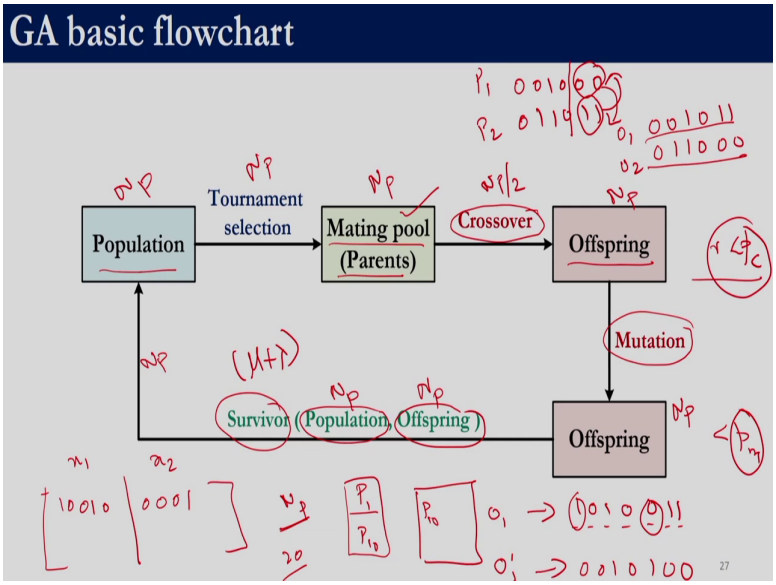
Computer Aided Applied Single Objective Optimization

Dr. Prakash Kotecha
Department of Chemical Engineering
Indian Institute of Technology, Guwahati

Lecture – 15
Real Coded Genetic Algorithm

Welcome back, in this session we will look into Real Coded GA. As you know most of the problems that we encounter do involve real variables and binary GA requires that we convert the real variables into binary strings and that increases the size of the problem; hence real coded GA is better suited for problems with real variables right.

(Refer Slide Time: 00:58)



So, before we get into the actual operators of real coded genetic algorithm; let us quickly recall what is genetic algorithm right. So, in genetic algorithm we initially have a population

which is randomly generated; in binary coded GA the population was made of binary strings right.

Let us say, if we have two variables then if we assume that variable x 1 is represented by 5 bits and variable x 2 is represented by 4 bits right. So, our population would look something like this right just binary variables in the population. So, once the initial population is generated; we employed tournament selection right, in tournament selection we randomly picked two solutions. Let us say population number 1 and population number 10, we conducted a tournament between P 1 and P 10 and whichever solution had lower fitness value or whichever solution was better was taken into the mating pool right.

So, let us say P 10; 1, so like this we conducted N P tournaments where N P is the size of our population. So, of our size of the population is 20 then we conduct 20 such tournaments; when we conduct 20 such tournaments there would be 20 winners right. So, and if you recall the best member will find two copies of itself right, the tournaments are to be conducted in such a way that each population member gets to play in two tournaments right.

So, once we had mating pool the solutions in the mating pool are known as parents and then we had that single point crossover. So, if our two parents are let us say 0 0 1 0 0 0 and 0 1 1 0 1 1 and if we decide to have a single point crossover and this is our crossover site, then the new offsprings are 0 0 1 0 and 1 1. So, this tale goes to this population member and 0 1 1 0 0 0; so, this tale comes to this population member.

So, this was our new offspring 1, this was our new offspring 2; these were our parents 1 and parent 2. So, here there is no need to bound because, the values would be either 0 or 1; so bounding does not come into picture in binary coded GA right. Once we had these offsprings we employed bit wise mutation right in bit wise mutation what we did was for every variable let say this is the solution that is to undergo mutation let us say this is offspring.

So, for generating the new offspring from mutation, what we will be doing is for each of this value we will generate a random number. If that random number happens to be less than mutation probability which is a user defined parameter. So, it happens to be less than mutation

probability, we mutate that bit from 1 to 0 or 0 to 1. So, for example, in this case let us say for this variable and for this variable the random number happened to be less than mutation probability. So, the new offspring would be 0 0 1 0; 1 0 0 right; so this would be the new offspring.

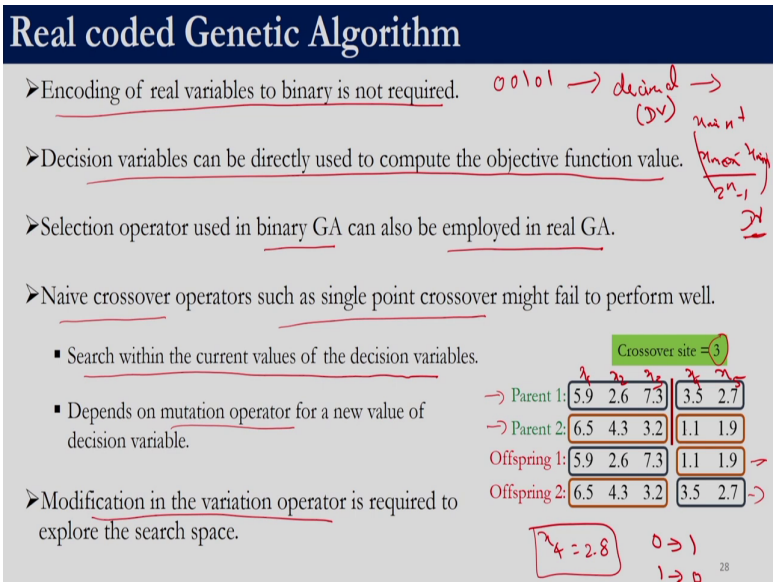
So, initially we started with N P right in mating pool also we require N P parents; to have N P parents will have to have N P tournaments each tournament will have a winner. So, once we have this mating pool we had to randomly select two parents and generate a random number. If the random number happens to be less than crossover probability, we employ this single point crossover right else we copy the parents themselves as offspring right. So, here we need to conduct N P by 2 crossovers; N P by 2 cross overs because each crossover gives us two offspring right.

So, if this condition is not valid that is the random number is not less than crossover probability, then we merely copy the parents as offspring. So, here we will again have N P population members I mean N P offsprings here also will have N P offsprings. So, now, we have N P population which we started with and then we have N P offspring. So, we two N P solutions; out of these two N P solutions we will evaluate the fitness of this offsprings because the fitness of the population as we performs the subsequent iteration would be known right.

So, we evaluate the fitness of this N P offspring right and out of these two N P solutions; we select the best N P solutions using this survivor operator. Survivor operator which we employ in GA is mu plus lambda, we combine the population which was at the start of the particular iteration and the offsprings generated in that iteration; we combine them we sort the fitness function value and the best N P values are taken right.

So, this is the basic flow chart of genetic algorithm which we had seen in the last session right. So, over here this step is going to remain the same; there is not going to be any change, the way we do crossover and the way we do mutation is what is going to change in P L coded genetic algorithm otherwise the scheme is going to remain the same.

(Refer Slide Time: 05:58)



So, in real code a genetic algorithm encoding of real variables to binary is not required; if you have understood binary coded GA then this converting real variables to binary variables actually increased the length of the population string. Right and also it comes with a particular precision right. So, if you want to increase the precision, we will have to increase the number of bits that we want to choose right. So, that was one big problem with binary coded GA.

So, in real code GA the decision variables can be directly used to compute the objective function value. In real coded GA, if you remember from whatever that binary string we have let say 0 0 1 0 1; if we have this has to be first converted into its decimal equivalent or what we discussed as decoded value.

And this decoded value was to be used in that expression x min plus x max minus x min divided by 2 power n minus 1 into the decoded value. So, we had to plug this decoded value

over here and then we were able to determine the actual value of the real decision variable; that was to be subsequently used for determining the fitness function value or the objective function value right.

So, whatever operator we used for binary GA can also be employed in real GA. For example, we can have a naive crossover which implements the single point cross over right. So but usually it is seen that the strategy does not work well right. So, for example, let us say in terms of real variables let us say we have five variables; x 1, x 2, x 3, x 4 and x 5. Let us say this is parent 1, this is parent 2 and between the lower and upper bounds let us say these are the values that we have right 5.9, 2.6, 7.3, 3.5, 2.7 its one solution that we have; 6.5, 4.3, 3.2, 1.1, 1.9 is the second solution we have.

So, in single point crossover what we did is; we randomly selected a crossover site let us say if we selected a crossover site of 3. So, this has to be randomly generated between 1 and decision variables right. So, here if we do a crossover then the offsprings can be 5.9, 2.6, 7.3 and this 1.1 and 1.9; this is our first offspring and 3.5 and 2.7 will form the tale of the second parent giving us the second offspring.

So, this can be done in a real coded GA also. So, for example, let us say the optimized for the fourth variable x 4 in the optimal solution along with other variables x 4 happens to be 2.8. So, if we keep swapping like this; we will not get this 2.8 at all; whereas, for the other techniques which we discussed in this course, we saw that there was some operation involved which gives rise to new values of the decision variables right. So, this crossover does not help us to get new values.

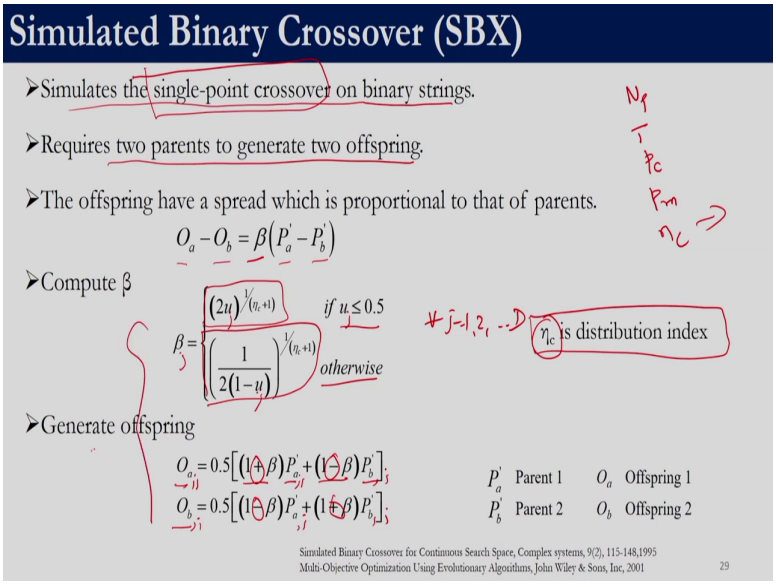
So, what we are doing is we are searching within the current values of the decision variables. So, we do not get a new value in crossover and will have to depend only on mutation operator, because in mutation for binary GA, it was straightforward that if it is a 0 we converted into 1 and if it was 1, it was converted to 0. So, that is the only place where the value of the decision variable can change right. So, that is why we say that these operators do not perform well in real coded GA right. So, what we want is a modification in the variation

operator. So, that the search space is effectively explode; in this course we will be discussing SBX crossover.

So, SBX crossover stands for Simulated Binary Crossover and polynomial mutation. Again you need to remember there are various other types of crossover and mutation operators. So, here we are discussing what we can say are the commonly used operators right. So, SBX and polynomial mutation are widely used; so that is why we are looking into the SBX crossover and polynomial mutation.

Again in this course, we will not go into the in depth as to how SBX crossover and polynomial mutation were designed, we will give you the appropriate reference if you want to further read on it you can read it. But here we will see the basic properties of SBX crossover and polynomial mutation and we will see how to use them.

(Refer Slide Time: 10:21)



SBX crossover as the name itself says it simulates the single point crossover on binary strings. So, in binary strings what we were using was single point crossover so, but here in SBX crossover, it simulates the single point crossover on binary strings. So, for SBX crossover we require two parents to generate two offspring. So, this is similar to binary coded GA; we need two parents when we employ SBX crossover we will get two offsprings right; the offsprings that we generate have this property right.

So, P a dash and P b dash are the two parents; we have used that P a dash in order to distinguish it from the population, it is not from the population, but it is from the parents. Remember, we had a population and then from the population we filtered out few solutions which are known as parents using the tournament selection right. So, this P a prime and P b prime are the parents right O a and O b are our offsprings right.

So, the way SBX crossover works is if that distance between the parents is large, the distance between the offsprings which we generate will be large and if the distance between the parents is smaller the distance between the offsprings would also be smaller right. So, this beta value if we see this beta value this beta value is computed using this formula. So, this u indicates a random number right.

So, for every variable we need to generate a random number; if the random number happens to be less than or equal to 0.5 right. So, we need to calculate beta using this part where eta c is known as the distribution index and it is a user defined parameter right. So, if we kind of try to consolidate, we had to fix the number of population, the termination criteria crossover probability we had previously seen that crossover probability is required mutation probability and now we have this eta c also right. So, this is also to be provided by the user right.

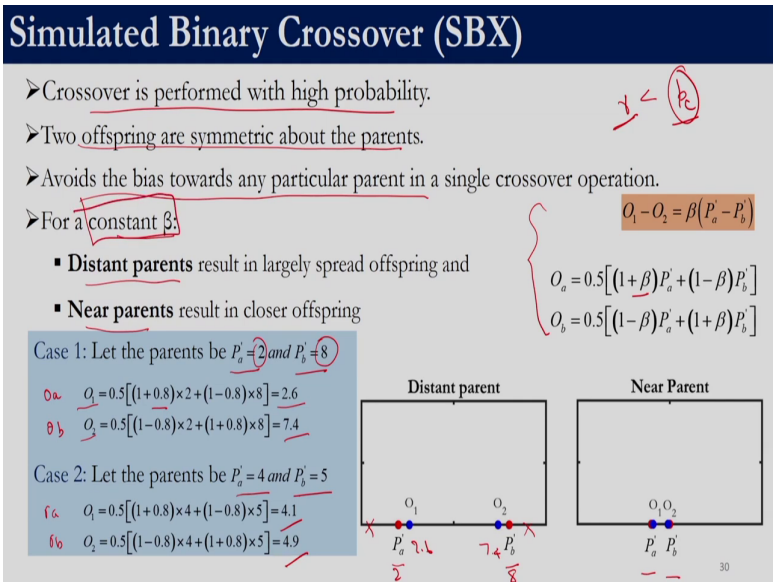
So, once we are calculating beta right then we can calculate the new offspring using these two formulas. So, if the random number is less than or equal to 0.5, beta is calculated using this for the first part if u is greater than 0.5 right. So, then beta is calculated using this the second part of the equation right. So, once we have determined beta; so the offspring is O a and O b right. So, 1 plus beta into parent a plus 1 minus beta into parent b right whereas, the second

offspring is the same formula instead of plus we have minus over here and instead of minus, we have plus over here right.

So, this is how we generate the offsprings; remember this beta has to be generated for every decision variable. So, if we have D decision variables then this goes for j is equal to 1, 2; all the way up to D right this will be u j right this is also u j this would be u j and this is beta j. Remember, this distribution index is not to be varied for every variable right distribution index is common for all the decision variables, but this random number is to be generate; for generated for each variable and beta is calculated for every variable right.

So, over here also if there are more than one variables this is the j th variable, this is the j th variable and here also we will get the j th variable right. So, as we discuss an example towards the end of this session, you will be able to further understand the application of these equations to generate the new offsprings.

(Refer Slide Time: 13:57)



So, as usual crossover is to be performed with high probability whereas, mutation is performed with the lower probability; that means, this p c which is being set by the user should have a relatively larger value right, it has to be between 0 and 1 because it is a probability.

So, it has to be between 0 and 1 and since we are generating random numbers if this is of high value, lot of random numbers that we generate will satisfy this condition, r is less than p c and a lot of crossover would happen right. So, that is why crossover is to be performed with high probability right. So, the two offsprings which we generate are symmetric about the parents; this avoids the buyers towards any particular parent; these are the properties of SBX crossover. So, for example, let us consider a case where in the beta is constant and there are the parents are distant and the parents are near.

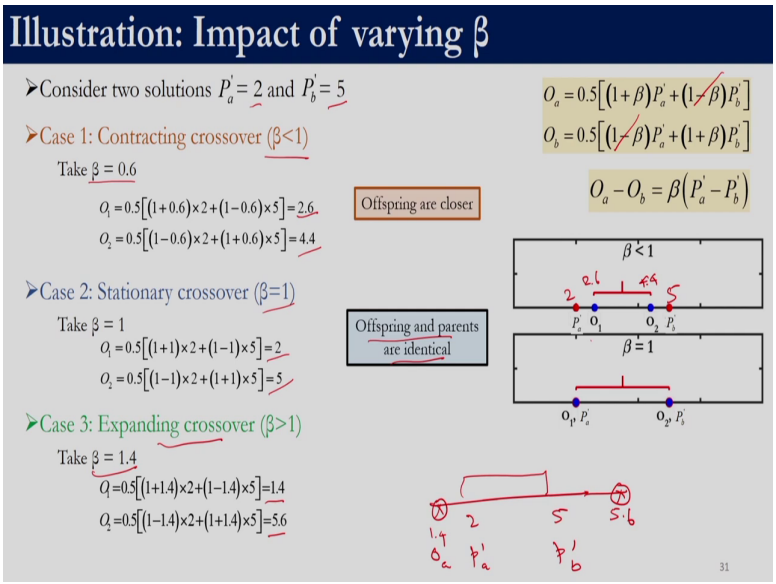
So, if we take an example; so these are the equations which we discussed in the previous slide right. So, let us say we take a beta value of 0.8 and the parents to be 2 and 8. So, it is a one variable problem. So, if we take P a dash as 2 and P b dash as 8; so these are our parents. So, we want to see where would the offsprings lie right with a particular value of beta. So, the beta value is 0.8. So, if we apply this formula 0.5, 1 plus 0.8 into P a prime is 2 plus 1 minus beta P b prime is 8.

So, if we calculate that we will get 2.6; similarly if we calculate the second offspring we will get 7.4 right; this can this is either O 1 or O 2 or you can even call it as O a and O b. So, these are the two offsprings. So, now, if we see pictorially over here the parent is 2, the other parent is 8; the solutions which we have generated are 2.6 and 7.4. So, the solutions are within the parents right they have not gone out over here or over here right and the distance if you see both the offsprings are closer to their parents right.

So, had this distance been far right the distance between the offspring which we generated would also be far; if this parents had been even closer this offsprings would have been even closer right. So, let us say the other example. So, instead of 2 and 8; let us say if the parents were 4 and 5. So, now in this case the parents are closer to one another when compared to 2 and 8; 4 and 5 are closer to one another. So, if we calculate the offsprings, it turns out to be 4.1 and 4.9. So, that if we see is much closer to 4 and 5 right.

So, if we pictorially see only a small distance is there between the parents and the offspring. So, this is when we had a constant beta value right.

(Refer Slide Time: 16:49)



So, let us see if we vary beta what happens. So, we have these three cases, the first one is were in beta happens to be less than 1 right; so this is called as contracting crossover. So, beta let us say is 0.6 and if the parents are 2 and 5 right then the solutions that we get is 2.6 and 4.4. So, if we look at that pictorially these are our parents 2 and 5 and offsprings are 2.6 and 4.4, the offsprings are contained within the parents right and depending upon the distance between the parents the distance between the offsprings would be proportional to the distance between the parents.

So, if beta happens to be 1 right; so in this formula if you substitute this would go off and this would go away right. So, those two terms go away and then since this is 0.5, this is 2 right. So, 0.5 into 2 will come out to be 1. So, what we will get is the original parent itself. So, beta is equal to 1 does not help us to generate new solutions will be generating the same solution.

So, in this case the parents and offsprings are identical ok. So, expanding crossover as you can probably now intuitively guess.

So, if beta is 1.4; so in this case the offsprings would be 1.4 and 5.6. So, if we plot this on the scale if 2 is here, 5 is here. So, these are our parents P a dash and P b dash are our parents. So, what we have got is 1.4 right; so this is the new solution this is offspring a and the offspring b is 5.6; 5.6 this is our new offspring right this is 1.4.

So, the offsprings if you see; it is not contained within the parents, it is outside the parents right whereas, here in beta less than equal to 1 it was contained within the parents. In all the cases, the distance between the offsprings will be proportional to the distance between the parents. So, to consolidate SBX crossover for each decision variable we had to generate a random number depending upon whether the random number is less than equal to 0.5 or greater than 0.5 we will have to calculate beta for each variable right.

And again when we are calculating beta, we require a user defined parameter eta c which is known as distribution index for crossover. Once we have calculated beta, we can calculate the two offsprings; for us to use SBX operator we need two parents right and then we will get two offsprings. So, now if you see depending upon eta c which is fixed by the user and the random numbers which are generated b; we will get a beta value and that beta value will help us to generate the offsprings.

And now if you see we do not have the same problem as we had in naive crossover, we can actually generate new values for the decision variables. Unlike, the naive crossover which we discussed in the beginning this SBX crossover operator will actually help us to get new values for the decision variable right.

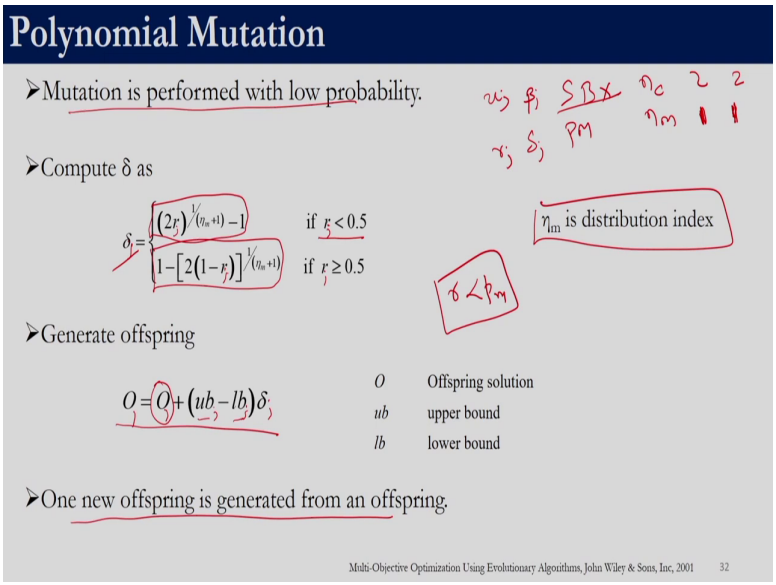
So, and as we have seen this SBX crossover has unique properties depending upon the values of beta right; we will be generating the offsprings which are contained within the parents or which are outside the parents. But in any case difference between the offspring will be in

proportion to the distance between the parents. If you recall the basic GA flow chart right, so after crossover we had mutation right.

So, in mutation for binary coded GA it was very easy because we generated a random number for every bit and if the random number happen to be less than mutation probability, we converted a 0 into 1 and 1 into 0, because those were the only two values permissible in binary coded GA right. Whereas, in real coded GA that is not the case because the variable has said domain right which is not resisted to either 0 or 1 right.

So, the question is if we decide to do mutation; how do we actually mutate this bit right. So, that is what we will see now using polynomial mutation.

(Refer Slide Time: 20:56)



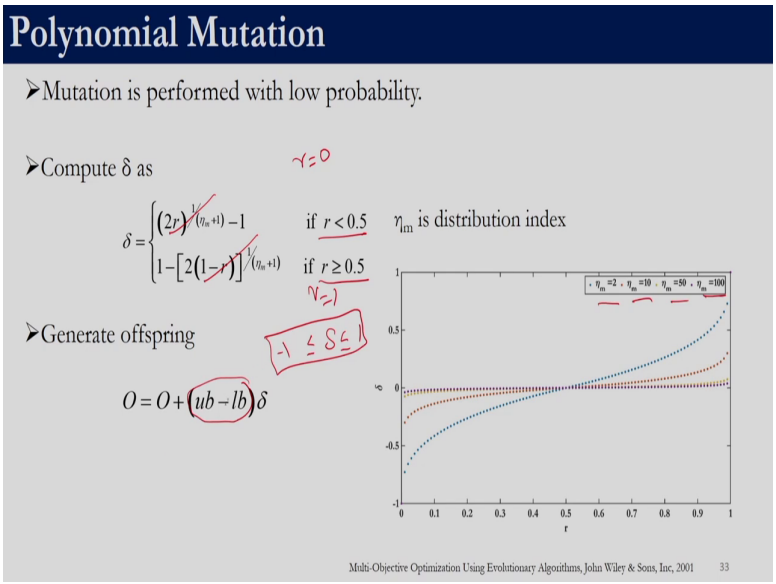
So, as usual mutation is performed with low probability; crossover is performed with high probability. So, just as we computed beta in SBX operator; we need to compute this del in polynomial mutation right. Similarly, as to we had this distribution index for crossover, we have a distribution index for mutation which is eta m right.

So, similar to SBX crossover for each decision variable we need to generate a random number. If the random number happens to be less than 0.5, we calculate del using this expression else we use this expression depending upon the value of random number we will be able to calculate del. And similar to SBX crossover just like we had beta for every variable, we have del for every variable right. So, for calculating beta for every variable we had a random number generated for every variable. So, here also we will generate a random number for every variable.

So, if there are 5 decision variables we will generate 5 random numbers depending upon whether they are less than 0.5 or greater than equal to 0.5; we will calculate this delta j for each variable right. So, once we calculate delta j for each variable; we can generate the offspring using this expression right. So, this is offspring because polynomial mutation is employed after crossover right; so we already have offsprings.

So, offspring plus this is the upper and lower bound of the respective variable right. So, of the j th variable right and this delta is again what we computed for that particular variable. So, this will give us the j th variable of the offspring right.

(Refer Slide Time: 22:40)



So, one property of this polynomial mutation is that as we increase eta m right. So, as you see here we have increased eta m from 2, 10, 50 and 100. So, this curves flat ends up and the delta value becomes smaller and smaller. So, this delta value can actually vary between minus 1 and 1. So, for example, take the case of r is equal to 0 right.

So, if r is equal to 0; it falls in this condition and this term will go away. So, delta will be equal to 1 and if r is equal to 1; it falls under this case right. So, this term would go away and delta would be 1; delta is going to vary between minus 1 and 1. This does not hold for beta, you can try it out right beta does not have this property delta is between minus 1 and 1. So, what we are essentially saying is this is the range of that particular decision variable right.

So, depending upon the value of delta; we are either adding anywhere between minus 1 to 1 time the range to the current value of the decision variable; one new offspring is generated

from an offspring right. So, unlike in SBX crossover where in two parents would undergo to produce two offspring, here one offspring would undergo mutation again with that condition.

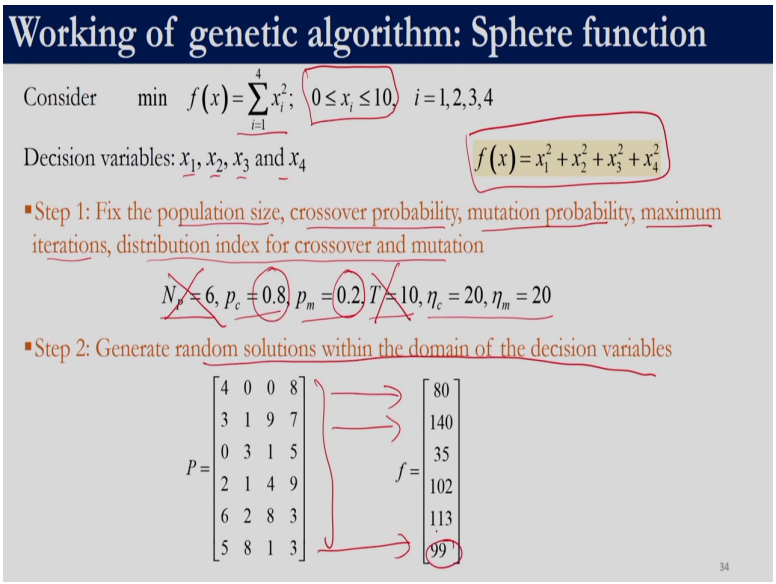
If the random number is less than the mutation probability right, then it would undergo polynomial mutation right in that case we require for every offspring will get a new offering; provided this condition holds that the random number which is generated is less than the mutation probability. Now, we have seen simulated binary crossover for the crossover operation and polynomial mutation for the mutation.

Now, we are in a position to solve the problem using real coded GA because in that flowchart; if you remember there were four steps four important step one was tournament selection which does not change whether we are doing binary coded GA or real coded GA right. The second was crossover operation right; so crossover for binary and real variables though we can use the same crossover operator which we used for binary coded GA; it is usually inefficient right; so that is why we looked into the SBX crossover.

Then the next operation is mutation for mutation, we have seen polynomial mutation and the survival strategy remains the same. So, out of four operations; four major operations two operation remain the same whether we are working with binary coded GA or the real coded GA; that is the tournament selection and the survivor strategy. The two operators that require some change is crossover and mutation.

We have studied variants for both of them SBX crossover and polynomial mutation; we should be able to solve optimization problem involving real variables using GA right without necessarily resorting to encoding them into binary strings.

(Refer Slide Time: 25:32)



So, let us look into this sphere function again get it. So, this is the problem that we have been working with for all other metaheuristic techniques. So, a scalable problem we take four decision variables x 1, x 2, x 3, x 4; the decision variables vary between 0 and 10 right. The objective function or the fitness function is x 1 square plus x 2 square plus x 3 square plus x 4 square; there are no constraints to this problem right.

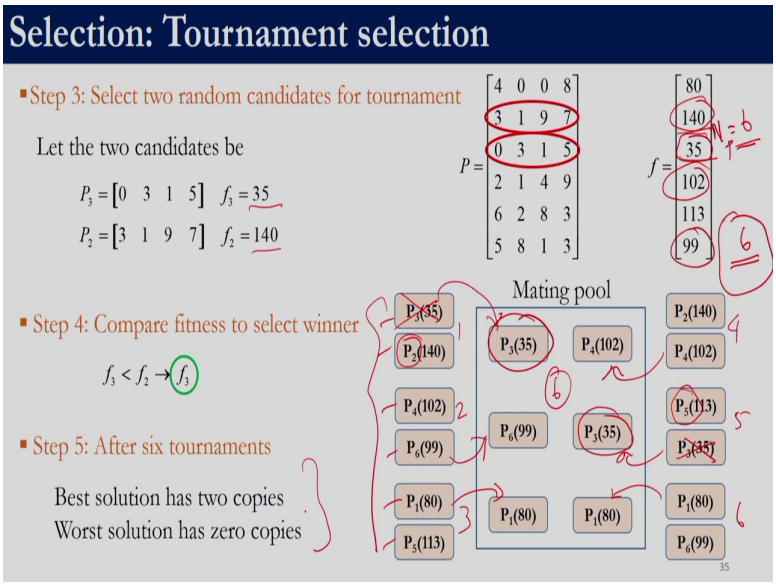
So, the first step is to fix these parameters; one is population size. So, in this case we take it as 6, then crossover probability we take it as 0.8, mutation probability we take it as 0.2. So, crossover probability if we see is higher than mutation probability right. So, that more solutions get an opportunity to undergo crossover as compared to mutation. We need to specify the maximum number of iterations here we have taken 10 iterations. So, that it helps us to stop the technique some point right.

So, as discussed earlier this can be any other criteria instead of maximum iteration; you could have fixed time or you could have seen if there is how much difference is there between the past few iterations and so on and so forth. And then we also need to give this distribution index for crossover and mutation. So, we let us take 20; 20 for each of them right. So, now, if you see this N P and this T is common for all the metaheuristic techniques which we discussed, but in addition to that we need to fix this 4 more parameters eta c, eta m, p c and p m right.

So, for an a naïve user it might become difficult to fix this these values and; obviously, the performance of the algorithm is sensitive to these parameters right. So, that can be said as one of the drawback of genetic algorithm that it requires these many user defined parameters. So, the next step is to generate random solutions within the domain of the decision variable. So, here it is between 0 and 10. So, as usual we have taken integer values so that it is easy to compute. So, we have six solutions randomly generated in this 0 to 10 bound and then we plug them into this fitness function and calculated its fitness.

So, the fitness if we say 5 square plus 8 square plus 1 square plus 3 square that should come out to be 99 right. So, this is just the fitness determined using the objective function.

(Refer Slide Time: 28:01)



The next step would be to conduct tournaments right. So, we have 6 members initially right and in the mating pool also we have decided that we will require 6 solutions right. So, that is a choice that we took that if N P is equal to 6; the size of the mating pool also needs to be 6 that is a choice we have made, we can tweak around with that right one can say that I want 12 members in the mating pool you know right.

So, in that case you will have to conduct more tournaments. So, in our case we have decided that we will do binary tournaments right as so each tournament will give us one winner. So, if we conduct 6 tournaments we will have 6 winners that will be the same as our population size right. So, for conducting tournaments; we need to randomly pick two solutions. So, let us say we pick solution 2 and solution 3, the fitness function of 2 and 3 are 140 and 35 respectively right.

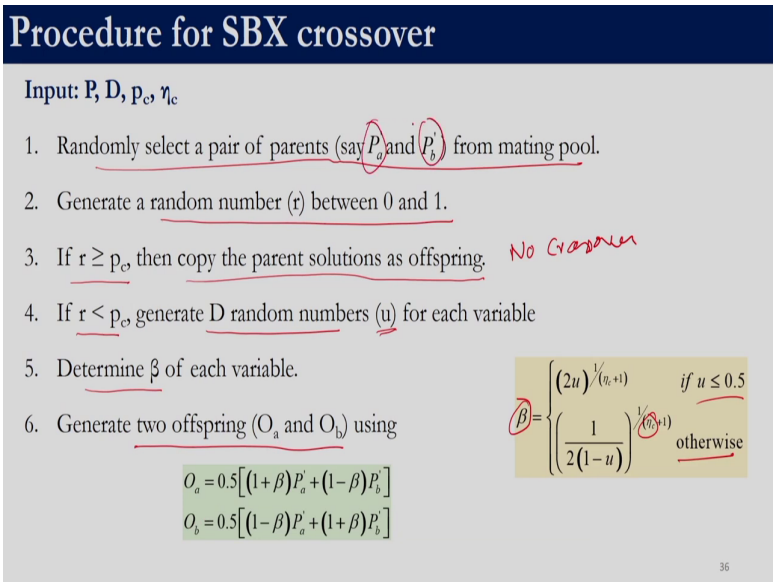
So, between these two if we conduct a competition; so obviously, 35 is going to win right. So, P 3 a copy of P 3 is placed in the mating pool. So, next time let us say we chose P 4 and P 6; so the fitness function is 102 and 99; so between them P 6 wins, so a copy of P 6 is placed in the mating pool. Similarly, we conduct one more tournament between the two left out right out of 6; 2, 3, 4, 6 have participated. So, 1 and 5 where the one which did not have a competition; we conduct a competition between 1 and 5 and a copy of 1 is placed in the mating pool because 1 wins.

As of now, every member of the population has participated once right. So, similarly we conduct one more set of competitions right. So, between P 2 and P 4 P 5 and P 3 and P 1 and P 6; so depending upon the fitness of each of them, so here again P 3 wins, here P 4 wins, here P 1 wins; so this is our mating pool right. So, we conducted 6 competitions 1, 2, 3, 4, 5, 6 and we have 6 winners who are in the mating pool. Here we need to make sure that each member participates twice right.

So, for example, P 3 had participated twice right; first time it competed with P 2; that is randomly chosen right and the second time it competed with P 5 right. So, like that if you check each member participates in at least two competitions right. So, the best solution among these if we see is P 3; 35 that is the least value and we will have two copies definitely two copies. So, one copy is here, the other copy is here right that is because no matter with whom P 3 plays; it is always going to win and P 3 is playing twice right.

Similarly, if we see the worst solution in this is 140 which is P 2, so there would not be any copy of P 2 because even though it plays twice, no matter with whom it plays it is always going to lose right. So, this we had also seen in binary coded GA right. So, now, we have our mating pool.

(Refer Slide Time: 31:08)



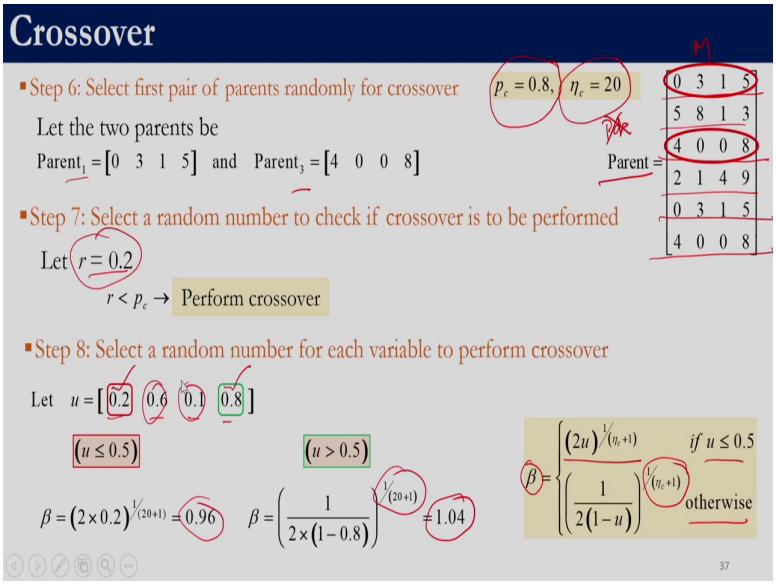
So, before we implement SBX crossover let us quickly recollect the steps. So, we need to randomly select a pair of parents from mating pool, then we need to generate a random number between 0 and 1. If the random number happens to be greater than or equal to the crossover probability, then we merely copy the parent solutions of and offspring.

So; that means, no crossover right, but if this condition is not satisfied and if the random number is less than crossover probability; then we need to generate D random numbers right; D random numbers because for each decision variable we will have to calculate this beta right. So, this random numbers which we are generating is actually the u right. So, for each variable we will have a u, depending upon the value of u; the value of beta would be computed.

So, depending upon the value of u and eta c which is fixed by the user; we need to determine the value of beta for each variable. Once we have determined the value of beta for each

variable, we can generate two offsprings O a and O b depending upon the values of the parents P a dash and P b right.

(Refer Slide Time: 32:16)



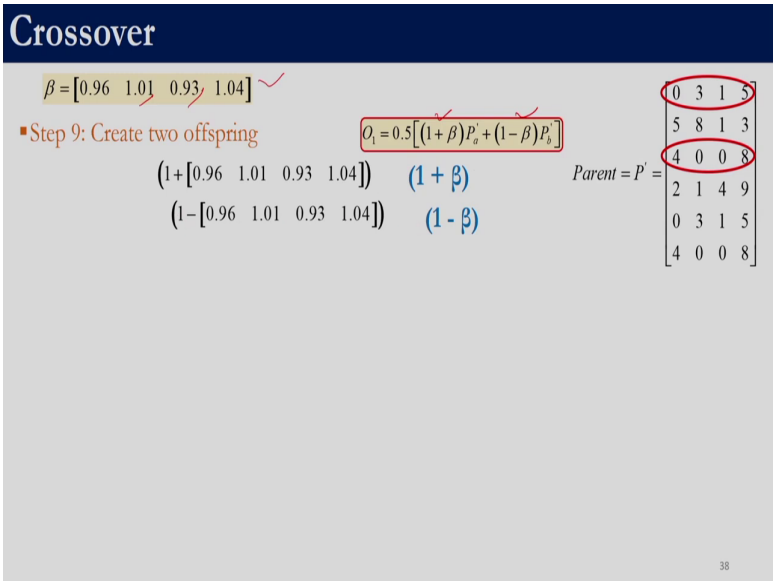
So, this is our mating pool. So, here if we see 0, 3, 1, 5 is over here as well as 0, 3, 1, 5 is over here; 4, 0, 0, 8 is over here and 4, 0, 0, 8 is over here and then we have the two other winners. So, this is our mating pool; so this is not population right these are our parents. So, in crossover we need to first randomly select two parents right. So, let us say we select parent 1 and parent 3 and then we need to generate a random number, let us say the random number is 0.2 and that our crossover probability is 0.8. So, since 0.2 is less than 0.8 we will have to perform crossover right.

So, to perform crossover we need to have random numbers for each decision variable. So, we have a 4 variable problem. So, we generate 4 random numbers between 0 and 1 right and then

we apply this equation right. So, if the random number is less than 0.5; we calculate beta using the first part, if the random number is greater than 0.5; we calculate beta using the second part. So, here in the case of 0.2; it is less than 0.5. So, we need to do 2 into 0.2 because 2 into u to the power 1 by 20 plus 1 because eta c we have taken as 20; 20 plus 1. So, if we calculate this; this will work out to be 0.96 right.

So, similarly we can calculate for 0.8; so 0.8 is greater than 0.5; so this second condition. So, 1 by 2 into 1 minus 0.8; again to the power 1 by 20 plus 1 because we have this term in the to the power right; so this value come comes out to be 1.04. We have shown it only for these two variables, you are supposed to calculate for these two variables also right.

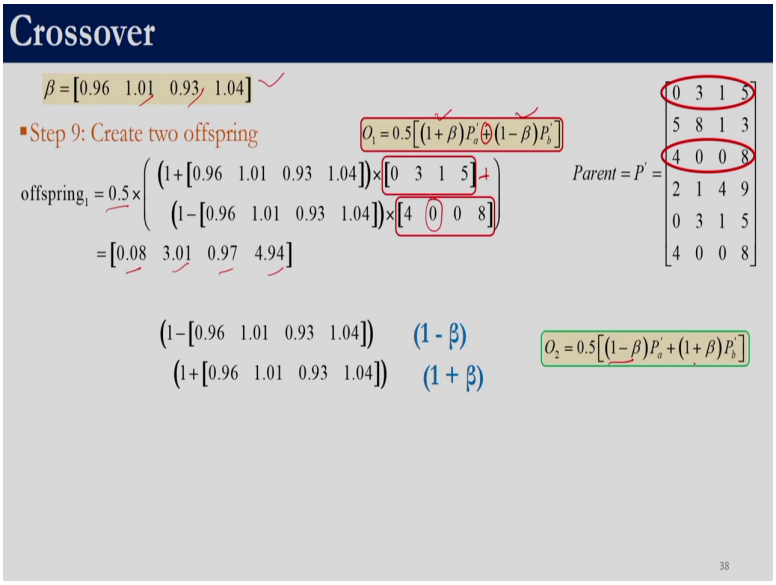
(Refer Slide Time: 34:05)



So, if you calculate the beta for those two variables it should come out to be 1.01 and 0.93. So, now, we have computed the beta value; then we need to generate the two offsprings right.

So, the first offspring can be generated using this formula 0.5; 1 plus beta into parent a plus 1 minus beta into parent b; so this is our beta value right. So, 1 plus beta and 1 minus beta, this part is right now shown. So, 1 plus 0.96, 1.01, 0.93, 1.04 and 1 minus beta is 1 minus again the same values multiplied by the respective parent.

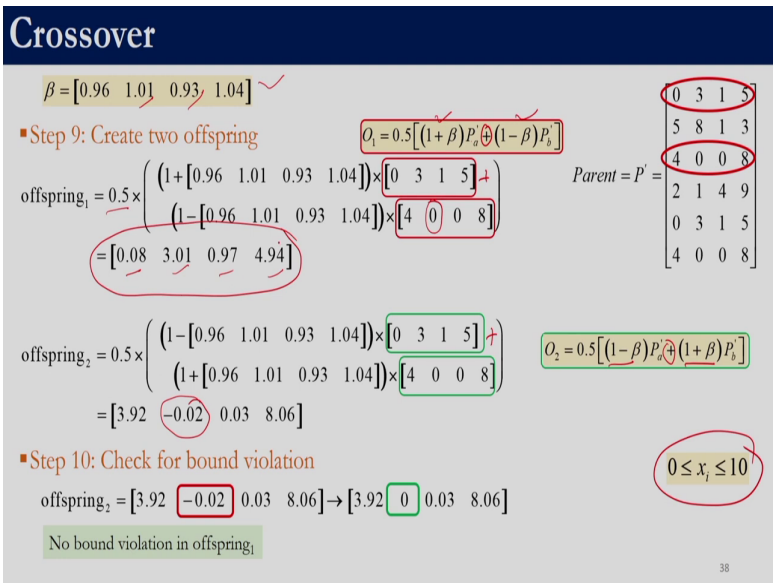
(Refer Slide Time: 34:44)



So, the first parent P a prime is 0 3 1 5; the second parent is 4 0 0 8 right and together multiplied with this 0.5. First, we need to add this 1; so 1.96, 1.96 into 0 that 1.96 into 0, there has to be a plus over here because of this plus; plus 1 minus 0.96. So that would be point 0 4 into 4 right, this into 0.5; this will give the value of the first variable right that should work out to be 0.08.

Similarly, the value of the second variable is 1 plus 2; 1.01; so that would be 2.01 into 3 right plus 1 minus 1.01 into 0. So, that into 0.5 if you do that will give 3.01 and similarly we can calculate the rest of the two values; so this is our first offspring.

(Refer Slide Time: 36:00)



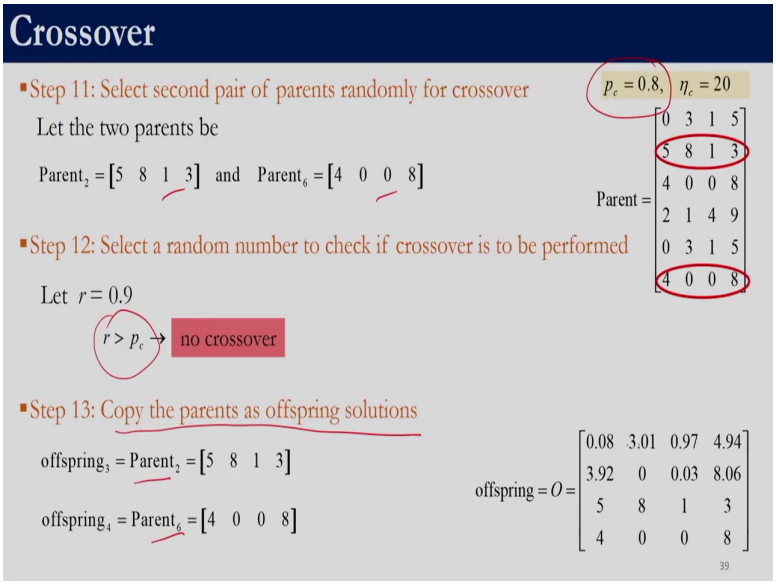
Similarly, we can calculate our second offspring right 1 minus beta 1 plus beta; so 1 minus beta and 1 plus beta multiplied by the two parents P a and P b right because of this place; we need to have a plus over here right. So, if we compute this expression right; so we will get the second offspring.

So, this is the second offspring right now we have both the offsprings. So, now we need to check for bounds in binary coded GA we were only working with 0 and 1, the permissible values are 0 and 1 and any operation we do; we will get either 0 and 1. So, there was no need to check for bounding; whereas, in real coded GA because of the binary crossover and

polynomial mutation, we can get any value for the decision variable. But we need to make sure that the variable is bounded back right; if it is violating the bonds we use the same corner bounding strategy that we had discussed previously to bound it back right.

So, in this case if we see our bounds are 0 and 10 for all the four variables. For offspring one all are between 0 and 10 right. So, offspring 1 is fine right for offspring 2 if we see this particular variable is out of bounds; the other three variables are within bounds. So, we again do that corner bounding strategy since this violates the lower bound right; the lower bound is 0, since it is violating the lower bound we will push it back to the lower bound which is 0. So, this minus 0.02 has been converted to 0. So, now we have two offsprings 0.083, 0.01, 0.974, 0.94 and 3.92, 0, 0.03 and 8.06.

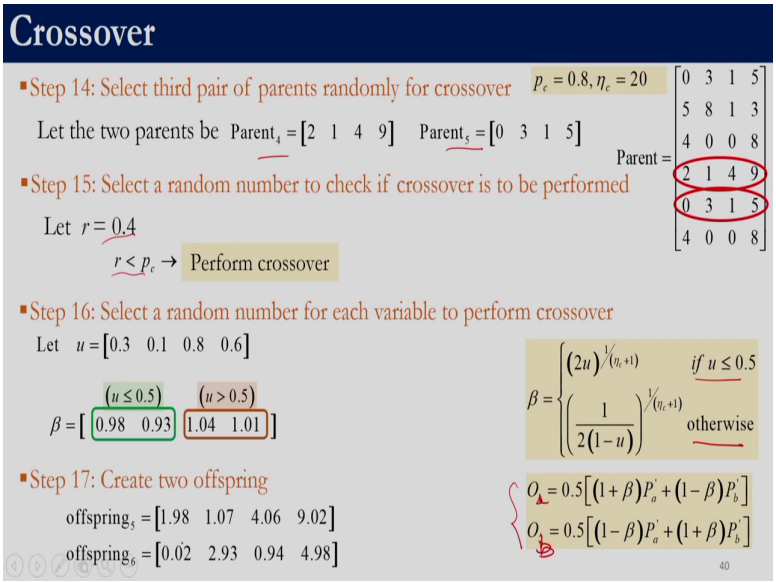
(Refer Slide Time: 37:27)



So, again we need to select two parents right. So, we need to conduct 3 crossovers because with each crossover we will get 2 offsprings. So, we need to conduct 3 crossovers; so as to have 6 offsprings. So, let us say in this case we take parent 2 and parent 6 to undergo crossover; again these are randomly selected right. So, here in this case our crossover probability is 0.8 and since r is greater than p c, no crossover is required.

So, in this case the offspring 3 and 4 would be nothing, but parent 2 and 6. So, we are just copying the parents as offspring solutions. If the crossover condition is met we perform SBX crossover else we merely copy the parents as the respective offsprings. So, we have now this 4 offsprings right.

(Refer Slide Time: 38:18)

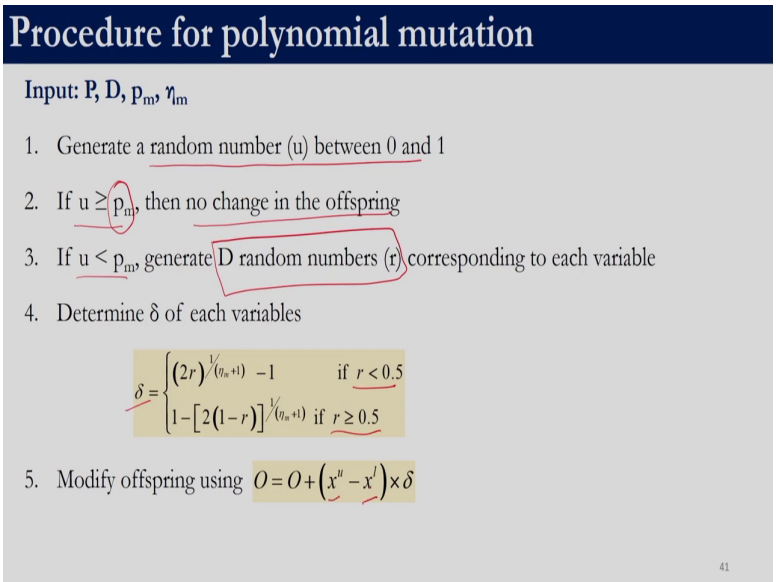


And then we conduct the final crossover the third time. So, here we are selecting parent 4 and parent 5 right and the random number that we generate is let us say is 0.4; so it is less than this

0.8; so we need to perform a crossover. So, if we perform a crossover we need to have a random number for each variable depending upon the value of random number; we need to apply one of these conditions and generate the beta value for each decision variable got it and then we need to calculate the offspring right.

So, here we have not shown you the calculations; we expect you to work it out and see if these values which are shown here are correct right. So, with these two expressions ah; we will get two more offspring. So, this has to be O a and O b right, just to avoid confusion that we already have two offsprings O a and O 1 and O 2; now we have 6 offsprings right. So, here probably since none of this is corner bounded I think so the values themselves would not be violating the bounds, please work it out and see whether you are getting the same two offsprings right.

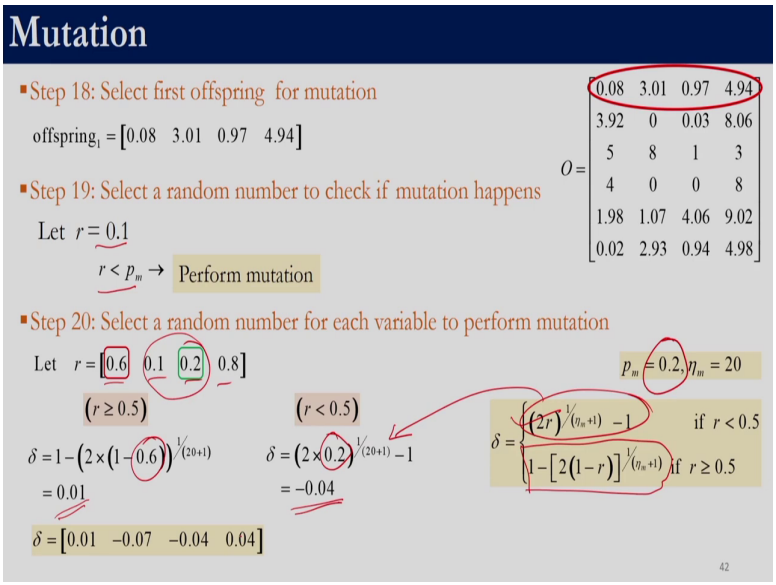
(Refer Slide Time: 39:30)



So, that completes SBX cross over for this; now let us quickly look into the polynomial mutation right before implementing let us recall the procedure we need to generate a random number between 0 and 1. So, if that random number happens to be greater than or equal to the mutation probability which is set by user; then there is no change in the offspring; so, no mutation is to be carried out.

If the random number happens to be less than mutation probability, we need to generate D random numbers for each variable right and then we need to calculate del value for all the variables right. So, if it is less than 0.5; we need to use this expression, if it is greater than 0.5 we need to use this expression and then depending upon the del value; we will have to calculate the offspring right. So, this is the upper bound and lower bound.

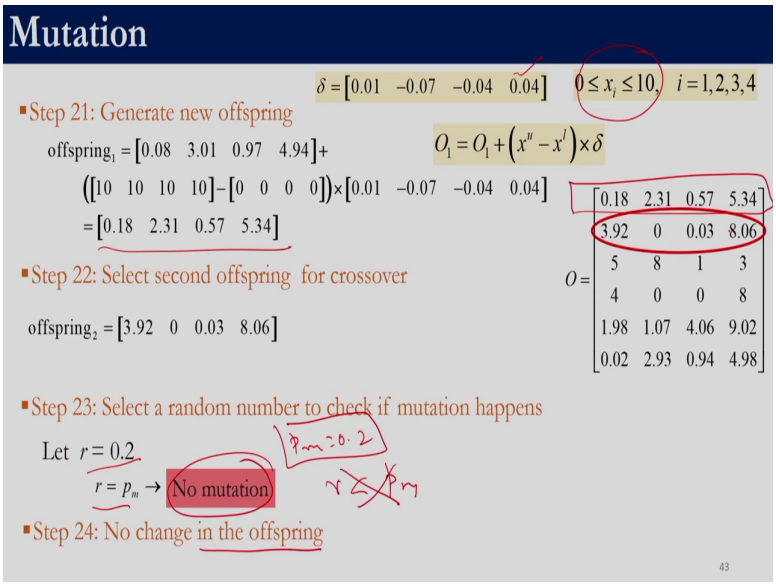
(Refer Slide Time: 40:15)



So, let us say we is for the first member we generate a random number; let us say it is 0.1. So, we need to perform mutation because it is less than the mutation probability which is 0.2 in our case. So, now, we need to generate a random number for each variable right. So, let say the random numbers are 0.6, 0.1, 0.2, 0.8. So, 0.1 and 0.2 are less than 0.5; so this expression will have to be used to calculate the del value; whereas, for the other two we will have to use the second expression right.

So, since it is greater than 0.5; for 0.6 we employ this expression 1 minus 2 into 1 minus r; 1 by 20 plus 1. So, this works out to be 0.01. So, if we take for 0.2; so 0.2 is less than 0.5, so we will have to employ this expression thus that is what is employed over here 2 into r to the power 1 by 20 plus 1 minus 1; so it works out to be minus 0.04. So, we have calculated it here for 0.6 and 0.2; we expect you to calculate it for 0.1 and 0.8 right. So, the del value for all the 4 variables would turn out to be 0.0; 0.01 minus 0.07, minus 0.04 and 0.04 right.

(Refer Slide Time: 41:32)

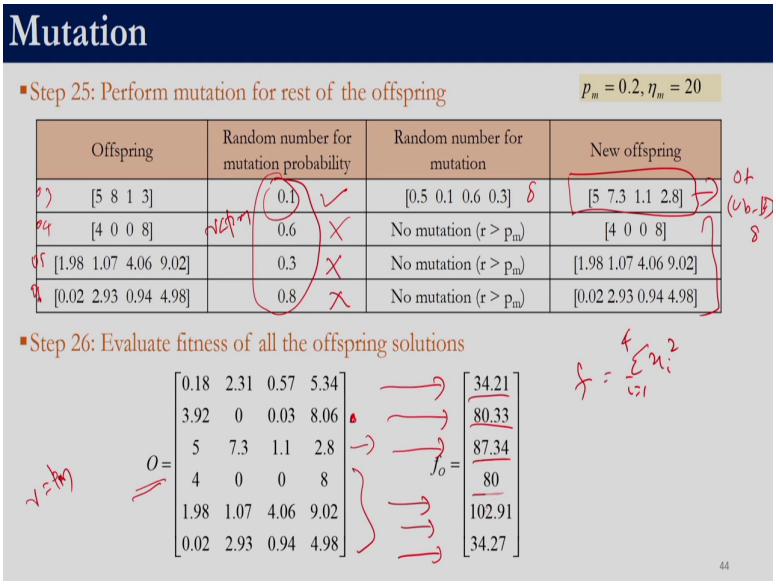


And then we employ this expression that our new offspring is nothing, but the old offspring plus the difference between the upper and lower bound into del right. So, the del value has been calculated over here and the current offspring number was this one right. So, if we compute this operation right; we will get this as the new offspring and since it is already bounded right our bounds are between 0 and 10; all these variables are between 0 and 10, so we do not need to bound it back.

So, for second offspring we will again generate a random number. So, in this case the random number is 0.2 and the mutation probability that we had taken was p m was also 0.2 right. So, since that condition is not satisfied; since r is not less than p m right this condition is not satisfied because r is equal to p m, we do not need to perform any mutation. So, if we do not

perform any mutation then there is no change in the offspring. So, the second offspring remains the same 3.92, 0, 0.03, 8.06.

(Refer Slide Time: 42:39)



So, similarly you can generate for all the rest of the 4 offsprings right. So, this is offspring 3, 4, 5 and 6. So, the random numbers which we generate to decide whether mutation is to be performed or not are given here right. So, for this case we will have to perform polynomial mutation, for the rest of the 3 case no mutation is required because it does not satisfy the condition r to be less than p m right.

So, for this again we need to have 4 random numbers 0.5, 0.1, 0.6, 0.3 right based on this we need to generate the del values and from the del values we need to calculate the new offspring is old offspring plus u b minus l b into del right. So, that if we do; we should be getting this

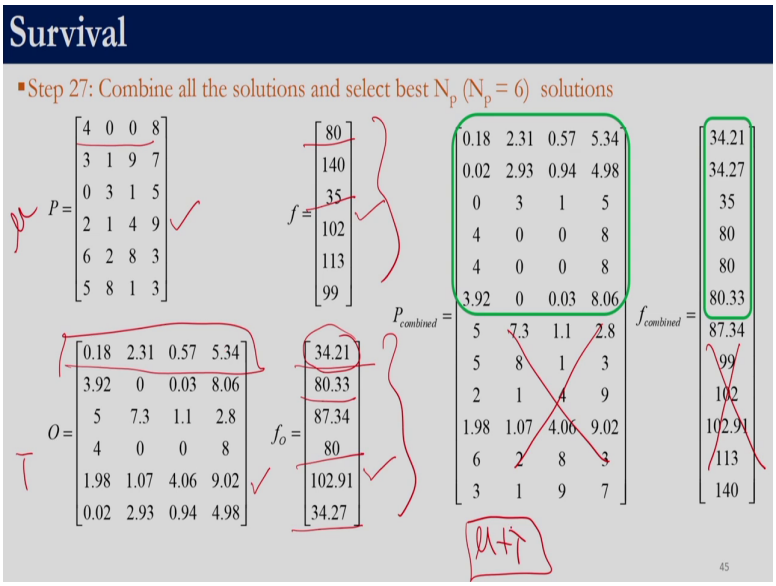
solution right. So, please go back and see if you are getting that same offspring which is shown here right. So, now we have completed polynomial mutation.

So, we initially had a population we selected parents from them and from the parents we performed cross over we got offsprings and those offsprings underwent mutation right and we have those modified offspring. So, here what we are showing is the modified offspring right.

So, these three remain the same over here right and these three this was the newly generated offspring and if we remember this offspring was also the same as we what we obtained from crossover, it did not undergo mutation because the random number that we generated was equal to the mutation probability and this had undergone mutation the first thing right.

So, at the end of it we have this 6 offsprings now we need to evaluate the fitness of all of them; for each of them we use the fitness function summation of x i square; i is equal to 1 to 4 right. So, this will give us the fitness which is what is given over here. So, we initially started with 6 solutions in our population and now we have 6 more solutions right.

(Refer Slide Time: 44:42)



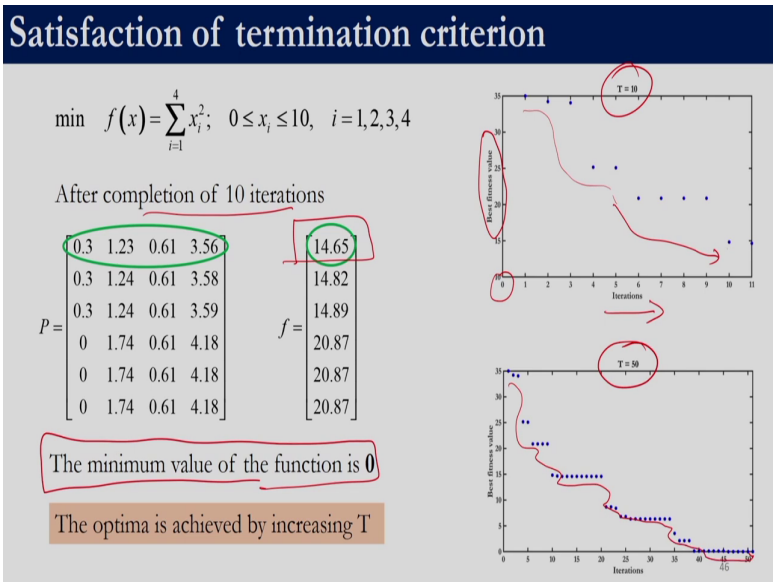
So, we have a total of 12 solutions now; this was our initial population right. So, remember this is not parent right and this is the fitness function corresponding to the population, these are the new offsprings which we have generated and this is the fitness of the offspring. So, now we have this 12 solutions; we are supposed to employ the survivor strategy because now we want to go to the next iteration, to go to the next iteration we require only 6 population members, but we have 12 right.

So, what we will do is we are combining all this 12 right and then sorting it right. So, if we sort it you will see that 34.21 is the least among these 12 right, this and this; the least is 34.21, 34.27; we have a 35 here or 2 times 80 we have and then we have a 80.33 right. So, remaining population numbers also you can sort it out right.

So, here if we see these are the 6 best members right. So, this is mu, this is lambda, this is mu plus lambda. So, from this population from this 12; we select the best 6 because that is our population size. So, we discard all these 6 solutions right. Again you need to remember especially when you are implementing you need to select the corresponding solution right the decision variable. So, there is no point writing 4 0 0 8 and corresponding to it writing 34.21 right.

So, for 34.21 the solution is 0.18, 2.31, 0.57 and 5.34. So, this is how we perform the mu plus lambda strategy. Now, we have 6 new solutions which will again undergo tournament selection will have a mating pool from that mating pool we will select parents who will undergo cross over and then mutation right. Similarly, we will end up with again 6 new solutions; we will combine all the 12 solutions, take the best 6 solutions and keep repeating this process till we reach the termination criteria which in this case is the number of iterations.

(Refer Slide Time: 46:52)

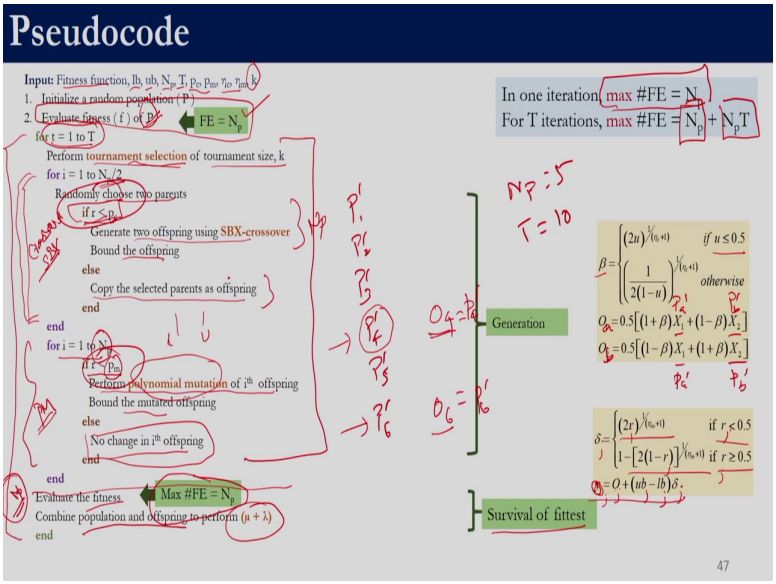


After 10 iterations, if you see the best value that we would have is 14.65 whereas, the optima is actually 0 right which means that the current parameters which we set did not help us to reach global optima. So, this shows the progress of GA; so with respect to the 10 iterations and the ordinate is the best fitness value at the end of each generation. If we plot the best fitness function value that we have, this is monotonically decreasing right because mu plus lambda when we are doing a solution can come in only if it is better than any of the other solutions right.

So, that ensures monotonic convergence in genetic algorithm. So, if we change the termination criteria right; if we instead of 10 iterations, if we perform 50 iterations right then we are able to reach the globally optimal solution. So, this again shows the convergence curve with respect to the iterations right. So, here if we see it is kind of converging almost to 0 right.

So, that completes our study of genetic algorithm. So, before concluding let us quickly compare binary coded GA and real coded GA right.

(Refer Slide Time: 48:19)



So, we had a pseudo code for binary coded GA; will quickly look into the pseudo code of real coded GA and see what are the major differences between binary coded GA and real coded GA. So, for real coded GA the input that is required from the user is the fitness function the lower and upper bounds of the problem, the population size, the number of iterations that we need to perform, crossover and mutation probability, distribution index for crossover and mutation and the number of solutions that are going to participate in a tournament which in our case is 2 right.

We both in real coded GA, as well as in binary GA; we had taken that two members will compete and a copy of the winner will be placed in the mating pool. So, the next step is to

initialize a random population right and evaluate the fitness of the population right. So, in real coded GA; there is no conversion we are not working with the binary strings alone right.

So, we do not need to find out the D decimal values or the decoded values and then find the real values to determine the fitness. Here, we can directly use the random population to determine the fitness of the population and then we need to repeatedly perform something for t times right.

So, this is our termination criteria. So, for t is equal to 1 to T; we need to perform the following. So, we need to perform a tournament selection right; so we will take of a specified tournament size. So, if the tournament size is 2, we will take 2 members and then we will have a competition with them; whoever is winner a copy of the winner will be placed in the mating pool right. Then we will have to do crossover; so this is for crossover.

So, we will have to do N P by 2 crossovers that is why if you remember; for all other materialistic techniques we were taking 5 population members, over here we had taken 6 because we wanted this N P by 2 to be an integer. So, N P usually is taken as an even number right. So, we will have to do N P by 2 crossovers; N P by 2 crossover because for each crossover we will get two population; two offsprings. So, if we do N P by 2 crossovers, we will get N P offsprings right. So, in this case we need to randomly choose two parents and we need to check for this condition whether crossover is to be done or not.

So, if crossover is to be done right then we will generate two offsprings using SBX crossover. So, SBX crossover if you remember we need to generate a random number for each variable and calculate this beta value for each variable; depending upon the random number we will use the appropriate expression and then we will generate the two offsprings right. So, this here it is x 1, x 2 are the parents or to be consistent with whatever we have discussed; we can have it as P a prime and P b prime right and this is O a and O b right.

So, most books would refer it as x 1 and x 2 we wanted to avoid that so that it does not confuse with the decision variables which we typically call it as x 1 and x 2 right. So, this is P a prime and P b prime right; so we generate the offsprings. Once we have generated the

offsprings we need to bound them; if this condition had failed that the if the random number which we selected happened to be greater than the crossover probability or equal to crossover probability, we do not need to perform this SBX crossover, but we need to merely copy the selected parents as offspring right.

So, no matter whether we are here or whether we are here; we will get N P solutions at the end of this crossover operation right; so this is our crossover. In this case, we are discussing SBX crossover there are various other types of crossover too. So, after this we need to perform mutation. So, mutation is to be performed for every offspring right. So, the number of offspring is N P because this loop ran for N P by 2 times and each time we got two solutions; so we have N P offsprings.

So, for each offspring well have to see whether we need to do mutation or not. So, for every time we need to generate a random number and check for this condition; this p m is mutation probability again user defined parameter. If r is less than p m, then we need to perform polynomial mutation of the i th offspring; if this condition is not satisfied we do not need to do any change in the i th offspring. So, we say that there is no need to mutate it right, but if we perform polynomial mutation we need to bound the offspring because there is no guarantee that polynomial mutation will give the solution within the lower and upper bounds right.

So, if it is violating we need to bring it back to the bounds; for performing polynomial mutation we need to generate a random number for every decision variable. And depending upon the value of random number we will have to either use the first part or the second part of the expression to determine the value of this del. Once we have determined the value of del, we will have to use this expression that the new offspring is equal to the old offspring plus upper bound minus lower bound of that particular variable into del of that particular variable.

For simplicity we are not adding this j over here right, but for each variable that has to be implemented right. So, this is upper bound of j th variable lower bound of the j th variable; this will give the j th variables mutate at offspring this is our polynomial mutation right; once we have completed crossover and polynomial mutation. So, we have N P offsprings, but if you

see nowhere we have calculated in this section the fitness right because the offerings which we generated over here can get changed in mutation.

So, that is why we did not evaluate fitness anywhere; now that all offsprings are determined we will evaluate the fitness of the offspring right. So, once we evaluate the fitness of offspring, we have this initial population and we have the offspring O which we generate over here in polynomial mutation right. So, we combine the population P and the offspring and perform this mu plus lambda survivor strategy. In mu plus lambda survivor strategy, we are not doing anything, but selecting the best N P solutions. So, this is the pseudo code of real coded GA.

So, if we see this part constitutes the generation. So, crossover and polynomial mutation helps in generation of new solution, tournament selection directly does not generate any new solution it helps us to get the mating pool which will be used to generate new solutions, but tournament selection itself does not give us new solutions right. So, the new solutions are generated either in crossover or the mutation and over here the survivor strategies survival of fittest right. So, if we calculate the functional evaluations; we will have N P functional evaluations here before we begin even begin the iterations right the iterations start over here.

We have N P population members will have to determine their fitness because we will require the fitness in the tournament selection. So, we will have to evaluate their fitness. So, we definitely require N P functional evaluations over here right and over here we again have N P offsprings right. So, we do N P by 2 crossover so, we will have N P offsprings and those offsprings may or may not get changed in the mutation phase right so, but at the maximum we will have N P new solutions.

So, here if you see we are saying maximum number of functional evaluations, it is not possible to say exactly how many fitness function evaluation will be required, but the maximum number of fitness function evaluation is N P. So, why we are saying here that it is not possible to exactly specify the actual number of functional evaluation is let us say we have 6 parents right. So, let us say P 1 prime, P 2 prime, P 3 prime, P 4 prime, P 5 prime and P 6 prime right.

Let us assume that for in one of the crossover we had selected P 4 prime and P 6 prime right and this condition happened that the random number which we are chose over here was actually greater than P c we did not have to perform SBX crossover. So, our new offspring 4 and new offspring 6 is nothing, but P 4 prime and P 6 prime right.

So, and the fitness function of P 4 prime and P 6 prime is already known right because in tournament selection; we merely selected the members from population right and they are being copied. So, we already have their fitness function value right and let us say when this offspring 4 was undergoing polynomial mutation right and it failed to meet this condition r less than p m right.

So, then there is no change in the offspring; P 4 prime did not get to undergo crossover right, it did not get to undergo mutation also right. So, it is nothing, but the same initial parent which was right whose fitness function is already known. So, there is no point of evaluating its fitness function again right. So, that is why we do not know how many such cases will happen because, it depends upon the random number which we generate at various instances right.

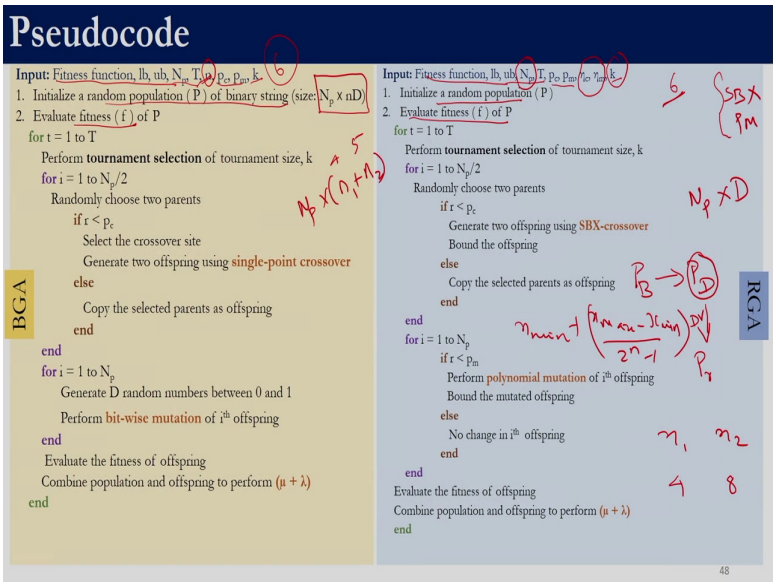
So, that is why we say that we cannot exactly specify how many functional evaluations will be required right the maximum that would be required is N P. Let us say this O 4 and O 6 are nothing, but the parent population itself right. So, whatever was the solution in the population came in the offspring also right and we can still go and estimate their fitness function right, but we will get the same value. So, we are unnecessarily spending functional evaluations to compute a value which is already known right ah. Otherwise even if we compute it there is nothing wrong in it as such like the answer should not change but it is waste of computational effort right.

So, that is why we say in the worst case N P functional evaluations will be generated assuming that every solution is actually undergoing crossover or mutation and is getting changed right. So, to consolidate in one iteration we will have N P functional evaluations that is the maximum right it can be less than that. So, for T iterations we have N P into T and

additionally we had N P functional evaluations over here right. So, if you specify N P is equal to 5 and T is equal to 10 right.

We can only estimate what would be the maximum number of functional evaluations; it is not possible to specify what would be the actual number of functional evaluations that would depend upon the various conditions that would be either satisfied or not right. So, now that we have looked into binary coded GA as well as real coded GA, let us just quickly have a look at their pseudo codes to see what are the major differences between binary coded GA and real coded GA right.

(Refer Slide Time: 59:17)



So, in binary coded GA again here the assumption is that for real coded GA; we are taking SBX crossover operator and polynomial mutation for mutation right. So, here if we see these three details belong to the problem; the fitness function lower bound upper bound, so that is

common to both of them. In this case we need to in binary coded GA we need to specify the population size the termination criteria; even in real coded GA we require them. And then we require crossover and mutation probability in binary coded GA as well as in real coded GA.

And this k denotes the number of solutions that will participate in a given tournament; in this case also we require that k right. So, here in binary coded GA since the real variables are represented by binary strings; we need to decide the length of the string right. So, here we have just taken N assuming that for all the decision variables we have a equal string length, but that need not be the case right.

If you have two variables x 1 and x 2 right; we can choose to represent x 1 with the 4 bit string and x 2 with the 8 bit string right that is perfectly fine right. So, in that case instead of n we will have to give n 1 as well as n 2 right. So, that is additionally required in binary coded GA which is not required in real coded GA right and in real coded GA; if we are going to use this SBX and polynomial mutation; we will have to provide these two additional parameters which denote the distribution index for crossover and the distribution index for polynomial mutation right.

So, here if we see apart from the problem definition 6 parameters the user has to be this way assuming this k is binary tournament selection right. So, again over here 1, 2, 3, 4, 5, 6 again here we have 6 parameters that the user has to specify. In this case this n need not be a constant value for different variables we can have different string lengths right.

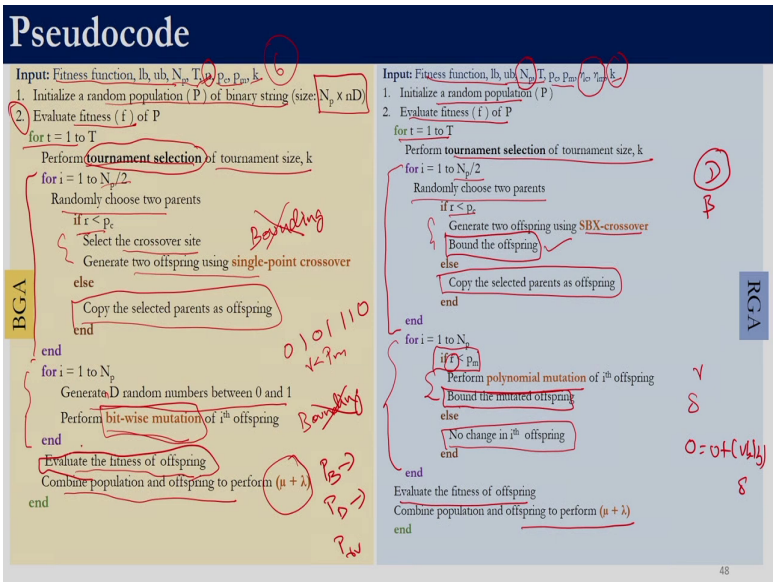
So, that is the difference as far as input is concerned right. The first step in binary coded GA is to initialize a random population of binary string. Here we need to initialize a random population within the bounds lower and upper bounds; it may not be binary strings right. So, here the length of the population is let us say if we take N P, in this case in real coded GA the size of the population matrix will be N P cross D where D is the dimension of the problem; all the real variables will be kept as real variables itself will not be converting them into binary variables whereas, in binary coded GA the size of the population matrix will be N P cross n D.

Because there are D real decision variables and each of those real decision variable is represented by a n bit string right. So, it is N P cross n D; otherwise it can be N P cross n 1 plus n 2. So, for variable 1; if we take 4 bit string and variable 2 if we take 5 bit string; then it will be N P into 9. So, there will be 9 columns and N P rows that is the difference in the initialization of binary coded GA and real coded GA right and then to evaluate fitness we need to convert the binary population right.

So, if I call that as P B that binary population we need to find out the equivalent decimal value right let us say that is P D or this is also known as decoded value and then we need to actually find out the values in terms of the actual variables using that expression x min plus x max minus x min into divided by 2 power n minus 1 into the decoded value. After finding the real values that has to be plugged into the objective function to find the fitness right. So, here we can directly plug those variables into the fitness function and determine the fitness.

Over here we need to first convert the binary strings into their decimal equivalents and then use that appropriate expression to find out the actual values only with those values we will be able to find the fitness function. So, that is a difference while evaluating the fitness in binary coded GA and the real coded GA.

(Refer Slide Time: 63:40)



So, then in both the cases we need to have the termination look for t is equal to 1 to T right. So, in both cases we are assuming that specified a maximum number of iterations; we need to perform the tournament selection, crossover and mutation for T times capital T times right.

So, that is identical in both of them and then here in both of them we need to perform a tournament selection of size k right. So, that is also identical in binary coded GA and real coded GA, there is no difference right and then the crossover phase begins. So, in both the cases we will have a loop for I equal to 1 to N P by 2 right because, we do N P by 2 crossover operations because for each crossover operation; we will get two offsprings right. So, we need to perform N P by 2 crossovers in both cases it is the same.

We need to randomly choose two parents which is identical in both of them, we need to again generate a random number if it is less than crossover probability right, we need to perform the

crossover right else we just copy the selected parents as offspring that is identical in both the cases right.

Whereas, if we happen to do a crossover in binary coded GA we need to again randomly select a crossover site and then generate two offsprings; that is what we do in single point crossover right; whereas, in SBX crossover over here we need to generate D random numbers, we need to calculate the beta value depending upon the value of the random numbers that we have selected; we need to calculate beta for each variable and then we need to use the appropriate expression to generate both the offsprings right.

One major difference in real coded and binary coded GA is there is no bounding right. So, in at the end of offspring generation; we do not have any bounding operation whereas, we need to bound the solutions in SBX crossover. The offspring which we get from SBX crossover is not guaranteed to be in the bounds; so we need to bound the offspring and then we have mutation over here and over here right.

So, the mutation is slightly different in binary coded GA and real coded GA right. So, in binary coded GA what we do is we generate D random numbers between 0 and 1. So, this and we implemented bitwise mutation. So, for example, let us say our string was 0 1 0 1 1 1 0 right. So, we generate random numbers for each of this decision variable right and wherever the random number is less than the mutation probability right; we convert that 1 to 0 or 0 to 1 right.

So, when here we have just written perform bit wise mutation, but here we also need to check for that we generate a random number and if the random number is less than mutation probability; then we need to perform the mutation right. So, that is how we implement mutation in binary coded GA; whereas, in real coded GA we just generate one random number initially right. If that is less than mutation probability, then we perform the polynomial mutation for each variable we do not check this condition right.

So, there is a small difference, but we need to be careful about it; in binary coded GA we generate a random number for each bit of the string whereas, in real coded GA, we generate

only one random number for a solution. If the random number happens to be less than mutation probability, we perform polynomial mutation else we do not perform polynomial mutation and the offspring will remain as such ok. So, here when we perform polynomial mutation while performing polynomial mutation; we need to generate a random number for each decision variable right.

And for each decision variable well have to calculate that delta value and then for that delta value; we will have to calculate the new offspring right, the new offspring is old offspring plus u b minus l b into delta. So, we need to use that expression to calculate the new offspring and then we need to bound the mutated offspring whereas, that bounding is not over here right; no bounding is involved whereas, in polynomial mutation, we need to bound it and then that would complete the mutation process; once the mutation process is complete we need to evaluate the fitness of offspring.

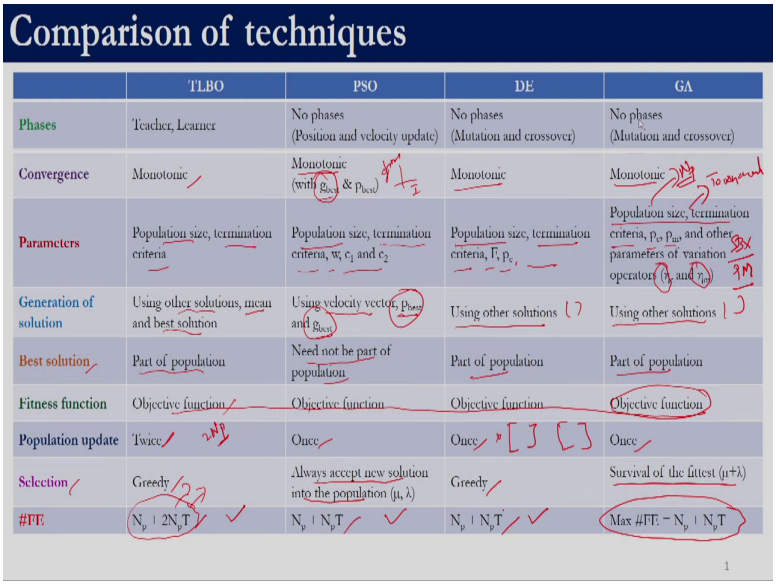
So, since here the offspring would be binary right. So, since it is binary we need to first convert it into its decimal equivalent and then find out the appropriate real variable values and then plug into the fitness function. So, that is the same as we implemented in this step 2 right; the step 2 and this evaluation of fitness of offspring; we will have to do 2 to 3 operations to be able to determine the fitness of the offspring in binary coded GA; whereas, in real coded GA we can directly plug the values into the fitness function and determine the fitness function value.

And the final step is identical in both of them that we need to combine the population; remember this is not the parent population and the offspring and we need to perform a mu plus lambda strategy which is the same in real coded GA. The major difference between real coded GA and binary coded GA is in the encoding of the decision variables right that is one big difference and then when we are performing this mutation right.

So, here if there are D decision variables; we need to generate n into D random numbers for each of this bit we will have to generate a random number and then we will have to use that. So, now, that we have looked into GA. So, as usual let us compare all the 4 algorithms which we have studied so far. So, here we are only accounting for real coded genetic algorithms. So,

we will be basically comparing teaching learning based optimization, particle swarm optimization, differential evolution and genetic algorithm right.

(Refer Slide Time: 69:28)



So, the first thing that we are looking at is the number of phases that is involved. So, in TLBO; if you remember we have two phases, the teacher phase and the learner phase. In PSO, as such we did not have any phase right there was a position update and then there was a velocity update, but there were no phases as such right. Similarly, in DE there were no phases; we had a mutation operation and the crossover operation. Number in DE we first perform mutation and then perform crossover whereas, in GA we first perform crossover right and then we perform mutation and here also there are no phases.

When we say there are no phases; it means that there is no intermediate update of the population. So, if we talk about convergence as you know TLBO exhibits monotonic

convergence because it has a greedy selection mechanism. PSO always accepts the new position right, but if we are plotting only the g best versus the iteration right. So, iteration on the x axis and g best and the y axis; then the convergence curve will be monotonic similarly for D it will be monotonic because again we are employing a greedy selection mechanism and in GA, we are doing a mu plus lambda right.

So, we are combining all the solutions, sorting those solutions and taking that best N P solutions. So, here also the convergence will be monotonic. So, if we talk about the parameters TLBO we required only two parameters population size and termination criteria. For this course, we have always taken the termination criteria to be the number of iterations that has to be performed. In particle sum optimization, we need to give the population size, the termination criteria; w which is the inertia weight, c 1 and c 2 which are the acceleration coefficient.

So, all these need to be fixed by the user; in DE we had to specify population size the termination criteria, the scaling factor F and the crossover probability. Coming to GA, again population size and termination criteria is common; in addition we had to provide the crossover probability, mutation probability. And if we are choosing SBX crossover and polynomial mutation; then we need to also provide this eta c and eta m; remember here the choices also we are making.

So, for example, here we take this SBX crossover and polynomial mutations. So, that is also a choice that we are employing right; there are other operators which can also be used. Similarly, here we also are selecting tournament selection. So, instead of this tournament selection someone else could choose to work with a row level or some other mechanism. Again, here we had something called as pool size right; so pool size we fixed it as N P, but it is not necessary that it be fixed at N P.

When we were conducting tournament in GA; we fix that the number of solutions competing in one tournament will be 2, but that need not be the case right; we can even take 3 or 4 solutions and we can perform tournament selection. Similarly, in DE if you remember we also

had those 5 strategies DE slash rank slash 1, DE slash rank slash 2 and those 5 strategies. So, we will be selecting any one of them. So, that is also a choice that the user has to make.

So, if we talk about generation of new solutions; so in TLBO we use other solutions in the population to generate a particular solution. And in teacher phase we also use a mean of the population and the best solution in the population whereas, in PSO we use the velocity vector, as well as the personal best and the g best. So, if you recollect while updating the velocity; we had v is equal to w into v plus the difference between p best and the excise solution and the difference between g best and the excise solution.

So, that is why we have written over here p best and g best are used to generate a new solution. In DE, we had to randomly select other available solutions same thing with GA. So, for example, when we are doing crossover we will be randomly selecting any other solution right. In GA and DE, we do not need to locate what is the best solution that is not required for generating new solutions; whereas, in TLBO as well as in particle swarm optimization that was required. If we talk about the location of best solution; so because TLBO employs a greedy selection procedure; the best solution is always part of population which is not the case in PSO right.

So, for PSO the best solution may not even be in the population; it could be in the g best right whereas, for DE and GA, the DE employs a greedy selection mechanism right whereas, GA employs a mu plus lambda strategy. So, that is why we will have the best solution being part of the population.

So, in all 4 algorithms; the fitness function is nothing, but the objective function for an unconstrained optimization problem. So, far we are only discussing about unconstrained optimization problem, we have chosen to include this over here because the next algorithm a b c which we will see there the objective function does not directly correspond to the fitness function value right.

So, for these 4 algorithms fitness function for an unconstrained optimization is nothing, but the objective function of the problem. So, here if we talk about population update; in TLBO we

would be updating the population twice right at the end of teacher phase and learner phase that we can do for every solution. So, for every solution at the end of teacher phase; the solution can get updated and at the end of learner phase the population can get updated.

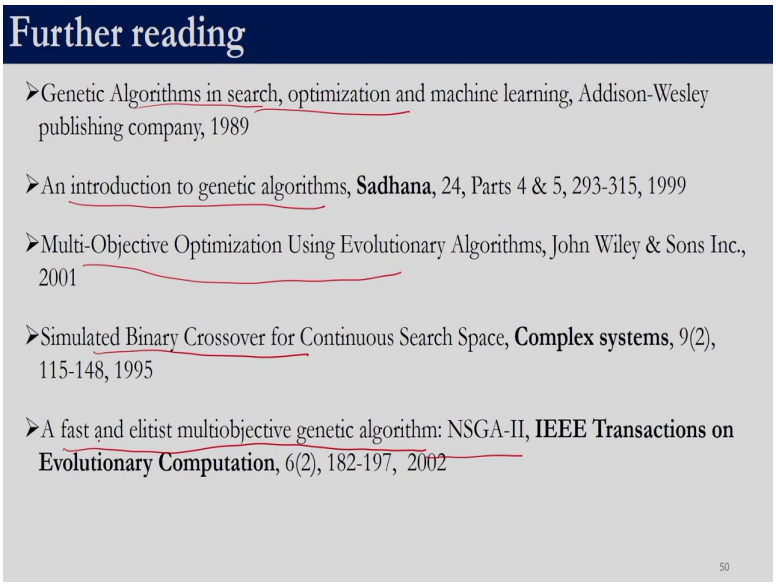
So, this is technically two N P, where N P is the size of the population whereas, in PSO DE and GA we will be updating the population only once. For example, in GA once we have completed crossover and mutation we will be updating the population. Same thing in DE before we complete one particular iteration; we would have generated all the solutions right and then we perform a greedy selection right. So, we would have this parents let say there are 10 members and over here there are 10 members.

So, these 10 are our target vectors, these 10 are our trial vectors; so we will compare 1 to 1 and whichever is better that will go into the population. So, this update happens only once. So, selection strategy we have discussed it multiple times even while discussing this slide right. So, here we have greedy selection mechanism in TLBO, as well as in DE whereas, in PSO will always accept the new solution into the population; whereas, in GA we will be employing this mu plus lambda strategy right.

So, the old solutions we will combine that with the new solutions and among these two N P solutions; we will take the best N P solutions. Talking about fitness function evaluation we have N P plus 2 N P t fitness function evaluation; here N P is the population size and t is the number of iterations right. So, we can exactly specify how many fitness function evaluation will be required in all these three algorithms right in TLBO PSO and DE right whereas, in GA we can only predict what would be the maximum number of fitness function evaluation it can be less than this right.

So, in PSO and DE it is N P plus N P T right whereas, in GA the maximum is N P plus N P T; it can be even be less than this.

(Refer Slide Time: 76:42)



So, before concluding genetic algorithm these are few references right. So, this is the classical book on genetic algorithm right; Genetic algorithms in search optimization and machine learning. This paper gives an introduction to genetic algorithm it is written by Kalyanmoy Deb. This is again the book by Professor Kalyanmoy Deb right; it is a book on multi objective optimization, but there are a couple of chapters which talk about single objective optimization.

So, all the different types of crossover is b x crossover polynomial mutation, additional types of crossover mutation both for binary coded GA real coded GA; if you are interested you can look into that book right, you will be able to find a lot of information over there. So, simulated binary crossover as we discussed over here; here we have only discussed its properties right, we have not looked into its derivation stuff like that. So, if you are interested you can look into this paper right.

So, this paper in this paper they had proposed the simulated binary crossover. So, you can have a look into it and then for multi objective optimization; this is one of the most site at paper NSGA II right. So, non dominated sorting genetic algorithm II; if you are interested in multi objective optimization and would like to apply genetic algorithm, then you can look into this particular paper. With that will conclude this session.

Thank you.