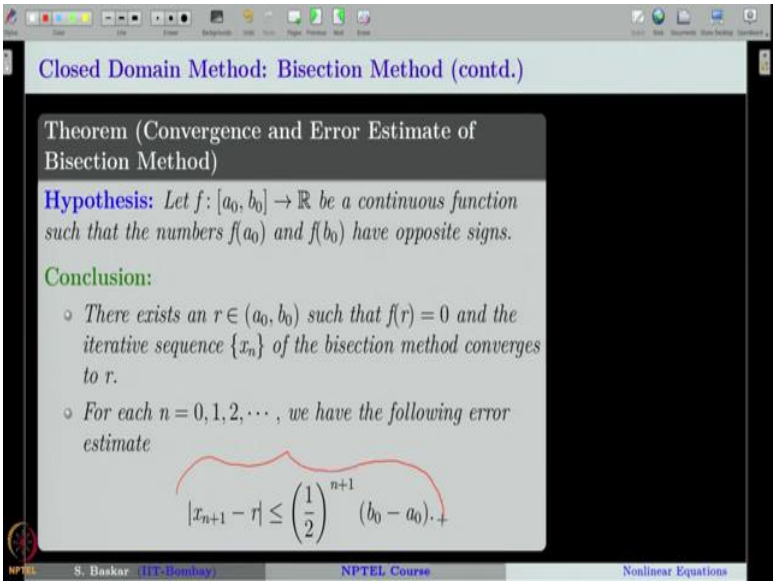
Numerical Analysis
Prof. S. Baskar
Department of Mathematics
Indian Institute of Technology – Bombay

Lecture – 29
Nonlinear Equations: Implementation of Bisection Method

Hi, in the last lecture we have studied the convergence theorem for bisection method. In this lecture we will continue our discussion on the convergence of bisection method. We will prove an interesting corollary of the convergence theorem of bisection method and then we will also see how to implement the bisection method as a python code.

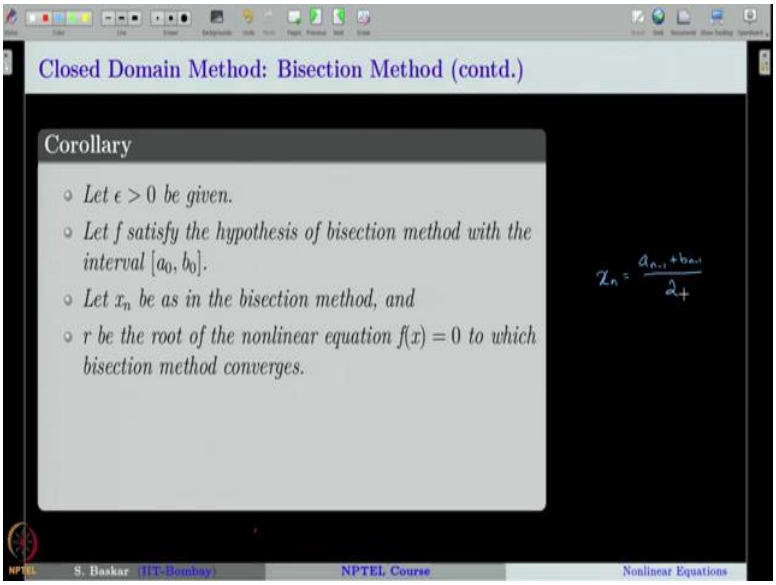
(Refer Slide Time: 00:46)



Recall that in the last class we have proved this convergence theorem, where we have assumed a function f which is defined on a closed and bounded interval $[a_0, b_0]$ and the function f is continuous on this interval. And the points a_0 and b_0 are chosen in such a way that the sign of f at a_0 and b_0 are opposite. Then we have seen that the sequence generated by the bisection method converges to a root in the interval (a_0, b_0).

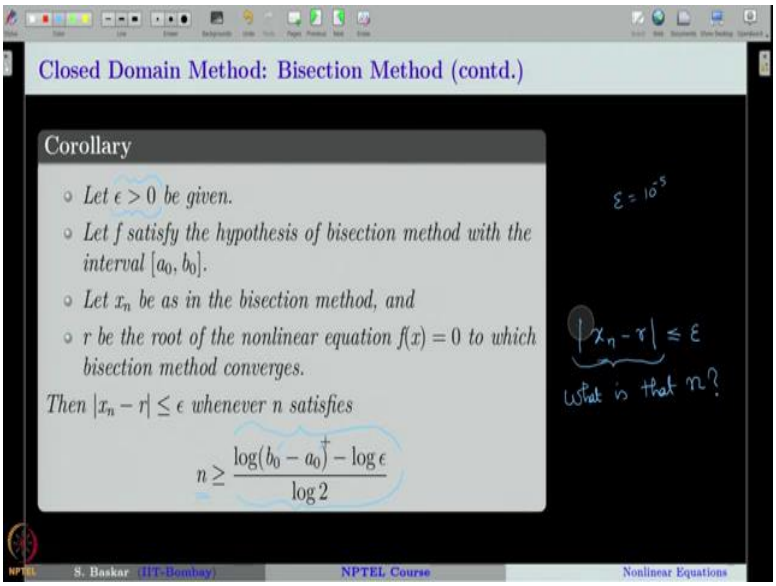
And further we also proved a nice estimate for the error involved in the bisection method at every iteration. The error estimate is given like this. Now, from this error estimate, we can also derive an interesting formula. That is what we are going to see in this lecture.

(Refer Slide Time: 01:56)



Let us state this formula in the form of a corollary. Let us take a real number $\epsilon > 0$ and let us assume all the hypothesis that we have assumed in our convergence theorem and also let x_n be as in the bisection method. That is x_n's are given by $x_n = \frac{a_{n-1}+b_{n-1}}{2}$. That is how we defined x_n's and r is the root of the non-linear equation $f(x) = 0$, for which the bisection method is converging.

(Refer Slide Time: 02:47)



So, these are given to us then what the corollary says is that if you take the integer n greater than or equal to this number. Then surely $|x_n - r|$ will be less than or equal to the number that we have chosen that is ϵ. How can we use this corollary? Well, suppose you say that I want my approximate solution x_n to be very close to the exact solution. Remember we do not know what is the exact solution?

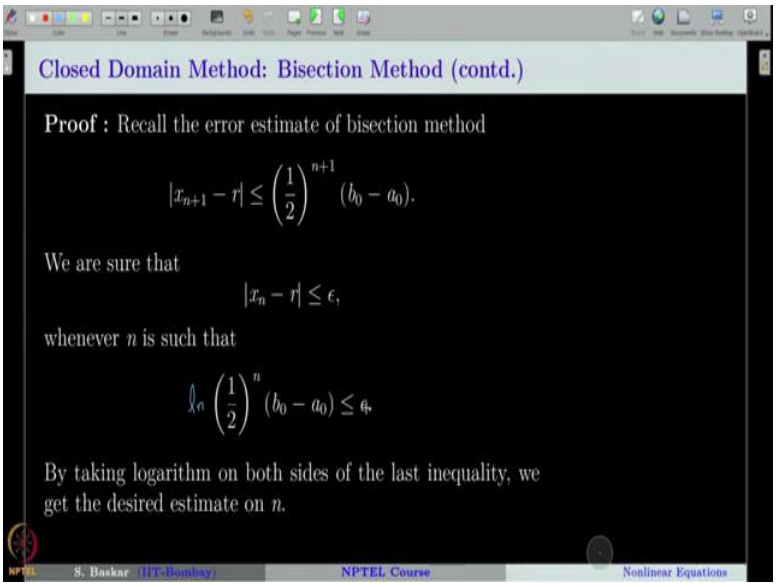
We only know what is x_n because we are computing it through bisection method. Now, what we are demanding is that, I want my approximate solution, x_n to be very close to r. How close it is? Well, it should be less than or equal to the ϵ that I have given. See ϵ is something which we take, say suppose ϵ you take as 10^{-5} or something.

Then I want to compute the iterative sequence of bisection method up to some n such that $|x_n - r| < \epsilon$. If so, what is that n? That is how many iterations should I compute in order to make my error, this is the error you see, in order to make my error less than or equal to ϵ. If you ask this question then our corollary says well you have to at most compute this many iterations. What is that?

That n should be an integer greater than or equal to this number. You see the interesting part of this number is all the quantities involved in this expression are known to us b_0 and a_0 are given as the input to our method. ϵ is something which we have chosen generally called the tolerance parameter and of course, log 2 is a constant you can always compute.

So, you can in fact tell how many iterations are needed in order to make sure that the error involved in x_n when compared to the exact root, is less than ϵ. That is the interesting part of bisection method.

(Refer Slide Time: 05:39)

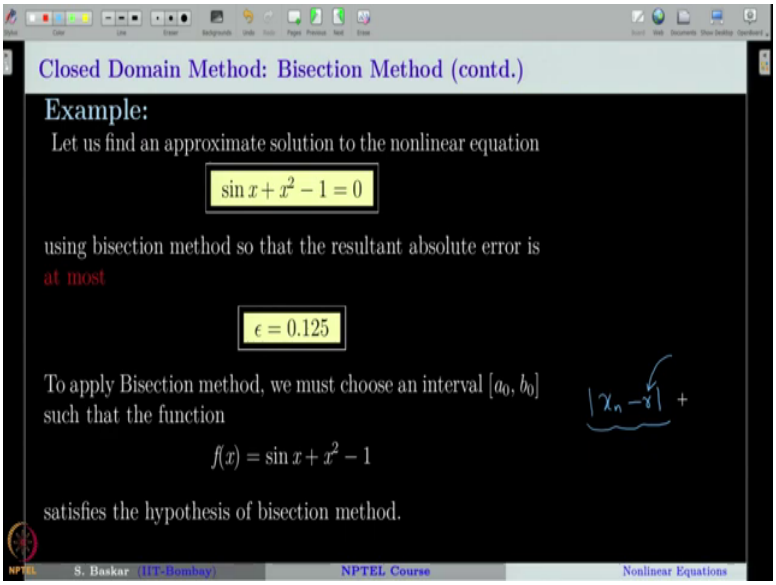


Let us prove this corollary. The proof of this corollary is not very difficult. You just take this estimate which we have proved in our convergence theorem. Now, what we want, we want $|x_{n+1} - r| \leq \epsilon$. Now, we can achieve this by imposing the condition that the right-hand side is

less than or equal to ϵ. That is all is the idea. To make sure that $|x_n - r| \leq \epsilon$, we will impose that the right-hand side is less than or equal to ϵ.

Now what we want? We want this *n*. That is all we want. That can be obtained by taking logarithm on both sides. Now, I leave it to you to derive the expression by taking log on both sides and then finding the *n* from that expression. I will leave it to you as an exercise.

(Refer Slide Time: 06:54)

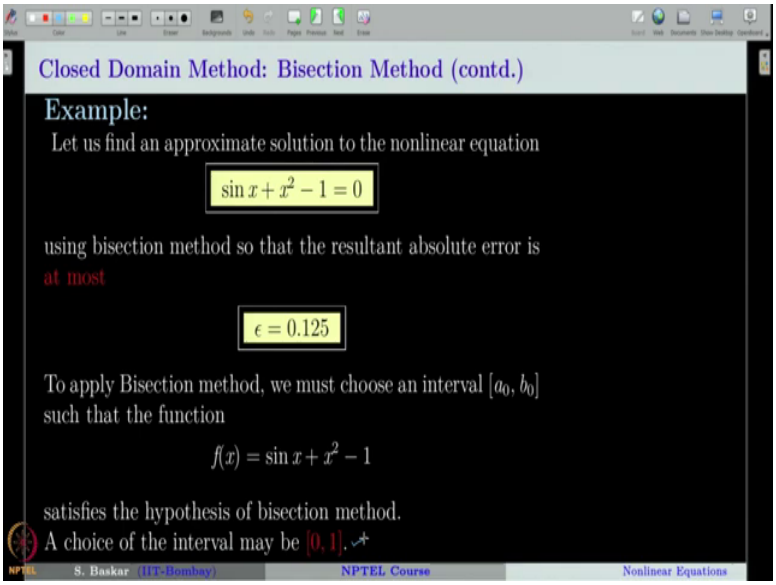


Now, let us take an example let us consider this equation $\sin x + x^2 - 1 = 0$. Now, I want to use bisection method to approximately find a root of this equation. Now, what I want? I will tell you that I want my approximate solution x_n to be such that $|x_n - r| \leq \epsilon$ and that ϵ is taken to be 0.125. This is what I want. Now, my question is how many iterations should I compute?

Now, you can use the above formula to find the number of iterations. One way is that you can just go on computing the iterations and at every iteration you check the number $x_n - r$ but for that you need to know the root *r*. But we actually, do not know the root. That is the problem, so, you just cannot check the condition that we need but we can use our formula to actually, give *n* without going to the explicit computation of the iterations.

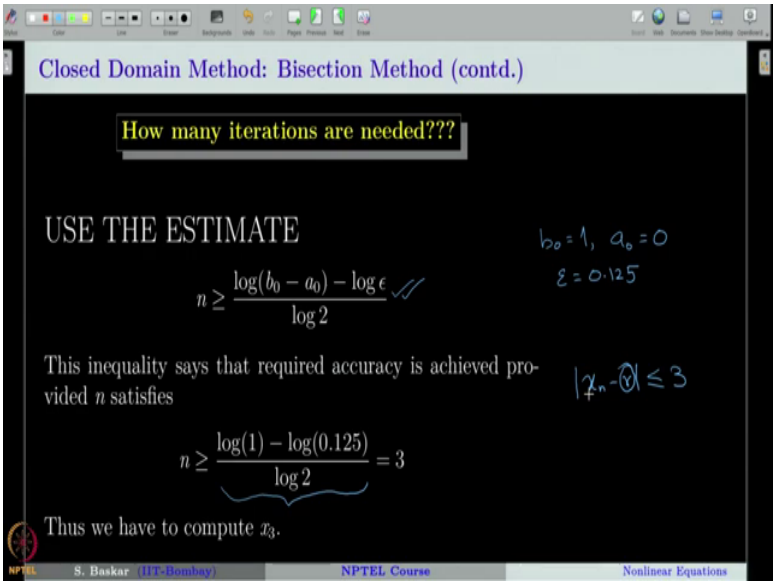
Now, for that remember, first step is to choose the interval $[a_0, b_0]$ in such a way that the function $f(x) = sin\, x + x^2 - 1$ takes opposite signs at a_0 and b_0.

(Refer Slide Time: 08:47)



Let me take the interval as [0, 1]. You can check that $f(0)$ is negative and $f(1)$ is positive. Therefore, the interval [0, 1] can be a choice for our initial interval.

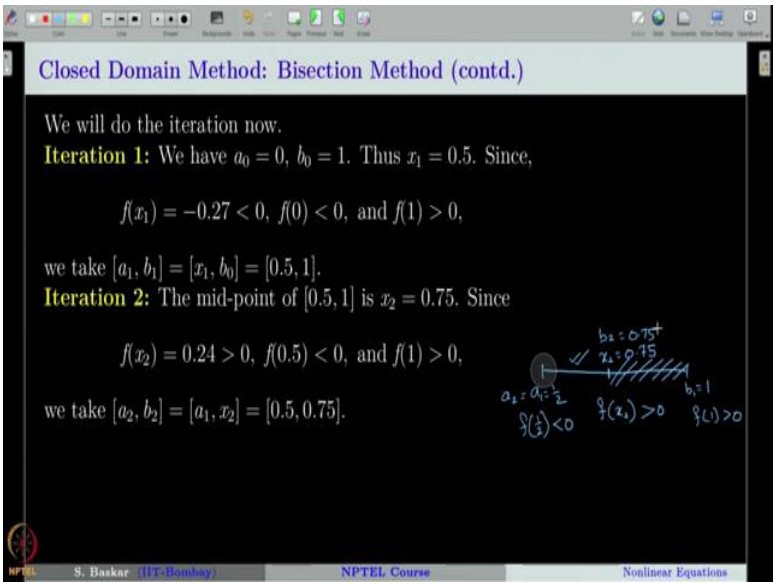
(Refer Slide Time: 09:08)



Now, coming to the question of how many iterations are needed? Well, we will use this formula that we have derived just now with $b_0 = 1$, $a_0 = 0$, $\epsilon = 0.125$. For that you can see that the value of the expression is nearly three and therefore, you may have to compute at most three iterations, remember it is at most three iterations. Why? Because we are using an upper bound that is the worst error that we can have in the computed root.

That is what the estimate says therefore, you need to compute at most three iterations in order to achieve an accuracy such that $|x_n - r| \leq \epsilon$. In this you see, we do not know what is r? Without knowing, what is r we are able to conclude this.

(Refer Slide Time: 10:30)



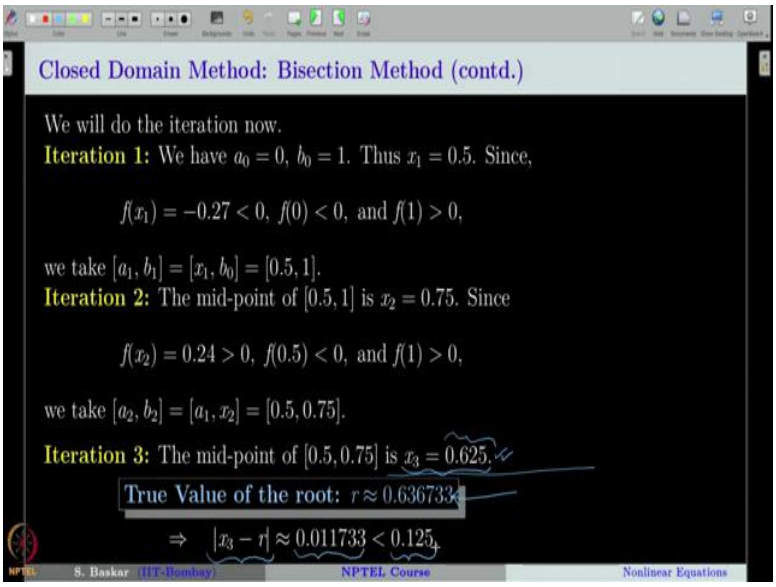
Let us go ahead and use bisection method for the given equation and compute three iterations and see how it goes. Well, for the first iteration, we have to give our interval a_0 as 0 and b_0 as 1. And now, for x_1 we have to use the formula which is nothing but the midpoint of the interval [0, 1] that is nothing but 1/2. Now, we have to see whether the function *f* changes sign in this interval or in this interval.

For that you have to find $f(0)$ you can see already it is less than 0 and $f(1)$ is greater than 0. Now, $f(1/2)$that happens to be – 0.27. That is what we have computed here. Therefore, this is also less than 0. That shows that the root is not going to lie in this interval. That is the root that the bisection method is trying to capture. That is not going to lie in this interval but it will be lying in this interval that is what we can see from this.

Therefore, we will discard the interval [0, ½] and we will only take the interval [½, 1] and then we will proceed to the next iteration and see how the next iteration is going to go. For the next iteration you have $a_1 = ½$ and $b_1 = 1$. Again, you have to take the midpoint of this which is nothing but x_2 = 0.75 and what is the value of $f(x_2)$ and that is 0.24 which is greater than 0.

Remember, $f(½)$ we have seen in the previous iteration that it is greater than 0. And similarly, $f(1)$ is also greater than 0. Therefore, you have to discard this interval and take the interval [½, 0.75]. Therefore, a_2 is taken as 0.5 and b_2 is taken as 0.75 and that completes our second iteration. Remember our analysis says to achieve the required accuracy you have to do at most three iterations. So, we will do three iterations.

(Refer Slide Time: 13:37)



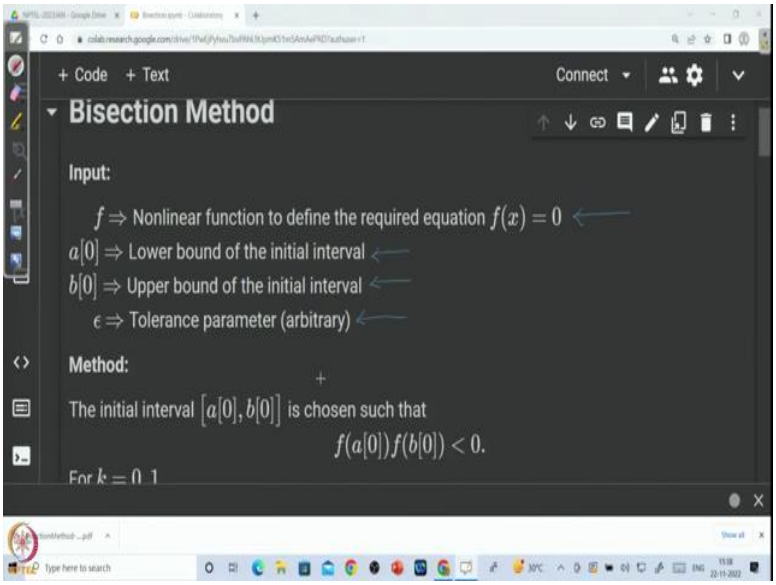
Let us go to the third iteration. You can see that the midpoint of the interval [0.5, 0.75] is given by 0.625 and that is it. We will stop here because we want the solution with accuracy, only 0.125 and our analysis says that you just have to carry three iterations. We have done the third iteration and take that value as the root. The problem ends here but let us cross check whether what our analysis says and what is the reality.

Our analysis says that you have to stop at the third iteration. Let us see what is the reality? The true value of the root is actually, approximately given by this value 0.636733 and what we got is 0.625. Therefore, the error involved in x_3 when compared to r is something like 0.01 which is well below what we can tolerate. We told, that we can tolerate ϵ which is equal to 0.125, for that our analysis said that you do at most three iterations.

We did three iterations and we saw that we got the accuracy in fact more we got that does not matter. We want at least this much accuracy. For that at most, we need to do three iterations. This is what the analysis says and the reality is well respecting the analysis that we did. So, with that we finish our discussion on bisection method. We will develop a python code for bisection method. Well, let us go to the collab to develop our python code.

Let us quickly recall what are all the inputs? And how the method goes? And what should be the output?

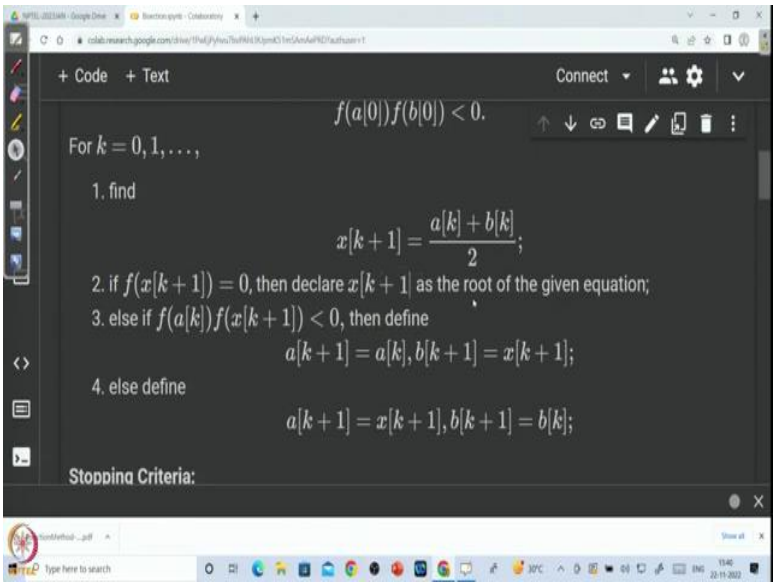
(Refer Slide Time: 15:56)



For that let us first see what are all the inputs needed for the code. Well, we need to give the function *f* that defines our equation $f(x) = 0$ and then we have to give the initial interval. For that we have to give a_0 and b_0. Also, we will take the tolerance parameter ϵ, that we will take as per our wish. There is no special property for ϵ you can choose this arbitrarily.

But it the understanding is that you will give ϵ very small that is the understanding. Then the method goes like this remember that we have to choose a_0 and b_0 such that $f(a_0)f(b_0) < 0$. In our code we are not asking the computer to do that but we are choosing such a_0 and b_0 and supplying those information as input to our code. In that way, our code is very simple.

(Refer Slide Time: 17:06)



The method goes like this for every k = 0, 1, 2 and so on, we know what is the interval in the previous iteration. You just take k = 0 and understand this. You know what is a[0] and b[0]

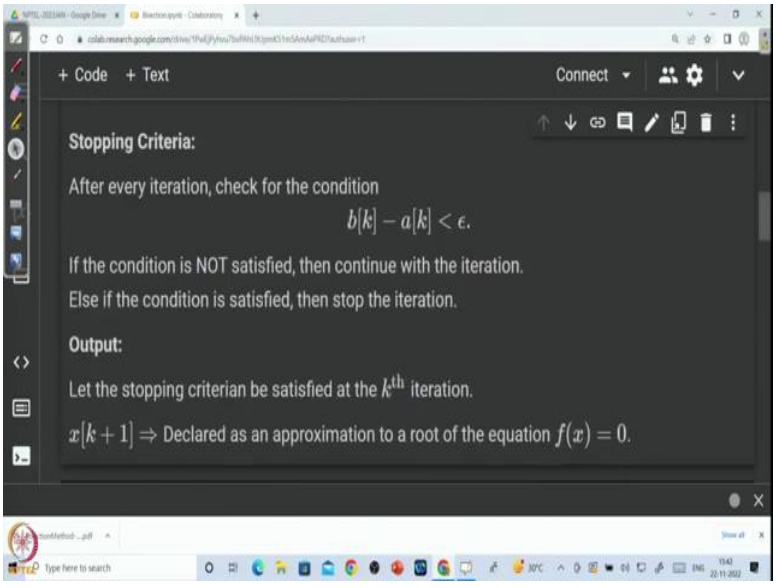
because that is given as an input and then you will compute x[1]. That will be the midpoint of the interval, a[0] and b[0]. Similarly, any k = 0 or 1 or 2 anything, the understanding is that we know what is a[k] and b[k].

From there we will find x[k + 1] and then we will go to check the condition that whether f(x [k+ 1]) = 0. If it is so then we will come out of the iteration declare that x[k+ 1] is the root of the given equation. Remember mostly, this condition will not be satisfied because on a computer, some rounding error will be involved, even if that quantity is going to be exactly equal to 0 mathematically, it may not be so on the computed quantity.

Therefore, mostly this condition will not be satisfied in your code but just for the sake of completeness, we will just check this. If this condition is not satisfied then we will go to check whether f(a[k]) f(x[k+ 1]) is negative. That is if there is a sign change between a[k] and x[k+ 1]. If so then we will take the interval for the next iteration as a[k + 1] = a[k] and b[k + 1] = x[k + 1].

If not suppose, if this is not satisfied then surely your root is going to sit in the second half of the interval (a[k], b[k]). Therefore, we will take the interval for the next iteration as a[k + 1] = x[k+ 1]. That is this one and b[k + 1] = b[k].

(Refer Slide Time: 19:35)



So, this is how we have built our algorithm in our previous class itself. I am just trying to recall it here. Now coming to the stopping criteria. We have to check whether b[k] – a[k] that is at every iteration. The length of the interval is less than ϵ or not. If the length of the interval is

less than ϵ then you will stop the iteration. If it is not less than ϵ then again you will go to the first step.

You increment your k value and then find the midpoint of that interval (a[k], b[k]) and then you just check for where the root lies and then this process should go on in a loop. Now finally, when the stopping criteria is satisfied then your loop has to be stopped and then declare the value x[k + 1] as an approximation to the root of the equation $f(x) = 0$. This is what we are going to do in this code. Let us see how to develop the code.

(Video Starts: 20:57) First part of our code is the input. Here we have to take our function *f* for that the format is f equal to this is something which is a part of this command and then x. Up to this is the part of the declaration of a function. Now, from here onwards, we are starting the expression of the function. We are taking the expression of our function as (x – 0.09) * (x – 0.15) * (x – 0.063).

It is a cubic polynomial you also know, what are all the roots of this polynomial. The next is, we have to give the initial condition a0 = 0. If you give a0 = 0, you can check that f of 0 is negative. And then we are taking b0 = 1 you can also check that f of 1 is positive. Therefore, there is at least 1 root lying between a0 and b0. Now, let us take the tolerance parameter ϵ is equal to 0.0001.

So, I want to continue my iteration till my computed value x[n] – the exact root r is less than epsilon. That is what I want but we have seen at least in the bisection method. It is enough to check the length of the interval. If the length of the interval goes below epsilon then surely your computed value will be close to the exact root of the equation, whatever it may with error which is less than this epsilon. Now, let us see how to code the iteration process.

Well, we have already taken b0 and a0, therefore, take the error, err as b0 – a0 and if the error is less than epsilon then we do not need to compute anything. We will just come out and say that x = (a0 + b0) / 2 is the root of the equation at least approximate root of the equation. We will say, therefore, if this error is less than epsilon that is this part is less than epsilon.

Then process will not get into the loop at all. If that number is greater than epsilon then the process will get into the loop and let us see what the loop does. It first finds the value of x which is the midpoint of a0 and b0. That is actually, the formula for the bisection method. Therefore, if you are developing a code for some other bracketing methods like we will be seeing Regula-Falsi method in the next class.

So, if you want to develop a code for Regula-Falsi method, the only line that you have to change is this line. Where you have to substitute the formula for bisection method with the formula for the Regula-Falsi method otherwise the algorithm remains the same. So, since we are developing bisection method, the formula is given by (a0+b0) / 2. Now, you see, we have to check three conditions if f of x itself is 0 then just say that the root of our equation is x.

So, for that you have to write this line, so, what the processor will do. If this condition is satisfied then it will print root is equal to whatever may be the value stored in this variable x. It will print that value and then comes to the next line. What next line says next line says break the loop whenever the processor sees break, it breaks the main loop in which it is currently running. It is now currently running in this loop.

Whatever is the body of this loop, it is given with an indent here. So, you can see there are two points. It means this line is written with an indent and therefore, it is a part of this while loop. And similarly, this, if condition is a part of this while loop and these two are written with two indents. Therefore, these two are part of this if condition. Once it comes into this, if part, it will break the loop and comes out it comes out and then simply it will go and compute this.

But then, when it comes to this, it will check whether f(x) is not equal to. 0. If f(x) is not equal to 0 then it was an approximate root otherwise, it is since you are coming from here that if condition is not satisfied. Therefore, it will not get into the. If condition it will print the residual. What is the residual? Residual error is equal to it will compute f(x). What is x? x is computed here because you came from this if condition you have directly jumped to this part.

Therefore, it will take the value of x computed there and it will supply that to the expression f given here and it will find the value of f as per the x computed here. So that will be the residual error. All this will happen if this, if condition, is satisfied. As I told you most likely this, if condition is not going to be satisfied. Almost sure that is, you can say that the probability that

the computed value f(x) is exactly equal to 0 on a computer is almost probability 0 because of the rounding error.

Therefore, it mostly it will not be satisfied, so, your control will only come to the next line which says else if, that is if this is not satisfied then what? Else if, you check this condition f(a0) * f(x) < 0. You keep the algorithm in hand and then see the program. Then you will understand f(a0) * f(x) < 0. If so then, a1 should be equal to a naught.

But I do not want to spend one more memory to store a1 because once I have a1 and b1, a0 and b0 are not going to be used at all. Therefore, I will always update my interval of the next iteration. In the same variable, a0 and b0. You can carefully observe that once an iteration is over then we will never use a0 and b0. We will have a1 and b1 and that only will be used to get a2, b2 like that it goes on.

Therefore, on a code you in order to minimize the usage of memory, you just simply store the value of a1 and b1 in the variables called a0 and b0 itself. This is just to minimize the usage of the memory. In this case, I have to take my a1 = a0 which I am now going to do whereas b1, I should take as x. Since I am going to use the same memory for a1 and b1.

So, I am storing x in b0 only, if that also does not satisfy. That is if this condition is also not going to be satisfied then I will go into the else, where b0 is retained as it is, whereas a0 is overwritten by the value x. Therefore, once the loop is done then your interval is bisected and you are getting that part of the bisected interval in which you have at least 1 root and now you go to check the length of that interval.

Remember now it is b1 – a1 theoretically, why I wrote b0 – a0 simply because I want to minimize the usage of memory. Therefore, I am not using a separate set of variables, a1 and b1. I am just using the same old variable but the value stored in these variables are now updated ones. Therefore, the error now will be surely different from this error. It will be half of that error surely.

Now, again once the error is done that is all this loop is over. How do you know that the loop is over? It is not because of this line these lines will not be executed at all. These lines are not going to indicate that your loop is over but any line which starts without an indent is the line

which is outside this loop. So, this is the first line which has no indent therefore, this is the first line that lies outside this loop.

So, when the control of the Python code executes this line then it sees that. That is the end of this line. So, it goes back to the while loop and then checks once again the condition. Now, when it checks the condition. The value in the variable err is now stored with this new value. Now, the value in this err is the one which is now newly computed. Now, it will check whether that value is greater than equal to epsilon.

If so, it will again continue to execute all these lines and it will keep on going at some stage when this interval length becomes less than epsilon. Then it breaks the loop comes out and takes the midpoint of that interval and declares that as the approximate root of our equation. And of course, it will also find the residual error at this x. Now, it will find at this x. Let us try to run this code and see what is the root that the bisection method is capturing?

If f is given by this expression and the initial guesses are a0 = 0 and b0 = 1. As I told you when you run the code for first time, it has to make some space in the server. So, it takes some time for the first run only well, it has given a space in the server and it also ran the program. The approximate route that it obtained is 0.1499 and so on. You can see that 0.15 is a root of this polynomial.

So, the bisection method, with this initial condition is converging to the root 0.15 and the residual error is something really very small. So that is very nice I am just giving you a slight variation of this program here, where I am not telling you what is the tolerance error that I want to take. Rather, I am telling you how many iterations I want to perform in the code. Then what you have to do is you can simply use the for loop.

The format of the for loop is that for k = 0 to n – 1. That is once it gets into the for loop. It takes k = 0 it performs whatever is given under the for loop. And then goes back take k = 1, does the loop then k equal to 2, does the loop like that it does still whatever n value that we have given and once it performs that many iterations it comes out and gives the answer? So, let me not explain this. You know how this main part of the program goes on.

The only difference is now the way the loop is made. Previously we made a while loop, where we have checked the error. Now, we have made a for loop, where we have specified how many iterations we want to run and then we made the loop to run that many iterations. I hope you have understood this code.

You can also run this code and see if you want to play around with different functions, you can change this expression. You can also play around with different intervals and see how the convergence is happening. **(Video Ends: 35:54)** Thank you for your attention. Thank you.