Scientific Computing Using Python

Professor. Vivek Aggarwal and Professor. Mani Mehra

Department of Mathematics

Indian Institute of Technology, Delhi

Lecture No. 10

Hello, and welcome to Scientific Computing Using Python. In the last lecture, we discussed the fixed-point methods and the bisection method for finding the roots of equations. Today, we will continue from there and discuss methods like the Regula Falsi and Secant methods. So, let's get started.

Our topic for today is Regula Falsi and Secant Method.

We will first discuss this because—what is our main purpose? Our main purpose is that we are given an equation of a certain type, and we have to find its root. To find the root, we determine an interval with the help of the Intermediate Value Theorem. From this, we come to know that the root lies somewhere between a and b. Now, we previously did this using a fixed-point method. What was the problem with the fixed-point method? We had to rewrite the equation in the form $x = g(x)$ and choose $g(x)$ such that the convergence condition, $|g'(x)| < 1$, is satisfied. Then there's the Bisection Method. In the bisection method, we were sure that we would get our solution. The only drawback was that it converged slowly—it converged gradually toward the root.

So, we have other methods which are a little faster. The first one that we will discuss is the Regula Falsi Method. The Regula Falsi Method, which is also known as the False Position Method, is a numerical technique for finding the roots of a nonlinear equation. You know, it is a nonlinear equation. It is a method that also needs two initial points to start. So, in this method as well, we will need two initial points to begin with. These initial points are denoted as x_0 and x_1.

Now, just like we show it in the graph, what we have to do is this: suppose, for starting, I take an element from this point, say a_0, and take b_0 as the initial point. Now, we know from the Intermediate Value Theorem that the root will lie somewhere between them.

So, what will we do? We have an equation, code code—which starts from this point, because this point has its coordinates. Now, if we look here, $(a_0, , f(a_0))$ and $(b_0, f(b_0))$ are its coordinates. So, what will we do? We will draw a line from this point to this point. Then, we will write the equation of this line. Now, we will see where this line cuts the x-axis—where it crosses the y axis. We observe that it crosses the x-axis here. So, what do we do? Wherever it crosses, we will choose that point. We name this point , b_1. So, from , a_0 b_0 , we obtained a new point , b_1. Now, what do we do at , b_1? We find the value of the function. It turns out to be positive—look here.

So, what do we do next? Earlier, b_0 was over here. Now, we take b_0 to here. So now, our new b_0 is here. Then, we get this new point.

Next, we draw a line from this point to the this point, then we will see where it is crossing the x-axis. We take that point. In this case, we have a point, and we observe that the value of the function at that point is negative.

So now, what has happened? Our a_0 reached here, a_0 moved closer to the root. The interval is narrowing from both ends, and the values are getting closer to the actual root. This iterative process continues, and ultimately, as we go further, we reach the root.

This method is called the Regula Falsi Method or the False Position Method, and it's also known as a bracketing method, because the bracketing interval keeps getting smaller.

The method starts with two initial values, x_0 and x_1 , and we must ensure that the function values at those points, $f(x_0).f(x_1) < 0$, have opposite signs. This indicates that a root lies between them.

So, this tells us for sure that the root will lie somewhere in the middle. Okay. And the code that we have written is that unlike the bisection method, which splits the interval into half, the Regula Falsi Method uses the line connecting $(x_0, f(x_0))$ and $(x_1, f(x_1))$ to approximate the root. The intersection of this line with the x-axis gives the next estimate for the root.

Now, we have to see how we can make it an iterative method. Because to make it an iterative method, we know that we have to put it in a certain form—typically something like: x n should be or x n+1 taken or take anything g x n-1. If we have to write it in this form only then we will be an iterative method.

So now, let us see how to make it an iterative method.

Let’s consider the formula. We know that this approach approximates the root. So, how do we find that?

Let me draw this function. Suppose I take two points on this function—one point as x_0 and another as x_1. Here, these are values, so this point corresponds to x_1, and the y-value will be $f(x_1)$. Similarly, the other point will be x_0, and the y-value will be $f(x_0)$.

We are trying to find the root of this function, and the process will continue iteratively.

Now, what do we do? With the help of these points, we draw a line connecting them. This line may go from anywhere, depending on the values, but the idea is that the line connects the two points. So what do we need to do? Basically, we need to write the equation of this line.

We know how to write the equation of a line between two points. y minus y1 because y1 means this y1 because our function y = f is one, okay so I will tell you that If y1 is there then in place of y1 I want to write f(x1). So let me write it directly so that y minus y1 is f(x1) equal to y2 minus y1 so this will mean that f(x1) equal minus f(x0) equal divided by x1 minus x0 into x minus x0. Okay, I'll take it as zero because I consider it to be that. This is the equation of line which is passing from here.

Now we have to see where the values are. So, if I want to write this y as f(x0) plus x minus x0, and this is my f(x1) equal minus f(x0) divided by x1 minus x0. Now, I have to check where our line will pass on the x-axis. So, the point where it crosses the x-axis — then what will happen? It will be zero where it passes. It will be zero for sure.

So, if I substitute this to zero, then it will give me at which point it is passing. So, if I put zero in it, from here I will have an equation. If you see, I manipulate it a little bit — see, I take it minus f(x0) and here (x1-x0) = 0 .I take it here, divided by f(x1) minus f(x0), okay, and it will become equal to x minus x0. So, from here we have to take out one.

So, from here the x that we get will become x0 minus f(x0) , x1 minus x0, divided by f(x1) minus f(x0). This comes, so it will tell us the value of that x where it crosses. So what will happen? In this way we will keep getting x and we will get the root. It means we will get it.

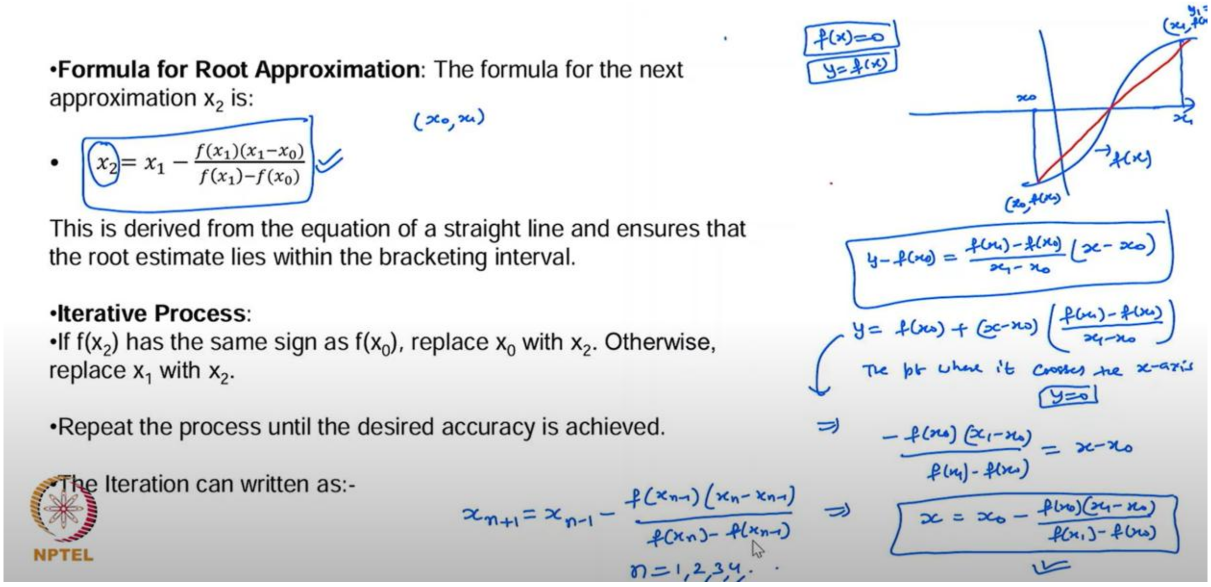
So this equation is the same like this, just the points that we have taken here, we have taken x0, x1. I have made it in iteration form. So, now what do we have? This has become — so we have to make it an iterative method. So, what is in this method? How to make it an iterative method?

So, see, to make this an iterative method, now we know x0 and x1. We will just get a new x. So, to make it an iterative method, I write it like this — x2. So x2 means I have written n or n plus 1. So, see here, there is zero and one, so I have to go to two places — n and n minus 1.

So, I will have to write n minus 1 minus $f(x_{n-1})$, x_n minus x_n minus 1, divided by $f(x_n)$ minus $f(x_{n-1})$ where n is — where do we start from? We start from zero or from one. So on, as soon as my n is done, it becomes x_0. This also becomes f_0. This becomes eq - x_0. This becomes $f(x_1)$ - x_0. And this becomes x_2.

So, what we have got is ours. Let us assume that the value that we will get in the first iteration will be x_2. This will be obtained with the help of this one. So, we first got this value, then we substituted two for the new value that came. So we have x_0, then x_1 goes, and x_2 is left. Then x_2 and x_3 are left. Then x_3 and x_4 are left. So, in this way new values will keep coming to us, and the solution we have will keep getting formed.

(Refer slide time: 14:02)



•**Formula for Root Approximation**: The formula for the next approximation x_2 is:

- $\boxed{x_2 = x_1 - \frac{f(x_1)(x_1 - x_0)}{f(x_1) - f(x_0)}}$ ✓

This is derived from the equation of a straight line and ensures that the root estimate lies within the bracketing interval.

•**Iterative Process**:
•If $f(x_2)$ has the same sign as $f(x_0)$, replace x_0 with x_2. Otherwise, replace x_1 with x_2.

•Repeat the process until the desired accuracy is achieved.

•The Iteration can written as:-

$$x_{n+1} = x_{n-1} - \frac{f(x_{n-1})(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})}$$

$n = 1, 2, 3, 4, \ldots$

(x_0, x_1)

$\boxed{f(x) = 0}$

$\boxed{y = f(x)}$

$$\boxed{y - f(x_0) = \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0)}$$

$$y = f(x_0) + (x - x_0)\left(\frac{f(x_1) - f(x_0)}{x_1 - x_0}\right)$$

The pt where it crosses the x-axis $\boxed{y = 0}$

$$\Rightarrow -\frac{f(x_0)(x_1 - x_0)}{f(x_1) - f(x_0)} = x - x_0$$

$$\Rightarrow \boxed{x = x_0 - \frac{f(x_0)(x_1 - x_0)}{f(x_1) - f(x_0)}}$$

So, what do we call this? Iterative process. So what will happen in iterative process is that you have to check how we have to take x_2. So, if x_2 is — $f(x_2)$ the same sign as $f(x_0)$ — then we have to replace x_0 with x_2, which we had shown in the previous; otherwise, replace it with this and keep repeating this process until we believe that we have accuracy or not. So, we know the accuracy. How do we do it? Till $|x_{n+1} - x_n|$ does not fall below our tolerance, then we will say that this process should continue. It should keep repeating.

So, what is the advantage of this? So, what is the advantage of this — that if you see that when we are putting the value in the root of the function, then we are using the slope of that line. Because we have to — if we look here — what is this thing? This is the slope of this line. So, we are using this slope. Wherever we are using slope, in this case, we are getting an advantage from this — that our convergence rate will increase.

And how will it increase? And by taking the derivative — what is the slope? The slope is the derivative. So, what can we get by taking the derivative? So, we will check this.

So, what is the advantage of faster convergence than bisection? So, what is it? That we knew that the bisection method gives good results, and it is sure that we will get the root. But the problem in it was that it was a very slow method. So, if we want to increase its convergence, then we need a faster convergence. So, what did we do? We went through this process and we found out that by incorporating the slope of the function that we defined, the method converges more quickly than the bisection method.

Okay, why "converge"? Because if you have seen the bisection method, it is linear — in that it gives linear convergence — but the regular-falsi is a simple method, so it comes under the category of super-linear method. And after that, its advantage is also that it is guaranteed — the root of the interval — as long as the bracketing condition is maintained, the method ensures that the root lies within the interval. So, it is sure that we will definitely get the root.

What is the disadvantage? It is slow convergence for certain functions if the function slope is very small. Okay, so if the slope of the function is very small on one of the interval boundaries, the method may converge slowly because the approximation will slowly favor the boundary with the smaller slope. Okay, so now what is happening in this is that, like we have a function f(x)= 0, we are solving it. So, we saw that in the beginning, we had taken the initial approximation of x0 and x1. These are the two initial guesses we took. So, we saw that the value of — and at some x, whatever it is — at x0 or xi , these values are very small. This is very small, right? So, these values are very small. So, what will happen in this is that the process will become very slow for that type of function for which it will give small values — dependency on small guesses. So, like all numerical methods, poor initial guesses can lead to slow convergence. So, we have to keep in mind that our initial guesses should be very close, because if it is not close, it is obvious that — what will happen in it — the mislead will happen a little in the beginning and the iteration will keep on increasing, and after that, it will take more time to have the convergence. So, in this process, we have to keep in mind that the initial guesses should be close, very close.

Now we have done this, so now there is a method — the learn method. So, now if we see, there is not much difference between the regular-falsi and the false. So, we can do it together.

So, what is the secant method? In the secant method, as the name itself suggests — secant method — we know that there is a tangent and there is a tangent.

So, if we take the tangent value or correspondingly use a numerical method to find the root of the nonlinear equation, it is closely related to Newton's method, which we will do next, but it eliminates the need for computing derivatives.

So, we know that there is any function—like in the previous case, we calculated this thing—that is, f(x1)−f(x0) divided by x1 −x0 right? So, like we did in the previous case, we know that by the mean value theorem, we can find out that here there is a slope; the slope of the tangent and the slope of this will be the same. So, we know that we can find out.

So, what are we doing in this regula falsi or secant method? In the first method, the convergence of the function at any point—for example, take the x-axis—we are approximating it with the help of this factor, and this factor is the slope of the function.

So, the secant method does this: its convergence is fast but slower than Newton's; it is superlinear and linear.

So, instead, it uses a secant line, a straight line connecting two points on the function to approximate it. The main thing is that bracketing is not needed. Now, it does not matter to us. We can do this: we choose x0 and x1, and it may be that f(x0) and f(x1) become positive and not negative.

It means that f(x0) and f(x1) are of the same sign—that is, both are positive or both are negative. Only then they will be greater than zero. So, now, what do we have to do? We should not choose the initial guess carefully.

What does it mean? That our bracketing method, which depends on the initial guess, can choose any value at any time. No derivative is required. Derivative is not required at all.

I have just explained that we have to find the derivative by a secant method. The secant method approximates the derivative, making it useful for functions where derivatives are difficult to compute. So, what can we do there? With the help of the secant method, we can solve it using its slope. It has faster convergence.

So, it typically converges faster than the bracketing methods like the regula falsi method.

So, I have seen that the regula falsi method is linear; that is, it gives convergence around one, but the secant method goes quite fast and gives convergence around 1.6, its rate of convergence is a little better. But what is the problem with this? That it may fail to converge if the initial guesses are poorly choosen.

Because now we have explained that if we satisfy this condition or not—which was the bracketing equation condition, right?—if it is satisfied in the regula falsi method, then it will be certain, so we will get a definite solution. But in the secant method, we can take initial guesses anywhere, so if we take wrong initial guesses, then it turns out that this method diverges; it does not converge at all. So, this is what we have to keep in mind.

So, this is the advantage: no derivative needed, faster convergence, simple implementation. See, its order of convergence is 1.67. It means that here we have an error at some step n+1 equal to some constant c , e raised to the power of 1.67. So, it means that it is superlinear. Its convergence is much better as compared to the bisection method and the regula falsi method.

The false method is also linear. So, if the initial guesses are close to the actual root, the method converges faster, showing superlinear convergence order compared to methods like bisection and regula falsi. Then this condition will come into play: if the guesses are close to the root, it converges well; if it is far from the actual root, it may diverge or fail to converge.

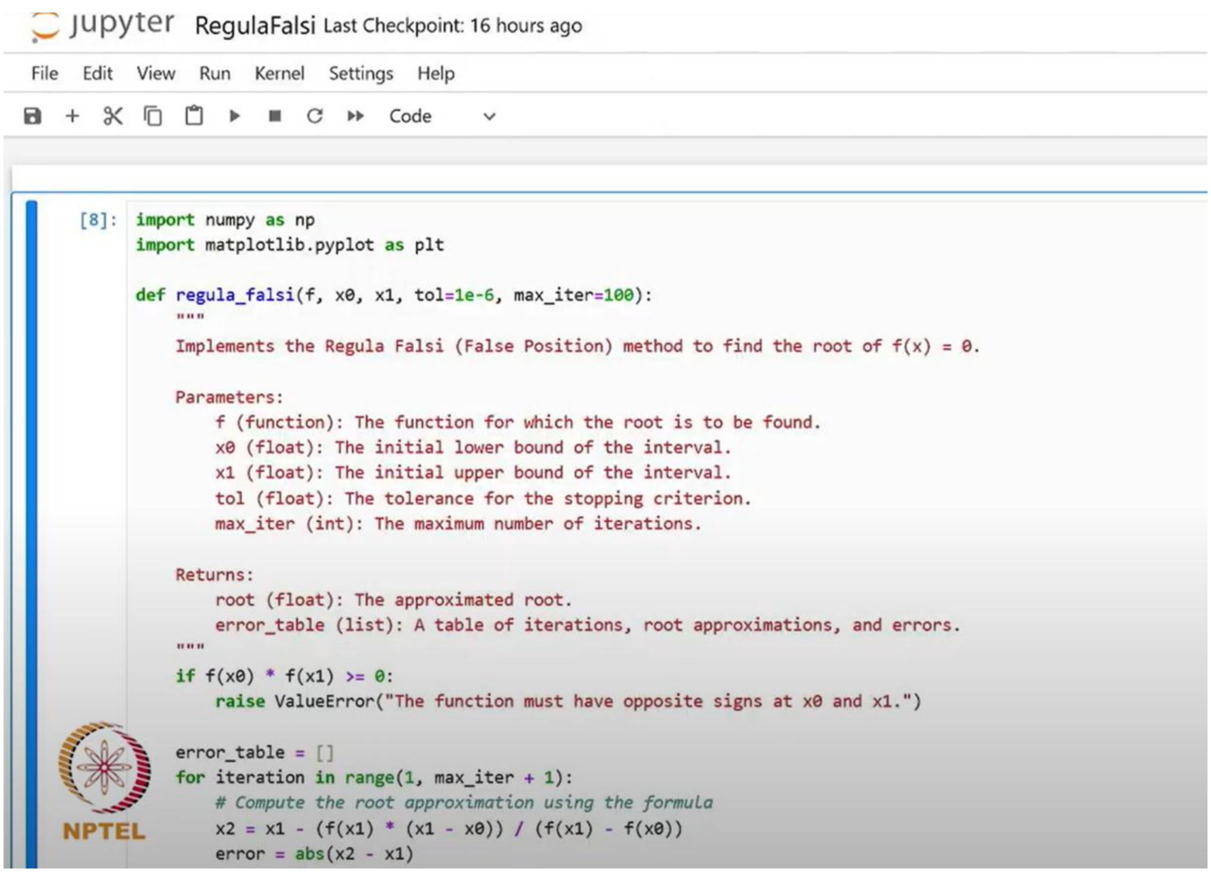
The implementation is basically the same as that of the regula falsi method. And what is the disadvantage? If the initial guesses are far from the root, the method may converge slowly or fail to converge. Not always, but the method does not guarantee convergence in all cases.

Another problem is that it may cause division by zero if even f(x1)is equal to f(x0); then the slope becomes zero, causing a division by zero. So, this problem can happen. How does this problem arise? Let me explain. If we see, I have done this: if the value of f was almost the same at both x0 and x1, then this quantity will become zero. And as it becomes zero, it will diverge and tend toward infinity.

So, in this method, we have to keep in mind that the difference between the function values at the two points should be sufficient so that it converges, right?

So, with the help of this, we can make the regula falsi and secant method. Now, let's see its code — how the Python code looks. So, I will save the regula falsi method code as it comes. We have created this code for the regula falsi method.

(Refer slide time: 26:55)



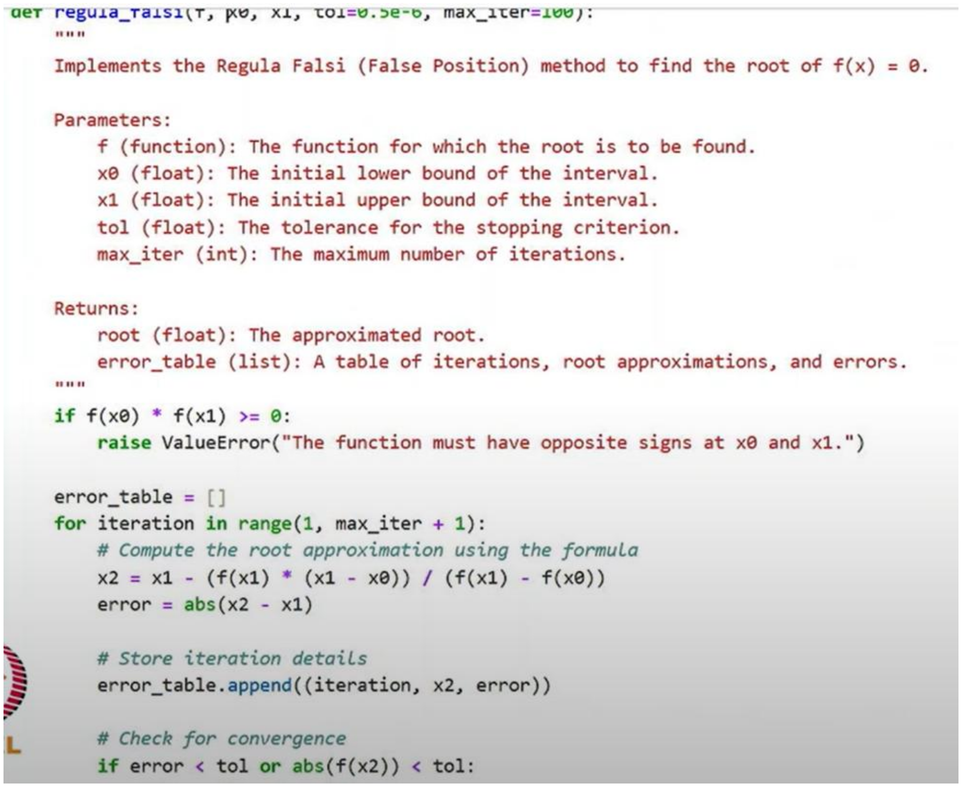
jupyter RegulaFalsi Last Checkpoint: 16 hours ago
File Edit View Run Kernel Settings Help
Code
NPTEL

```python
[8]: import numpy as np
     import matplotlib.pyplot as plt

     def regula_falsi(f, x0, x1, tol=1e-6, max_iter=100):
         """
         Implements the Regula Falsi (False Position) method to find the root of f(x) = 0.

         Parameters:
             f (function): The function for which the root is to be found.
             x0 (float): The initial lower bound of the interval.
             x1 (float): The initial upper bound of the interval.
             tol (float): The tolerance for the stopping criterion.
             max_iter (int): The maximum number of iterations.

         Returns:
             root (float): The approximated root.
             error_table (list): A table of iterations, root approximations, and errors.
         """
         if f(x0) * f(x1) >= 0:
             raise ValueError("The function must have opposite signs at x0 and x1.")

         error_table = []
         for iteration in range(1, max_iter + 1):
             # Compute the root approximation using the formula
             x2 = x1 - (f(x1) * (x1 - x0)) / (f(x1) - f(x0))
             error = abs(x2 - x1)
```

In this code, we have done, like before, that we imported NumPy for mathematical calculations and Matplotlib for plotting. So, we have already done this work for plotting, right?

So, what do we do with this? Let's write it down for ourselves: this is the program we are doing, it is the regula falsi method, and the root has to be found. The tolerance for it, we have given as 1 here, but we can also give it as 0.5. Okay, so here we have it. Now, f is the function which we will provide, x0 and x1 will be our initial values. We will give a tolerance — here, we want to do this much tolerance only, not more than this. After that, how many iterations do we have to take? So I assumed that let's take 100. Okay, so I have defined a function named regula falsi. What will it give us in return? As soon as we call it, it will give us the root in return and will give a table of iterations, root approximations, and errors, which will get saved.

(Refer slide time: 28:21)



```python
def regula_falsi(f, x0, x1, tol=0.5e-6, max_iter=100):
    """
    Implements the Regula Falsi (False Position) method to find the root of f(x) = 0.

    Parameters:
        f (function): The function for which the root is to be found.
        x0 (float): The initial lower bound of the interval.
        x1 (float): The initial upper bound of the interval.
        tol (float): The tolerance for the stopping criterion.
        max_iter (int): The maximum number of iterations.

    Returns:
        root (float): The approximated root.
        error_table (list): A table of iterations, root approximations, and errors.
    """
    if f(x0) * f(x1) >= 0:
        raise ValueError("The function must have opposite signs at x0 and x1.")

    error_table = []
    for iteration in range(1, max_iter + 1):
        # Compute the root approximation using the formula
        x2 = x1 - (f(x1) * (x1 - x0)) / (f(x1) - f(x0))
        error = abs(x2 - x1)

        # Store iteration details
        error_table.append((iteration, x2, error))

        # Check for convergence
        if error < tol or abs(f(x2)) < tol:
```

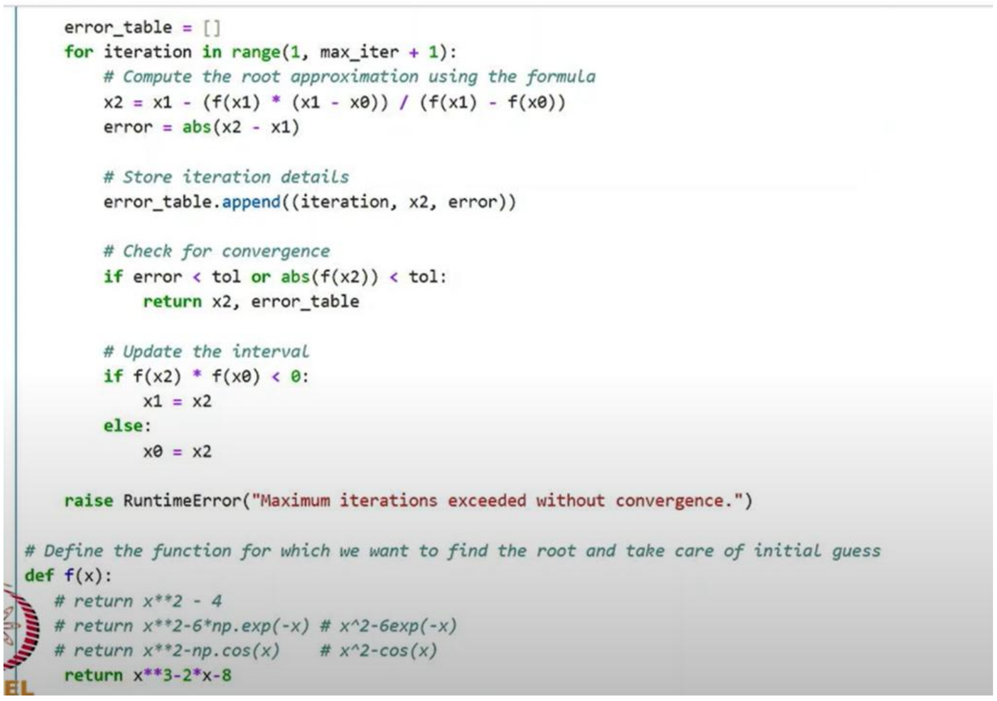
Now, we have given the initial values x0 and x1, and we check that the product of both f(x0) and f(x1) is greater than zero. So, we will say that if the initial guesses are not correct, then we will have to change the initial guesses. So the message will come from here: the function must have opposite signs at x0 and x1. So if it is a bracketing method, then the root should be inside the bracket. Now, just like we did earlier, we will create an error table whose dimension we do not know. So, we define it in the close bracket. Now, the iteration runs in range from 0 to maximum iterations plus 1. So, if it is 100, then we have given maximum iterations, it will go up to plus 1.

Now, what to do? See, the method that we had applied:

So, x2 is x1 minus f(x1) into x1 minus x0, divided by f(x1) minus f(x0). This was the function that we were using there, and from here we got this value. So, the values that we have, which we used here, came here. Like, this is the method: f(x1) and x - x1 = 0. So, we wrote it there. So, these values came, and what would be the error? Because there is x1 and there is x2 — between the two, the values between them. Now, what we did is we appended them in the error table and wrote it down. Here, we wrote the iteration. Here, our new value of x2 is the root, and the error here. Now, we have seen that for check for convergence, we have to see that if the error is less than tolerance, or the absolute value is less than tolerance — or is there any other? So, if we see, we applied this to us and used the logical gate operator. So, if even one of the two is satisfied, then return x2 and the error table which is being created should return it to us. Otherwise, update this value.

So, what did it do? It updated the interval. If x2 and x0 are negative with each other and are of opposite sign, then what does it mean? The x2 becomes our x1. Otherwise, the x2 becomes x0, because only one of the two will be formed, right?

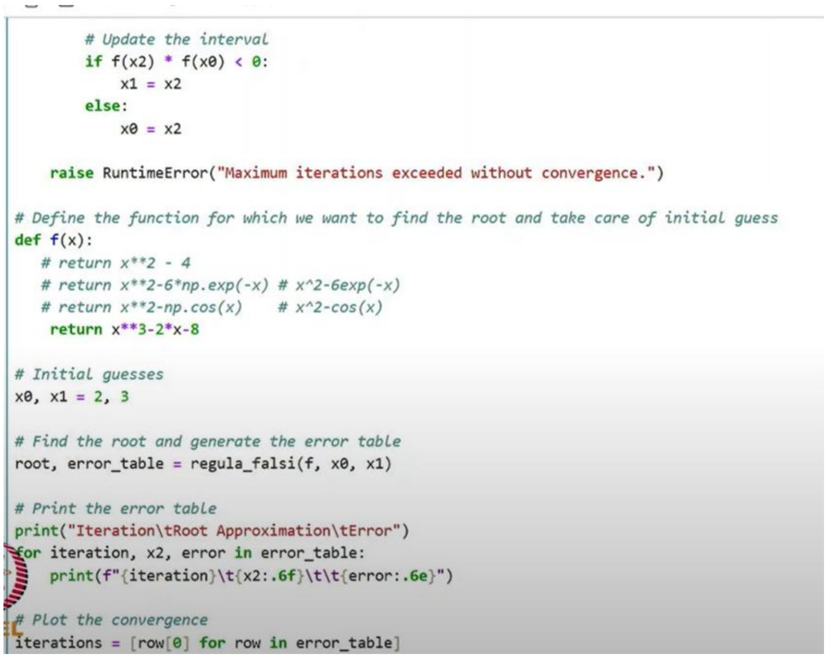
(Refer slide time: 30:59)



```
    error_table = []
    for iteration in range(1, max_iter + 1):
        # Compute the root approximation using the formula
        x2 = x1 - (f(x1) * (x1 - x0)) / (f(x1) - f(x0))
        error = abs(x2 - x1)

        # Store iteration details
        error_table.append((iteration, x2, error))

        # Check for convergence
        if error < tol or abs(f(x2)) < tol:
            return x2, error_table

        # Update the interval
        if f(x2) * f(x0) < 0:
            x1 = x2
        else:
            x0 = x2

    raise RuntimeError("Maximum iterations exceeded without convergence.")

# Define the function for which we want to find the root and take care of initial guess
def f(x):
   # return x**2 - 4
   # return x**2-6*np.exp(-x) # x^2-6exp(-x)
   # return x**2-np.cos(x)    # x^2-cos(x)
    return x**3-2*x-8
```

Here, we saw that we were going close, so we updated the interval. See, from here, we did it like: if we are going here, then if these are the values, then we will take these. And if we are going from here, then we will take these values. So, between both, we have to check which values — whether — because this condition, which is the bracketing condition, has to be fulfilled. And after that, what will we do? Raise runtime error: maximum iteration exceeded without convergence. If it has not converged in so many iterations, then we will get a message. Then we will change it. So, now what are we doing? Define the function for when we want to find the root. So, we have defined the function f(x) equal to. Now, it depends on which function it is. So here, let us first take this: x^3 - 2x - 8. So, the initial value that I have taken — the initial root of the function — I am taking this, okay? x^3 - 2x - 8. So, we have taken this ring, and I have taken the initial values 2 and 3.

(Refer slide time: 32:21)



```
        # Update the interval
        if f(x2) * f(x0) < 0:
            x1 = x2
        else:
            x0 = x2

    raise RuntimeError("Maximum iterations exceeded without convergence.")

# Define the function for which we want to find the root and take care of initial guess
def f(x):
   # return x**2 - 4
   # return x**2-6*np.exp(-x) # x^2-6exp(-x)
   # return x**2-np.cos(x)    # x^2-cos(x)
    return x**3-2*x-8

# Initial guesses
x0, x1 = 2, 3

# Find the root and generate the error table
root, error_table = regula_falsi(f, x0, x1)

# Print the error table
print("Iteration\tRoot Approximation\tError")
for iteration, x2, error in error_table:
    print(f"{iteration}\t{x2:.6f}\t\t{error:.6e}")

# Plot the convergence
iterations = [row[0] for row in error_table]
```

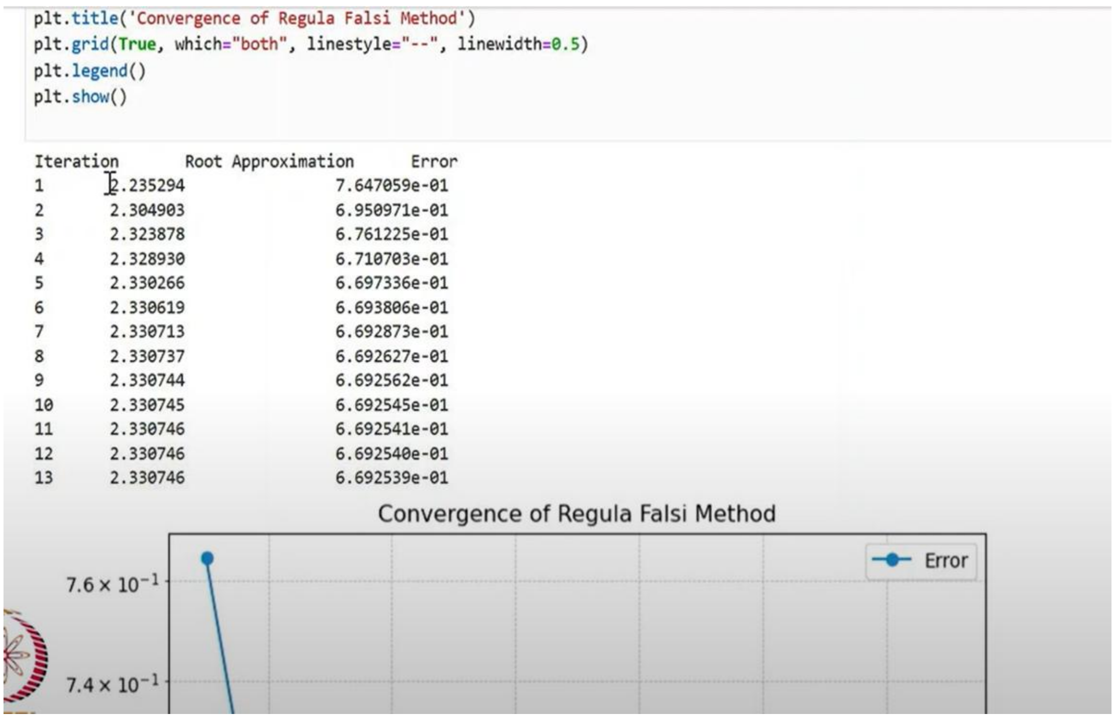
So, if you remember, I do this here — that the math that we are taking is $x^3 - 2x - 8$. So, y: $x^3 - 2x - 8$. This is our regula falsi; we have to apply it.

So, what will we do? We had seen it at zero. We had seen it at one. So, we have to see it at two. So, if you look at two, then 8 - 4 - 8 — so negative — minus 4 comes. So, this is negative. Then we look at three. As soon as we looked at three, we got 27 - 6 - 8. So, this came out positive. So, from here, we came to know that there is a bracket, and our two and three — so the root will lie somewhere in between it. So, we came to know this with the help of the intermediate value theorem. So, as soon as we came to know this, from here, we took the initial guess, and we supplied the initial guess here. Okay? So, the roots and error table will call regula falsi here, and the values that we are getting from this, which it returns, will be saved here. So, from here, we called this function. After that, when the process will be over, we will print all the tables. So, print iteration, root approximations — which is the number of iterations — which is as much as the two, and the root, and what is the error. So, all this will be printed here.

And we will do the plotting here — that iteration row, which will become our matrix, that the first column is okay, that gives me the iteration; second column gives us the error; and the third column gives us — the second column gives the root, and the third column gives us the error.

So, let me run this and see what answer comes. So, this came. So, we ran it. So, there is no error.

(Refer slide time: 35:07)



```
plt.title('Convergence of Regula Falsi Method')
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.legend()
plt.show()
```

```
Iteration       Root Approximation      Error
1       2.235294                7.647059e-01
2       2.304903                6.950971e-01
3       2.323878                6.761225e-01
4       2.328930                6.710703e-01
5       2.330266                6.697336e-01
6       2.330619                6.693806e-01
7       2.330713                6.692873e-01
8       2.330737                6.692627e-01
9       2.330744                6.692562e-01
10      2.330745                6.692545e-01
11      2.330746                6.692541e-01
12      2.330746                6.692540e-01
13      2.330746                6.692539e-01
```

So, see what it means. Now what happened — now, from here, we will know that in the first iteration, our x2 value came out to be 2.23, and the error in it was 7.64. Then we calculated a new value. In the second iteration, it came out to be 2.30, and the error came out to be this much. Then in the third, fourth, fifth, sixth — we kept taking it like this. Now, we have seen that our convergence was the tolerance — it was 0.5e to the power -6. So, the values we have — that is, till the sixth digit, there is zero and 0.5

So, if we see, we have this tolerance. In this case, we have 0.0000005 — this is the tolerance. So now, whenever we have to do any calculation, we have to see whether the same value comes till the sixth zero. Till then, it will keep on calculating.

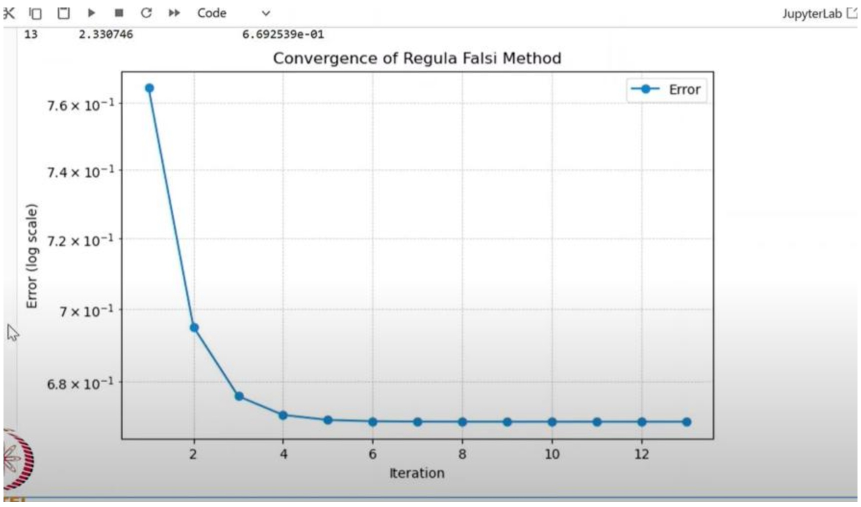
So now, what did it do? It kept on calculating. Now you see in this — that our iteration is going on. In the 10th iteration, our 11th iteration came to 2.33746.

If we see in the 12th iteration also, it comes up to six digits, but its value may differ in the seventh digit. So, then this came, and in the third iteration this came. So, you can see that the 11th, 12th, and 13th iterations, up to six digits, if we subtract both of them, then we will get zero value.

So, what does it mean? That our root is not changing till six digits starting from the 11th iteration up to the 13th iteration. So, it means that we have got our convergence. Now, there is no major change in the root.

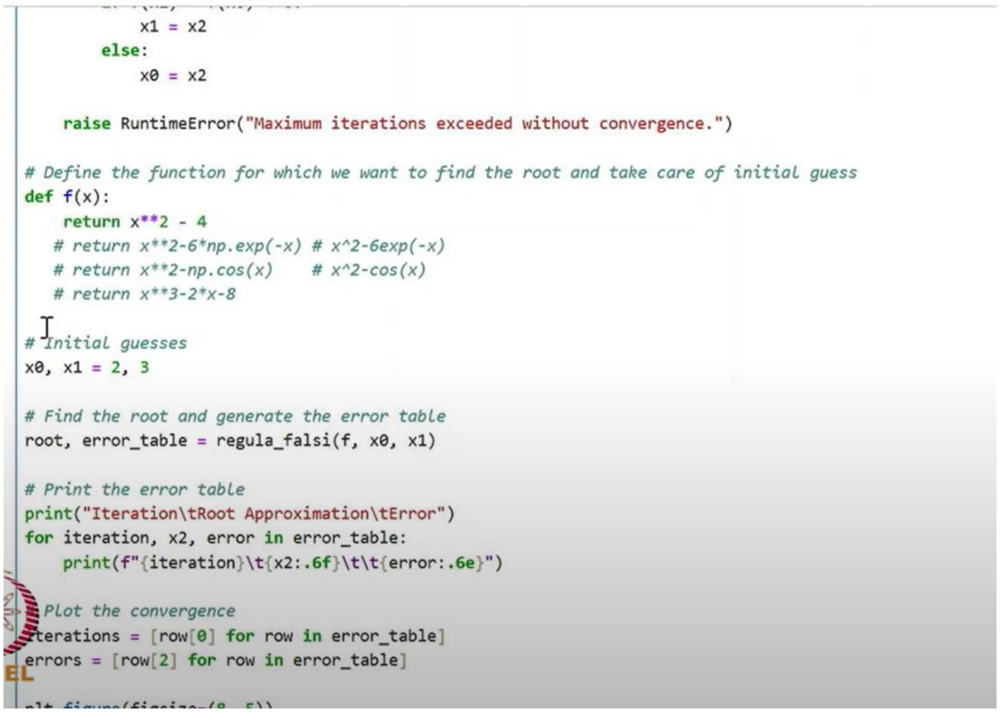
So then we have plotted the convergence according to iteration and according to error.

(Refer slide time: 37:22)


```
13      2.330746                6.692539e-01
```

The first iteration was here, so its value was 7.6 into 10 power -1, meaning 0.76. Then the second iteration, then the third iteration, then the fourth iteration, then the fifth iteration, then the sixth iteration. So, after the sixth iteration, look, there is a small change. There is a slight small change in the roots. Otherwise, their value will remain almost the same. So, here we have got an error of Regula Falsi. Now it is possible that this is for the function. Now what do I do? Let me change the function. So, let's take it and comment it out and write it as return. So this means we have taken $x^2 - 4$.

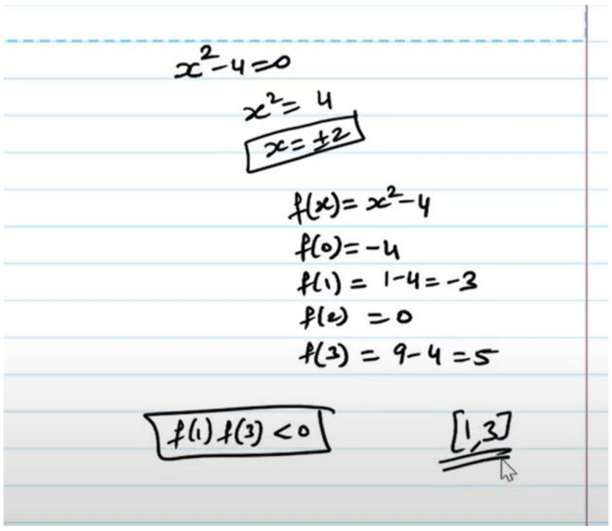
(Refer slide time: 38:13)


```
            x1 = x2
        else:
            x0 = x2

    raise RuntimeError("Maximum iterations exceeded without convergence.")

# Define the function for which we want to find the root and take care of initial guess
def f(x):
    return x**2 - 4
   # return x**2-6*np.exp(-x) # x^2-6exp(-x)
   # return x**2-np.cos(x)    # x^2-cos(x)
   # return x**3-2*x-8

# Initial guesses
x0, x1 = 2, 3

# Find the root and generate the error table
root, error_table = regula_falsi(f, x0, x1)

# Print the error table
print("Iteration\tRoot Approximation\tError")
for iteration, x2, error in error_table:
    print(f"{iteration}\t{x2:.6f}\t\t{error:.6e}")

# Plot the convergence
iterations = [row[0] for row in error_table]
errors = [row[2] for row in error_table]
```

So, now what am I doing? I am taking the values of this. I will write it here. So, what have we done? Now I have taken the value $x^2 - 4$ and taken the root of this equation. So, we know that $x^2 - 4$, and the x that comes out is plus minus 2. So, we will get two roots: one +2 and one -2. So, now let's see how to find out the root in this. Now, if I take this on f(x) equal to $x^2 - 4$, then it comes on one. See f at 0 comes to -4, f at 1 comes to 1 - 4 = -3. When we looked at f at 2, we got 0 there. Let's not know what it is. When we looked at f at 3, we got 9 -

4 = 5. So, now we have seen that this f(1) and f(3), so it means that the 1 and the 3 are doing a root lie somewhere in between them.

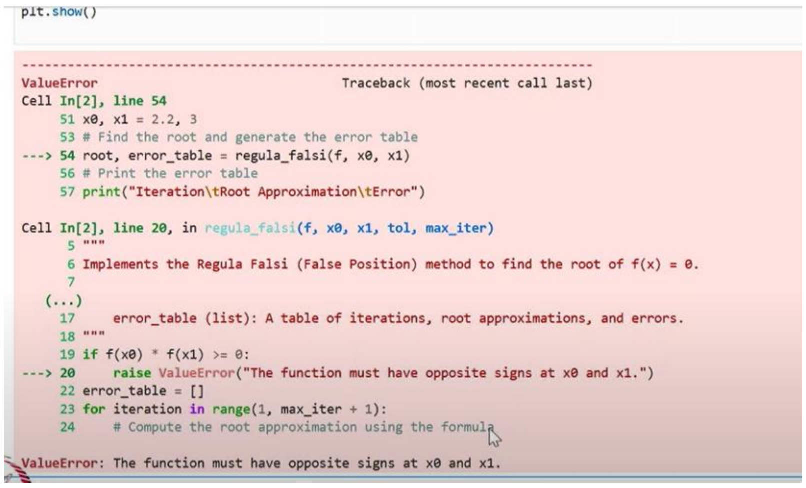
(Refer slide time: 39:59)



$$x^2-4=0$$
$$x^2=4$$
$$\boxed{x=\pm 2}$$
$$f(x)=x^2-4$$
$$f(0)=-4$$
$$f(1)=1-4=-3$$
$$f(2)=0$$
$$f(3)=9-4=5$$
$$\boxed{f(1)f(3)<0} \qquad \underline{\underline{[1,3]}}$$

So, we will have to look at this thing. So now, what do I do? I take $x^2 - 4$. So, the guess which I will give is 2.2, and 3, if I give the wrong guess, then it turns out that it did not converge at all. Now, like here, I give 2 point two as the initial guess and I keep this element here and run it. So, see what message came: that it did not converge at all. So, the function must have the opposite sign at x0 and x1. So, it is written that both should have opposite signs, but we know that they are not opposite signs. Both have come positive. So, from here we got an error.

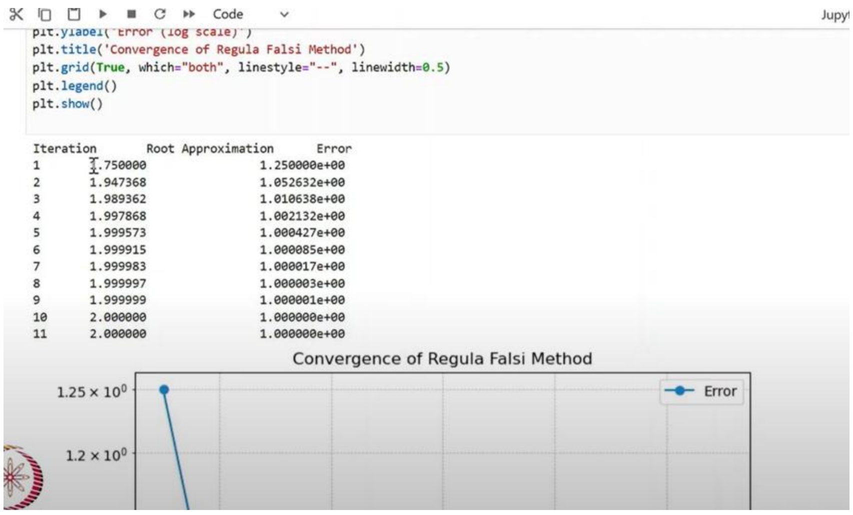
(Refer slide time: 40:48)



```
plt.show()

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[2], line 54
     51 x0, x1 = 2.2, 3
     53 # Find the root and generate the error table
---> 54 root, error_table = regula_falsi(f, x0, x1)
     56 # Print the error table
     57 print("Iteration\tRoot Approximation\tError")

Cell In[2], line 20, in regula_falsi(f, x0, x1, tol, max_iter)
      5 """
      6 Implements the Regula Falsi (False Position) method to find the root of f(x) = 0.
      7
   (...)
     17     error_table (list): A table of iterations, root approximations, and errors.
     18 """
     19 if f(x0) * f(x1) >= 0:
---> 20     raise ValueError("The function must have opposite signs at x0 and x1.")
     22 error_table = []
     23 for iteration in range(1, max_iter + 1):
     24     # Compute the root approximation using the formula

ValueError: The function must have opposite signs at x0 and x1.
```

So, now I will correct it. Now I will know from here. I thought that I have done something wrong, so what did I do? Let me write it as 1. Now let's run it and see.

(Refer slide time: 41:04)



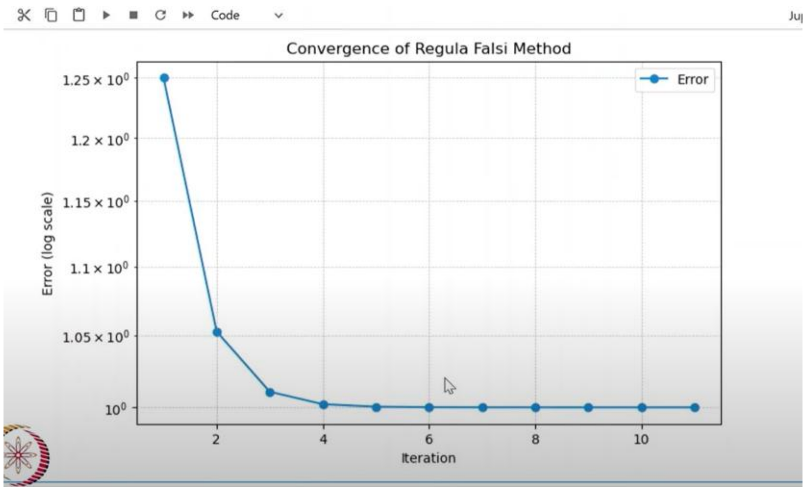
```
plt.ylabel('Error (log scale)')
plt.title('Convergence of Regula Falsi Method')
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.legend()
plt.show()
```

```
Iteration       Root Approximation      Error
1        1.750000                1.250000e+00
2        1.947368                1.052632e+00
3        1.989362                1.010638e+00
4        1.997868                1.002132e+00
5        1.999573                1.000427e+00
6        1.999915                1.000085e+00
7        1.999983                1.000017e+00
8        1.999997                1.000003e+00
9        1.999999                1.000001e+00
10       2.000000                1.000000e+00
11       2.000000                1.000000e+00
```

The answer is here. In the first iteration, the error was 1.75. Then after refining, we got another error in the next iteration. The root increased a little in this case. Then we refined it further. So, the root is going towards 2.

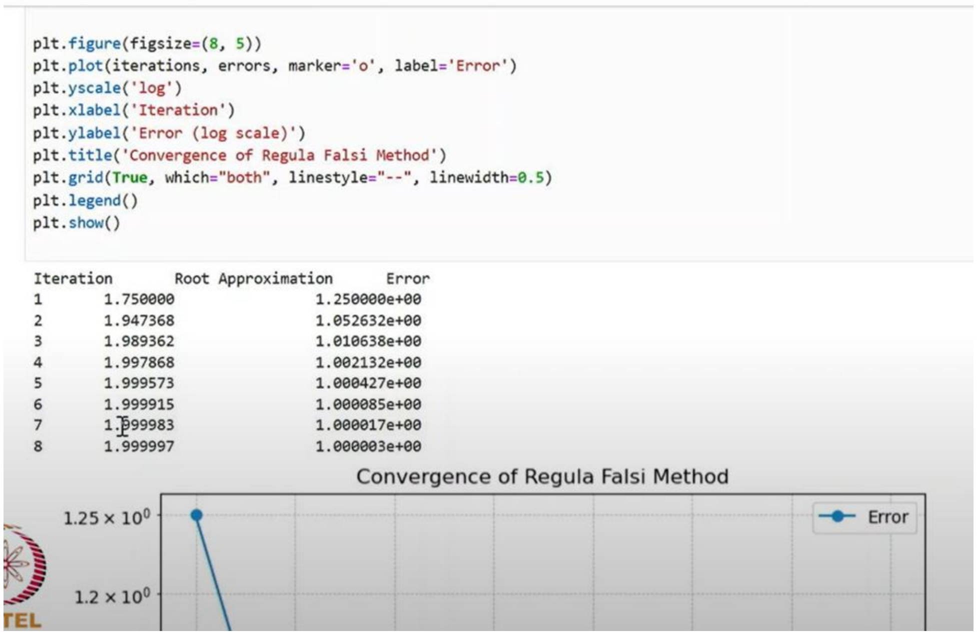
So, this root is converging towards 2. So, we keep refining it and keep reducing the error. So, see, the first error is 1.25, then 1.05, then 1.01. This way, the error keeps decreasing and the root we have, we keep converging towards the root. So, in this we see that in 11 iterations we have got the root which was our √2, and this is our convergence.

(Refer slide time: 41:52)



Now, if someone says that we are getting this root in 11 iterations, why are we taking so many offs? Then it all depends on what our tolerance is. If I reduce this toss further, then let's see. I have made it four. Now I want to add up to four digits, and the same should be added up to four digits. So now let's see what happens in this case. So, the number of iterations has reduced. In the eighth iteration, we got the answer. Because what we did was we changed the accuracy, so now we have to see only four.

(Refer slide time: 42:26)



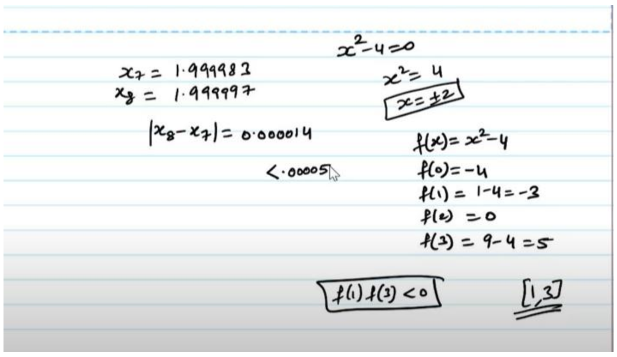
```
plt.figure(figsize=(8, 5))
plt.plot(iterations, errors, marker='o', label='Error')
plt.yscale('log')
plt.xlabel('Iteration')
plt.ylabel('Error (log scale)')
plt.title('Convergence of Regula Falsi Method')
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.legend()
plt.show()
```

```
Iteration       Root Approximation      Error
1       1.750000                1.250000e+00
2       1.947368                1.052632e+00
3       1.989362                1.010638e+00
4       1.997868                1.002132e+00
5       1.999573                1.000427e+00
6       1.999915                1.000085e+00
7       1.999983                1.000017e+00
8       1.999997                1.000003e+00
```

So, if you see, the difference between this and these two, how much has it come? 8 - 7 is how much it has become 0.000014 so one comes. So, what does it mean? It has become less than 0.0001. So, what does it mean? Look at the values that we were doing: 1.999983 and 1.999997, so we have got two values. We see this.

Now if you see, the root that we got was 1.999983 and 1.999997, how much has it come? This came, okay, and it is going towards 2. So, if we take the difference of these two, then suppose ours was x7 and this was x8, so we saw that x8 minus x7 is how much it is coming out. If we see, it will come out to be 0.000014, and we know that this is less than 0.0005, right? So, this will stop here and we will get the solution.

(Refer slide time: 44:04)



$x_7 = 1.999983$

$x_8 = 1.999997$

$|x_8 - x_7| = 0.000014$

$< .00005$

$x^2 - 4 = 0$

$x^2 = 4$

$\boxed{x = \pm 2}$

$f(x) = x^2 - 4$

$f(0) = -4$

$f(1) = 1 - 4 = -3$

$f(2) = 0$

$f(3) = 9 - 4 = 5$

$\boxed{f(1)f(3) < 0}$

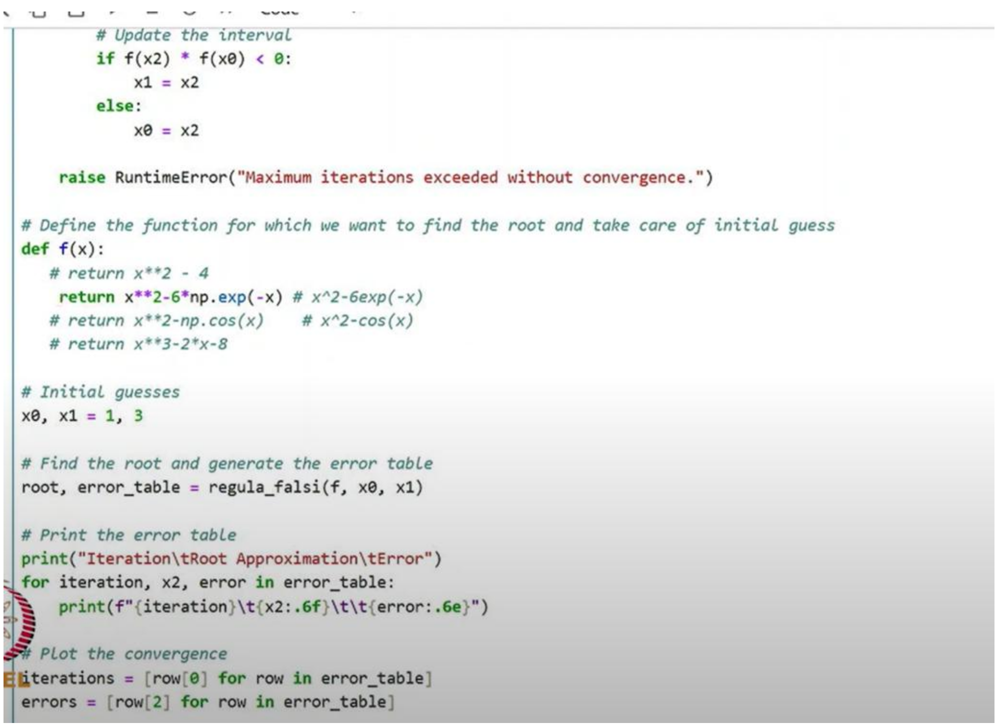
$\underline{[1,3]}$

If we see before this, if we see what is happening in the sixth, then 83 - 15, then you will see that the value is more than 0.5, so it will iterate one more time. So, everything depends on how much accuracy we need, how many iterations do we need. If I need more, then if we do this, then you will see that the answer comes in five iterations. So, how much accuracy we need, that much more the number of iterations will increase.

Now I should increase it to seven. Let's see what will happen. Look, the iteration has come to 12. So, this many have come. From here, we have got these roots. Now what do I do? I want to check that my method, this code of ours, is robust. So, what we need to do is we need to check this for a lot of functions.

So, now I take this other function: $x^2 - 6e^{-x}$.

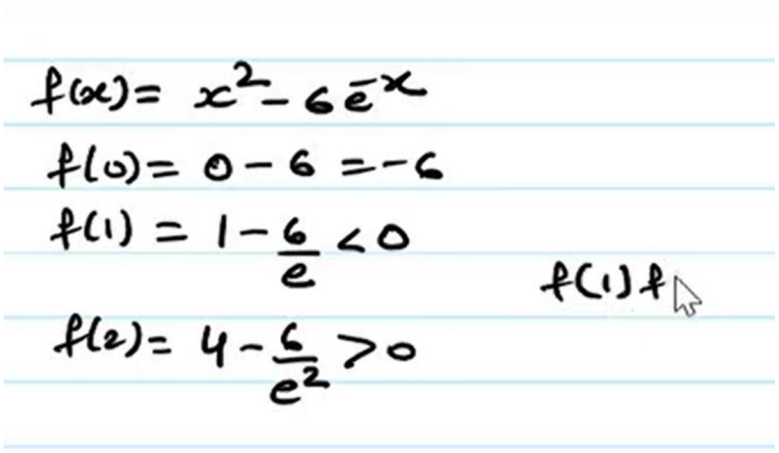
(Refer slide time: 45:12)



```
        # Update the interval
        if f(x2) * f(x0) < 0:
            x1 = x2
        else:
            x0 = x2

    raise RuntimeError("Maximum iterations exceeded without convergence.")

# Define the function for which we want to find the root and take care of initial guess
def f(x):
   # return x**2 - 4
    return x**2-6*np.exp(-x) # x^2-6exp(-x)
   # return x**2-np.cos(x)    # x^2-cos(x)
   # return x**3-2*x-8

# Initial guesses
x0, x1 = 1, 3

# Find the root and generate the error table
root, error_table = regula_falsi(f, x0, x1)

# Print the error table
print("Iteration\tRoot Approximation\tError")
for iteration, x2, error in error_table:
    print(f"{iteration}\t{x2:.6f}\t\t{error:.6e}")

# Plot the convergence
iterations = [row[0] for row in error_table]
errors = [row[2] for row in error_table]
```

So, what I did is, because I know that a function can be anything, or it can even be a transcendental. It can even be a simple algebraic equation. It can be anything. So, if we have this function, then what I did is I take this function and take another function: f equal to $x^2 - 6e^{-x}$. So, I want to solve this with regula falsi. So, I saw that if I take f(0), then this will be 0 - 6. So, this will become -6, and so f(1) is 1 - 6 by e so this value will come out to be less than 6. So, this is also negative. f(2), let's see what will happen: 4 - 6 by e square will come out to be square, which is a very big value. So, we will see that the value of this is greater than zero.

(Refer slide time: 46:33)



$$f(x) = x^2 - 6e^{-x}$$

$$f(0) = 0 - 6 = -6$$

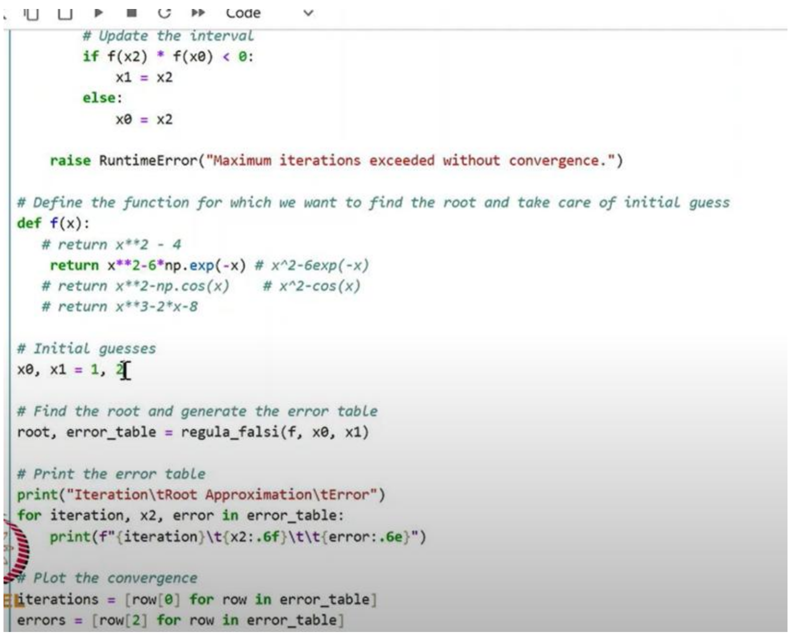
$$f(1) = 1 - \frac{6}{e} < 0$$

$$f(1)f$$

$$f(2) = 4 - \frac{6}{e^2} > 0$$

So now we have to do this. Now I got to know that if I have to solve this, then this condition is getting satisfied here. So, if we give one and two, then we can get a better result. So, what did we do? I wrote this, and I gave one and two.

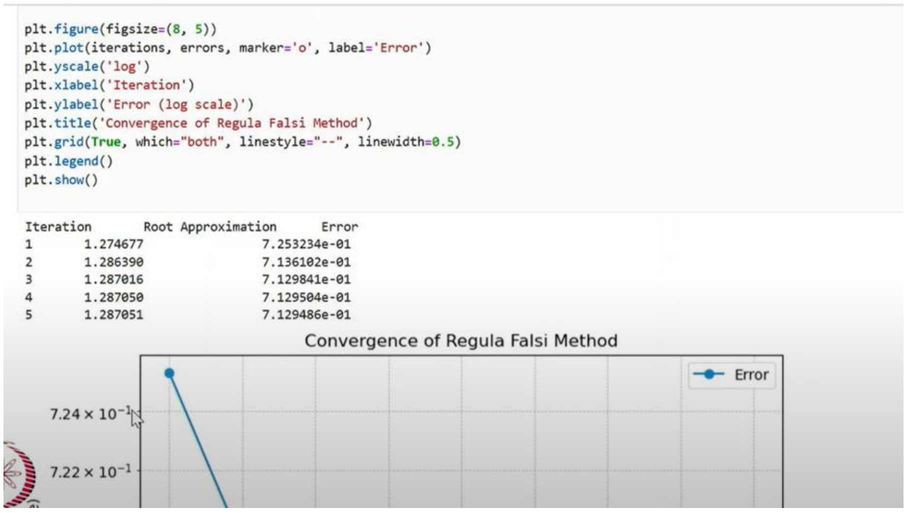
(Refer slide time: 46:55)



```
        # Update the interval
        if f(x2) * f(x0) < 0:
            x1 = x2
        else:
            x0 = x2

    raise RuntimeError("Maximum iterations exceeded without convergence.")

# Define the function for which we want to find the root and take care of initial guess
def f(x):
    # return x**2 - 4
    return x**2-6*np.exp(-x) # x^2-6exp(-x)
    # return x**2-np.cos(x)    # x^2-cos(x)
    # return x**3-2*x-8

# Initial guesses
x0, x1 = 1, 2

# Find the root and generate the error table
root, error_table = regula_falsi(f, x0, x1)

# Print the error table
print("Iteration\tRoot Approximation\tError")
for iteration, x2, error in error_table:
    print(f"{iteration}\t{x2:.6f}\t\t{error:.6e}")

# Plot the convergence
iterations = [row[0] for row in error_table]
errors = [row[2] for row in error_table]
```

One and three will also work, there is no problem. But what was our first condition? That it should be close to the guess so that we get sure that we have to get the root. So, I reduced the tolerance a little. Now let's see what the answer is. See, in five iterations, we got the solution.

(Refer slide time: 47:17)



```
plt.figure(figsize=(8, 5))
plt.plot(iterations, errors, marker='o', label='Error')
plt.yscale('log')
plt.xlabel('Iteration')
plt.ylabel('Error (log scale)')
plt.title('Convergence of Regula Falsi Method')
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.legend()
plt.show()
```

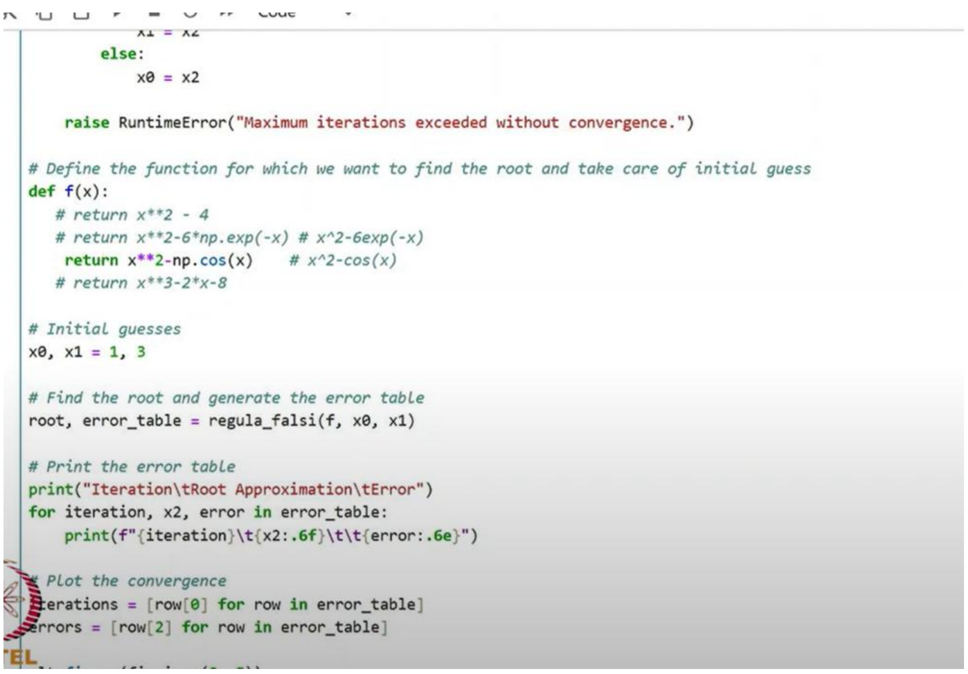
```
Iteration	Root Approximation	Error
1	1.274677		7.253234e-01
2	1.286390		7.136102e-01
3	1.287016		7.129841e-01
4	1.287050		7.129504e-01
5	1.287051		7.129486e-01
```

From 1.287, we got its solution, and we got the convergence. So, what did we do in this case? Just four digits, up to five digits basically. Here, we did , so here, up to five digits, it should be the same. So it means, see, this solution which is 1.28705 and 1.28705—so five digits are exactly the same. Six digits , which the difference between the two values should be less than 0.5. So, if it is less than 0.5, then this is our error, and we get this value.

If I look at it by giving it to 3 instead of 2 , let's see what happens. The number of iterations is not much different—just one more iteration increased, by two iterations. A little more work had to be done. In this case, our code had to work a little more, but we got our convergence. Okay, so in this case also, we got our initial solution.

Now let's look at another solution. Let's look at this. So this is our $x^2 - \cos(x)$.

(Refer slide time: 48:32)

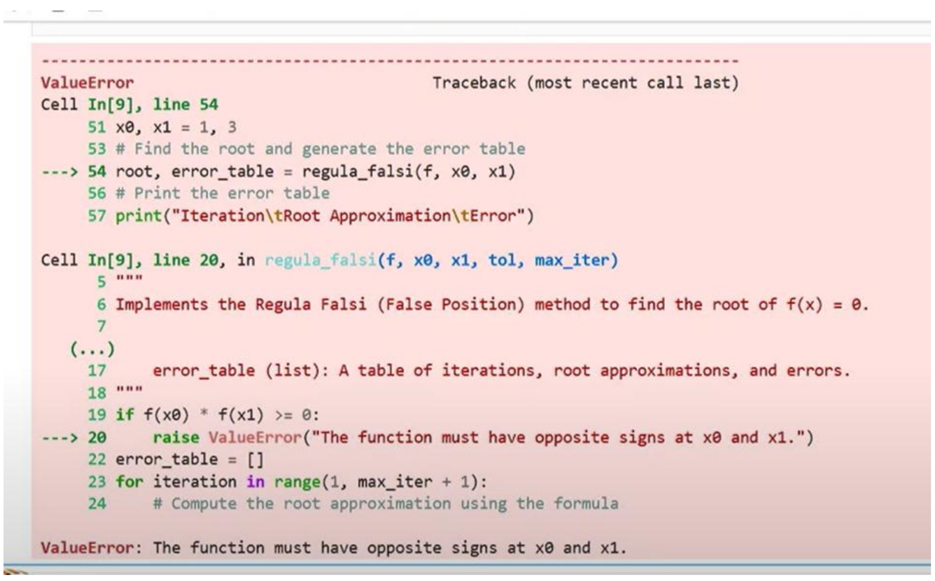


```
            x1 = x2
        else:
            x0 = x2

    raise RuntimeError("Maximum iterations exceeded without convergence.")

# Define the function for which we want to find the root and take care of initial guess
def f(x):
   # return x**2 - 4
   # return x**2-6*np.exp(-x) # x^2-6exp(-x)
    return x**2-np.cos(x)    # x^2-cos(x)
   # return x**3-2*x-8

# Initial guesses
x0, x1 = 1, 3

# Find the root and generate the error table
root, error_table = regula_falsi(f, x0, x1)

# Print the error table
print("Iteration\tRoot Approximation\tError")
for iteration, x2, error in error_table:
    print(f"{iteration}\t{x2:.6f}\t\t{error:.6e}")

# Plot the convergence
terations = [row[0] for row in error_table]
rrors = [row[2] for row in error_table]
```

So, let's see what is happening in this. Now, our second function is here. Suppose I took it and took $x^2 - \cos(x)$. Now, this is a transcendental function. So, if we want to solve this, we cannot solve it using pen and paper. But we came to know from approximations that if I take zero, then it will be 0 -1, that is negative. So now I have taken f(1) that's is 1 -cos1. and whatever the value of cos is, it will always be less than one. So it will always be positive. So from this, we got to know approximately that our root will be somewhere between zero and one, and it is neither zero nor one. So, we will use this mathematics to find out where the root is.

So now look at the root that I have taken, 1 and 3 if I take the same, but we know that it is between zero and one. So, if I run it by mistake, then we have got the opposite sign—the function must have opposite sign.

(Refer slide time: 49:51)

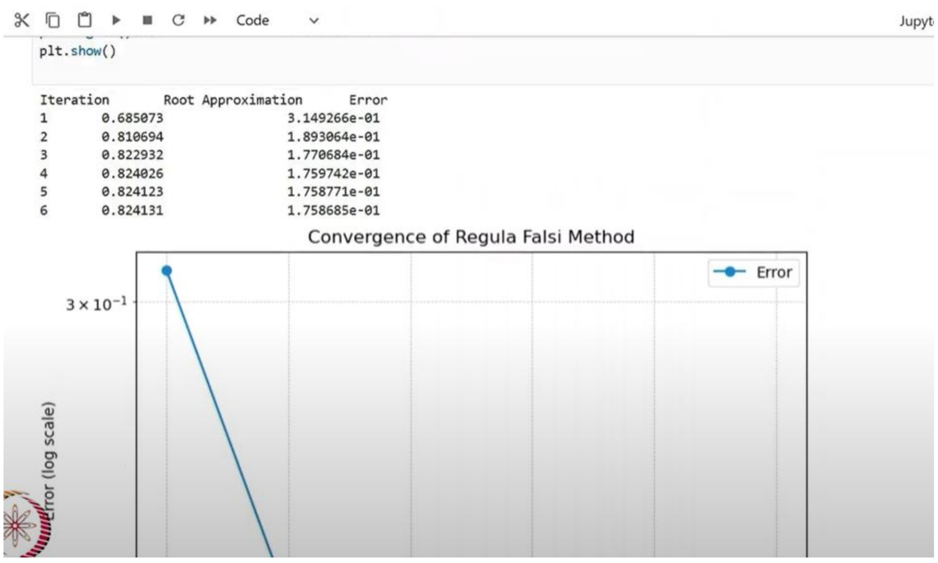


```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[9], line 54
     51 x0, x1 = 1, 3
     53 # Find the root and generate the error table
---> 54 root, error_table = regula_falsi(f, x0, x1)
     56 # Print the error table
     57 print("Iteration\tRoot Approximation\tError")

Cell In[9], line 20, in regula_falsi(f, x0, x1, tol, max_iter)
      5 """
      6 Implements the Regula Falsi (False Position) method to find the root of f(x) = 0.
      7
    (...)
     17     error_table (list): A table of iterations, root approximations, and errors.
     18 """
     19 if f(x0) * f(x1) >= 0:
---> 20     raise ValueError("The function must have opposite signs at x0 and x1.")
     22 error_table = []
     23 for iteration in range(1, max_iter + 1):
     24     # Compute the root approximation using the formula

ValueError: The function must have opposite signs at x0 and x1.
```

So as soon as it comes, I will get to know that I have taken it. So, I make it zero and make it one here and see. Look, we have the root, and it has iteration—it has six iterations only. So in the six iterations, you can see that the values are exactly the same up to five digits. And if we see the difference in the sixth digit also, it will be less than 0.5. So it will stop.

And why? Because to stop it, we have given a condition—that it should continue until it does not stop, until our tolerance is less than or its value is not less than the tolerance. So we did this. And see, here there was so much error in the initial guess, the error decreased completely. It became this in the second, and this in the third, and there was no change after that. So you can see that the method of Regula Falsi, its order of convergence is a little better as compared to the fixed-point. We saw that it was absolutely linear. So it is a little better than that.

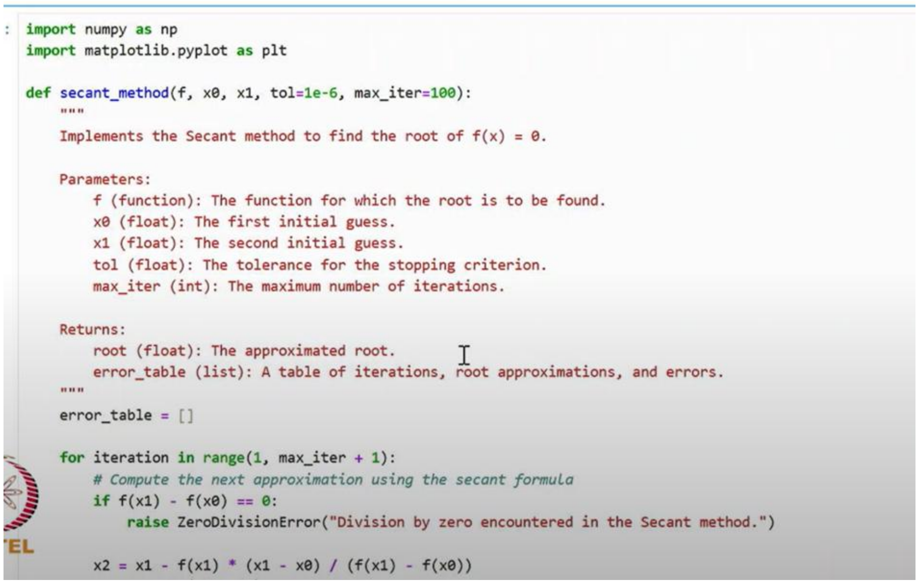
(Refer slide time: 51:09)



```
plt.show()

Iteration       Root Approximation      Error
1       0.685073                3.149266e-01
2       0.810694                1.893064e-01
3       0.822932                1.770684e-01
4       0.824026                1.759742e-01
5       0.824123                1.758771e-01
6       0.824131                1.758685e-01
```

So this is our Regula Falsi method, and that is done. Now similarly, we want to see what is the problem in the secant. So, as I told you that the secant method is like this—it is like Regula Falsi. There may be a problem in the initial guess. Okay, so what do I do? So let me go... let me write it down and it opens it. So, the secant method, and we have created a code for the learn method. It is the same code. There can be a change in the initial guess.

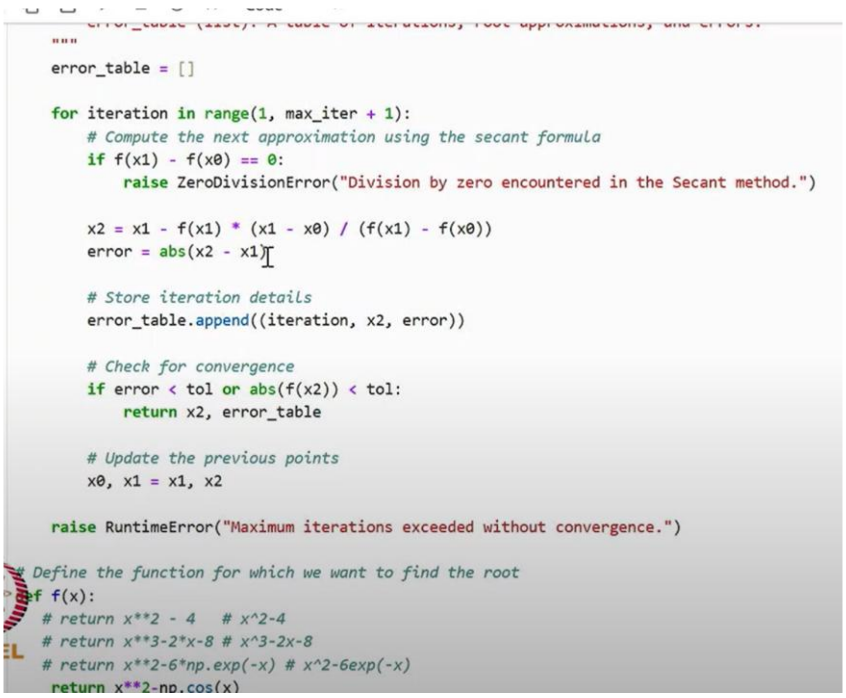
(Refer slide time: 51:48)



```
import numpy as np
import matplotlib.pyplot as plt

def secant_method(f, x0, x1, tol=1e-6, max_iter=100):
    """
    Implements the Secant method to find the root of f(x) = 0.

    Parameters:
        f (function): The function for which the root is to be found.
        x0 (float): The first initial guess.
        x1 (float): The second initial guess.
        tol (float): The tolerance for the stopping criterion.
        max_iter (int): The maximum number of iterations.

    Returns:
        root (float): The approximated root.
        error_table (list): A table of iterations, root approximations, and errors.
    """
    error_table = []

    for iteration in range(1, max_iter + 1):
        # Compute the next approximation using the secant formula
        if f(x1) - f(x0) == 0:
            raise ZeroDivisionError("Division by zero encountered in the Secant method.")

        x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))
```

So our secant method is the initial one. The tolerance, maximum, will be the same. The name of our secant method is just here—that the division one counter should not be zero. The same value should not be the same. We have to take care that both of them do not get the same value. Apart from that, the warning that we had in the previous section—"must be of opposite sign"—we have eliminated that warning. Now our warning is: division is zero. Okay.

(Refer slide time: 52:27)



```
    """
    error_table = []

    for iteration in range(1, max_iter + 1):
        # Compute the next approximation using the secant formula
        if f(x1) - f(x0) == 0:
            raise ZeroDivisionError("Division by zero encountered in the Secant method.")

        x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))
        error = abs(x2 - x1)

        # Store iteration details
        error_table.append((iteration, x2, error))

        # Check for convergence
        if error < tol or abs(f(x2)) < tol:
            return x2, error_table

        # Update the previous points
        x0, x1 = x1, x2

    raise RuntimeError("Maximum iterations exceeded without convergence.")

# Define the function for which we want to find the root
def f(x):
    # return x**2 - 4    # x^2-4
    # return x**3-2*x-8 # x^3-2x-8
    # return x**2-6*np.exp(-x) # x^2-6exp(-x)
    return x**2-np.cos(x)
```

So now we will go on in the same way and keep updating the new values. And after that, we will see the convergence in this case. So now see, I have done this function. Let us see x^2 −

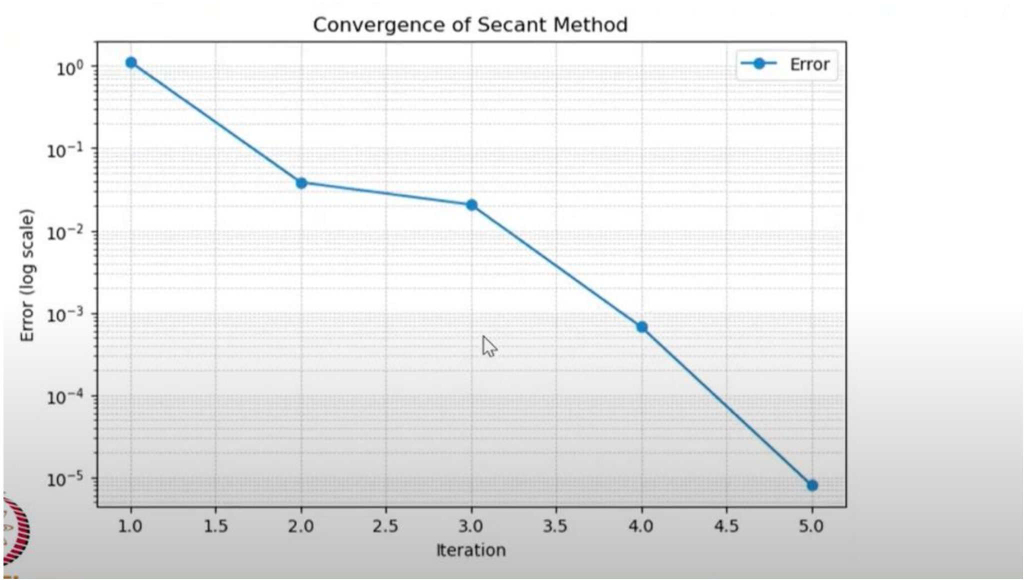
cos (x) which we have just done, and I have taken the initial guess between 0 and 2. I run it again. I ran this, and we got the convergence. So you can see that its solution is 0.8241. And see how fast the convergence has happened.

So this convergence is not linear. First, then second, and third—it is decreasing like this. And in the second iteration, we got this initial guess. See, what did we take—zero and 2. So what did we do in the previous iteration? We took zero and one for this function. See $x^2 - \cos(x)$,what did we take in this? 0 and 2. Let me change it and take 1 and 2, see what happens. See, it still came.

Now you think that one is also positive and two is also positive. If we see its value, then the values that we have of this function are positive at one, and what will happen at two? So, 4 - cos(2). So, both are positive values. So I took both these positive values, and after that I applied the secant method. So what would happen in this case if the same work is done on Regula Falsi? If I had done it in Falsi, it would have given an error. But I did not do so in this—it did not give an error.

Because in the secant method, we have put a condition that the error—that is, the initial guess—need not be of opposite sign. So, we have these roots, and it came like this.

(Refer slide time: 54:48)



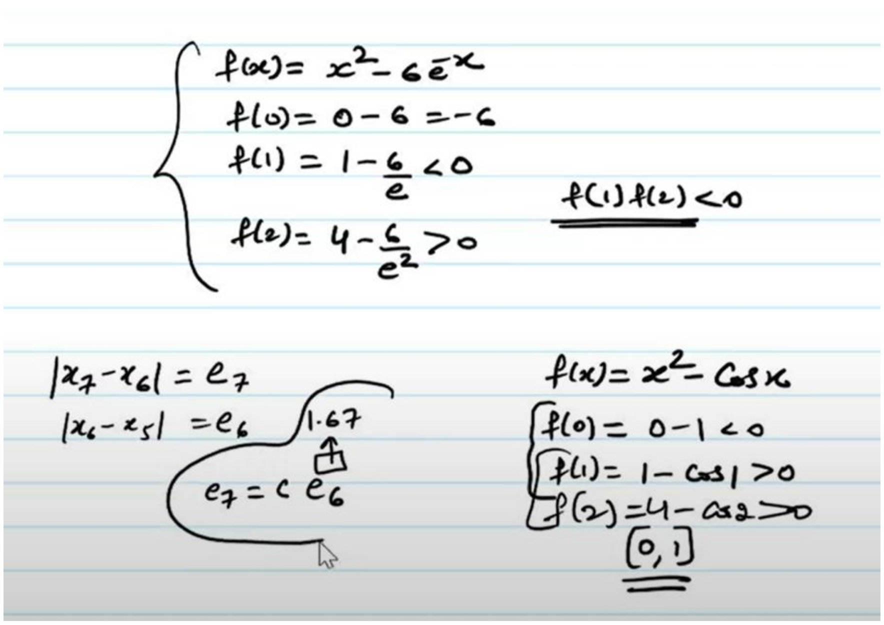
So this secant method converged because the secant method happens—in the first iteration, after one or two iterations—it works like a Regula Falsi type.

So in the same way, we can do the different-different method. I can remove this and take another one. I can take this. Now I do not have to worry whether it is the same or not. I looked at it. It came to 1.28, and we got an error. So the convergence of error, which is the difference of convergence, also depends on the derivative, the value of the function, and what is coming at that point—what is their difference. So I did this. Now I can remove this and do this. I don't know what its root is. See, the root was two, and I took the initial guess. And then the root is 2.33, and it also converged. So what happened from here is that the secant method gave us our convergence. So it happened with this method as well. Now what did I do? Let me change it and take this. We know its root. So what do I do with this? Let me take zero and one. I ran it. I got the answer in the seventh iteration. See, this convergence is giving you the faster convergence in this way. In Regula Falsi, initially it was converging fast. After that, let's not change it. But what is happening in the secant iteration? We are doing that at every step it is converging faster.

So we can check from here that there is nothing such that we cannot know the order of convergence. We can easily find out what we can do—we have taken two values. We can see from here that I have taken the value of, suppose, seventh iteration minus sixth iteration. So we have written it as error. Error at this—if this is a pass, then let me accept this as 7. Then I saw x6 - x5. This error comes 6. So you can see that e7 should be equal to e6, and what are the values coming here?

So that 1.67 , if we go theoretically, then 1.67 will come.

(Refer slide time: 57:41)



$f(x) = x^2 - 6e^{-x}$

$f(0) = 0 - 6 = -6$

$f(1) = 1 - \frac{6}{e} < 0$

$f(2) = 4 - \frac{6}{e^2} > 0$

$\underline{f(1)f(2) < 0}$

$|x_7 - x_6| = e_7$

$|x_6 - x_5| = e_6$

1.67

$e_7 = \boxed{c}\, e_6$

$f(x) = x^2 - \cos x$

$f(0) = 0 - 1 < 0$

$f(1) = 1 - \cos 1 > 0$

$f(2) = 4 - \cos 2 > 0$

$\underline{[0, 1]}$

Okay, it is not one because we can know from the graph that it is not one. See this in the secant. So its values have reduced immediately, then reduced, then reduced. Okay, so the convergence is quite fast.

So today we have discussed the Regula Falsi and secant method, and I have also discussed its Python code. And we have seen that it converges, and it gives the faster convergence basically by the bisection method or the fixed-point method.

So we will do some further Newton Raphson related to this. So thank you for watching this lecture.