Scientific Computing Using Python

Professor. Vivek Aggarwal and Professor. Mani Mehra

Department of Mathematics

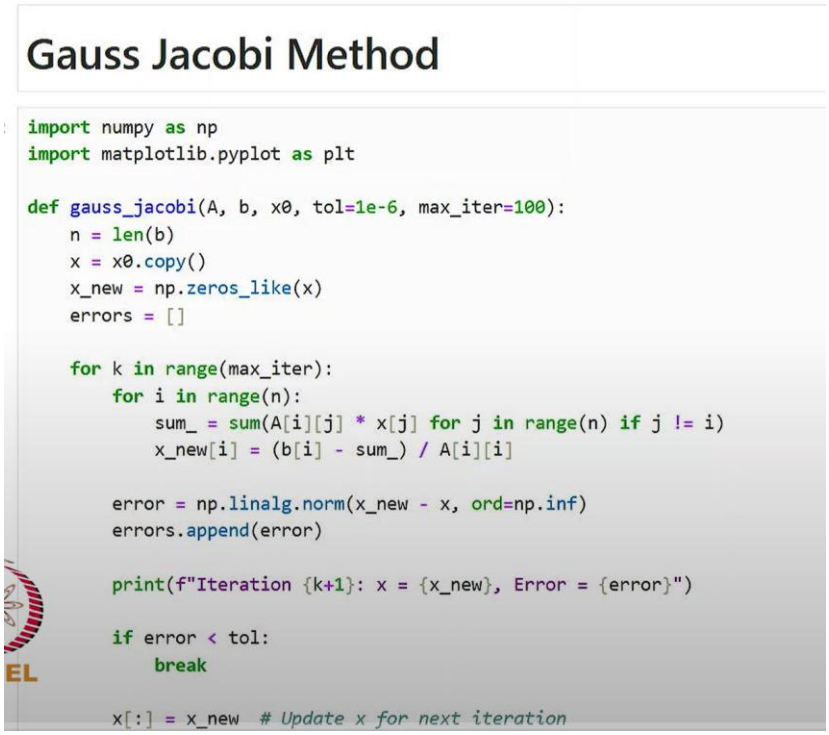
Indian Institute of Technology, Delhi

Lecture No. 17

So, in the previous lecture, we discussed the Gauss Jacobi, Gauss Seidel methods, their convergence, etc. So, today we will start from that. So, let's start.

In the previous lecture, we had discussed till here that we will apply Gauss Jacobi or Seidel method, and after that we will see its convergence matrix, and from there we will come to know. So, now let's discuss this Python code which is made for Gauss Jacobi.

(Refer slide time: 0:53)



See what is happening in Gauss method. We are using NumPy and Matplotlib. So, the tolerance that we have taken is our e to power minus 6. I can convert it and do 0.5 e to power minus 6. I have assumed 100. Let's take it up to 100 maximum. After that, we will apply all these things.

So, what did we do? I defined a subroutine or a function: define Gauss Jacobi. Okay, inside that, what will we input? A matrix b, the right-hand side vector, and x0 will be the initial starting point, and this will be the tolerance or iteration. So, this is an iterative method.

Okay, the length of n will be the length of b. There are that many number of variables. We know, copy x0, and the x_new is x, np.dot.zero_like(x), and error. Okay, so on each of it, the values we have started from zero.

Now what we have to do is, we have to define the range up to the maximum iteration. And on each iteration, we have to find the sum. See, the sum is the right side sum. Okay, and this x_new will be what will be bi minus the sum divided by the diagonal element or diagonal elements. So, from this, we will get x_new, and we will find the norm of error.

So, we used the inbuilt function of linear algebra, norm, so x_new minus the x which we get is infinity norm. We took it in its form. We will get an error, and we will keep writing that error here. In this way, our iteration will be printed in every iteration: what is the new x, error, what is the error.

If the error is less than the tolerance, then we will break and come out directly from the loop. Otherwise, we will go here and put this new in it, and we will keep updating this iteration. Now we know the rate of convergence.

(Refer slide time: 3:22)

```
        print(f"Iteration {k+1}: x = {x_new}, Error = {error}")

        if error < tol:
            break

        x[:] = x_new  # Update x for next iteration

    return x_new, errors

def rate_of_convergence(errors):
    rates = []
    for i in range(1, len(errors) - 1):
        rate = np.log(errors[i+1] / errors[i]) / np.log(errors[i] / errors[i-1])
        rates.append(rate)
    return rates

# Example system
A = np.array([[10, -1, 2, 0],
              [-1, 11, -1, 3],
              [2, -1, 10, -1],
              [0, 3, -1, 8]], dtype=float)
b = np.array([6, 25, -11, 15], dtype=float)
x0 = np.zeros_like(b)

# Solve using Gauss-Jacobi
solution, errors = gauss_jacobi(A, b, x0)

# Compute rate of convergence
roc = rate_of_convergence(errors)
```

That is the error rate of convergence. So, on the basis of the matrix that we have derived, we will find out the rate of convergence.

And here we have taken a matrix. So, like I have taken an array, an np.array, so we have defined this array. We have taken a four by four matrix, and this is our right-hand side vector. So, we have taken the matrix in such a way that it is a diagonal dominant matrix.

So, you see, if I look here, our system will be 10, and that is greater than 1 plus 2, 3, then 11. The next diagonal is also greater than 4. Then this 10 is greater than the sum of the remaining

element. 8 is greater than the sum of the remaining element. So, this matrix that we have taken is a diagonal dominant matrix.

(Refer slide time: 4:21)

```
        rates.append(rate)
    return rates

# Example system
A = np.array([[10, -1, 2, 0],
              [-1, 11, -1, 3],
              [2, -1, 10, -1],
              [0, 3, -1, 8]], dtype=float)
b = np.array([6, 25, -11, 15], dtype=float)
x0 = np.zeros_like(b)

# Solve using Gauss-Jacobi
solution, errors = gauss_jacobi(A, b, x0)

# Compute rate of convergence
roc = rate_of_convergence(errors)
print("Rate of Convergence:", roc)

# Plot the error decay
plt.figure(figsize=(8, 5))
plt.plot(range(1, len(errors) + 1), errors, marker='o', linestyle='-', label='Error')
plt.yscale("log")
plt.xlabel("Iteration")
plt.ylabel("Error (log scale)")
plt.title("Error Decay in Gauss-Jacobi Method")
plt.legend()
plt.grid(True)
plt.show()
```

So, we have taken a diagonal dominant matrix, and we have taken this right side vector. Now, after this, we have to start it with x0, which is the initial guess. We have to give it.

Now what will happen after that? We will call it Gauss Jacobi. From there we will get the solution, and we will get the error. And we will write the rate of convergence here. We will plot it like this.

Now, if I run it—I ran it—so after running it, see, this is what came out.

(Refer slide time: 5:00)

```
plt.legend()
plt.grid(True)
plt.show()

Iteration 1: x = [ 0.6        2.27272727 -1.1        1.875    ], Error = 2.272727272727273
Iteration 2: x = [ 1.04727273  1.71590909 -0.80522727  0.88522727], Error = 0.9897727272727272
Iteration 3: x = [ 0.93263636  2.05330579 -1.04934091  1.13088068], Error = 0.3373966942148763
Iteration 4: x = [ 1.01519876  1.95369576 -0.96810863  0.97384272], Error = 0.15703796487603305
Iteration 5: x = [ 0.9889913   2.01141473 -1.0102859   1.02135051], Error = 0.057718961307287486
Iteration 6: x = [ 1.00319865  1.99224126 -0.99452174  0.99443374], Error = 0.026916770226802944
Iteration 7: x = [ 0.99812847  2.00230688 -1.00197223  1.00359431], Error = 0.010065620869902814
Iteration 8: x = [ 1.00062513  1.9986703  -0.99903558  0.99888839], Error = 0.004705919560371252
Iteration 9: x = [ 0.99967415  2.00044767 -1.00036916  1.00061919], Error = 0.001777370422652913
Iteration 10: x = [ 1.0001186   1.99976795 -0.99982814  0.99978598], Error = 0.0008332116799193434
Iteration 11: x = [ 0.99994242  2.00008477 -1.00006833  1.0001085 ], Error = 0.00032252355187700754
Iteration 12: x = [ 1.00002214  1.99995896 -0.99996916  0.99995967], Error = 0.00014883338027238402
Iteration 13: x = [ 0.99998973  2.00001582 -1.00001257  1.00001924], Error = 5.957571911897833e-05
Iteration 14: x = [ 1.00000409  1.99999268 -0.99999444  0.9999925 ], Error = 2.6746168130586945e-05
Iteration 15: x = [ 0.99999816  2.00000292 -1.0000023   1.00000344], Error = 1.0943616738257056e-05
Iteration 16: x = [ 1.00000075  1.99999868 -0.99999899  0.99999862], Error = 4.825886316339734e-06
Iteration 17: x = [ 0.99999967  2.00000054 -1.00000042  1.00000062], Error = 2.0031180858470776e-06
Iteration 18: x = [ 1.00000014  1.99999976 -0.99999982  0.99999975], Error = 8.730860806549501e-07
Iteration 19: x = [ 0.99999994  2.0000001  -1.00000008  1.00000011], Error = 3.6581100559818225e-07
Rate of Convergence: [1.2946795842361207, 0.7106118089659382, 1.3087588693537238, 0.7621488718457785, 1.2894298344284432, 0.7729627560218412, 1.280652892
0911557, 0.7780766746646296, 1.2527829878148828, 0.8148133099490895, 1.1839152046377628, 0.8746996153848269, 1.1158484025826636, 0.9162152820067161, 1.07
38925055083792, 0.9444286866605083, 1.0475553412491385]
```

Now, in the first iteration, the error came out to be 0.6, 2.27. Okay, -1.1 and 1.87, and the error is 2.27. Okay, we started with 0,0,0.

In the second iteration, the error improved here. See, it was 2, it got reduced and reached here. It came close to 0.1. Okay. Then we took the third iteration, then it will be reduced. In the fourth iteration, it will be reduced. It got reduced in the fifth iteration. So, from the error, we can come to know that it is getting reduced.

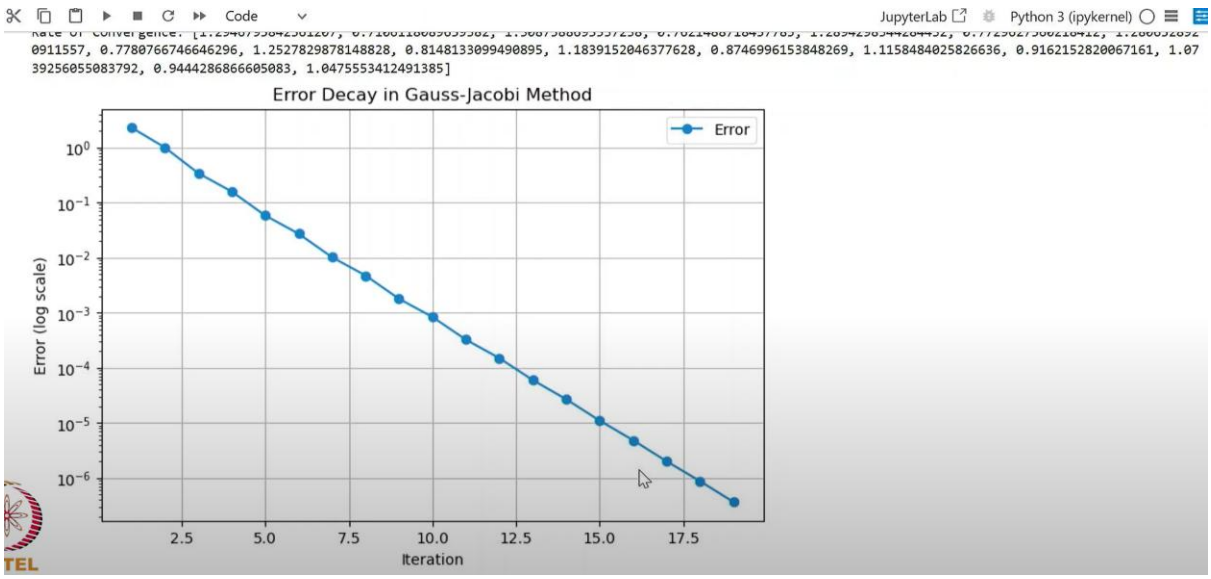
Now, see, 5 into 9 5 10 power minus 5 has gone up to this. So, the tolerance that we have given in it is 5 in 10 power minus 6, which we have given above. So, it means that our sum should be up to six digits. Only then we will stop.

So, we looked at this thing, and we see the error from here. Here it is 8.73, then it became 3.765, and the convergence, the solution that we got, we got this. See, there is one in it. 1, 1.99, okay, it became -999, and in the next, this became two, this became minus one, this became one.

So, the infinity value that we have applied between these two—that infinity norm—is now less than the tolerance. So, just like we have applied the tolerance, what did we get? Our solution came in 19 iterations, and this rate of convergence is 1.29, 0.79. After changing like this, we saw that it is almost near one.

The convergence rate is decreasing at some places, but mostly it is going around one—1, 0..87, then 1.1, then 0.9. We plotted it like this, and we saw that this is the convergence rate. Look, this shows that the convergence of this method is linear.

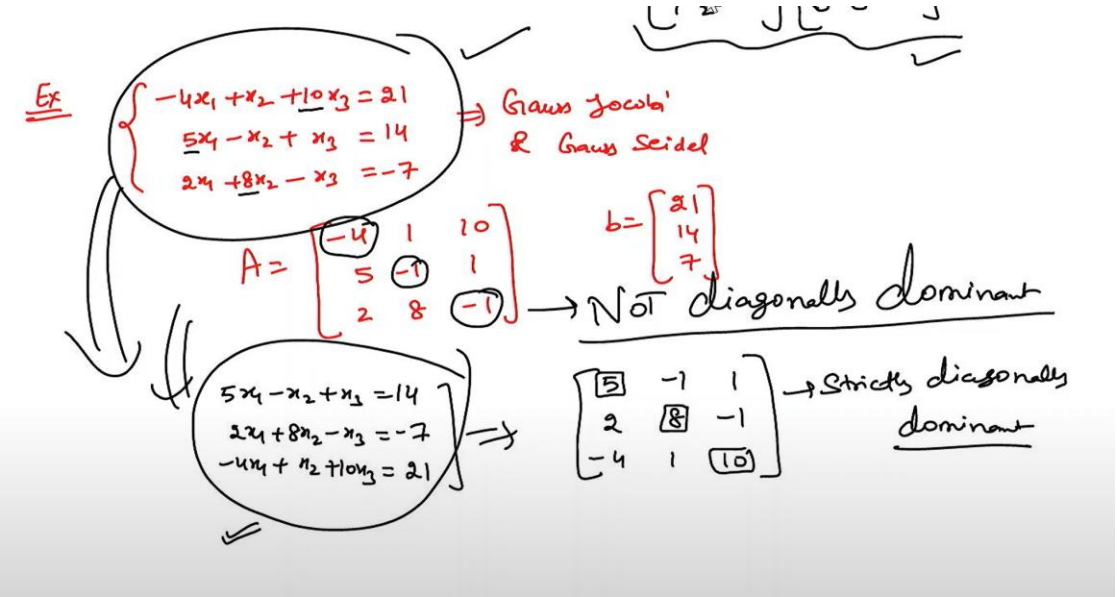
(Refer slide time: 7:18)



The order of convergence of this method is linear. How its iteration is increasing here, and the error here is getting reduced linearly. So, we will call this method that this method gives linear convergence.

So now, we have taken the matrix that was diagonal dominant. This is beneficial for us in this case. Now in the previous example, we had—where did I do this? So, let's solve this. First, I take this from both ways and then I take this. Then let's see what's happening in this case.

(Refer slide time: 7:58)



Okay, so what did I do to this? I'll convert it—Control + C, Control + V. Okay, I'll comment on this. Now what are we doing? I'm taking this -4. Okay, 1, 10. Now I'm taking the next one. I'm 5. After that, -1. After that, 1. Okay, I'm taking 3 by 3 system. After that, I took 2 and 8, and after that came -1. Okay, this. And I deleted the last one. this went, this went, and this one more. Okay, this came, and the right-hand side vector was 21. And I'll write here 14, 14, and here we have -7. Okay, so this matrix is not our diagonal dominant matrix.

(Refer slide time: 10:05)

```
# Example system
#A = np.array([[10, -1, 2, 0],
#               [-1, 11, -1, 3],
#               [2, -1, 10, -1],
#               [0, 3, -1, 8]], dtype=float)
A = np.array([[-4, 1, 10],
              [5, -1, 1],
              [2, 8, -1]], dtype=float)
b = np.array([21, 14, -7], dtype=float)
x0 = np.zeros_like(b)

# Solve using Gauss-Jacobi
solution, errors = gauss_jacobi(A, b, x0)

# Compute rate of convergence
roc = rate_of_convergence(errors)
print("Rate of Convergence:", roc)

# Plot the error decay
plt.figure(figsize=(8, 5))
plt.plot(range(1, len(errors) + 1), errors, marker='o', linestyle='-', label='Error')
plt.yscale("log")
plt.xlabel("Iteration")
plt.ylabel("Error (log scale)")
plt.title("Error Decay in Gauss-Jacobi Method")
plt.legend()
plt.grid(True)
plt.show()
```

I'll run it and see what comes out. So see.

(Refer slide time: 10:11)

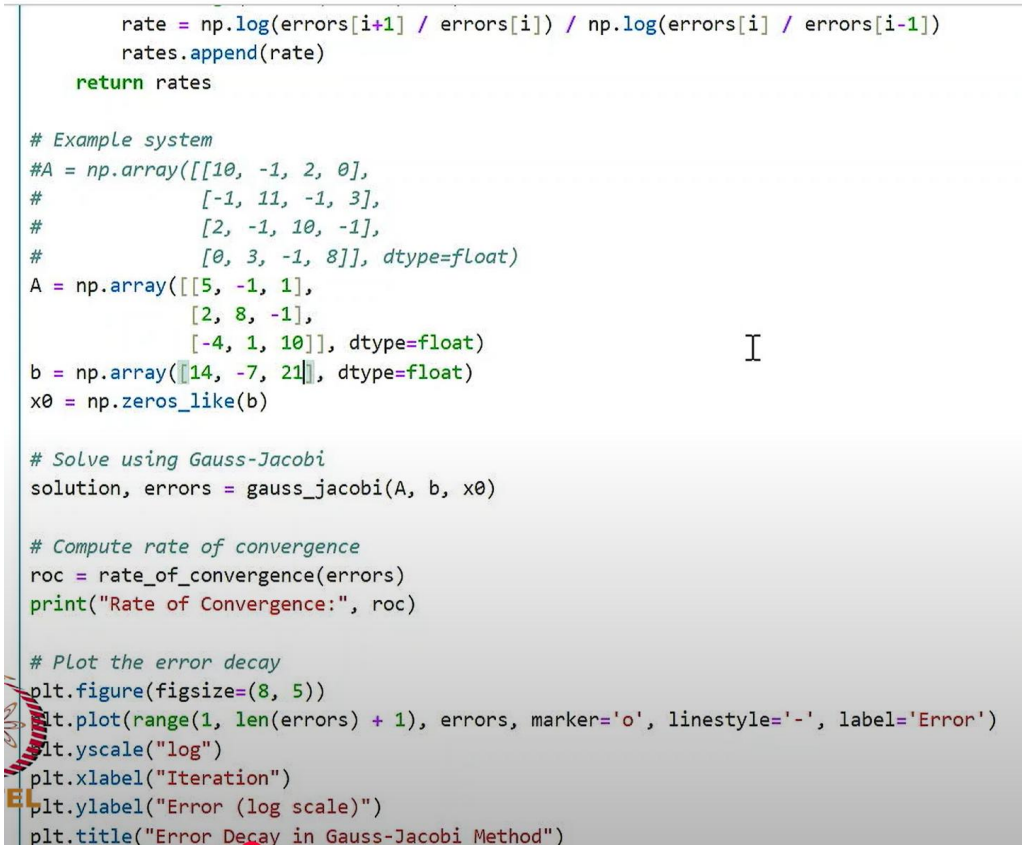
```
# Plot the error decay
plt.figure(figsize=(8, 5))
plt.plot(range(1, len(errors) + 1), errors, marker='o', linestyle='-', label='Error')
plt.yscale("log")
plt.xlabel("Iteration")
plt.ylabel("Error (log scale)")
plt.title("Error Decay in Gauss-Jacobi Method")
plt.legend()
plt.grid(True)
plt.show()
```

```
Iteration 1: x = [ -5.25 -14.     7.  ], Error = 14.0
Iteration 2: x = [   8.75  -33.25 -115.5 ], Error = 122.5
Iteration 3: x = [-302.3125  -85.75   -241.5   ], Error = 311.0625
Iteration 4: x = [ -630.4375 -1767.0625 -1283.625 ], Error = 1681.3125
Iteration 5: x = [ -3656.078125  -4449.8125   -15390.375   ], Error = 14106.75
Iteration 6: x = [-39593.640625 -33684.765625 -42903.65625 ], Error = 35937.5625
Iteration 7: x = [-115685.58203125 -240885.859375   -348658.40625   ], Error = 305754.75
Iteration 8: x = [ -931872.73046875  -927100.31640625 -2158451.0390625 ], Error = 1809792.6328125
Iteration 9: x = [-5627907.92675781 -6817828.69140625 -9280540.9921875 ], Error = 7122089.953125
Iteration 10: x = [-24905814.90332031 -37420094.62597656 -65798438.38476562], Error = 56517897.392578125
Iteration 11: x = [-1.73851125e+08 -1.90327527e+08 -3.49172380e+08], Error = 283373941.4296875
Iteration 12: x = [-9.20512837e+08 -1.21842802e+09 -1.87032246e+09], Error = 1521150078.1333008
Iteration 13: x = [-4.98041315e+09 -6.47288665e+09 -1.15884498e+10], Error = 9718127353.327148
Iteration 14: x = [-3.05893462e+10 -3.64905156e+10 -6.17439195e+10], Error = 50155469727.08313
Iteration 15: x = [-1.63482428e+11 -2.14690651e+11 -3.53102817e+11], Error = 291358897637.3086
Iteration 16: x = [-9.36429706e+11 -1.17051496e+12 -2.04449006e+12], Error = 1691387242620.313
Iteration 17: x = [-5.40385389e+12 -6.72663859e+12 -1.12369791e+13], Error = 9192488998903.5
Iteration 18: x = [-2.97741073e+13 -3.82562485e+13 -6.46208165e+13], Error = 53383837419888.97
```

This error is increasing. You see, before this, there was 14 error, then 121 then 311. Like this, the error is increasing, and the error keeps increasing in this iteration. Okay, so what happened in this—we had given 100 iteration maximum and the error increased by this much. So what happened in this method? What will we call this method? We will call this method that the method did not converge; it diverged in this case because the guarantee of convergence was there. So, we changed it. So the matrix that is there—was not diagonal dominant matrix. So if we did not take it, then it may converge or it may not.

So what will happen now in this case—what does it mean? That now I can try to make this matrix diagonal dominant. So, as we did, we made it diagonally dominant. How did we make it diagonally dominant? We did it by swapping it. So what did we do? We swapped the rows. So we brought the one with 5 here, we brought the one with 2 here, we brought the one with -4 here. So we do the same thing in this method. So I wrote the one with 5 here as 5, -1, 1. After that, 2, 8, -1, 2, 8, -1. And this is the biggest one—we will take minus 4, 1, 10. And this will also change—it became 14, it became -7, and here it became 21. It came, right?

(Refer slide time: 12:19)

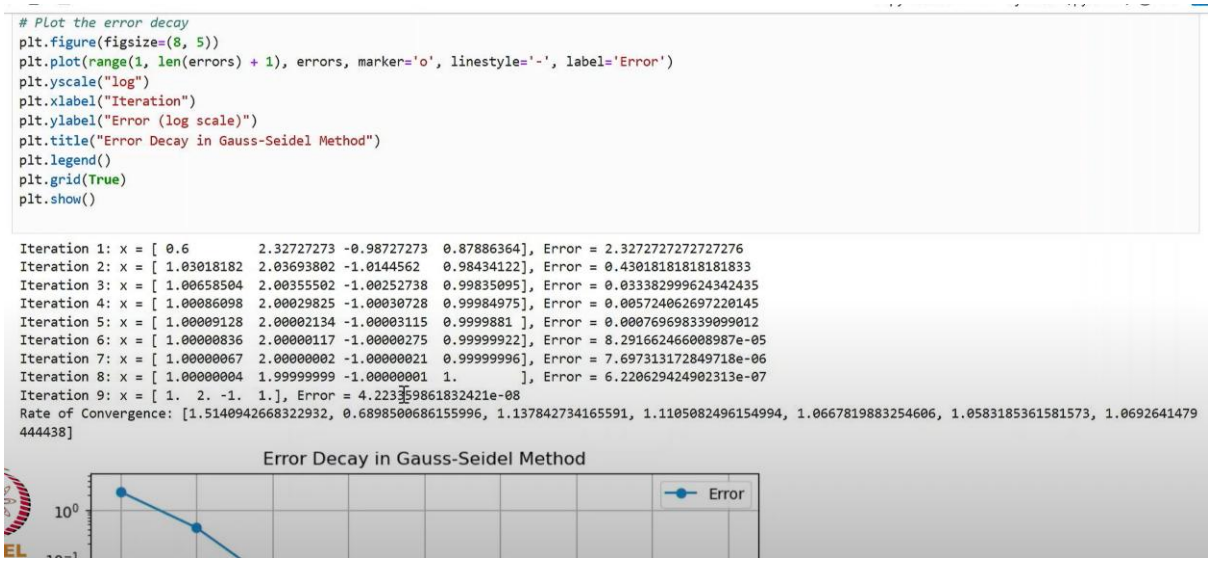


```
        rate = np.log(errors[i+1] / errors[i]) / np.log(errors[i] / errors[i-1])
        rates.append(rate)
    return rates

# Example system
#A = np.array([[10, -1, 2, 0],
#              [-1, 11, -1, 3],
#              [2, -1, 10, -1],
#              [0, 3, -1, 8]], dtype=float)
A = np.array([[5, -1, 1],
              [2, 8, -1],
              [-4, 1, 10]], dtype=float)
b = np.array([14, -7, 21], dtype=float)
x0 = np.zeros_like(b)

# Solve using Gauss-Jacobi
solution, errors = gauss_jacobi(A, b, x0)

# Compute rate of convergence
roc = rate_of_convergence(errors)
print("Rate of Convergence:", roc)

# Plot the error decay
plt.figure(figsize=(8, 5))
plt.plot(range(1, len(errors) + 1), errors, marker='o', linestyle='-', label='Error')
plt.yscale("log")
plt.xlabel("Iteration")
plt.ylabel("Error (log scale)")
plt.title("Error Decay in Gauss-Jacobi Method")
```

So we have made this matrix diagonally dominant. Now let's see. Now I ran it. See, the solution came. After 18 errors, the convergence came and the solution came—it was 1, -1, 3. So in that case, it was not converging, but it did converge. And see, its convergence is linear. So, as the error iteration is increasing, the errors are decreasing. So in this case, we will say that our matrix was diagonal dominant. And because this is a sufficient condition for us, so what happened with this—we got the solution from Gauss Jacobi, right? So Gauss Jacobi also gave us the solution.

We can do the same thing with Gauss Seidel also. So I do it with Gauss Seidel. I will do this—Gauss Seidel, I have taken out the Gauss Seidel. Okay, so in Gauss Seidel also, I convert it by 0.5. After that, we had taken the same matrix earlier. So now let us see it. This convergence has come. In the previous case, which was coming in 19 iterations, it has come in 9 iteration.

(Refer slide time: 13:36)



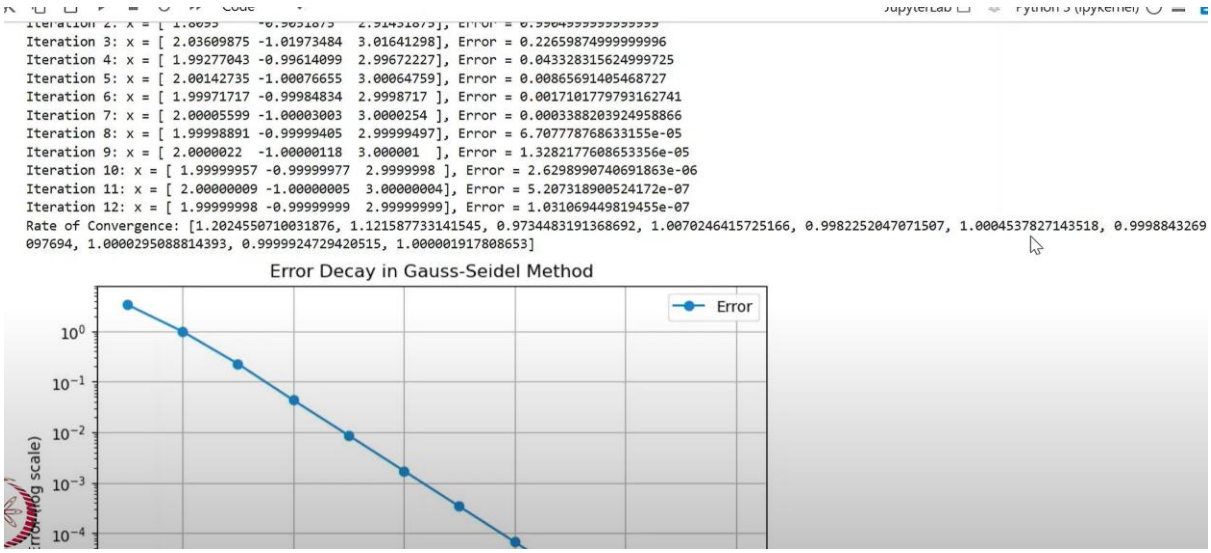

```
# Plot the error decay
plt.figure(figsize=(8, 5))
plt.plot(range(1, len(errors) + 1), errors, marker='o', linestyle='-', label='Error')
plt.yscale("log")
plt.xlabel("Iteration")
plt.ylabel("Error (log scale)")
plt.title("Error Decay in Gauss-Seidel Method")
plt.legend()
plt.grid(True)
plt.show()
```

```
Iteration 1: x = [ 0.6         2.32727273 -0.98727273  0.87886364], Error = 2.3272727272727276
Iteration 2: x = [ 1.03018182  2.03693802 -1.0144562   0.98434122], Error = 0.4301818181818833
Iteration 3: x = [ 1.00658504  2.00355502 -1.00252738  0.99835095], Error = 0.033382999624342435
Iteration 4: x = [ 1.00086098  2.00029825 -1.00030728  0.99984975], Error = 0.005724062697220145
Iteration 5: x = [ 1.00009128  2.00002134 -1.00003115  0.9999881 ], Error = 0.000769698339099012
Iteration 6: x = [ 1.00000836  2.00000117 -1.00000275  0.99999922], Error = 8.291662466008987e-05
Iteration 7: x = [ 1.00000067  2.00000002 -1.00000021  0.99999996], Error = 7.697313172849718e-06
Iteration 8: x = [ 1.00000004  1.99999999 -1.00000001  1.        ], Error = 6.220629424902313e-07
Iteration 9: x = [ 1.  2. -1.  1.], Error = 4.223359861832421e-08
Rate of Convergence: [1.5140942668322932, 0.6898500686155996, 1.137842734165591, 1.1105082496154994, 1.0667819883254606, 1.0583185361581573, 1.0692641479
444438]
```

And this convergence of this is also linear, right? But it is a little faster as compared to the Gauss Jacobi . So, in 9 iteration we got the solution. We had said that there is not much difference between Gauss Jacobi and Seidel. We just know what it is doing in Seidel, and it is using the updated value. So, the updated value is being used. Hence, the solution that we have is very easy, but it is not easy, but we are getting it in a faster way. So, we took it like this.

And in this case, we saw that our matrix is diagonal dominant. So, it is done. Now I will see this, check it and try. So, let me see what the second matrix became. Now I have taken this. Okay, and I will copy it from here. It is here. Now let us solve this. Let us see. We got the solution in 12 iterations, and it took 18 iterations to solve that. See, after the same 18 iterations, the solution came 1, -1, 3, and in this, the 1, -1, 3 is where we have it. Basically, it is 2, -1, 3. We got the solution in 12 iterations, and the rate of convergence is around, order of convergence is near 1 in this case.

(Refer slide time: 15:46)




```
Iteration 3: x = [ 2.03609875 -1.01973484  3.01641298], Error = 0.22659874999999996
Iteration 4: x = [ 1.99277043 -0.99614099  2.99672227], Error = 0.043328315624999725
Iteration 5: x = [ 2.00142735 -1.00076655  3.00064759], Error = 0.00865691405468727
Iteration 6: x = [ 1.99971717 -0.99984834  2.9998717 ], Error = 0.0017101779793162741
Iteration 7: x = [ 2.00005599 -1.00003003  3.0000254 ], Error = 0.0003388203924958866
Iteration 8: x = [ 1.99998891 -0.99999405  2.99999497], Error = 6.707778768633155e-05
Iteration 9: x = [ 2.0000022  -1.00000118  3.000001  ], Error = 1.3282177608653356e-05
Iteration 10: x = [ 1.99999957 -0.99999977  2.9999998 ], Error = 2.6298990740691863e-06
Iteration 11: x = [ 2.00000009 -1.00000005  3.00000004], Error = 5.207318900524172e-07
Iteration 12: x = [ 1.99999998 -0.99999999  2.99999999], Error = 1.031069449819455e-07
Rate of Convergence: [1.2024550710031876, 1.121587733141545, 0.9734483191368692, 1.0070246415725166, 0.9982252047071507, 1.0004537827143518, 0.9998843269
097694, 1.0000295088814393, 0.9999924729420515, 1.000001917808653]
```

Okay, and both the methods are linear methods, and it depends on how fast the difference of the matrix is decreasing. So now, from the work that we have done, we have come to know

that the method is converging if our matrix is diagonally dominant. Then now we see what is happening with our correspondingly convergence matrix. How many eigenvalues does it have? We will check this. So we have written a code for Gauss Jacobi and Gauss Seidel's which is based on their convergence matrix. So we first decomposed the matrix as we did in the form of L D U. We got D, diagonal elements, triangular matrix lower and this upper. So, we converted it. In Gauss Jacobi, we know that this is one of our matrices. This is the convergence matrix. In Gauss seidel, this is the convergence matrix. Okay, we found the spectrum radius and we did this.

So now see, we first -4, 1, 10 solved this, which I just did. So we ran it. Then it is a lower triangular, D and this.

(Refer slide time: 17:08)

```
# Rate of convergence (approximate)
rate_J = -np.log(rho_J) if rho_J < 1 else float('inf')
rate_G = -np.log(rho_G) if rho_G < 1 else float('inf')

print(f"\nRate of Convergence (approximate) for Gauss-Jacobi: {rate_J:.4f}")
print(f"Rate of Convergence (approximate) for Gauss-Seidel: {rate_G:.4f}")
```

```
L:
 [[0. 0. 0.]
 [5. 0. 0.]
 [2. 8. 0.]]
D:
 [[-4.  0.  0.]
 [ 0. -1.  0.]
 [ 0.  0. -1.]]
U:
 [[ 0.  1. 10.]
 [ 0.  0.  1.]
 [ 0.  0.  0.]]

Gauss-Jacobi Iteration Matrix:
 [[0.   0.25 2.5 ]
 [5.   0.   1.  ]
 [2.   8.   0.  ]]

Gauss-Seidel Iteration Matrix:
 [[  0.     0.25   2.5 ]
 [  0.     1.25  13.5 ]
```

And now we saw that the Gauss Jacobi iterative matrix is formed and Gauss seidel is formed and found the spectrum radius. So see what came out. 5 came out, and its value came out to be 1.14. See, that is why it was not able to converge. So, will the rate of convergence come to infinity. So, what happened in this—our method was not converging at all because we knew that the spectrum radius should be less than one.

(Refer slide time: 17:38)

```
 [[-4.  0.  0.]
 [ 0. -1.  0.]
 [ 0.  0. -1.]]
U:
 [[ 0.  1. 10.]
 [ 0.  0.  1.]
 [ 0.  0.  0.]]

Gauss-Jacobi Iteration Matrix:
 [[0.   0.25 2.5 ]
 [5.   0.   1.  ]
 [2.   8.   0.  ]]

Gauss-Seidel Iteration Matrix:
 [[  0.     0.25   2.5 ]
 [  0.     1.25  13.5 ]
 [  0.    10.5  113.  ]]

Spectral Radius of Gauss-Jacobi Matrix: 5.657965150125579
Spectral Radius of Gauss-Seidel Matrix: 114.25437619999015

Rate of Convergence (approximate) for Gauss-Jacobi: inf
Rate of Convergence (approximate) for Gauss-Seidel: inf
```

So, if our matrix was not diagonal dominant, then that did not converge. And if it did not converge, then one of its conditions was that the eigenvalues would become greater than one. So, definitely it will go towards infinity. So, what happened in this case is that it did not converge.

Now I have written this matrix in this form. Okay, now whatever work we have done, I will write it like this. I had commented it out, and I removed it.

(Refer slide time: 18:19)

```
    return T_G

def spectral_radius(T):
    """ Computes the spectral radius (max absolute eigenvalue) of matrix T """
    eigenvalues = np.linalg.eigvals(T)
    return max(abs(eigenvalues))

# Example usage
A = np.array([[5, -1, 1],
              [2, 8, -1],
              [-4, 1, 10]], dtype=float)  # Example matrix
#A = np.array([[-4, 1, 10],
#              [5, -1, 1],
#              [2, 8, -1]], dtype=float)  # Example matrix


L, D, U = decompose_matrix(A)

print("L:\n", L)
print("D:\n", D)
print("U:\n", U)

T_J = gauss_jacobi_matrix(L, D, U)
T_G = gauss_seidel_matrix(L, D, U)

print("\nGauss-Jacobi Iteration Matrix:\n", T_J)
print("\nGauss-Seidel Iteration Matrix:\n", T_G)
```

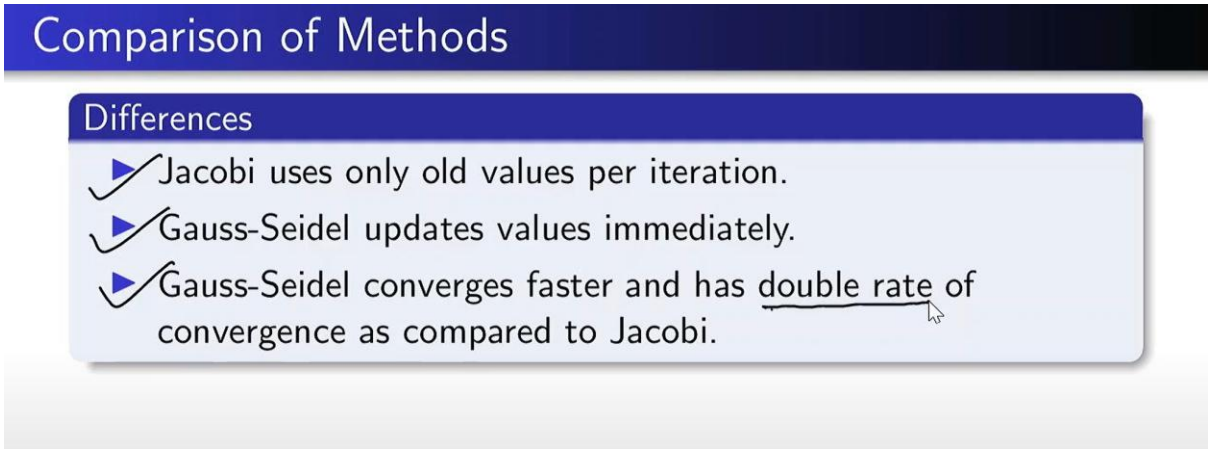
There is no use of the right-hand side vector, so we made this. Now this is the diagonal dominant matrix. Let me see what will happen. Look, L, D, U matrix of Gauss Jacobi also has been formed—this. And Gauss Seidel—this has been formed, which is our convergence matrix.

Now look, now if you see the eigenvalue, then what is its eigenvalue? Maximum is 0.37, and Gauss Seidel's is minus 0.19. So look, if we see the rate of convergence, then Gauss Jacobi is coming close to 0.97 and Gauss Seidel's is coming to 1.6195. So it is almost double. So we can say that Gauss Seidel, which its rate of convergence is two times. Right? The C is two times, meaning that almost all the Gauss jacobi present in the cell, if you take it two times, then it comes in the Gauss seidel. But it depends on what is its largest eigenvalue.

Now see, the largest eigenvalue was 0.193 in the Gauss seidel, so its order of convergence increased, minus the log of that. Right? And like I had 0.37, then 0.97 came there. So if we take any such matrix, if we take the system, its corresponding Gauss jacobi and Gauss seidel convergence matrix, iterative matrix that we can write, can check there eigenvalues. So, they converge than definitely it will be less than one—if it is diverging, then you will see that it is greater than one. So, we can do it in this way.

So, what does this mean? That whatever we did in theory, we have seen it practically with Python that it is happening like this. If the matrix is not convergent—meaning it is not diagonal dominant—then this is happening in this case. So, if we compare what will happen to both the methods, then in comparison, we can say that Bhaiya Gauss Jacobi also using only old values on iteration, Gauss Seidel updates value immediately, and this Gauss Seidel converges faster and has double rate. As I have written, double rate is of convergence as compared to the Jacobi.

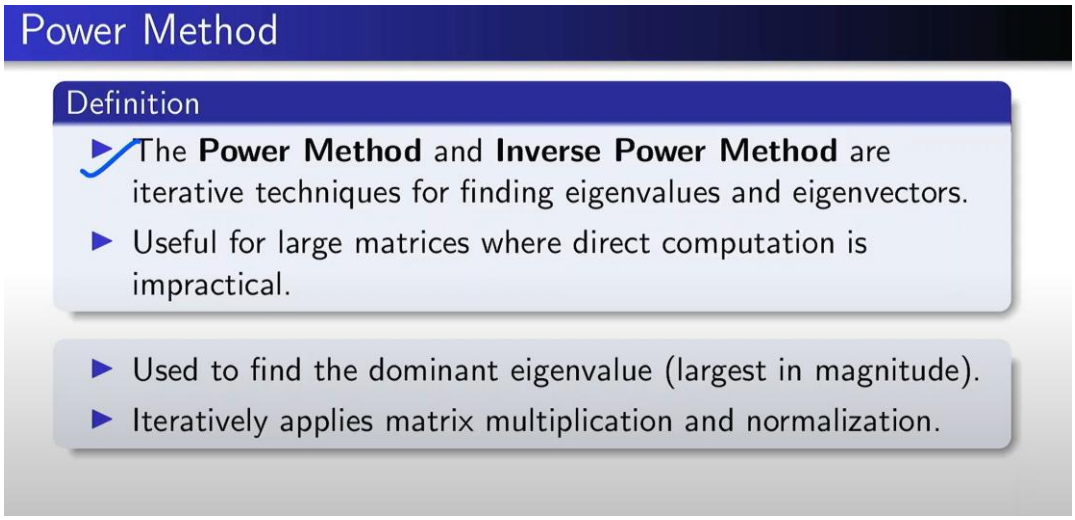
(Refer slide time: 20:47)



So, it will always be double if our matrix is diagonally dominant. Then, after that, we will see what are the eigenvalues coming, and we will solve it from there. So, this work we have done—this Gauss Jacobi and Gauss Seidel—is done. Now, if you see what is happening in this case, which values—one is telling us diagonal dominance and, secondly, we need spectral radius in this. So, spectral radius means now what we have to do is we have to find the eigenvalues, and the eigenvalue is also spectral radius is largest eigenvalue.

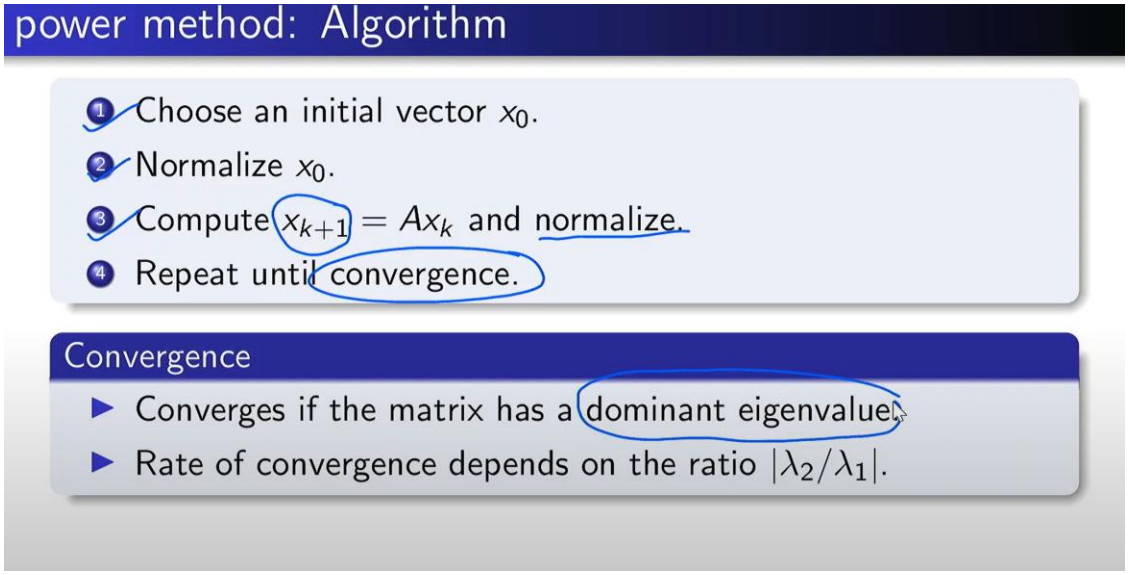
So, now what we have to do—we have to find the eigenvalue, but we have to find the largest eigenvalue. So, it means we will discuss a method which will give the largest eigenvalue. The name is power method. So, what is power method? So, what is this power method? Power method and inverse power method are iterative techniques for finding eigenvalues and eigenvectors. So, this gives us eigenvalues and eigenvectors. Power method is useful for large matrices where direct computation is impractical.

(Refer slide time: 22:23)



That is our main work. We are doing computationally. So, basically, we have to solve large matrices. Our main purpose is that—how can we solve our large matrices? Now, what is this? We use it to find dominant initial value. See, dominant largest magnitude means we are going towards the spectrum radius. And what is this? Iteratively applies matrix multiplication. This work that we are doing will require two operations. One operation will be normalization. We say that the largest value of the vector should be one, and it cannot be larger than that. And the matrix multiplication will be used in this, so we call it power method. Now, let's see what is power method. So, what happens in power method is iterative method. First, we will give initial vector, and we will normalize that vector. Then, we will put it in iteration and put new value, and we will repeat this. The value of x k+1will come, and we will normalize that too. So, we will keep repeating like this, and until we get its value, it will converge. If the matrix is dominant eigenvalue—so, if the eigenvalues are separated and different—then the method we will see in it will work very well.

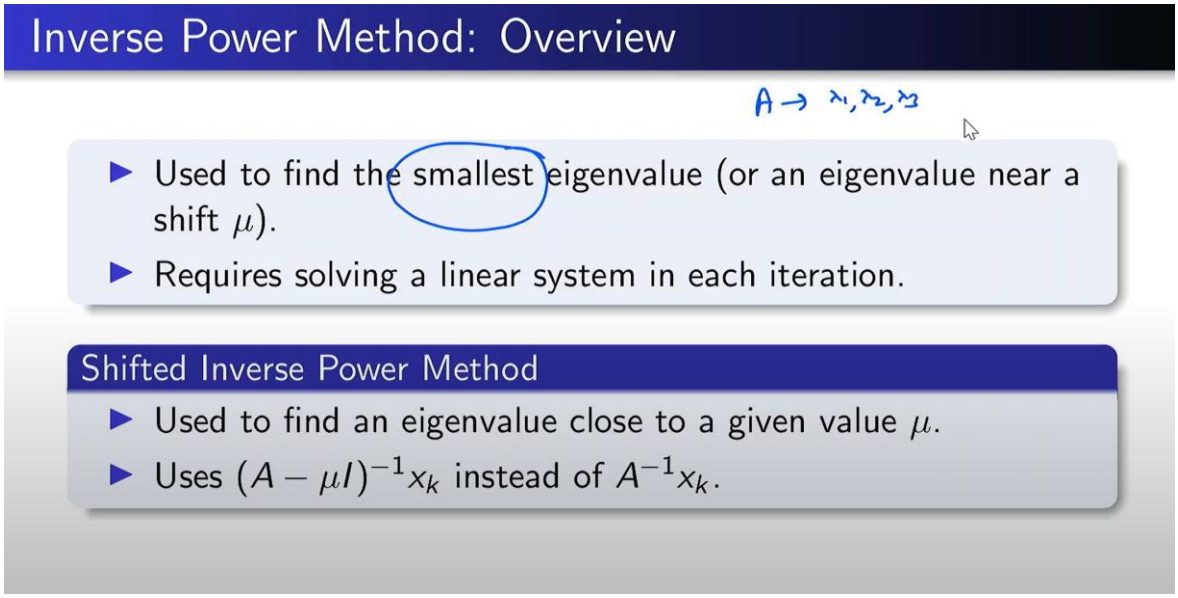
(Refer slide time: 24:05)



And the rate of conversion depends on the ratio, that is, the ratio between the largest eigenvalue and the next eigenvalue. And what is it? If the ratio is very high and the distance between them is very high, then the convergence will be faster. If the eigenvalues are very close, then their value will be—the rate of convergence will be very less. So now, we will see how the power method will be used. Now, we have found the largest initial value.

Now, if someone asks us to give them the smallest eigenvalue, then what do we do for the smallest eigenvalue? The inverse power method is that we know that the eigenvalue of matrix A is something like this: λ1, λ2, λ3 and suppose 3 by 3 matrix, its eigenvalues are these.

(Refer slide time: 25:05)



So if we have A inverse, then its eigenvalues will be 1 by λ1, 1 by λ2, and 1 by λ3. Okay. So its eigenvalues get reciprocal in this manner. So if we want to find the smallest eigenvalue then we apply power method on A inverse. So its largest eigenvalue will be smallest eigenvalue of A. This is called inverse Power method. Inverse power method can be written

in one another different way, Shifted inverse power method. So, we found the largest and also found the lowest one. Now, if someone tells us that the matrix is A and suppose it is 4 cross 4, then its four eigenvalues are $\lambda1, \lambda2, \lambda3, \lambda4$. We applied the power method, so from here we have this—suppose largest. I write it like this. I found it out from the inverse. So, we have two. Now, if someone tells us to find out $\lambda2$ and $\lambda3$ as well, how do you find it? We call it—how we can find out the eigenvalue. For that, we use shifted inverse power.

So, what is shift? Suppose we have to reach $\lambda2$. So, what will we do with our matrix? We don't know what are $\lambda2$ and $\lambda3$. We don't know anything. We just found out that our eigenvalue is between $\lambda1$ and λn. Suppose this was the largest and this was the smallest. So, our eigenvalue will come somewhere in between these. So, what do we do? We choose a random number. That is, I will call that mu. Let me write it, A minus mu. So, A minus mu I. So, we know that if A minus mu i, then its eigenvalues are also the same as that of one, and the values will just be shifted, mu, and the eigenvectors will be the same. So, we will write it like this, okay?

So, it will become λ minus mu x. So, this is what we have to do. So, let's see how to do it. The first thing is that we try to understand a little bit about how the power method works. So, see, what do we have to do in the power method? We have to find out the largest eigenvalue first. Then, after that, we will use that and find out the smallest. So, let's see. Let's first see how the power method will work.

So, what is in the power method? Let's assume that there is a matrix. We have A, n cross n matrix, okay? And this matrix has n eigenvalues, and it can be repeated as well. But we assume that a is an eigenvalue, and suppose we write the eigenvalue like this: $\lambda1, \lambda2, \lambda3$ like this, okay? So, we assume that we have written the eigenvalue like this. We can write the values in such an order that if this is the eigenvalues of A, then we will get n eigenvalues. So, what is happening here? What is our procedure to do now?

So, we are assuming that our matrix is A. Let the matrix be A, and n cross n has n linear independent eigenvectors. So, the eigenvectors are linearly independent. Some may be repeated, but our eigenvectors—if we see here, if any value is getting repeated, then the equality sign will come. If it is getting repeated somewhere, then it is certain that we will get n linearly independent eigenvectors. This means that the matrix is diagonalizable. Basically, we are getting n linearly independent eigenvectors.

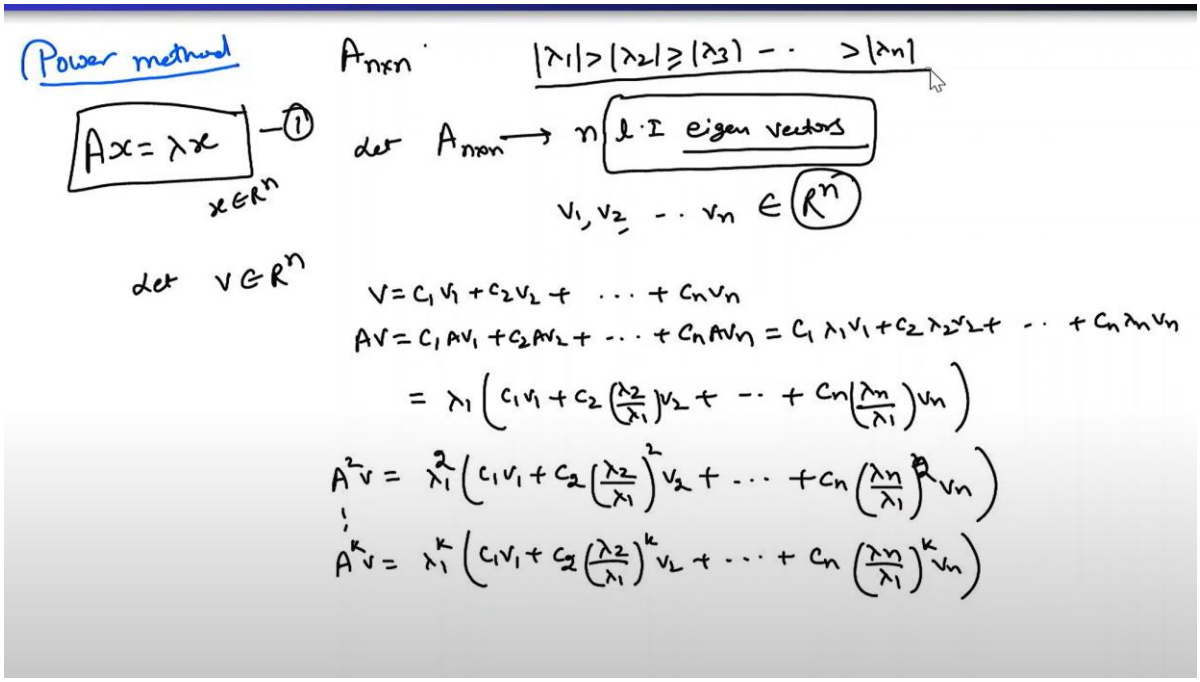
So we assume this eigenvector to be the eigenvalue vector. So, v1, v2, vn all eigenvectors will be included. Now what do we do? Our purpose is to solve $Ax = \lambda x$ and find out the λ. So let's name it, okay? So what do we do for this now? Let's choose a vector v that belongs to Rn. A solution which is x, x also belongs to Rn. We know that we have chosen a vector in Rn. So, we can write it as c1 v1+ c2 v2,..cn vn, because the basis will be of Rn. A. So we can represent it uniquely. The c1, c2, cn will be unique in this case.

Now what do I do? If I apply A of v, it will come out to be c of v1, c2 A of v2, cn A of vn. Now what is vn? What is the v1? It is eigenvalue. So what does it mean? I can write it like this: c1 $\lambda1$ v1, plus c2 $\lambda2$ v2, cn λn. It will come out to be okay. Now what do I do? I take out the common $\lambda1$ from this. It becomes c1 v1 plus c2 $\lambda2$ over $\lambda1$ v2, cn λn over $\lambda1$vn we have it now. What do I do then? I apply it. So if you see this from A, then this will come. It will

become the square of λ1, inside will come: sieve c1 v1, c2 λ2 over λ1 square v2. In the same way we applied it. We did not do much in it. We just applied it. This came, okay?

If I go on, then I write some suppose A k v, what will that be λ1 k c1 v1 plus c2 λ2 over λ1, k v2, cn λn over λ1, the power of k will come to us, okay? We will keep going like this. If we see in this case, now see, the λ which was bigger in magnitude, okay.

(Refer slide time: 33:27)

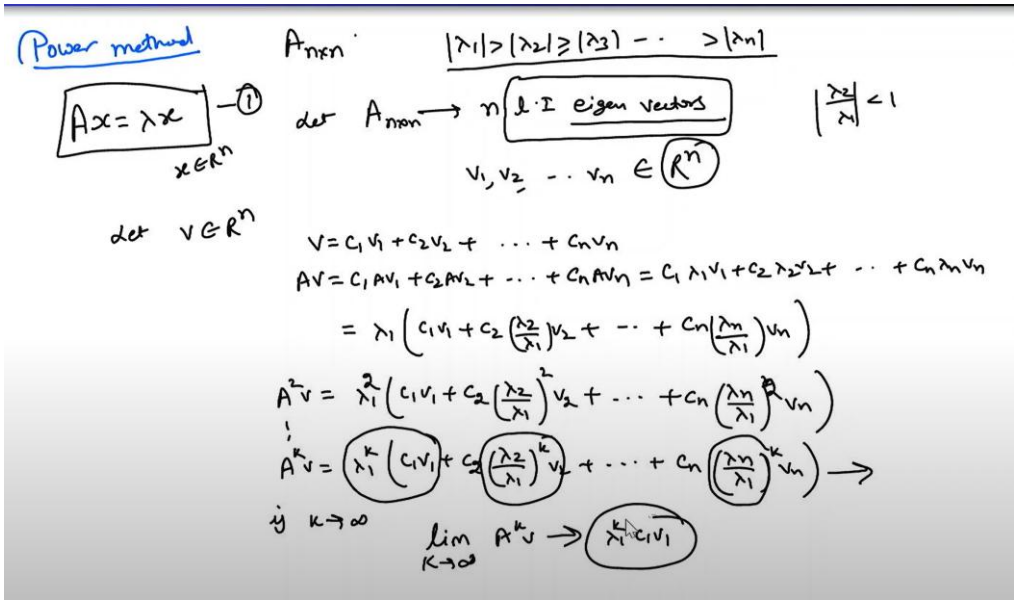


It can be negative or positive, but it is bigger in magnitude. So if it is bigger in magnitude, then if I write here, if k tends to infinity, because our iteration is increasing, then what will happen?

As the iteration increases, in this case, as it is going towards infinity, then what will happen to this quantity? What will happen to this quantity now? If I look at λ1 the magnitude is greater than the magnitude of λ2, λ3, so see these quantities λ2 over λ1is less than one. Okay. Let's take positive, take magnitude, now if keep taking there power they will become even more smaller. So, if I see and turn towards this so limit k tends to infinity if I seewhat will happen in this case? If k tends to infinity, then this quantity will converge to this. It will get this, right? All these will become very small. So for the large k, it will converge to this value, okay?

So if we assume this v, then what will happen? This will give us λ1 k, because k is also decreasing, c1 v1 it will converge towards this.

(Refer slide time: 35:03)

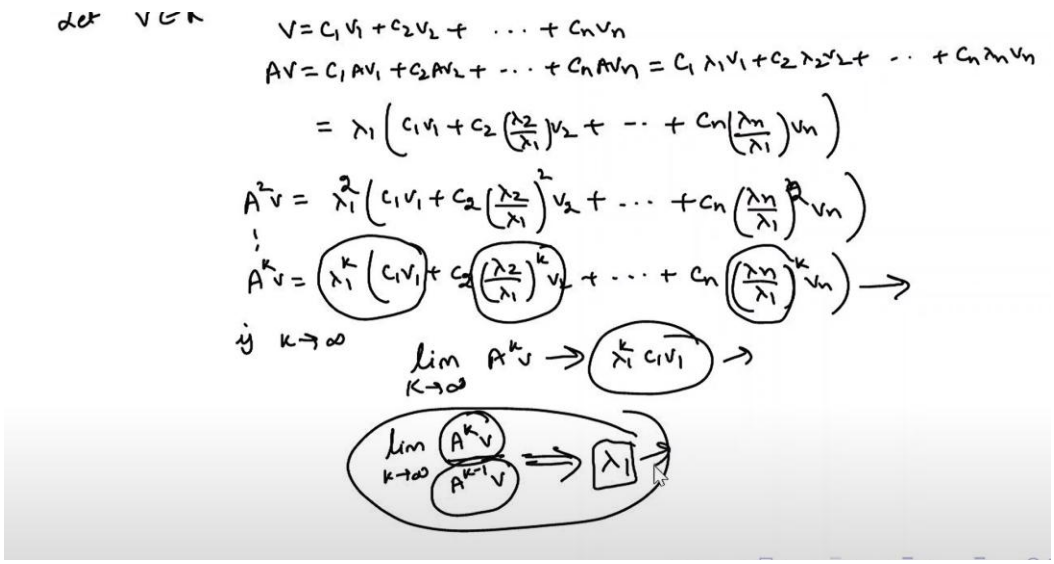


If this quantity becomes very small, all these will become zero. Only this will be left with us. So it is going towards this, okay? Now it is going towards this. Now see, this is λ1 is a scalar which is an eigenvalue and this is a vector. So now we have to see how this eigenvalue will come out. Now this λ1 ia an eigenvalue, for sure.

So let us see what we will do. I will try to see this. Limit k tends to infinity of A k v divided by A k-1 at limit k tends to , sorry, k tends to infinity we have already written. Now let's see its ratio. So if you see the ratio, you will see that this λ1 and this λ will cancel out. Brother, c1 v1 and c1 v1 both will converge. So if we see where is it going? This is going towards λ1, okay, right?

So it is converging to λ1. What does it mean? That the vector v A on the kth iteration, divided by the vector that comes out, will come out on the A k-1 of one if we divide both of them, okay? It will go towards λ1.

(Refer slide time: 36:38)



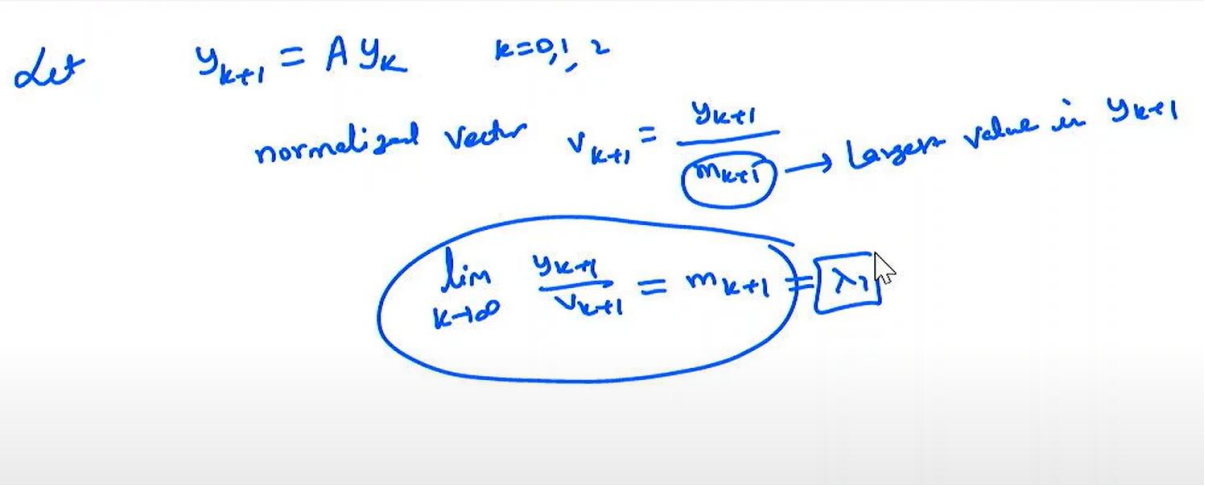
So what does it mean? If we adopt this process, then it will give us λ1. So what will be λ1? Will be our eigenvalue and corresponding to it, it will go in this way, so correspondingly, the vector we get will be our eigenvector, okay?

So if we see what is happening in this, we are getting λ1 and corresponding vector, v1 we are getting. So we are getting both things together. So in this case now we have to check how to do it, okay? So what procedure do we have to adopt for this? First, what will be the procedure? Because we also have to normalize the vector. So what are we doing now? What is normalization of a vector? So suppose, what did we do? Let us create a vector from A y k. This.

First we took y0, then y came, then y2 came like this. Now what do we have to do? We will take any vector and normalize it. So suppose my y was normalized before. I have to normalize the next one. So to normalize, I write suppose the normalized vector, k is 0, 1, 2 and this. I write it. Suppose vk, then how will the normalize vector of vk be formed? How will it be formed? I will divide the vector y k+1 by the largest eigenvalue. I will name it m k+1. This is the largest value in y k+1. I will divide it by that, okay?

And after that, if I take the limit k tends to infinity, then what will it become? Y k+1 over v k+1, if suppose I take this. So, if you from here y k+1 divided by v k+1, y k is a vector and v k is a normalized vector. If I look at this, you will see that this is taking us towards m k+1, right. And that will be our λ1, eigenvalue. So this is our procedure. We will have to do.

(Refer slide time: 39:27)



For example, like let me take an example. From this, it will become a little more clear what to do. It is very easy. Let's take one example. Now in one example, we have taken A and took its value: -15, 4, 3 and 10 , -12, 6 and second third took 20, -4, 2 so it comes. Now I have to find its eigenvalues. I will find the largest eigenvalue. Now I have to apply the power method. Okay, so what will I do in the power method?

Let me take the initial guess. So I took the initial guess. Let us start with the initial guess which we have. I took 1, 1, 1, which means this one one, we assumed that this is eigenvector; initially, its largest eigenvalue is one. So this is normalized. Now what did I do? I calculated it's a x0. So here we will put the A: -15, 4, 3, 10, -12, 6, 20, -4, 2, 1,1,1.

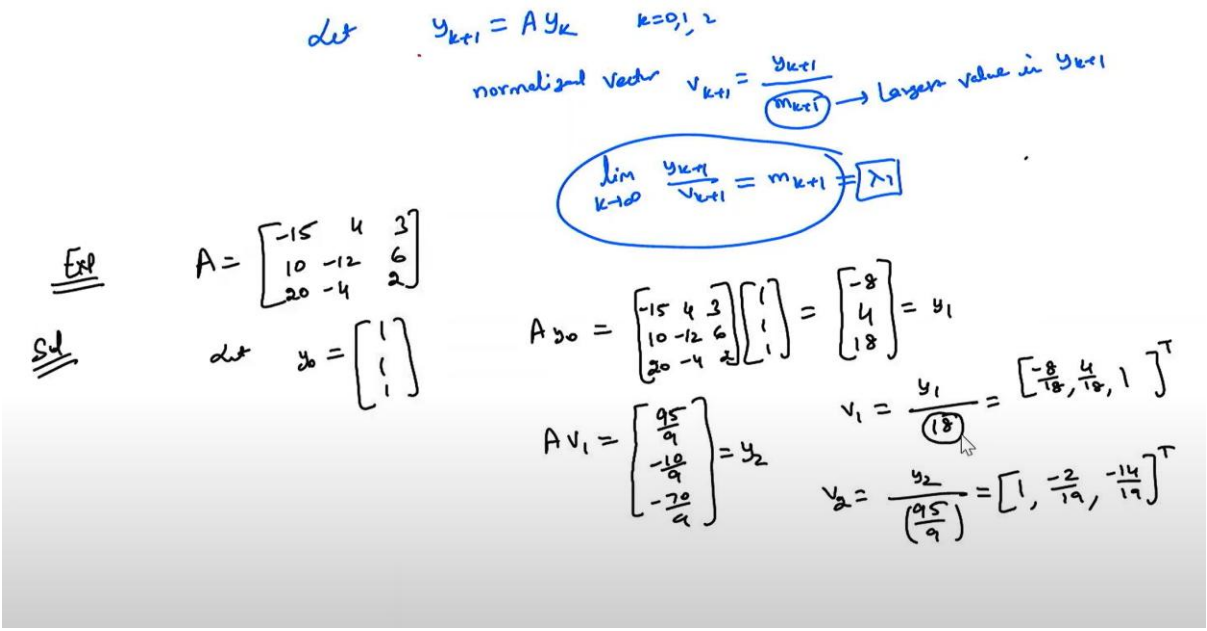
Now, when I calculated this, see: minus 15 plus 7, minus 8 came. And from here, 16, minus 4 came. And how much did it come? 22, minus 4, 18 came. This is what came. Let me take this. If I had not taken the x0, then it was y0. This was y0. So this is our y1.

Now we need the v1 from y1. So what will I do? I will divide it by the largest. Okay. So what did I do? I divided y1. What is the largest value? 18. I divided it by 18. So just like I divided by 18, I also got it like this in row So this come -8 by 18, 4 by 18, 18 by 18 transpose. Transpose means column vector. So 18 by 18 means one, right? So I deleted it and wrote it as one.

So now the quantity that we have is this. Now what did I do? Take v1. Now what will I do? Should I take out A of v1? Okay, right? Normalized. Now what did I do? I applied A v1. If I do A v1, then I will get the value. After calculating that, suppose this is what I am getting. So I calculated it. So I got this. Suppose 95 by 9, minus 10 by 9, minus 70 by 9. This is what we got.

Now what we got is y2. Now I have to take out v2 from here. So what was v2? What was v2, y2 divided by the largest element? The largest element in this is 95 by 9. I divided it by this. We got a new vector. And this is what we got: 1, -2 by 19, -14 by 19. This vector has come. Okay. So now I will go on like this: v2, then v3, then v4, v5. After that, we will keep checking that when our values are—how do we stop it? How do we stop it?

(Refer slide time: 43:49)



So see, the values—all these values—will be almost the same after some time. Okay, they will be the same or the vectors that we are getting, v1, v2, v3, v4, v5—they will start having the same values. So if they are the same, then if we divide them, the largest value will also be the same. So we have to go on till then, and after that, when this happens, we will say that this divisor, which was the largest eigenvalue, will become our eigenvalue. And these vectors that are being formed, we will have the eigenvector because this is what we did. See, we will divide it, or we will divide the y k+1 by the v k+1. So the values that will come to us will keep converging towards λ1, and the corresponding vector will become eigenvector. Okay.

So in this, if we keep doing it this way, then we will do it with the help of code. Okay. So now this is what we have. Now suppose someone tells us to find its smallest eigenvalue also. For this problem, like this, we have to find its smallest eigenvalue or we have applied the power method. Now what do I do? Find the smallest eigenvalue.

So I tell him the inverse power method. It is the power method, but is applied on A inverse, why apply the inverse method? Because now see what is happening. We have the eigenvalue $Ax = \lambda x$. Okay, what is λ, is its eigenvalue. We have assumed that the λ is non-zero. Okay. It means that the matrix A is non-singular. If it is non-singular, then all its eigenvalues will be non-zero.

Now what do I do? If it is non-singular, then suppose its inverse exists. So the inverse exists. So I can write it like this. See, A inverse $Ax= \lambda$ A inverse x where A inverse exists. We got it from here. If we see, what has happened is this, this has become I. I will take it here. So I wrote it like this: A inverse x. From this, what did we get to know, that the 1 by λ is the eigenvalue of A inverse, okay so the eigenvalues of λA, so 1 by λ has become eigenvalue of A inverse and the eigenvector remained the same in this case.

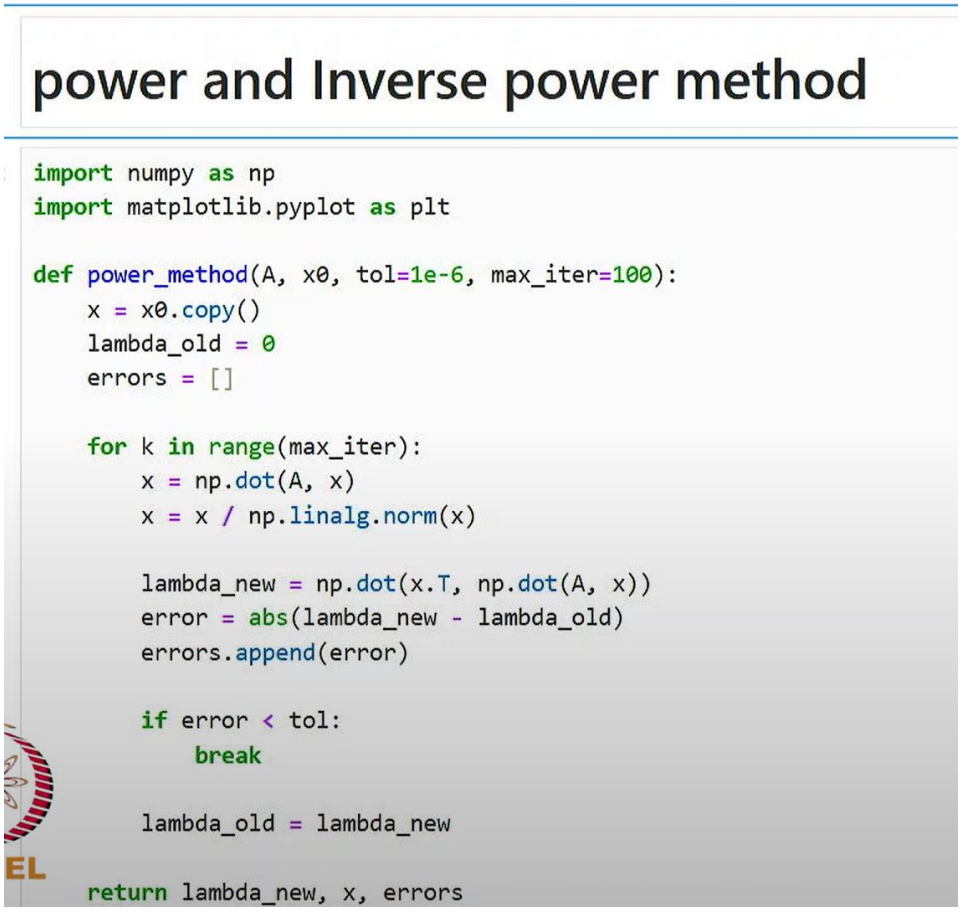
So what did we do? If we got the largest value, now we should also get the lowest. So what will we do? We will use A inverse and find its largest eigenvalue. So its largest value will become the minimum of A. So from there, we will get the minimum value. Right?

So in this case, we will get the minimum value. So like we got to know that its largest value has come and the minimum has also come, so what will we do? We have to find the value in between. So look, now if suppose we had a 3 by 3 matrix—A 3 cross 3—then we got $\lambda 1$ the largest eigenvalue and $\lambda 3$ the lowest eigenvalue. Suppose now we will find out. I thought that we need the eigenvalue somewhere in between. So what will I do now? I have taken A minus some number. I have taken some number in between them—mu. So I have seen what happened. So A minus mu I. I can write it like this: λ – mu x. Right? Just like in the matrix, we can do this. What is λ – mu x? It is the eigenvalue of A minus mu.

So if any of our eigenvalues is very close to mu, then what will happen in that case? This value, this quantity, will become small. Or it will become small. Right? And these values—this matrix—will become a new matrix whose eigenvalue is λ-mu. Okay? And if λ and mu are very close to each other, then this value will become very small. And if it is a small value, then if we want to find it, then what will we do? We will apply the power method on the inverse of A minus mu I. Okay? And by applying the power method, we will get the lowest eigenvalue from here. From there, it will become the largest. From that, we will subtract mu.

So the value that is in between will also come. So we call it shifted inverse method. So this is the value that we can find out in this way. So now let's take an example, then we will do it with this. I have a code of power method. Power method, okay, this is its code. So power method and inverse power method.

(Refer slide time: 50:11)



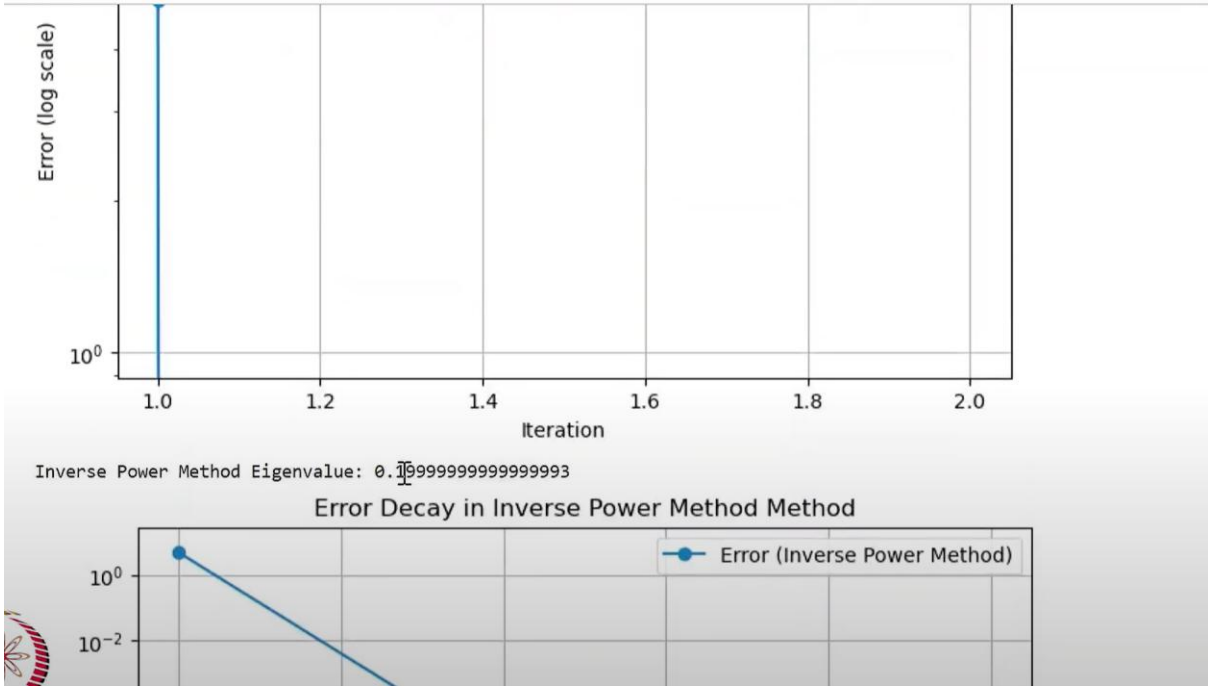
So if we see any program in it, then what did we do? We did it, okay. After that, we defined a function called power method. In it, we need A and the initial value x0. We need tolerance, okay. I put the tolerance as 0.5. I took iteration 100, okay. I copied the value of x in x0. I considered the old value as zero. Mostly, we go according to one one. Now what did we do? We have to normalize. So see what are we doing here. Whatever x comes, we will normalize it. Then the new one will come. We will take the difference of both, that error. So in this way, our method will keep on converging towards the eigenvalue. This is our inverse power method. What will it give us? It will give us the minimum eigenvalue, okay. So it will give us the minimum eigenvalue. We have to calculate this also like this. Just have to take the inverse. So if we have to find the inverse, then we will use the inbuilt function that we have: linear algebra dot inverse, okay. So this is an inbuilt library that we have. And we have to go till maximum iteration, and from there we will get these values. We will just call it, and we will get the solution.

(Refer slide time: 51:55)

```
        if error < tol:
            break

        lambda_old = lambda_new

    return 1/lambda_new, x, errors

def plot_errors(errors, method):
    plt.figure(figsize=(8, 5))
    plt.plot(range(1, len(errors) + 1), errors, marker='o', linestyle='-', label=f'Error ({method})')
    plt.yscale("log")
    plt.xlabel("Iteration")
    plt.ylabel("Error (log scale)")
    plt.title(f"Error Decay in {method} Method")
    plt.legend()
    plt.grid(True)
    plt.show()

# Example usage
A = np.array([[4, 1], [2, 3]], dtype=float)
x0 = np.array([1, 1], dtype=float)

# Power Method
lambda_power, eigenvector_power, errors_power = power_method(A, x0)
print("Power Method Eigenvalue:", lambda_power)
plot_errors(errors_power, "Power Method")

# Inverse Power Method
lambda_inverse, eigenvector_inverse, errors_inverse = inverse_power_method(A, x0)
```

So if we have any matrix, like we have taken the 2 by 2 matrix: 4, 1, 2, 3. So we have to find its eigenvalue, okay. So let's see. After running this, we ran it. So this is simple. So from here, our power method comes. So the largest eigenvalue comes, okay.

(Refer slide time: 52:15)



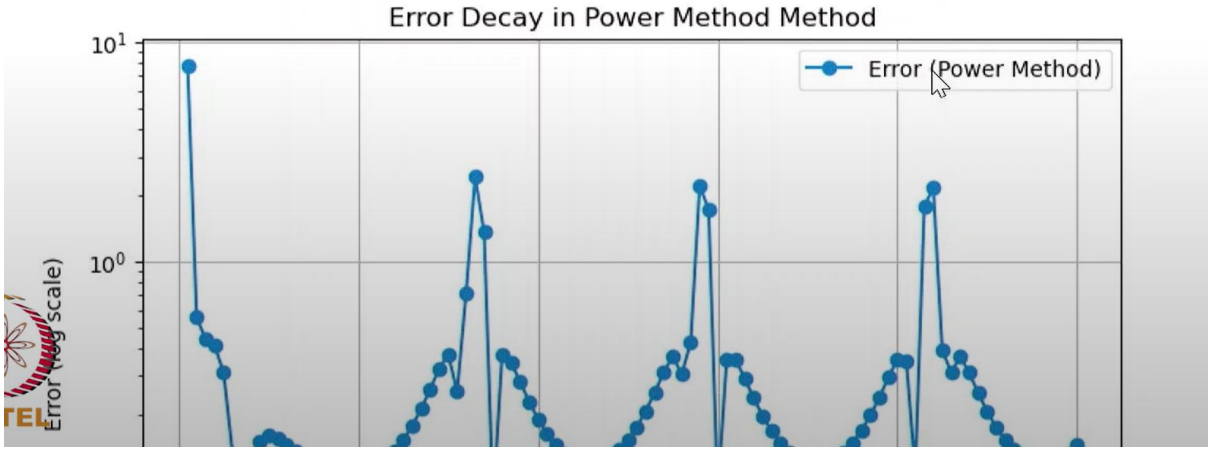
The smallest eigenvalue comes. So the largest eigenvalue is close to 5, and what is the lowest eigenvalue close to 0.1, okay. So we have found it out. Now we can do the same thing for any other matrix as well. So what did we do? Let's change the matrix to this. Here, what do I do?

Let's take A of the values that we had taken. Let's take this matrix and it is diagonal dominant we know. Let's see that its eigenvalues. And we also have to take the initial guess. So, let's take this in the initial guess, 1, 1, 1. Always take normalized and always keep this in mind. So, after normalizing this, we ran this. We got this.

(Refer slide time: 53:42)

```
# Power Method
lambda_power, eigenvector_power, errors_power = power_method(A, x0)
print("Power Method Eigenvalue:", lambda_power)
plot_errors(errors_power, "Power Method")

# Inverse Power Method
lambda_inverse, eigenvector_inverse, errors_inverse = inverse_power_method(A, x0)
print("Inverse Power Method Eigenvalue:", lambda_inverse)
plot_errors(errors_inverse, "Inverse Power Method")
```

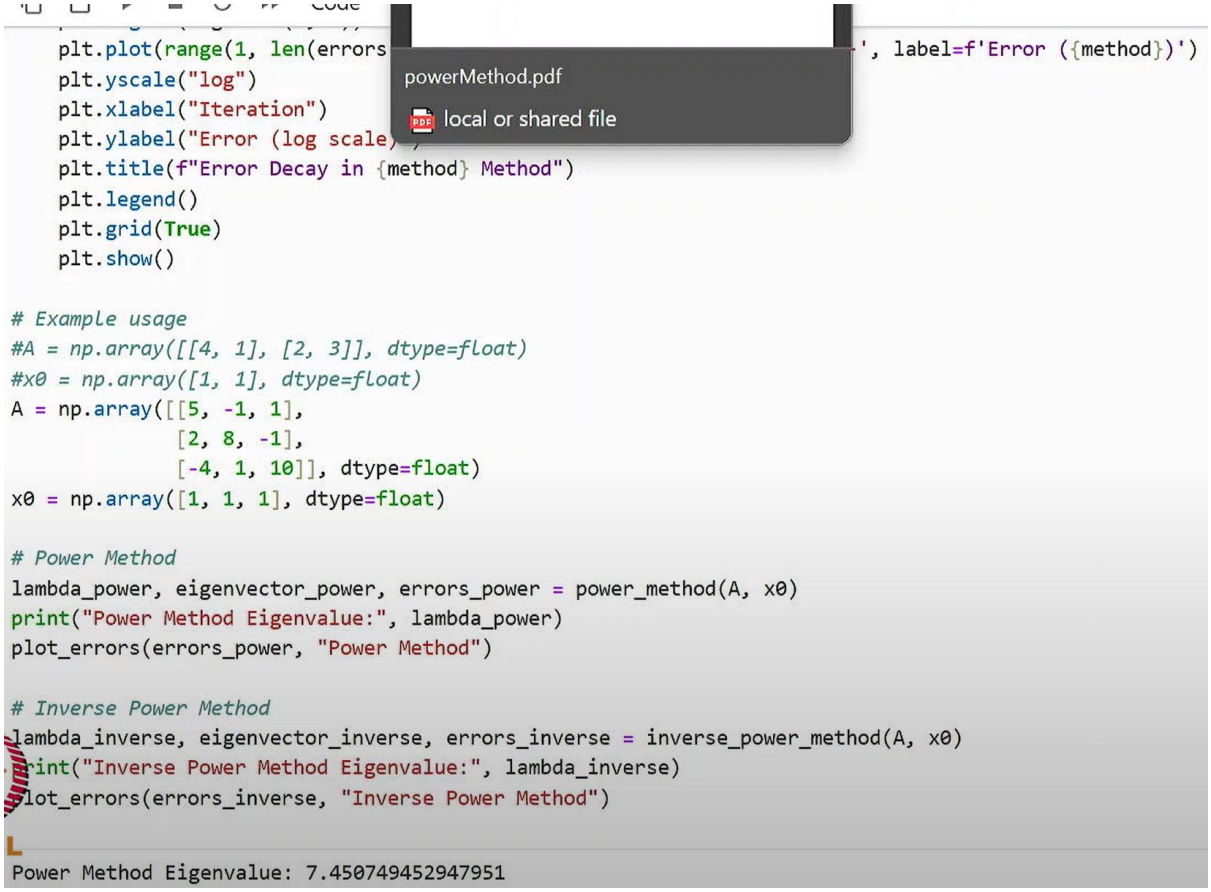
```
Power Method Eigenvalue: 7.450749452947951
```



So, by power method, we saw that 7.45 is the largest eigenvalue. We have this convergence. How was the power method converging? After that, the smallest eigenvalue is 0.14285. So, we got the largest from this, and we got the smallest from this.

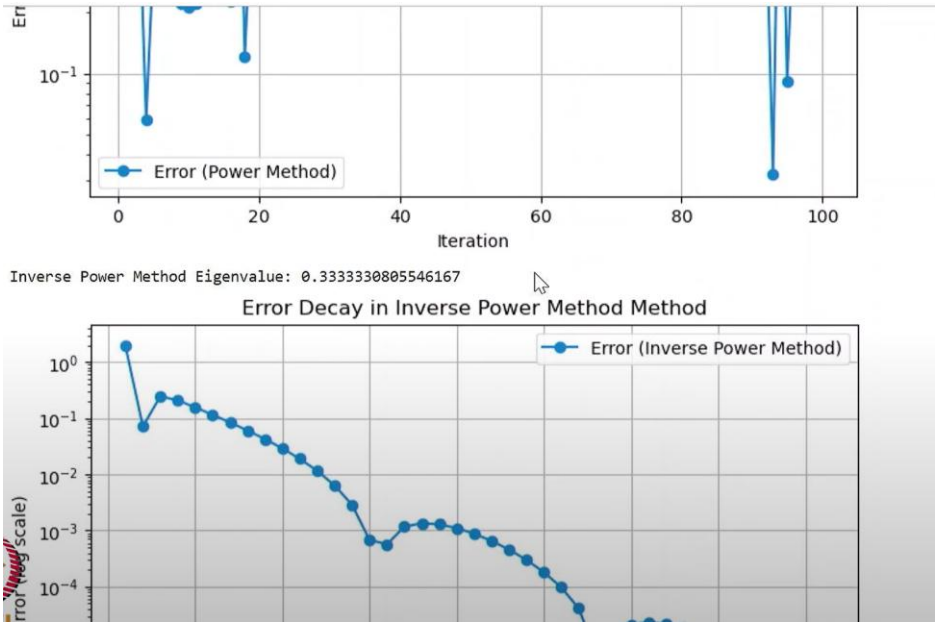
So now we have come to know that there is a value between point 1 and 7. So, what do we do now? We can solve this with the help of shifted. So what we have done in shifted is that we shift the values by some value. So look, now the value that is coming, suppose I write 5, -1,1. We had 5, -1, 1. You see what needs to be done with that.

(Refer slide time: 54:47)



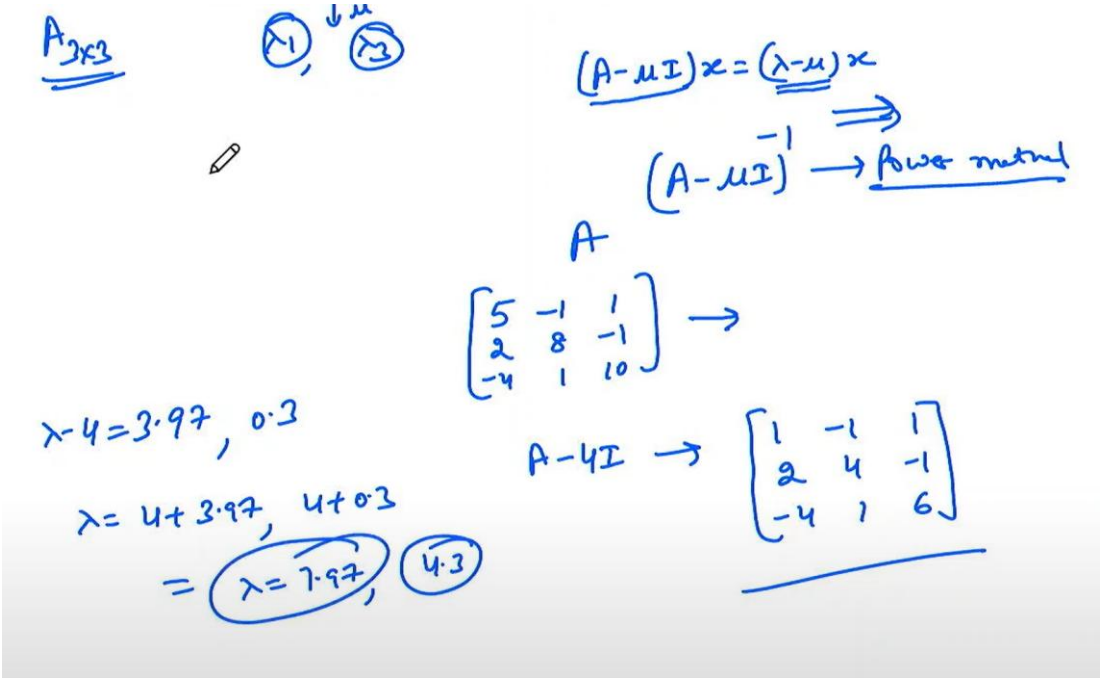
So this is this. So this value was 5, -1, 1. The matrix we had was 2, 8, -1, -4, 1, 10. So I will see how to calculate the other values. The values in the middle, we do not know what its value could be. It could be 7, and its value could be just a little bigger than 0.19. So we do not know anything. So just for the time being, I will shift it and write that this was our. Suppose I will write A – 4I. I will just write it. I do not know exactly what will come for 4. So this will come 1, -1, 1, 2, 4, -1, -4, 1, 6. This is what we have, okay. So I changed the matrix to 1, -1, 1. So now let's see what comes out. I will make it 1, -1, 1. This one came out to be four, and this one came out to be six. See, this one came out. Now let's see what came out. And from here we got this. Now we ran it. See, the value 3.97 came out to be the largest, and this one came out to be the smallest 0.333 came out, okay.

(Refer slide time: 56:39)



So if I add four to it, then it would be 7.97. That was okay. I will add four to it because we have done minus 4, right? So if we do plus minus, then I will add four to it. So if you see, the value that comes out is 4.33. So it means the eigenvalue in the middle is 4.33, right, with the help of this one. Why? Because we have seen now that λ - 4, we got this much value.λ - 4, this came out. From here it came out to be 3.97 and 0.3. We would have written 3.97 and 0.3 is there, then how much would the λ be? 4 + 3.97, 4+0.3 . This means how much is λ, okay? 7.97 was already coming, and this 4.3 also came to us. So we got this value, right? 4.3.

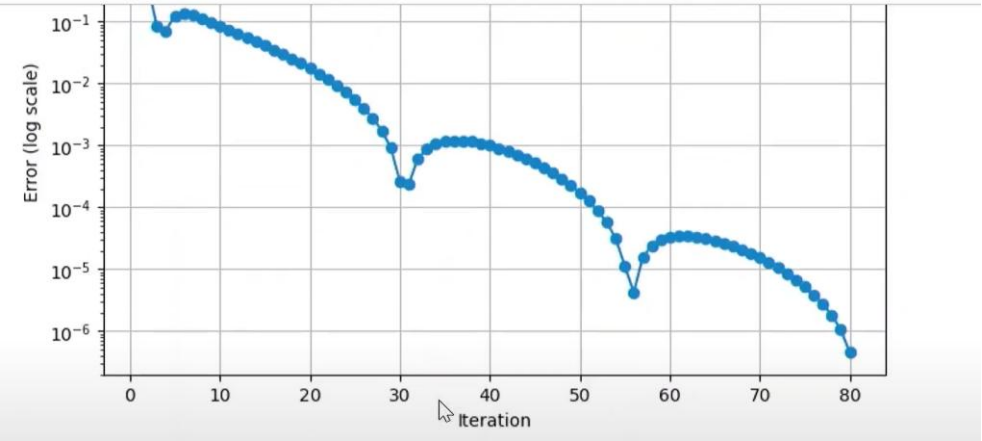
(Refer slide time: 58:02)



So if you see, in that case, earlier we were getting point one. Now this that has come, so the value in between is left to us. Now there is no problem at all. Now I will add four, four, four

to it. Let me go back to that. So 10 and solved it. See, this came. There is a little, actually iterative methods. So 7.45, 0.128, and the value in between will be 3, 4.3. So in this way, we can find out the eigenvalue.

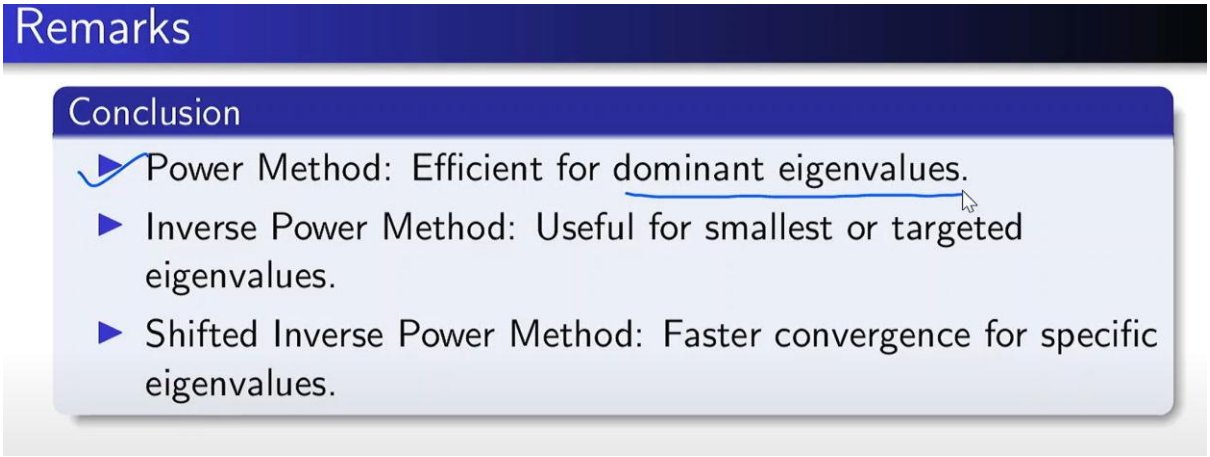
(Refer slide time: 58:46)



So the resulting value in between them can also be found out by inverse power method, okay. So what did we do with this? All the values will be found out.

So the remark, if we write it in power method, then what is the remark? Remark first: So it is efficient for dominant eigenvalue. Inverse power method is useful for smallest and targeted eigenvalue. Targeted means that we know that its eigenvalue is coming somewhere near it.

(Refer slide time: 59:22)



So, if it is coming somewhere near it, then we try to find it by the Inverse Power Method and shifted is right, right? So, faster convergence for specific eigenvalue. If we come to know that any eigenvalue is near it, then we can increase its convergence by applying the Shifted inverse Power Method. With the help of the Shifted inverse Power Method, we have already used this code. So, we can play with this code. We can define different, different matrices. We can check whether it is diagonally dominant or not. If it is not diagonally dominant, then what is the problem?

So, all this work which we have done is done with the help of this code which we have created. And this code is available in books, and you will also find it online. So, you can create your own code and try it with it.

All right, we have to take the dot product. This dot means dot product of A with respect to x. So, you calculate all these values. We have plotted their error. And from here, we have also applied its Inverse Power Method.

So, after doing this, today we saw that the important thing for us is the spectrum radius. As we saw in Gauss-Seidel and Jacobi, it is quite useful. So, to find the largest Eigenvalue, we discussed the Power Method. We discussed the Inverse Power Method. We discussed the Shifted Inverse Power Method.

So now, with the help of their Power Method, we can find out all the eigenvalues of the matrix.

So, I hope you liked today's lecture and understood it. Thank you for watching this.