

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

Lecture-22
Iterative Methods: Python Implementation of Jacobi Method

Hi everybody, we have introduced some iterative methods for solving linear systems, we have introduced Jacobi method, Gauss-Seidel method, we have also introduced successive over relaxation method. In this lecture we will implement the Jacobi method as a python code. Let us quickly recall the Jacobi method.

(Refer Slide Time: 00:45)

The slide displays the following content:

Iterative Methods: Jacobi Method (contd.)

Given initial guess $\mathbf{x}^{(0)} = (x_1^{(0)}, x_2^{(0)}, x_3^{(0)})^T$.
Define a sequence of iterates (for $k = 0, 1, 2, \dots$) by

$$\left. \begin{aligned} a_{11} \neq 0 &\Rightarrow x_1^{(k+1)} = \frac{1}{a_{11}}(b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)}) \\ a_{22} \neq 0 &\Rightarrow x_2^{(k+1)} = \frac{1}{a_{22}}(b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)}) \\ a_{33} \neq 0 &\Rightarrow x_3^{(k+1)} = \frac{1}{a_{33}}(b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k)}) \end{aligned} \right\}$$

which is the **Jacobi iterative procedure** in the case of 3×3 system.

NPTEL S. Baskar (IIT Bombay) NPTEL Course Iterative Methods

In our theory class, we have derived the Jacobi method in particular for a 3×3 system. Let us quickly recall the formula for the Jacobi method in the case of a 3×3 system, we are given a initial guess $\mathbf{x}^{(0)}$. In this case it is a 3-dimensional vector then the Jacobi iterative sequence can be computed using this formula and this has to be done for each $k = 0, 1, 2$ and so on.

Let us recall the formula. There are 2 important points to be noted in this formula as far as the coding is concerned. In each of the components of the vector $\mathbf{x}^{(k+1)}$ what we are doing is that we are keeping the diagonal element on the left-hand side and moving all the non-diagonal element to the right-hand side. Therefore, in each component we have 2 terms and in this we do not include the diagonal term.

That is the first point to note and second thing is, we are dividing both sides by the diagonal element a_{ii} . In the first equation it is a_{11} and second equation it is a_{22} and in the third equation it is a_{33} . Therefore, we have to make sure that all these diagonal elements are non-zero. These are the important points that we have to keep in mind when we are developing the code for Jacobi method. To have a clear idea let us put this formula in the general case when our system has n linear equations.

(Refer Slide Time: 02:41)

Iterative Methods: Jacobi Method (contd.)

Given initial guess $\mathbf{x}^{(0)} = (x_1^{(0)}, x_2^{(0)}, \dots, x_n^{(0)})^T$.

Define a sequence of iterates (for $k = 0, 1, 2, \dots$) as follows:

For each $i = 1, 2, \dots, n$, **FIRST** check for $a_{ii} \neq 0$. If so, then compute

$$\Rightarrow x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1, j \neq i}^n a_{ij} x_j^{(k)} \right)$$

In this case our initial guess will be an n -dimensional vector which is generally chosen as per our wish and then for each iteration you have to compute now n components of the vector $x^{(k+1)}$ and each component is computed using this formula again you note that the summation on the right hand side should exclude the diagonal term. That is the summation should run from $j = 1$ to n , where you should not include the term $j = i$ here.

That is one thing and second thing is when you compute the components you have to make sure that the diagonal elements are non-zero. Before going to the code let us see how many loops are involved in this computation. The first loop in our code should run for the iterations and the index is denoted by k . Therefore, our first loop is for the iteration and for each k we have to compute the vector $x^{(k+1)}$.

And remember $x^{(k+1)}$ has n components; therefore, our second loop should run for the components of the vector $x^{(k+1)}$ and that is your second loop and there is one more loop in each component of the vector $x^{(k+1)}$ we have to find the value of this sum. Therefore, the third

loop should run for $j = 1$ to n and that is your third loop and remember in this third loop you have to exclude one term that is the diagonal term.

Which means, you have to always check whether $j = i$ or not. If $j \neq i$ then you have to include the term into your summation. If $j = i$ then you have to exclude it. We can see that all these loops will be nested loops and therefore the code will be little confusing. Mathematically Jacobi method is a very simple and easy to understand method. However, the computer code will be little confusing. Therefore, we will carefully try to understand this.

(Refer Slide Time: 05:39)

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 #####Input#####
4 A = np.array([[1, 0.0, 1.0],
5               [-1.0, 1.0, 0.0],
6               [1.0, 2.0, -3.0]])
7 b = np.array([0.0, 0.0, 0.0])
8 x = np.array([1.0, 1.0, 1.0])
9
10 N = 12
11 #####

```

Handwritten notes on the right side of the code:

$$\begin{cases} x_1 + 0x_2 + x_3 = 0 \\ -x_1 + x_2 + 0x_3 = 0 \\ x_1 + 2x_2 - 3x_3 = 0 \end{cases}$$

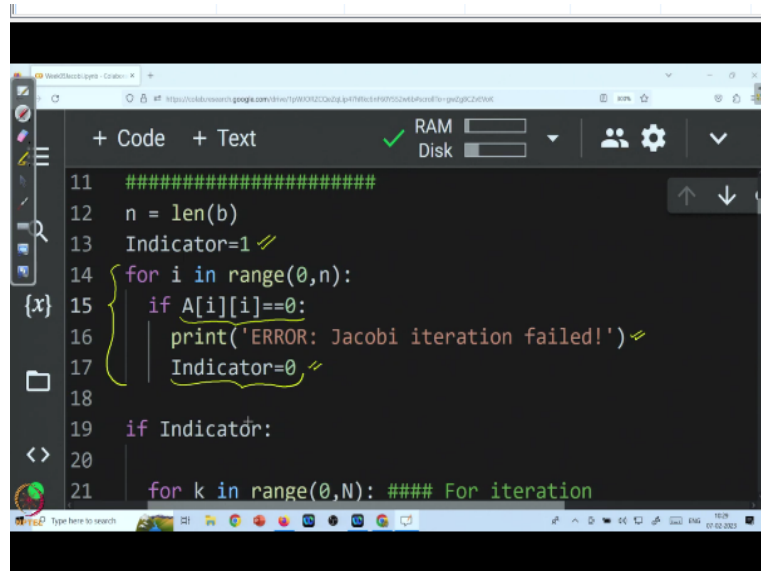
$x^{(0)} = (1, 1, 1)$

Let us go into the code now. The first part of our code is to include some modules into our program. Here we are including 2 modules; one is numpy which is mainly used to declare the arrays and the second one is the matplotlib which is used to plot some graphs mainly. In our program, at the end we will plot the l_2 norm of the error involved in each iteration. For that purpose, I am also importing matplotlib module.

The next part of our code is input. Here we are taking the left hand side coefficient matrix in the variable called A and we are taking the right hand side vector in the variable b and finally the initial guess is taken in the variable x. You can see that A is a 2-dimensional array whereas B and X are one dimensional array. With this you can see that we are trying to solve the linear system $x_1 + 0x_2 + x_3 = 0$, $-x_1 + x_2 + 0x_3 = 0$ and finally $x_1 + 2x_2 - 3x_3$.

And the initial guess $x^{(0)}$ is taken to be (1, 1, 1). And this variable corresponds to the number of iterations that we wish to run. I have taken it as 12, if you want to have more iterations you can increase the value of N. With all these inputs, let us go to the main part of the program.

(Refer Slide Time: 07:51)



```
11 #####
12 n = len(b)
13 Indicator=1
14 for i in range(0,n):
15     if A[i][i]==0:
16         print('ERROR: Jacobi iteration failed!')
17         Indicator=0
18
19 if Indicator:
20
21     for k in range(0,N): #### For iteration
```

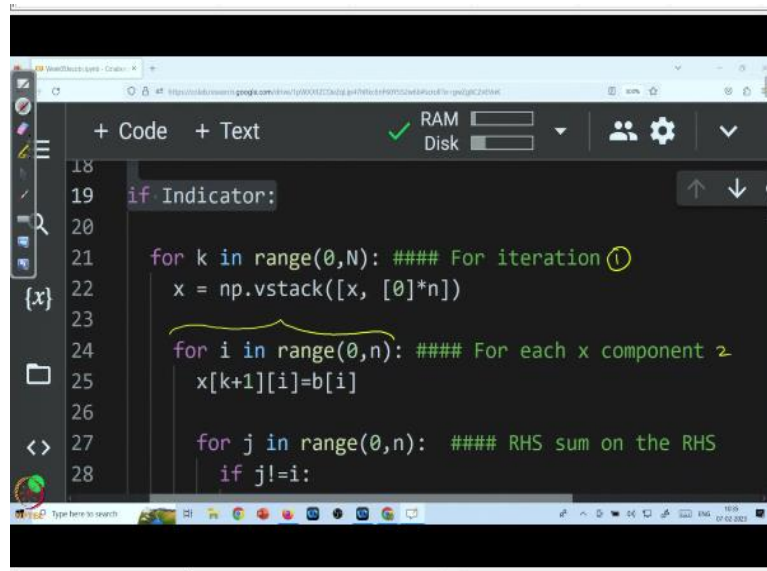
At the first stage we are trying to see what is the length of the array b and we are taking that into the variable n. This is to sense what is the dimension of our system. In this particular input we have taken a 3×3 system; therefore we could have directly written 3 here but I want to make our code general. So, that you go to enter a matrix A which is say 4×4 matrix or 5×5 matrix and correspondingly the vectors b and x, then that should be automatically taken into consideration in our main program.

That is why I do not want to directly feed the value 3 in this variable n rather I want to find it from the length of the array b which will precisely be the dimension of our system. Then, I am declaring a variable called indicator, we will see why we are using this variable. Then the next part of our code is to check whether all the diagonal elements are non-zero.

This is one of the main points which we have noted from our previous slide that we have to check that all the diagonal elements are non zero. If at least one diagonal element is 0 then we do not want to go ahead with the Jacobi method. So, what we are doing is we are simply printing that the Jacobi method fails and then we are putting the value in the variable indicator as 0.

Remember we have initialized the variable indicator as 1. Now we have put the value 0 into the indicator. That says that we should not compute the Jacobi iteration. Let us see how we are sensing that? Let us get into the main part of the Jacobi method.

(Refer Slide Time: 10:08)



```
18
19 if Indicator:
20
21     for k in range(0,N): ##### For iteration ①
22         x = np.vstack([x, [0]*n])
23
24         for i in range(0,n): ##### For each x component 2
25             x[k+1][i]=b[i]
26
27         for j in range(0,n): ##### RHS sum on the RHS
28             if j!=i:
```

And this main part is executed only when the value of the indicator is 1. If the value in the indicator is 0 then the python control will not enter into the if statement. Since from our previous for loop you can see that if any one of the diagonal elements is 0 then the indicator will become 0 and therefore the control will not enter into this if statement. If all the diagonal elements are non-zero then this part of the statement is never executed.

Therefore, the indicator will hold the value 1 which was initialized previously and therefore the python will enter into this if statement and that is the idea. Remember for the system that we have given you can see that all the diagonal elements are non-zero. Therefore, we will get into this if loop and we will start computing the iterations. As we have seen the first Loop of our method is going to be for the iteration and it is denoted by k, mathematically.

And the k runs from 0 up to N - 1. Remember N is the variable we have used to store how many iterations we want to compute in our method. If you recall we have taken N = 12 in the input. Therefore, this loop will run for k = 0, 1, 2 up to 11. Then the first step of this loop is to allocate memory for x. If you recall we already allocated memory for $x^{(0)}$ and we have stored the vector (1, 1, 1) in that memory.

Now what we are doing is we are allocating another memory. For this for the variable $x^{(1)}$ which is the vector from the first iteration of the Jacobi method. When we create the memory for this variable we are initializing it as (0, 0, 0) and then we will try to compute each of these coordinates using the Jacobi formula. That is the idea. If you recall what is the Jacobi formula?

The Jacobi formula is given like this for the coordinates of the vector $x^{(k+1)}$. Let us see how to compute that. The next loop is for the components of the vector $x^{(k+1)}$ and that is our second loop.

(Refer Slide Time: 13:24)

```

21 for k in range(0,N): #### For iteration ①
22     x = np.vstack([x, [0]*n]) // x(0) = (0,0,0)
23
24     for i in range(0,n): #### For each x component ②
25         x[k+1][i]=b[i]
26         xi(k+1) = 1/aii ( bi - ∑j=1, j≠in aijxj(k) )
27         for j in range(0,n): #### RHS sum on the RHS ③
28             if j!=i: //
29                 x[k+1][i] = x[k+1][i] - A[i][j]*x[k][j]
30
31         x[k+1][i] = x[k+1][i]/A[i][i] // xi(k+1) = bi - ai0x0(k) - ai1x1(k) - ...

```

Let us write the formula step by step and see how to implement it here. If you recall $x[k + 1][i]$ is equal to you have $1/a_{ii}$ and then you have the bracket in that you have b_i minus some summation. In that we are first taking the value of b_i into $x[k+1][i]$ here. Remember when we initialized this vector $x^{(k+1)}$. Remember when $k = 0$ it is simply $x^{(1)}$ and we have initialized it as (0, 0, 0).

And then we came into the second loop which runs for each component of the vector $x^{(1)}$. In that we are first storing the value of b_i into $x_i^{(1)}$ and then we are getting into our third loop. What is our third loop? Third loop is for the summation which is involved on the right hand side of this formula. If you recall that is $\text{sigma } j = 1 \text{ to } n$ and you have to exclude the diagonal element.

And then it is $a_{ij}x_j^{(k)}$, the previous iterations value. So, let us see how to implement this summation. For that we are running this innermost loop with the index as j. The first line is to check whether j is not equal to i and this summation will happen only if j is not equal to i then what we do is, we will take the previous value of $x[k+1][i]$ and with that we will subtract $a[i][j]*x[k][j]$.

This is something like initially we had $x_i^{(k+1)} = b_i$. That was the first value which is assigned into $x_i^{(k+1)}$ and then when $j = 0$ what you are doing is you are subtracting $a_{i0}x_0^{(k)}$, the previous iteration. You see you have k here that is because you are taking the value from the previous iteration and this is like the first component of the ith row and then when it goes to the second time in the loop j becomes 1.

Therefore, when it comes for the second time of course it will check the if condition. If this if condition is satisfied then $x_i^{(k+1)}$ is holding now the value given by this expression. From that value now you are again subtracting $a_{i1}x_1^{(k)}$ and like that this loop will go on till j touches n - 1. When you come out of this loop you will have the value of this expression for the corresponding ith coordinate.

Finally, what you have to do once you finished computing this expression inside the bracket then you have to divide that value by a_{ii} . That is what we are doing now in this step, after executing this loop you are coming out of this loop and then taking whatever value you obtained that is precisely the value of this expression inside the bracket and then you are dividing it by $A[i][i]$. You see here you are dividing it by $A[i][i]$.

(Refer Slide Time: 17:54)


```

21 for k in range(0,N): #### For iteration
22     x = np.vstack([x, [0]*n]) ←
23
24     for i in range(0,n): #### For each x component
25         x[k+1][i]=b[i]
26
27         for j in range(0,n): #### RHS sum on the RHS
28             if j!=i:
29                 x[k+1][i] = x[k+1][i] - A[i][j]*x[k][j]
30
31         x[k+1][i] = x[k+1][i]/A[i][i]
32     print('Approximate solution after '+str(N)+' iterations = '+str(x[N])+
33
#####Calculation Ends#####

```

That completes the calculation of one component of the vector $x^{(k+1)}$. That is $x_i^{(k+1)}$. And this has to go for each $i = 0, 1, 2$ up to $n - 1$. Once that is done then you get the vector $x^{(k+1)}$. Once $x^{(k+1)}$ is done then you will go back to the outermost loop, you will increment k by 1 more and then you will initialize the memory for the new x which is the vector of the next iteration and then you come to compute the value of the components of that vector.

Like that the loop will go on, how many times? Well, the number of times that we have specified in the variable N . Once the iterations are calculated for N number of times then python will come out of this entire loop and print this statement which says that approximate solution after that N number of times. Remember we have given the value of N as 12. Therefore, it has to print approximate solution after 12 iterations equal to it will print the vector $x[N]$. That is whatever the iteration it has calculated at the N th stage will be printed as the output.

(Refer Slide Time: 19:49)


```

32 print('Approximate solution after '+str(N)+' iterations = '+str(x[N]))
33
34 #####Calculating Error#####
35 errl2 = [0]*(N+1)
36 for k in range(0,N+1):
37     for i in range(0,n):
38         errl2[k] = errl2[k] + x[k][i]**2
39     errl2[k] = np.sqrt(errl2[k])
40     print('l^2 error = '+str(errl2[N])+'.')
41 #####Plotting error graph#####
42 # #####Plotting error graph#####
43 xaxis = list(range(0, N+1))
44 plt.plot(xaxis, errl2, color='blue', linestyle='-', linewidth = 1.2)
45 plt.xlabel('$k$')

```

Handwritten notes on the code editor:

- $\|e^{(k)}\|_2$
- $e^{(k)} = x - x^{(k)}$
- $= 0 - x^{(k)}$
- $\|e^{(k)}\|_2 = \sqrt{e_1^{(k)2} + e_2^{(k)2} + \dots + e_n^{(k)2}}$

Next, we are also interested to have a feeling on how the error looks like. For that what we are doing here is that we are calculating the l_2 norm of the error involved in the k th iteration. That is we are computing $e^{(k)}$ and then taking the l_2 norm of it. First let us see what is the error involved in the k th iteration. By definition the error involved in the k th iteration is equal to the exact value that is the exact solution of our system minus the vector computed at the k th iteration.

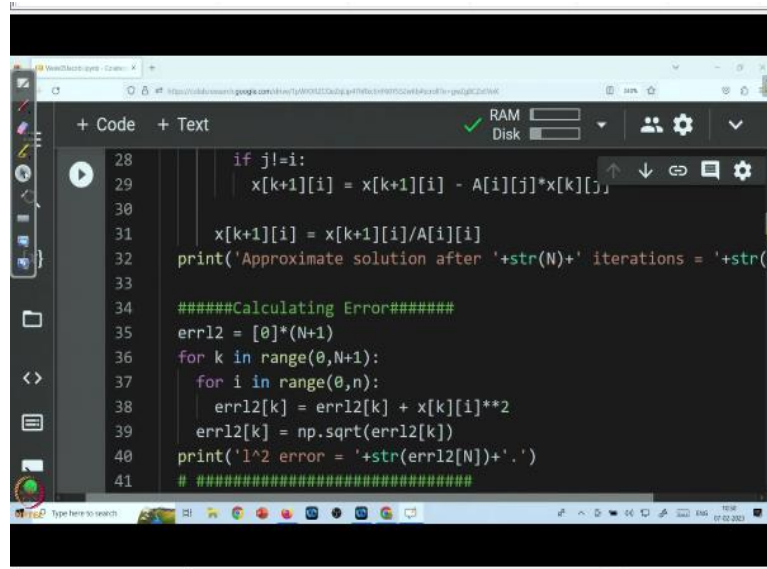
Recall that we have taken the right hand side vector of our system as the zero vector. Therefore, the exact solution is 0 because our matrix A is invertible and hence the error involved in the k th iteration is simply $-x^{(k)}$ only. Now let us see what is the formula for the l_2 norm of any vector in particular what is the l_2 norm of $\|e_2^{(k)}\|$? That is nothing but square root of e first component square + e second component square and so on till the n th component of the vector $e^{(k)}$ square.

So, that is the formula for the l_2 norm and you can observe that we have computed l_2 norm for each $k = 0$ to N . As a first step we are initializing the variable `errl2`. This is going to have the l_2 norm of the error of all the iterations; we are initializing this variable with 0. Remember it is created as an array of dimension $N + 1$ and then for each k from 0 to N .

Therefore, you can see that this for loop will run for $N + 1$ times. For each time it will compute the l_2 norm of the error involved in the k th iteration of the Jacobi method and then once it computes the inner sum then it takes the square root and that will finally give you the l_2 norm

of the error at the k th iteration. I am also printing the l_2 norm of the last iteration here and finally we are plotting the graph of $errl2$ as a function of k .

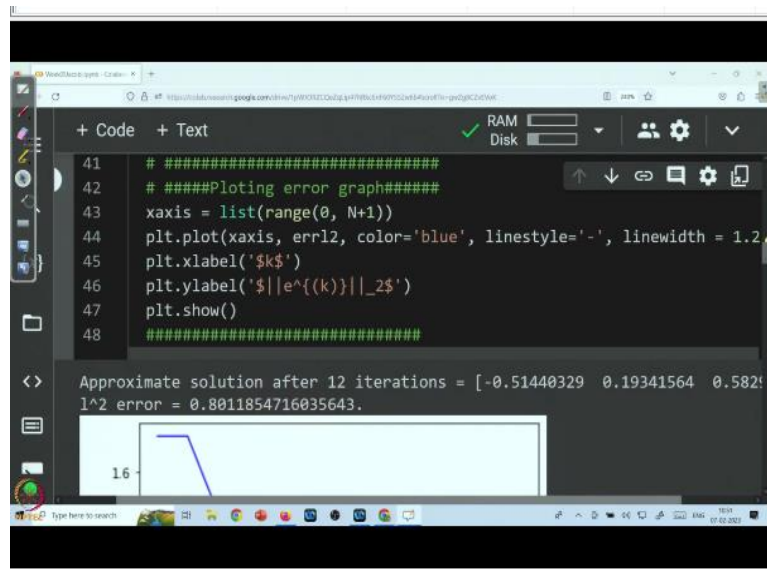
(Refer Slide Time: 22:58)



```
28     if j!=i:
29         x[k+1][i] = x[k+1][i] - A[i][j]*x[k][j]
30
31     x[k+1][i] = x[k+1][i]/A[i][i]
32     print('Approximate solution after '+str(N)+' iterations = '+str(
33
34     #####Calculating Error#####
35     errl2 = [0]*(N+1)
36     for k in range(0,N+1):
37         for i in range(0,n):
38             errl2[k] = errl2[k] + x[k][i]**2
39         errl2[k] = np.sqrt(errl2[k])
40         print('l^2 error = '+str(errl2[N])+'.')
41     #####
```

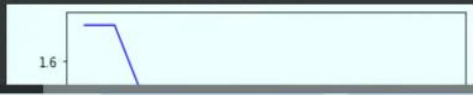
That is done in this part of our code. I will leave it to you to see how this is done; there is nothing to explain here. Let us see the output of this code. The code is executed and let us see the output recall that as the first stage we are printing the approximate solution after 12 iterations. So, that is what we expect to be printed as the first output of our code.

(Refer Slide Time: 23:30)



```
41     #####
42     # #####Plotting error graph#####
43     xaxis = list(range(0, N+1))
44     plt.plot(xaxis, errl2, color='blue', linestyle='-', linewidth = 1.2)
45     plt.xlabel('$k$')
46     plt.ylabel('$||e^{(k)}||_{2}$')
47     plt.show()
48     #####
```

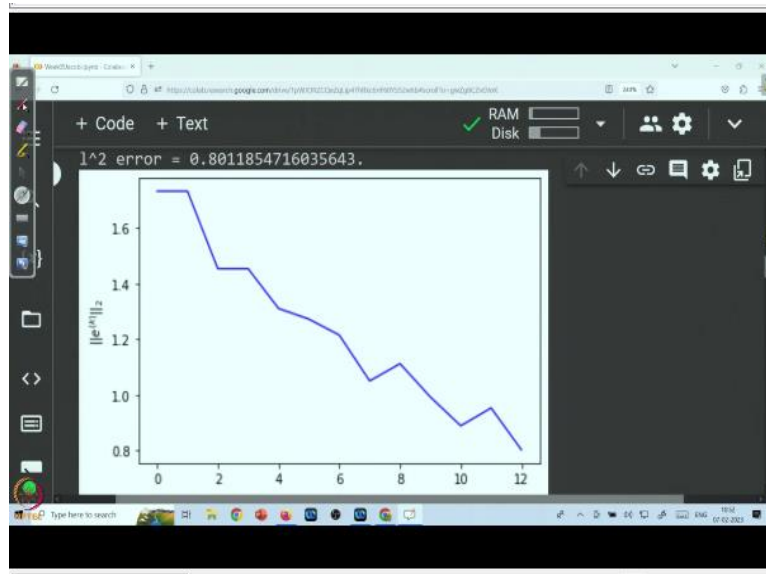
Approximate solution after 12 iterations = [-0.51440329 0.19341564 0.5821
l^2 error = 0.8011854716035643.



You can see that it is printed here approximate solution after 12 iteration is given by this vector; you can see that it is still far away from the zero vector. Recall the exact solution of the system that we have taken as an input has the exact solution as the zero vector. Here you can see that after 12 iterations it has only given this as the approximate solution which is rather a bad

approximation. And also you can see what is the l_2 error involved in this. Obviously, it is around 0.8. Let us see the plot of the l_2 norm of the error and that is given here.

(Refer Slide Time: 24:21)



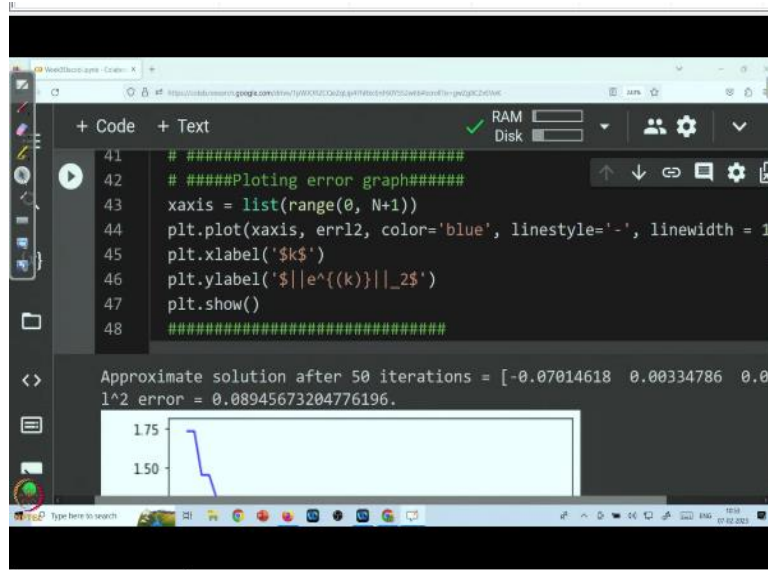
The iteration started somewhere here and the error kept on decreasing. It came up to 0.8 around and then it stopped, because we asked the code to compute only 12 iterations. Maybe if you go on further in the iteration you may get better and better approximation to our solution. Let us also see that.

(Refer Slide Time: 24:46)

```
6 [1.0, 2.0, -3.0]]
7 b = np.array([0.0, 0.0, 0.0])
8 x = np.array([1.0, 1.0, 1.0])
9
10 N = 50
11 #####
12 n = len(b)
13 Indicator=1
14 for i in range(0,n):
15     if A[i][i]==0:
16         print('ERROR: Jacobi iteration failed!')
17         Indicator=0
18
19 if Indicator:
```

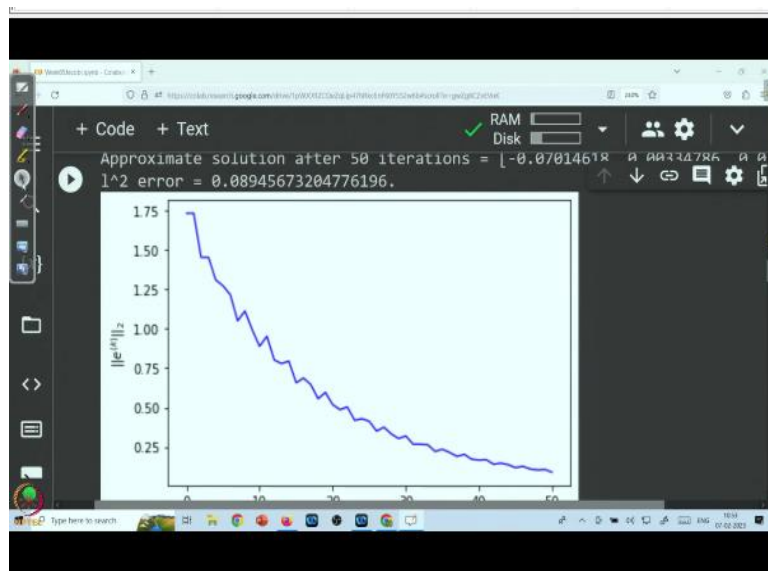
Let us do say 50 iterations and see how the approximation is improved. Let us run the code with $N = 50$ and see what is the improvement in our result.

(Refer Slide Time: 25:10)



Now you can see that the l_2 error is almost 0.09.

(Refer Slide Time: 25:13)



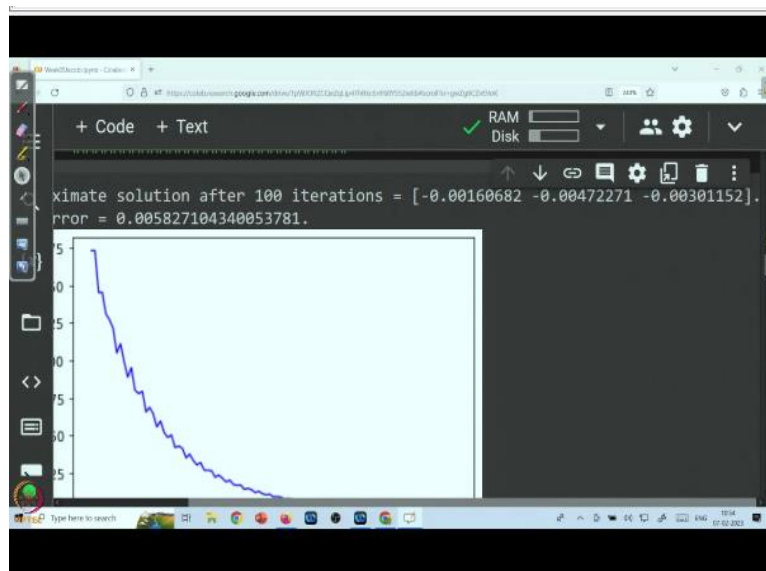
Also let us see the plot of the l_2 norm of the error; you can see that somewhere here and then further the error decreased and going closer and closer to 0. Let us go further in the iteration and see what is happening.

(Refer Slide Time: 25:36)

```
6 [1.0, 2.0, -3.0]]
7 b = np.array([0.0, 0.0, 0.0])
8 x = np.array([1.0, 1.0, 1.0])
9
10 N = 100
11 #####
12 n = len(b)
13 Indicator=1
14 for i in range(0,n):
15     if A[i][i]==0:
16         print('ERROR: Jacobi iteration failed!')
17         Indicator=0
18
19 if Indicator:
```

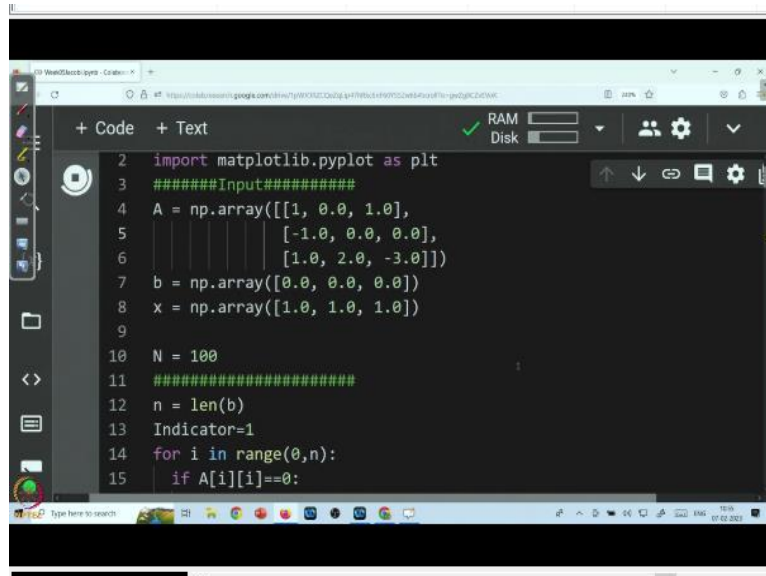
Let us say go up to 100 iterations and see what is the result now.

(Refer Slide Time: 25:42)



You can see that the approximation is quite good now. The l_2 norm of the error is around 0.005, you can also see the solution. The solution is now pretty close to the zero vector. So, you can see here the computed solution is pretty close to the exact solution and we can also see that the error is considerably reduced and going very close to 0. Now if you go on with the iteration it will go more and more close to 0.

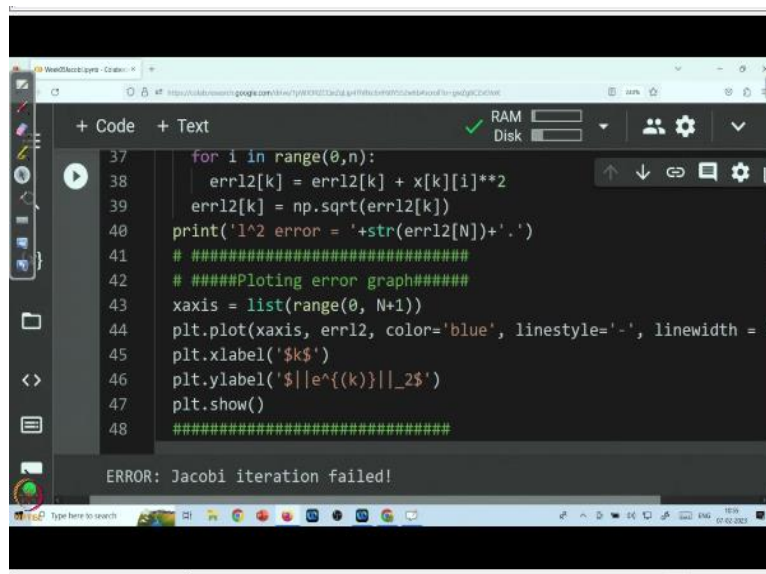
(Refer Slide Time: 26:22)



```
2 import matplotlib.pyplot as plt
3 #####Input#####
4 A = np.array([[1, 0.0, 1.0],
5              [-1.0, 0.0, 0.0],
6              [1.0, 2.0, -3.0]])
7 b = np.array([0.0, 0.0, 0.0])
8 x = np.array([1.0, 1.0, 1.0])
9
10 N = 100
11 #####
12 n = len(b)
13 Indicator=1
14 for i in range(0,n):
15     if A[i][i]==0:
```

Let us also see what happens if one of the diagonal elements is 0. Say for instance let us take a_{22} as 0 and see what is happening. Let us execute the program.

(Refer Slide Time: 26:39)



```
37     for i in range(0,n):
38         err12[k] = err12[k] + x[k][i]**2
39         err12[k] = np.sqrt(err12[k])
40     print('1^2 error = '+str(err12[N])+'.')
41     #####
42     #####Plotting error graph#####
43     xaxis = list(range(0, N+1))
44     plt.plot(xaxis, err12, color='blue', linestyle='-', linewidth = 2)
45     plt.xlabel('$k$')
46     plt.ylabel('$|e^{(k)}|_{2}$')
47     plt.show()
48     #####
ERROR: Jacobi iteration failed!
```

Now you can see that the code hit the error message, it just flashed the error message that Jacobi iteration failed and it did not get into the main loop.

(Refer Slide Time: 26:55)


```

10 N = 100
11 #####
12 n = len(b)
13 Indicator=1
14 for i in range(0,n):
15     if A[i][i]==0:
16         print('ERROR: Jacobi iteration failed!')
17         Indicator=0
18
19 if Indicator:
20
21     for k in range(0,N): #### For iteration
22         x = np.vstack([x, [0]*n])
23

```

That is because $A[2][2]$ is 0. Therefore, it satisfied this if condition. So, python entered into this if statement and printed this command and put indicator as 0. Since indicator is 0 this statement is false and therefore python could not enter into the main part of the program.

(Refer Slide Time: 27:20)

```

20
21 for k in range(0,N): #### For iteration
22     x = np.vstack([x, [0]*n])
23
24     for i in range(0,n): #### For each x component
25         x[k+1][i]=b[i]
26
27         for j in range(0,n): #### RHS sum on the RHS
28             if j!=i:
29                 x[k+1][i] = x[k+1][i] - A[i][j]*x[k][j]
30
31         x[k+1][i] = x[k+1][i]/A[i][i]
32     print('Approximate solution after '+str(N)+' iterations = '+str(x))
33

```

That is python could not further compute the iterations of the Jacobi method and therefore it could not also print the solution and so on. It just ended the program by printing the error message only and that is why we only see the error message as the output here. There are 2 main disadvantages in this program the first disadvantage is the way we have managed the memory.

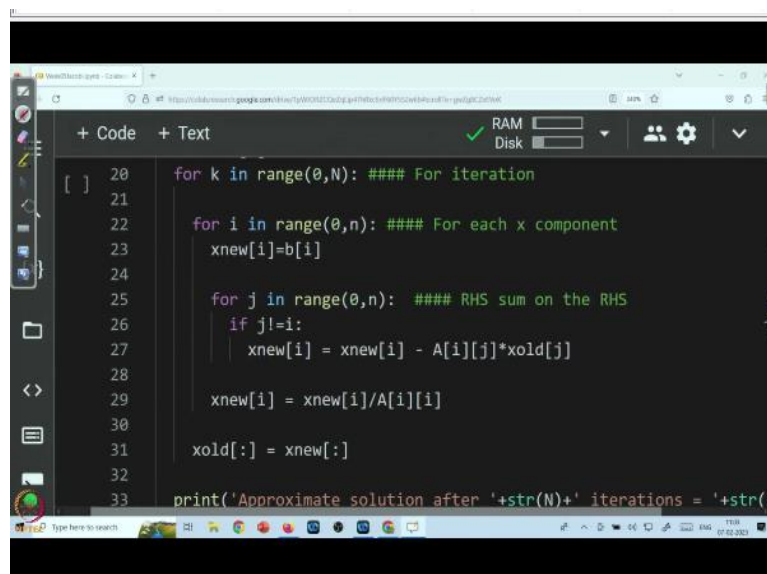
You can see that we are allocating a fresh memory for x at every iteration. That is we have first initialized $x^{(0)}$ and then when you come to compute $x^{(1)}$ you have allocated a fresh memory

for this $x^{(1)}$ and then stored the computed values of the components of $x^{(1)}$ in that memory and when you go to the second iteration again you created a fresh memory for $x^{(2)}$ and store the values of the components of $x^{(2)}$ in this memory and so on.

Every time you go for the fresh iteration you are creating a memory for that. This can be quite costly if your system is very large and you want to do many iterations then you need lot of memory while running this code, but if you carefully observe once you compute $x^{(1)}$ you no longer need the values stored in $x^{(0)}$. Because $x^{(2)}$ is computed by only keeping $x^{(1)}$ values, we never use $x^{(0)}$ values.

In general, to compute $x^{(k+1)}$ we only need $x^{(k)}$, we do not need $x^{(k-1)}$, $x^{(k-2)}$ and so on. Therefore, remembering all these computed values is unnecessary as long as the Jacobi method computation is concerned. Unless you want to remember all these vectors from the method you really do not need to remember all these vectors.

(Refer Slide Time: 29:55)

A screenshot of a code editor window showing Python code for the Jacobi method. The code is as follows:

```
20 for k in range(0,N): #### For iteration
21
22     for i in range(0,n): #### For each x component
23         xnew[i]=b[i]
24
25         for j in range(0,n): #### RHS sum on the RHS
26             if j!=i:
27                 xnew[i] = xnew[i] - A[i][j]*xold[j]
28
29         xnew[i] = xnew[i]/A[i][i]
30
31     xold[:] = xnew[:]
32
33 print('Approximate solution after '+str(N)+' iterations = '+str(xold))
```

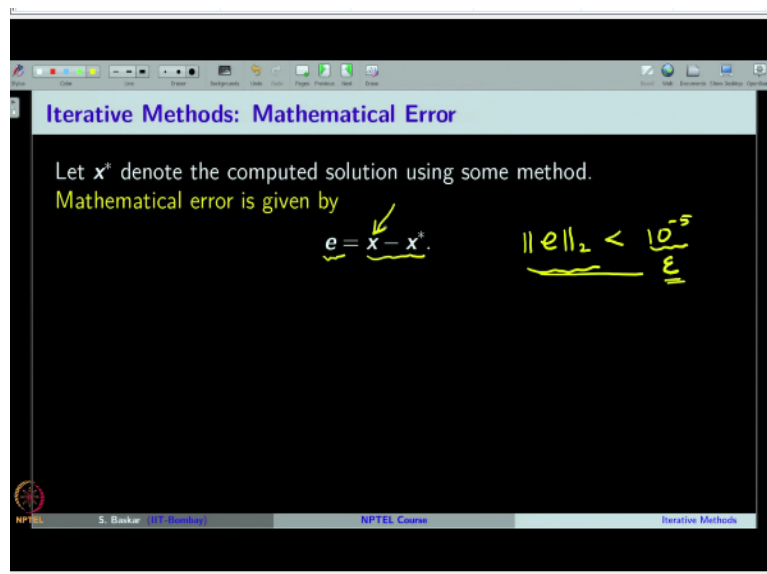
There is one more main disadvantage in our code. If you see we are asking the code to run the iterations for some pre-assigned number of times. Now we really do not know after computing the iterations this many times whether we have achieved the accuracy that we want. Say for instance at the first stage we have given only 12 iterations and we have not achieved a good accuracy.

In this particular example we know the exact solution. Therefore, we can see that we have not achieved a good accuracy and so we increase the number of iterations from 12 to 50. Then we saw that it is good but still not enough. So, we increased from 50 to 100; like that you can go on. But for that you need to know the exact solution. If you do not know the exact solution then blindly you have to specify some big number.

It may happen that after computing so many iterations you might have not still achieved the required accuracy or you might have achieved the accuracy much before than what you specified here. So, in that way going with giving number of iterations and computing the approximation is rather a blind approach. We need a good stopping criterion. That is, we want some condition that will tell us that we have achieved the required accuracy.

Therefore, we can stop the iteration at this stage. Let us see how to design a good stopping criterion.

(Refer Slide Time: 31:47)



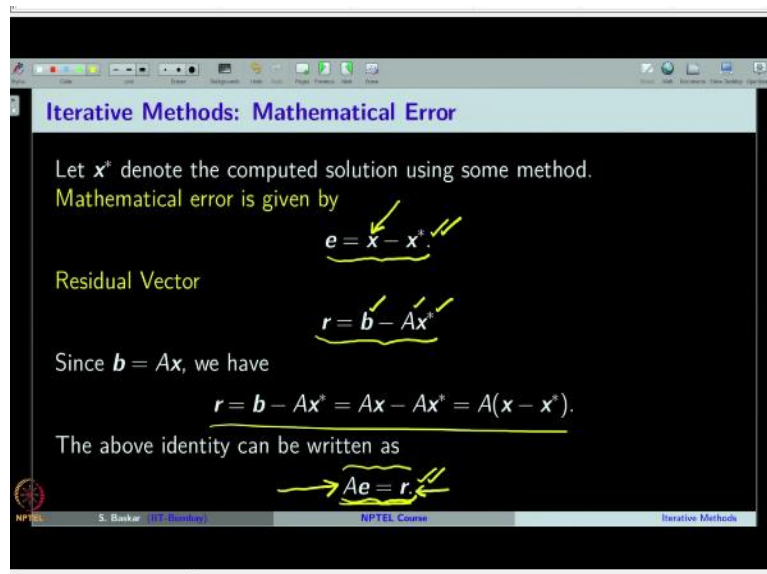
For that let us see what are all the ways that we can measure the error involved in our computed solution. Let x^* denotes the computed solution from any method. Remember, this is not only restricted to Jacobi method what we are discussing here as a stopping criteria can be used for any method. So, we are assuming x^* to be the computed solution from some method.

Then we know that the error involved in x^* when compared to x is defined as $x - x^*$ and it is denoted by e . Now remember you may also go for finding the l_2 norm or l_∞ norm or l_1 norm

of this error and then check whether this is less than. Say for instance 10^{-5} or any tolerance parameter that you specify. Say for instance you can give some ϵ value to your code.

And every time once you finish the iteration you can check for the condition that the error involved in that iteration whether it is less than that ϵ or not, but unfortunately this cannot be implemented practically because to check this condition you need to know the exact solution.

(Refer Slide Time: 33:19)



Iterative Methods: Mathematical Error

Let x^* denote the computed solution using some method.
Mathematical error is given by

$$e = x - x^*$$

Residual Vector

$$r = b - Ax^*$$

Since $b = Ax$, we have

$$r = b - Ax^* = Ax - Ax^* = A(x - x^*).$$

The above identity can be written as

$$Ae = r.$$

NPTEL S. Bankar (IIT Bombay) NPTEL Course Iterative Methods

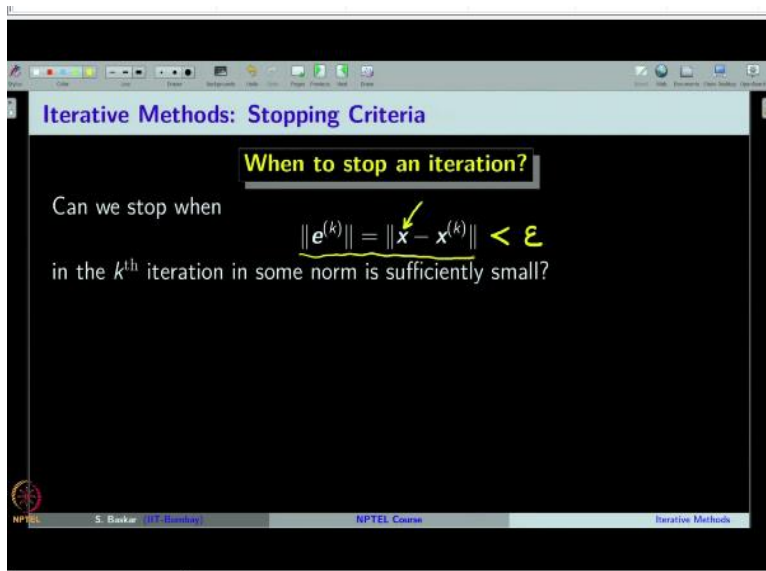
There is another way of sensing this error that is using the residual vector. Residual vector is defined as $r = b - Ax^*$. You can see that if x^* is very close to x then Ax^* will be very close to Ax which is actually equal to b . Therefore, when x^* is very close to the exact solution then r will be very close to 0, just like how if x^* is very close to x then e will be very close to 0.

Similarly, even r will go closer and closer to 0, if x^* goes closer and closer to the exact solution. You can also see an interesting fact that r can be written as Ae because simply $Ax = b - Ax^*$ and since A is a linear transformation you can also write this as $A(x - x^*)$ and therefore this can be written as $Ae = r$. An interesting observation here is that the error e can be computed as a solution of this system.

And therefore, you can compute e without knowing the exact solution. Remember if you want to use this formula to compute e you need to know the exact solution, whereas by computing e from this system you just do not need to know the exact solution. You can compute e from all the available information. Remember r is fully known to you because x^* is known to you, b is known to you and A is also known to you.

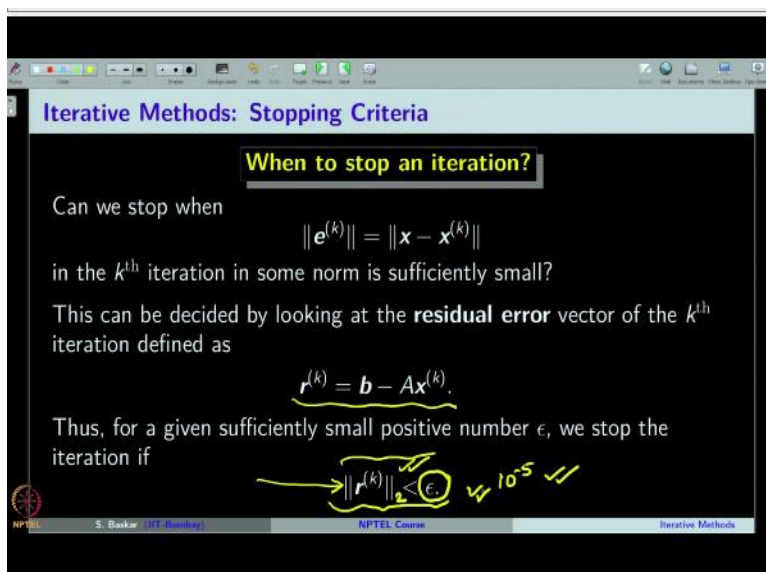
Therefore, r can be computed explicitly and then it can be plugged in as the right hand side vector and you can solve this system to get e without knowing the exact solution of your original system.

(Refer Slide Time: 35:34)



Coming back to the stopping criteria as I told, ultimately we would like to go for checking this norm to be less than ϵ , where ϵ is some pre-assigned positive quantity which is the tolerance of your error. But this is not practically applicable, because you just do not know this x or if you want to go for this condition then you may have to use this system. Solve this system to get e that may be equivalent to solving your original system itself.

(Refer Slide Time: 36:12)



An alternate idea is to compute the residual vector which can be computed directly from your approximate solution and then check for this condition. Say for instance in our code we could have computed this with a l_2 norm and we could have checked whether that quantity is less than some number say 10^{-5} or whatever the tolerance number that you give. You can check this condition after computing each iteration.

That is for each k once you finish the iteration for that particular k you can compute the residual error and then find the l_2 norm of that residual error and check this condition. If that condition is satisfied you come out of the loop, otherwise you go for the next iteration. In that way you can get the solution as accurate as you want and you can stop the iteration as soon as you achieve your accuracy.

So, this is one popular way to stop your iteration computationally. However, it has a danger that if you are working with a system where A is almost a singular matrix then you may have a very small residual error, but still you may be far away from the solution. Therefore, one has to be very careful in using this stopping criterion. However, if your A is a nice matrix then this is a good stopping criterion. With this note let us finish this lecture. Thank you for your attention.