0.1423
```

So, the random number generated now is 0.7008. So, since that is greater than this 0.5, we will be using this equation to calculate the delta.

(Refer Slide Time: 57:40)

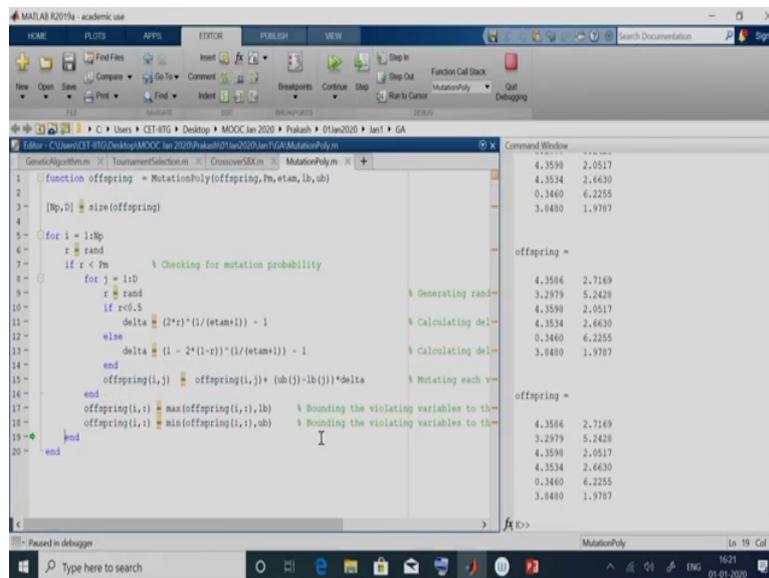


The screenshot shows the MATLAB R2019a environment. The main window displays a function named `offspring = MutationPoly(offspring, Pm, etam, lb, ub)`. The function iterates over each element of the offspring population. For each element, it generates a random number and checks if it is less than the mutation probability `Pm`. If so, it calculates a delta value based on the current value and the mutation rate `etam`. The delta is then used to mutate the variable. The output window shows the following values:

```
delta =  
-0.0425  
  
offspring =  
4.3586 2.7169  
3.2979 5.2428  
4.3598 2.0517  
4.3534 2.6630  
0.3460 6.2255  
3.8480 2.0137  
  
z =  
0.9446  
  
delta =  
-0.0035
```

So, since it is for the 6 solution, i is equal to 6 right. So, for i is equal to 6 if you see the solution initially was 4.2732 and that has changed to 3.8480 right that is because of the mutation that particular variable has been mutated so far right. So, if we continue this right this is for the second variable right, we determine delta again if it see look at the offspring, so 2.0137 has been changed to 1.9787.

(Refer Slide Time: 58:05)



The screenshot shows the MATLAB R2019a environment. The main window displays a function named 'MutationPoly' with the following code:

```
1 function offspring = MutationPoly(offspring, Pm, etam, lb, ub)
2
3 [Np, D] = size(offspring)
4
5 for i = 1:Np
6     r = rand
7     if r < Pm % Checking for mutation probability
8         for j = 1:D
9             r = rand % Generating rand
10            if r < 0.5
11                delta = (2*r) * (1/(etam+1)) - 1 % Calculating del
12            else
13                delta = (1 - 2*(1-r)) * (1/(etam+1)) - 1 % Calculating del
14            end
15            offspring(i,j) = offspring(i,j) + (ub(j)-lb(j))*delta % Mutating each v
16        end
17        offspring(i,:) = max(offspring(i,:), lb) % Bounding the violating variables to th
18        offspring(i,:) = min(offspring(i,:), ub) % Bounding the violating variables to th
19    end
20 end
```

The Command Window on the right shows the output of the function, displaying a 2x5 matrix of offspring values:

```
offspring =
4.3590 2.0517
4.3534 2.6430
0.3460 6.2255
3.8480 1.9787
4.3586 2.7169
3.2979 5.2428
4.3590 2.0517
4.3534 2.6430
0.3460 6.2255
3.8480 1.9787
4.3586 2.7169
3.2979 5.2428
4.3590 2.0517
4.3534 2.6430
0.3460 6.2255
3.8480 1.9787
```

So, in this case it happens that all of the variables are within the bound. So, these two statements will not have any effect; but if it had been out of the bounds, so these two statements would have helped us to bring it into the bounds. So, if we complete this right. So, now, we have generated offspring from the mutation right. So, we have this offspring which is within the bound. So, we need to calculate the fitness function of this right.

(Refer Slide Time: 58:48)

The screenshot shows the MATLAB IDE with a script named 'GeneticAlgorithm.m'. The script is currently paused at line 49. The Command Window on the right displays the following output:

```
23.2176
26.0434
NaN
NaN
OffspringObj =
26.3794
39.3629
23.2176
26.0434
39.8767
NaN
OffspringObj =
26.3794
39.3629
23.2176
26.0434
39.8767
18.3224
```

So, now we have the population P right and the off springs; offspring is 6 rows 2 columns and P was already the initial population right. So, that was also 6 row 2 column.

(Refer Slide Time: 59:01)

The screenshot shows the MATLAB R2019a interface with a script editor and a Command Window. The script, named 'GeneticAlgorithm.m', contains the following code:

```
38 %% Tournament selection
39 MatingPool = TournamentSelection(f,Pp) % Performing the tournaments to select
40 Parent = P(MatingPool,1) % Selecting parent solution
41
42 %% Crossover
43 offspring = CrossoverSBX(Parent,Pc,etaa,lb,ub)
44
45 %% Mutation
46 offspring = MutationPoly(offspring,Pm,etaa,lb,ub)
47
48 for j = 1:Np
49     OffspringObj(j) = prob(offspring(j,:)) % Evaluating the fitness of the offspring
50 end
51
52 CombinedPopulation = [P; offspring]
53 [f,ind] = sort([OffspringObj]); % mu + lambda selection
54
55 P = f(1:Np);
56 P = CombinedPopulation(ind(1:Np),:);
57
58 end
59
60 [bestfitness,ind] = min(f)
61 bestsol = P(ind,1);
```

The Command Window displays the following output:

```
OffspringObj =
    26.3794
    39.3429
    23.2174
    26.0434
    39.8747
    19.7224

CombinedPopulation =
    4.3599    2.0465
    0.2593    4.1927
    5.4964    2.9965
    4.3532    2.6483
    4.2037    4.2113
    3.3033    5.2914
    4.3586    2.7189
    3.2979    5.2428
    4.3530    2.0517
    4.3534    2.6438
    0.3440    4.2258
    3.8480    1.9787
```

So, if we combine them right. So, now, will have combined population which has 12 rows right and 2 columns. So, this top one is the parent or the variable P and the bottom 6 are the offspring right. So, now, what we will have to do is we will combine the objective function and the sort right. So, this within the square brackets, we are stacking the objective function similar to the way we stack the solutions and we are sorting it right.

(Refer Slide Time: 59:31)

The screenshot shows the MATLAB IDE with a script named 'GeneticAlgorithm.m' open. The script implements a genetic algorithm with the following steps:

- 38: Tournament selection
- 39: MatingPool = TournamentSelection(f, Np) % Performing the tournaments to select
- 40: Parent = P(MatingPool, i) % Selecting parent solution
- 41: % Crossover
- 42: offspring = CrossoverSBX(Parent, Pc, eta, lb, ub)
- 43: % Mutation
- 44: offspring = MutationPoly(offspring, Pm, eta, lb, ub)
- 45: for j = 1:Np
- 46: OffspringObj(j) = prob(offspring(j, :)) % Evaluating the fitness of the offspr
- 47: end
- 48: CombinedPopulation = [P; offspring]
- 49: [f, ind] = sort([f; OffspringObj]) % mu + lambda selection
- 50: P = f(1:Np)
- 51: P = CombinedPopulation(ind(1:Np), :)
- 52: end
- 53: [bestfitness, ind] = min(f)
- 54: bestsol = P(ind, :)

The Command Window on the right displays the output of the algorithm, showing a list of fitness values and their corresponding solutions. The 12th solution is highlighted in blue, indicating it is the best solution found.

Iteration	Best Fitness	Best Solution
1	4.3599	2.0445
2	0.2593	6.1927
3	5.4966	2.9945
4	4.3532	2.6683
5	4.2037	6.2113
6	3.3033	5.2914
7	4.3586	2.7169
8	3.2979	5.2429
9	4.3590	2.0517
10	4.3534	2.6630
11	0.3480	6.2235
12	18.7224	23.1973
13	23.1973	23.1973
14	23.2176	26.0434
15	26.0434	26.0702
16	26.0702	26.3794
17	26.3794	39.3429
18	39.3429	38.4169
19	38.4169	38.6767

So, this indicates that the 12th solution is actually the best solution 18.7224. So, that corresponds to this particular solution.

(Refer Slide Time: 59:43)

The screenshot shows the MATLAB IDE with a script titled 'GeneticAlgorithm.m' and a Command Window. The script includes the following code:

```
38 % Tournament selection
39 MatingPool = TournamentSelection(f, Np) % Performing the tournaments to select
40 Parent = P(MatingPool, i) % Selecting parent solution
41
42 % Crossover
43 offspring = CrossoverSBX(Parent, Pc, etac, lb, ub)
44
45 % Mutation
46 offspring = MutationPoly(offspring, Pm, etam, lb, ub)
47
48 for j = 1:Np
49     OffspringObj(j) = prob(offspring(j, :)) % Evaluating the fitness of the offspr
50 end
51
52 CombinedPopulation = [P; offspring]
53 [f, ind] = sort([f; OffspringObj]) % mu + lambda selection
54
55 P = ind(1:Np)
56 P = CombinedPopulation(ind(1:Np), :)
57
58 end
59
60 [bestfitness, ind] = min(f)
61 bestsol = P(ind, :)
```

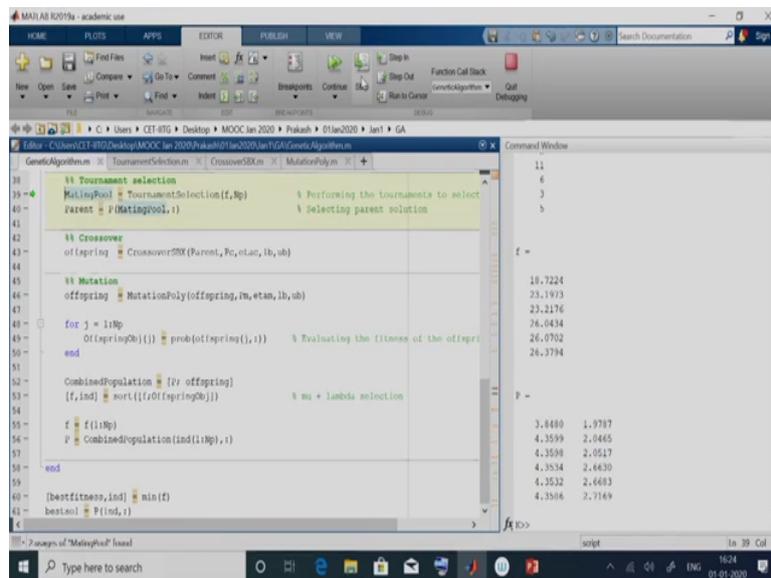
The Command Window displays the following output:

```
26.0702
26.3794
26.3629
39.4169
38.9767
39.9112
39.1902
56.2516

ind =
12
1
9
10
4
7
8
2
11
6
3
5
```

And next best solution is the first solution, the third best is 9 and the fourth best is 10 and so on right ah. So, these are the fitness function. Now, we need to appropriately extract that solution right. So, 18.7224 is here which the solution corresponding to this that is 3.8480 and 1.987. So, we need to extract that particular solution.

(Refer Slide Time: 60:13)



```
GeneticAlgorithm
38
39 % Tournament selection
40 % MatingPool = TournamentSelection(f,Np) % Performing the tournaments to select
41 % parent = P(MatingPool,1) % Selecting parent solution
42
43 % Crossover
44 offspring = CrossoverSRX(Parent, Pc, nlat, lb, ub)
45
46 % Mutation
47 offspring = MutationPoly(offspring, Pm, etan, lb, ub)
48
49 for j = 1:Np
50   Offspring0(j) = prob(offspring(j,:)) % Evaluating the fitness of the offspr
51 end
52
53 CombinedPopulation = [P offspring]
54 [f, ind] = sort(f(Offspring0)) % mu + lambda selection
55
56 f = f(1:Np)
57 P = CombinedPopulation(ind(1:Np),:)
58 end
59
60 [bestfitness, ind] = min(f)
61 bestsol = P(ind, :)
```

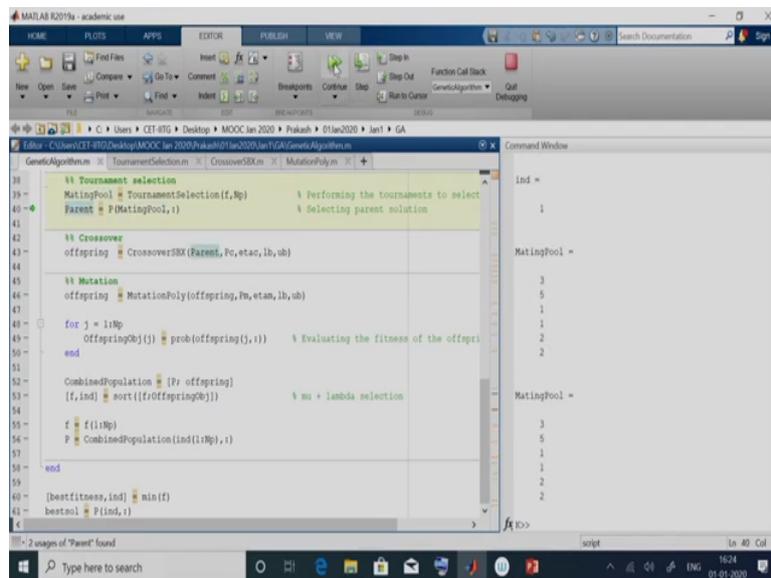
Command Window

```
11
12
13
14
15
16 f =
17 19.7224
18 23.1973
19 23.2176
20 26.0434
21 26.0702
22 26.3794
23
24 P =
25 3.8480 1.9787
26 4.3599 2.0465
27 4.3598 2.0517
28 4.3534 2.6430
29 4.3532 2.6683
30 4.3506 2.7149
```

So, before extracting right, we are just reducing the f to the first N_p elements right. So, N_p is no longer 12 right it is just the first 6 better solution right. So, this first 6 better solution are taken and the rest of the values are discarded right. This 6 values are discarded. Similarly, from this combined population we are extracting the N_p best members right.

So, when we do ind of 1 to N_p , it will actually correspond to 1, 2, 3, 4, 5, 6; this 6. So, we will extract this 6 solutions from the combined population and store it in the variable P right. So, P is now again 6 that completes one iteration of genetic algorithm right.

(Refer Slide Time: 60:59)



The image shows the MATLAB R2019a interface with a script editor and a Command Window. The script editor contains the following code:

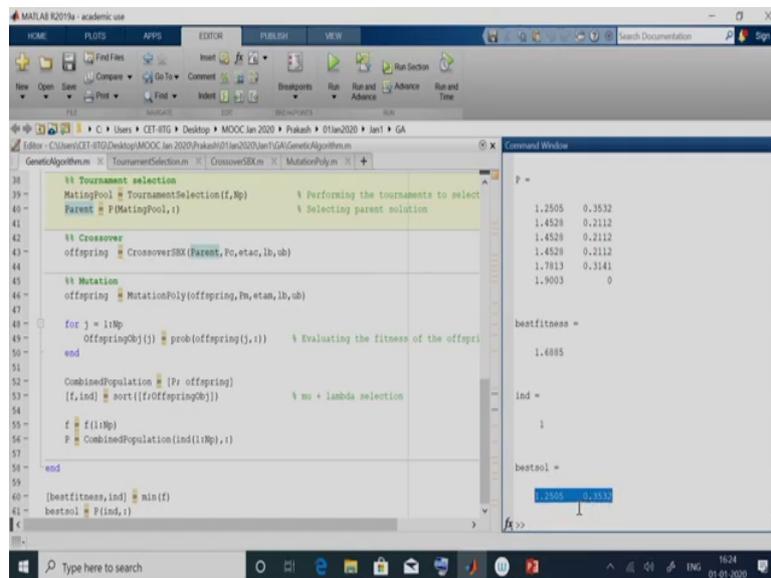
```
38 % Tournament selection
39 MatingPool = TournamentSelection(f,Np) % Performing the tournaments to select
40 Parent = P(MatingPool,1) % Selecting parent solution
41
42 % Crossover
43 offspring = CrossoverSX(Parent, Pc,etac,lb,ub)
44
45 % Mutation
46 offspring = MutationPoly(offspring,Pm,etam,lb,ub)
47
48 for j = 1:Np
49     OffspringObj(j) = prob(offspring(j,:)) % Evaluating the fitness of the offspring
50 end
51
52 CombinedPopulation = [P offspring]
53 [f,ind] = sort([f OffspringObj]) % mu + lambda selection
54
55 f = f(1:Np)
56 P = CombinedPopulation(ind(1:Np),:)
57
58 end
59
60 [bestfitness,ind] = min(f)
61 bestsol = P(ind,1)
```

The Command Window shows the following output:

```
ind =
     1
MatingPool =
     3
     5
     1
     1
     2
     2
MatingPool =
     3
     5
     1
     1
     2
     2
```

So, this we can implement it multiple times. So, in this case let me just continue so that it will complete everything.

(Refer Slide Time: 61:06)



The screenshot shows the MATLAB R2019a interface. The main editor window displays a script for a Genetic Algorithm (GA) with the following code:

```
38 % Tournament selection
39 MatingPool = TournamentSelection(f,Np) % Performing the tournaments to select
40 Parent = P(MatingPool,i) % Selecting parent solution
41
42 % Crossover
43 offspring = CrossoverSBX(Parent,Pc,etaa,lb,ub)
44
45 % Mutation
46 offspring = MutationPoly(offspring,Pm,etaa,lb,ub)
47
48 for j = 1:Np
49     OffspringObj(j) = prob(offspring(j,:)) % Evaluating the fitness of the offspr
50 end
51
52 CombinedPopulation = [P; offspring]
53 [f,ind] = sort([f;OffspringObj]) % mu + lambda selection
54
55 f = f(1:Np)
56 P = CombinedPopulation(ind(1:Np),:)
57
58 end
59
60 [bestfitness,ind] = min(f)
61 bestsol = P(ind,i)
```

The Command Window on the right shows the output of the script:

```
P =
    1.2505    0.3532
    1.4528    0.2112
    1.4528    0.2112
    1.4528    0.2112
    1.7813    0.3141
    1.9003         0

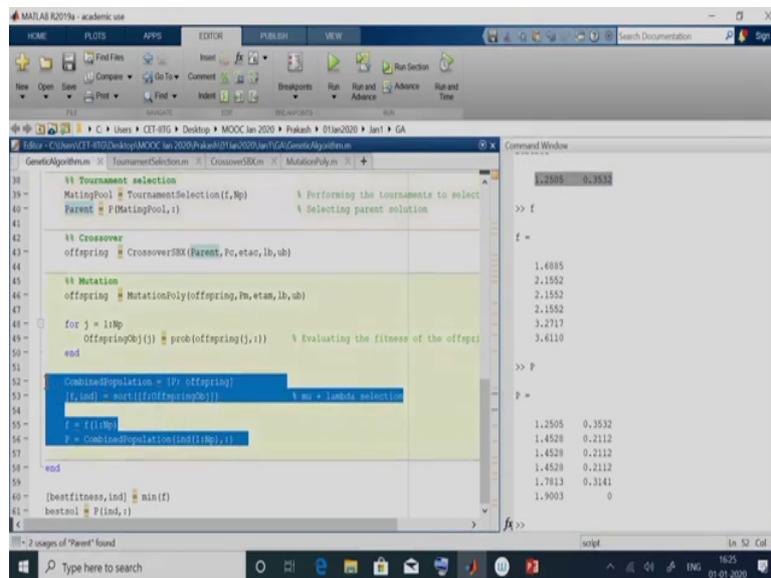
bestfitness =
    1.6885

ind =
     1

bestsol =
    1.2505    0.3532
```

So, here we had taken only 10 iterations right. So, at the end of 10 iteration, the best fitness function value we obtained is 1.6885 and the solution corresponding to it was 1.2505 and 0.3532 right.

(Refer Slide Time: 61:24)



```
%% Tournament selection
39- MatingPool = TournamentSelection(f,Np) % Performing the tournaments to select
40- Parent = P(MatingPool,1) % Selecting parent solution
41-
42- %% Crossover
43- offspring = CrossoverSBX(Parent, P0, eta0, lb, ub)
44-
45- %% Mutation
46- offspring = MutationPoly(offspring, Pm, eta0, lb, ub)
47-
48- for j = 1:Np
49-   OffspringObj(j) = prob(offspring(j,:)) % Evaluating the fitness of the offspring
50- end
51-
52- [bestIndObj, ind] = min(OffspringObj) % mu + lambda selection
53- f = f(ind)
54- P = CombinedPopulation(ind(1:Np,:), f)
55-
56- end
57-
58- [bestfitness, ind] = min(f)
59- bestsol = P(ind,1)
```

Command Window

```
>> f
f =
    1.6885    0.3532
    2.1552    0.2112
    2.1552    0.2112
    2.1552    0.2112
    3.2717    0.3141
    3.6110    0.3141

>> P
P =
    1.2505    0.3532
    1.4528    0.2112
    1.4528    0.2112
    1.4528    0.2112
    1.7813    0.3141
    1.9003     0
```

So, f also if we see the first value will be 1.6885 and if you look at this P the first value will be the same 1.2505 and 0.3532. So, this happens only in genetic algorithm because here we actually sort the population; rest of the meta heuristic techniques, we have discussed we never sort that the fitness function value right. So, that is why the best solution could be located anywhere and we were using the main function to identify where it is located and what is its location and we were extracting the corresponding solution.

Over here, it is always guaranteed because we are employing the $\mu + \lambda$ strategy. It is always guaranteed that the first solution would be the best solution right. At the end of iterations, all the iteration, the first solution will definitely correspond to the best solutions. So, that completes the implementation of genetic algorithm right. So, when compared to the rest of the meta heuristic techniques, it might seem slightly tricky to implement right.

But once you look into the code couple of times, it will become much easier for you to understand the code and the functioning of genetic algorithm right. As we have discussed multiple times, this meta heuristic techniques are useful only if you are able to do a large number of iterations with the larger population size fairly quickly right and since this are stochastic techniques, we need to again run it multiple times right. So, it just not possible to use this techniques using pen and paper right.

So, we need to have an automated way and that is best implemented using programming language. So, in this case, we had taken it as MATLAB. We will also circulate the codes which we have developed right. So, you can actually look into the code in the debug mode and try to further understand and as usual if you have any doubts, you can drop an email; we will get back to you.

That is the end of this session, in the next session we will be looking into artificial b colony optimization and its implementation using MATLAB.

Thank you.