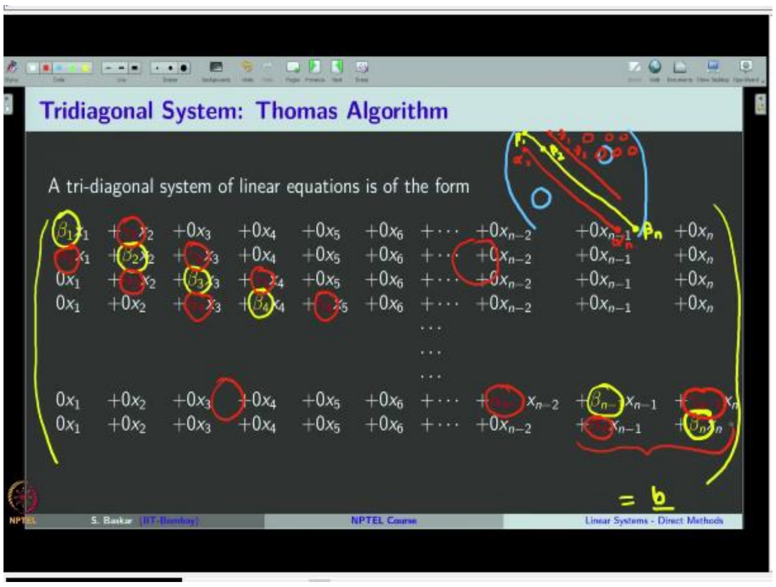
Numerical Analysis

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

Lecture-14
Linear System: Python Coding for Thomas Algorithm

Hi, in this lecture we will learn to implement a slight variation of Gaussian elimination method called Thomas algorithm which is applicable if the coefficient matrix A is a tridiagonal matrix. We will first see how to do Thomas algorithm and then we will also learn to implement Thomas algorithm in this lecture.

(Refer Slide Time: 00:43)



Tridiagonal System: Thomas Algorithm

A tri-diagonal system of linear equations is of the form

$$
\begin{array}{cccccccccc}
\beta_1 x_1 & +\gamma_1 x_2 & +0x_3 & +0x_4 & +0x_5 & +0x_6 & +\cdots & +0x_{n-2} & +0x_{n-1} & +0x_n \\
\alpha_2 x_1 & +\beta_2 x_2 & +\gamma_2 x_3 & +0x_4 & +0x_5 & +0x_6 & +\cdots & +0x_{n-2} & +0x_{n-1} & +0x_n \\
0x_1 & +\alpha_3 x_2 & +\beta_3 x_3 & +\gamma_3 x_4 & +0x_5 & +0x_6 & +\cdots & +0x_{n-2} & +0x_{n-1} & +0x_n \\
0x_1 & +0x_2 & +\alpha_4 x_3 & +\beta_4 x_4 & +\gamma_4 x_5 & +0x_6 & +\cdots & +0x_{n-2} & +0x_{n-1} & +0x_n \\
 & & & & & & \cdots & & & \\
 & & & & & & \cdots & & & \\
 & & & & & & \cdots & & & \\
0x_1 & +0x_2 & +0x_3 & +0x_4 & +0x_5 & +0x_6 & +\cdots & +\alpha_{n-1} x_{n-2} & +\beta_{n-1} x_{n-1} & +\gamma_{n-1} x_n \\
0x_1 & +0x_2 & +0x_3 & +0x_4 & +0x_5 & +0x_6 & +\cdots & +0x_{n-2} & +\alpha_n x_{n-1} & +\beta_n x_n
\end{array}
$$

$= b$

Let us consider a tri-diagonal system. A tri-diagonal system has non-zero elements only at its diagonal part and then the lower diagonal part and the upper diagonal part. Other than this all other entries are 0, this is shown in this system. Here I am only showing the left-hand side of the system $Ax = b$. So, this entire thing should be equal to the b vector on the right-hand side. Here you can see that the diagonal elements are non-zero denoted by β_i's.

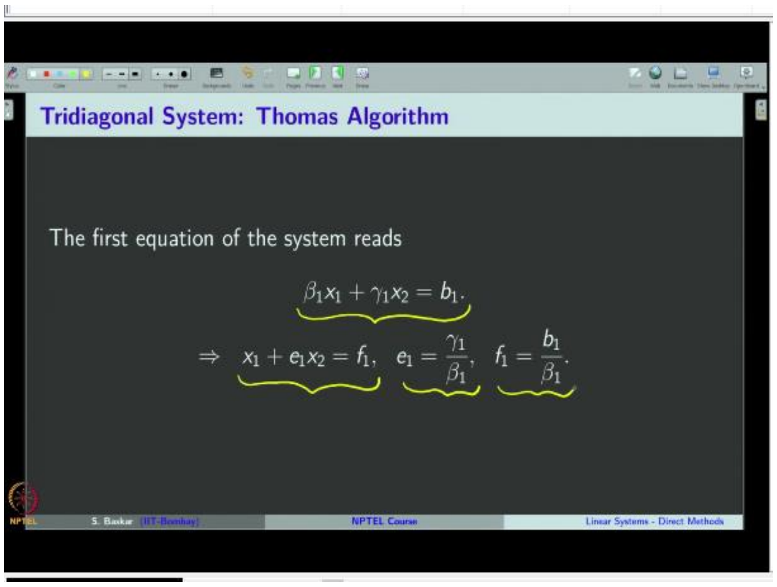
And similarly, the lower diagonal elements α's are non-zero, shown here, and the upper diagonal part γ's are also non-zero entries in the coefficient matrix. Remember few of them may be 0, but generally we will assume them as non-zero elements. Other than that everybody else will be 0. So, how the structure of the matrix will look like, therefore we have diagonal element which is taken to be non-zero then the lower diagonal elements and then the upper diagonal elements.

All other elements are zeroes here and this is how a tridiagonal system will look like. So, the first element of the matrix is denoted by β_1 and the first row second element is γ_1 and all others are 0s and then in the next line the first element is α, let us call it as α_2 then β_2, γ_2 and then all other elements are 0 like that it goes and finally at the last row you will have β_n and previous to that you will have α_n.

That is what you can see here and this is how the structure of the tridiagonal system will be. Tridiagonal systems often occur in certain applications especially when you try to build a numerical method for certain PDEs we apply finite difference methods or even finite element methods. PDE can be approximated by a linear system and that linear system in some cases will turn out to be a tridiagonal system.

And therefore, solving tridiagonal system is a very important problem. Thomas algorithm is often used in such cases. Thomas algorithm is a slight variation of the Gaussian elimination method. The underlying idea is exactly the same, but we will take the tridiagonal structure and adopt the Gaussian elimination method for this. Let us try to understand the Thomas algorithm.

(Refer Slide Time: 04:25)



Tridiagonal System: Thomas Algorithm

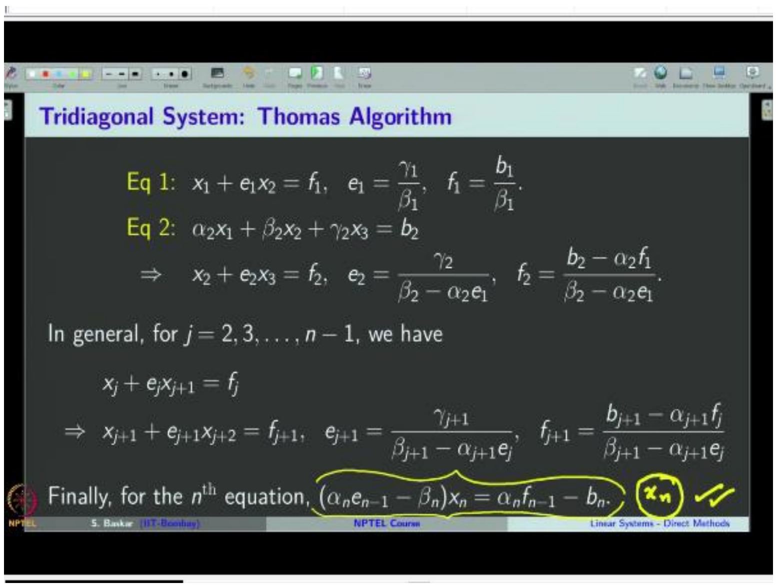
The first equation of the system reads

$$\underbrace{\beta_1 x_1 + \gamma_1 x_2 = b_1}.$$

$$\Rightarrow \quad \underbrace{x_1 + e_1 x_2 = f_1}, \quad \underbrace{e_1 = \frac{\gamma_1}{\beta_1}}, \quad \underbrace{f_1 = \frac{b_1}{\beta_1}}.$$

The first equation is given like this $\beta_1 x_1 + \gamma_1 x_2 = b_1$, we have to rewrite the first equation like this where e_1 is given by $\frac{\gamma_1}{\beta_1}$ and f_1 is given by $\frac{b_1}{\beta_1}$.

(Refer Slide Time: 04:48)



Tridiagonal System: Thomas Algorithm

Eq 1: $x_1 + e_1 x_2 = f_1, \quad e_1 = \dfrac{\gamma_1}{\beta_1}, \quad f_1 = \dfrac{b_1}{\beta_1}.$

Eq 2: $\alpha_2 x_1 + \beta_2 x_2 + \gamma_2 x_3 = b_2$

$$\Rightarrow \quad x_2 + e_2 x_3 = f_2, \quad e_2 = \frac{\gamma_2}{\beta_2 - \alpha_2 e_1}, \quad f_2 = \frac{b_2 - \alpha_2 f_1}{\beta_2 - \alpha_2 e_1}.$$

In general, for $j = 2, 3, \ldots, n-1$, we have

$$x_j + e_j x_{j+1} = f_j$$

$$\Rightarrow \quad x_{j+1} + e_{j+1} x_{j+2} = f_{j+1}, \quad e_{j+1} = \frac{\gamma_{j+1}}{\beta_{j+1} - \alpha_{j+1} e_j}, \quad f_{j+1} = \frac{b_{j+1} - \alpha_{j+1} f_j}{\beta_{j+1} - \alpha_{j+1} e_j}$$

Finally, for the n^{th} equation, $\underbrace{(\alpha_n e_{n-1} - \beta_n) x_n = \alpha_n f_{n-1} - b_n.}$ $\boxed{x_n}$ ✓✓

NPTEL S. Baskar (IIT-Bombay) NPTEL Course Linear Systems - Direct Methods

Now once you reduce the equation 1 then let us go to the equation 2. In equation 2 we have only 3 terms all other terms are 0. We have $\alpha_2 x_1 + \beta_2 x_2 + \gamma_2 x_3 = b_2$. Now what we do is you have to do an elimination process just like what we do with the Gaussian elimination method to eliminate the variable x_1 here. Once you do that then the resulting equation which involves only x_2 and x_3 on the left-hand side will be finally written in this form with an appropriate definition of e_2 and f_2.

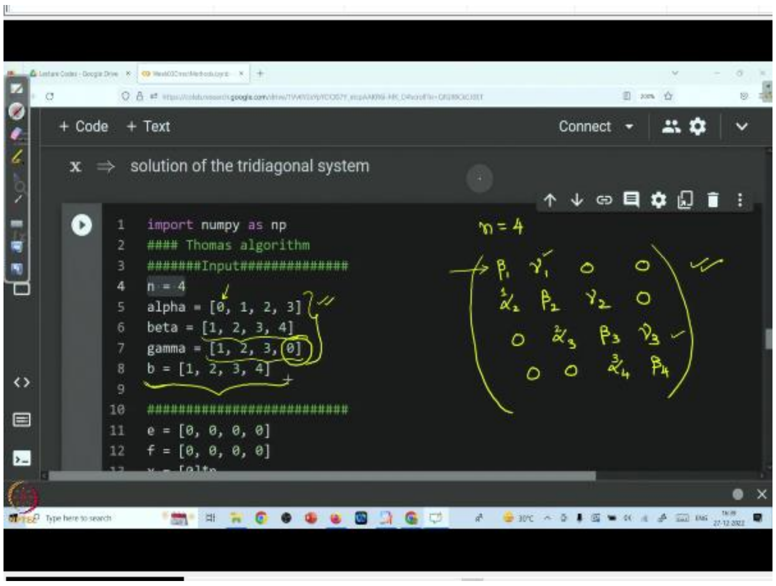
You can check this how we are getting it, just do the elimination process similar to the step 1 in the Gaussian elimination method then you will get this equation. You have to replace this equation with the second equation of our original system. Now you continue like this. In general the *j*th equation is written as $x_j + e_j x_{j+1} = f_j$ and once you write this then the next equation that is *j* + first equation is again put in this form with e_{j+1} given like this and f_{j+1} given like this.

So, this is the general form of the Thomas algorithm. Once you finish the elimination from 2 to *n* – 1. Remember the first equation is done directly, there is no elimination process for that we are only rewriting the first equation. From the second equation onwards we have the elimination process just explained in the case of second equation you have to carry over a similar idea for all the other equations starting from the second equation up to last but one equation. That is up to *n* – 1 equation.

*n*th equation again can be done in a rather direct way and you can get the *n*th equation like this. Note that we have to do an elimination process in the *n*th equation also, but once you do that the *n*th equation is just involving x_n and therefore we can get x_n directly from this equation. Once you get x_n from the *n*th equation you can substitute that into the *n* – 1 equation and get x_{n-1}.

Once you know x_{n-1} and x_n you can substitute them into the previous equation and get x_{n-2} and so on. So, in other words after doing this elimination process you can get the solution of the tridiagonal system by doing the backward substitution process. So, this is the idea of Thomas algorithm. Let us try to see how to code this method.

(Refer Slide Time: 08:23)


$\mathbf{x} \Rightarrow$ solution of the tridiagonal system

```python
import numpy as np
#### Thomas algorithm
#######Input##############
n = 4
alpha = [0, 1, 2, 3]
beta = [1, 2, 3, 4]
gamma = [1, 2, 3, 0]
b = [1, 2, 3, 4]

###########################
e = [0, 0, 0, 0]
f = [0, 0, 0, 0]
```

+ Code
+ Text
Connect
$n = 4$
$$\begin{pmatrix} \beta_1 & \gamma_1 & 0 & 0 \\ \overset{1}{\alpha_2} & \beta_2 & \gamma_2 & 0 \\ 0 & \overset{2}{\alpha_3} & \beta_3 & \gamma_3 \\ 0 & 0 & \overset{3}{\alpha_4} & \beta_4 \end{pmatrix}$$

Let us first see what are all the inputs for our code, well the dimension of the system *n* is the first input; I am just taking it as 4 and then instead of defining the entire matrix we can just define the lower diagonal elements, upper diagonal elements and the diagonal elements. Recall that we have used the notation α, β and γ for lower diagonal and upper diagonal elements.

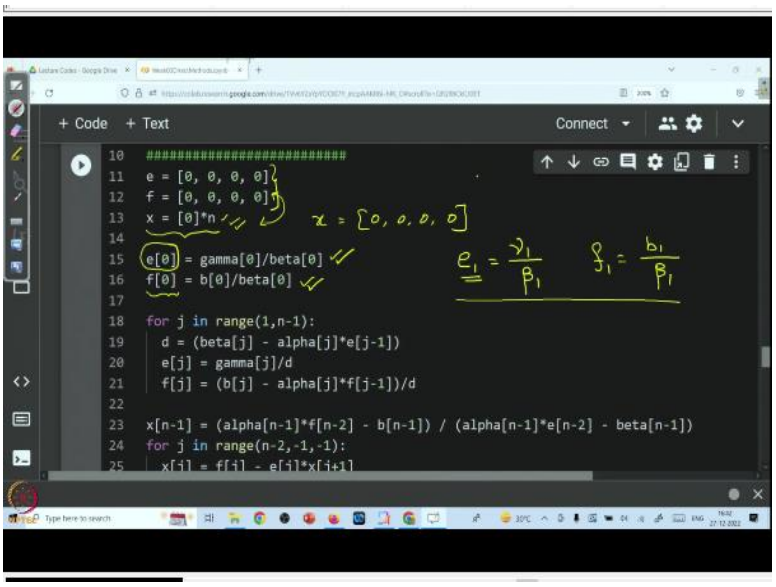
The same notation, I am using here also. Recall that the equation will look like β_1, γ_1, 0, 0 then $\alpha_2, \beta_2, \gamma_2$, 0 and then 0, $\alpha_3, \beta_3, \gamma_3$ and then 0, 0, α_4, β_4. This is the tri-diagonal system in the case of *n* = 4. Just keep this structure in mind and see how I have defined αs, βs and γs. I am defining α as a list or array where I have put 0 at the first position because there is no α in the equation 1.

That is why I put 0 here and then I am taking α_2 as 1, α_3 as 2 and α_4 as 3. Similarly, β is defined like this and you can see that β has 4 components, it appears in all the equations and similarly γ has only 3 components, it starts from the first equation and goes up to the third equation. Last equation will not have γ. That is why I have put 0 on the last component of γ.

And then I am taking b as an input and I have just given it as [1, 2, 3, 4]. Now let us go for the elimination process. If you recall we have to obtain two parameters, one is e and another one is f in the method, e is sitting here and f is coming from the right-hand side elimination process. Let us see how these things have to be incorporated in the code, well we will use this formula for e_1 and f_1.

And then for e_2 and e_3 we will use this formula. Again, for the last equation we have to plug in this expression. Let us see how to do that.

(Refer Slide Time: 11:31)



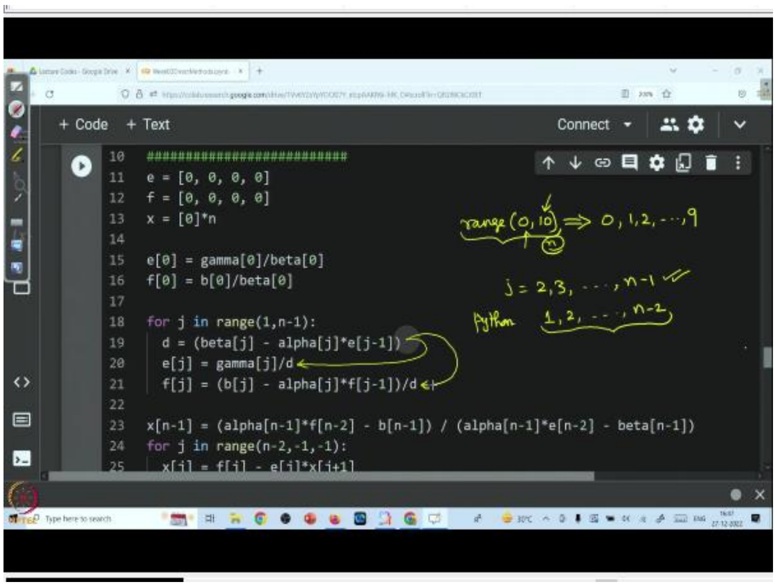
```
##########################
e = [0, 0, 0, 0]
f = [0, 0, 0, 0]
x = [0]*n

e[0] = gamma[0]/beta[0]
f[0] = b[0]/beta[0]

for j in range(1,n-1):
  d = (beta[j] - alpha[j]*e[j-1])
  e[j] = gamma[j]/d
  f[j] = (b[j] - alpha[j]*f[j-1])/d

x[n-1] = (alpha[n-1]*f[n-2] - b[n-1]) / (alpha[n-1]*e[n-2] - beta[n-1])
for j in range(n-2,-1,-1):
  x[j] = f[j] - e[j]*x[j+1]
```

We will first initialize e and f as a four-dimensional vector and we will initialize x as again a four dimensional vector, you can also write x = [0, 0, 0, 0], but just to show that we can also initialize these vectors in this form, I have just written like this. These two are equivalent in python and now we have to define e_1 that is nothing but $\frac{\gamma_1}{\beta_1}$. That is what we have seen in the method.

That is what is written in this line; remember e_1 mathematically means e[0] in python because python starts its index from 0 and goes up to n – 1. I am always emphasizing this because this is very important and often it is a confusing terminology. Therefore, I am just emphasizing it again and again. Similarly, f_1 is defined as $\frac{b_1}{\beta_1}$ and that is what given here. Now once we have the first equation, second equation onwards till n minus first equation we can go in a loop.

That is what we are theoretically writing here from j = 2 to n - 1 we can define the parameters e and f in a loop. Let us see how to do that.

(Refer Slide Time: 13:21)



```
##########################
e = [0, 0, 0, 0]
f = [0, 0, 0, 0]
x = [0]*n

e[0] = gamma[0]/beta[0]
f[0] = b[0]/beta[0]

for j in range(1,n-1):
  d = (beta[j] - alpha[j]*e[j-1])
  e[j] = gamma[j]/d
  f[j] = (b[j] - alpha[j]*f[j-1])/d

x[n-1] = (alpha[n-1]*f[n-2] - b[n-1]) / (alpha[n-1]*e[n-2] - beta[n-1])
for j in range(n-2,-1,-1):
  x[j] = f[j] - e[j]*x[j+1]
```

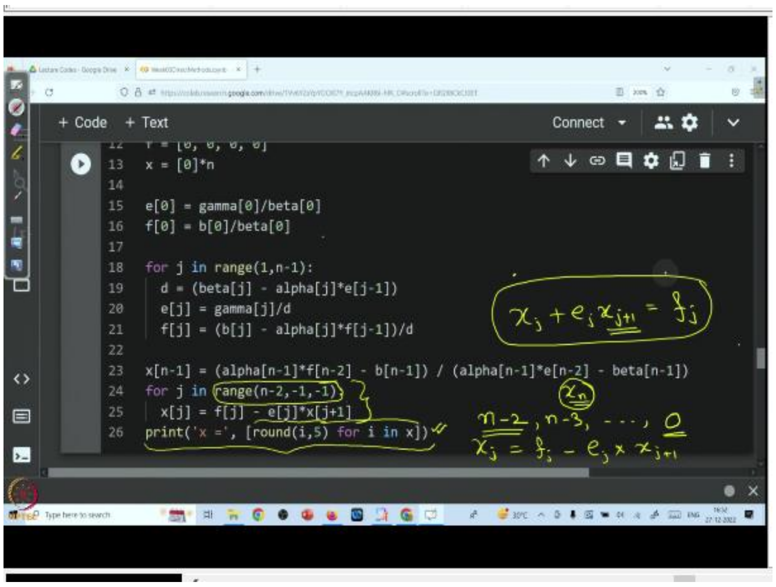
For that we will use the for loop that runs from 2 to n – 1, theoretically j runs from 2 to n - 1 and in python equivalently it has to run from 1 to n - 2 that is what is happening here. Remember range of 0, 10 will create a sequence starting from 0, 1, 2 up to 9 only, it will not include 10. That is what we have seen in one of our python session that we have to remember this confusing terminology that when you say range from some number to some number n it will go from this number till one number less than this.

Similarly, here it starts from 1 and goes up to n - 2 only, it will not include n - 1 in the sequence that the range generates. Therefore in order to create a sequence like this we have to use this command and we are taking j varying from 1 to n - 2 mathematically that is equivalent to saying 2 to n - 1 that is what precisely we want in our method and I am just going to define e[j] as, if you recall what is the expression for e_{j+1}, $\frac{\gamma_{j+1}}{\beta_{j+1}-\alpha_{j+1}e_j}$.

And a clever observation is that the denominator in e_{j+1} is the same as the denominator in f_{j+1} also. Therefore what you can do is instead of defining e [j] and f [j] with the denominator calculated separately each time you can just calculate the denominator once, store it in the variable d and then use it here as well as use it in the denominator of f. In that way what you are doing is you are saving some operations.

You can also put this expression explicitly here as well as here, but that will unnecessarily involve one additional arithmetic operation. Just to avoid that I have computed once and then plugged in those values in the expression of e and f.

(Refer Slide Time: 16:22)



```
f = [0, 0, 0, 0]
x = [0]*n

e[0] = gamma[0]/beta[0]
f[0] = b[0]/beta[0]

for j in range(1,n-1):
  d = (beta[j] - alpha[j]*e[j-1])
  e[j] = gamma[j]/d
  f[j] = (b[j] - alpha[j]*f[j-1])/d

x[n-1] = (alpha[n-1]*f[n-2] - b[n-1]) / (alpha[n-1]*e[n-2] - beta[n-1])
for j in range(n-2,-1,-1):
  x[j] = f[j] - e[j]*x[j+1]
print('x =', [round(i,5) for i in x])
```

Now you can see that we have found e[0], f[0] and then all the others right from j = 1 to n - 2. We are left out with the last equation n – 1, that is the last equation as far as the python is concerned theoretically, *n*th equation is the last equation and from the last equation we can directly get the last component of x that is x_n which is in python's notation x[n – 1] and if you recall that can be directly obtained from this last equation.

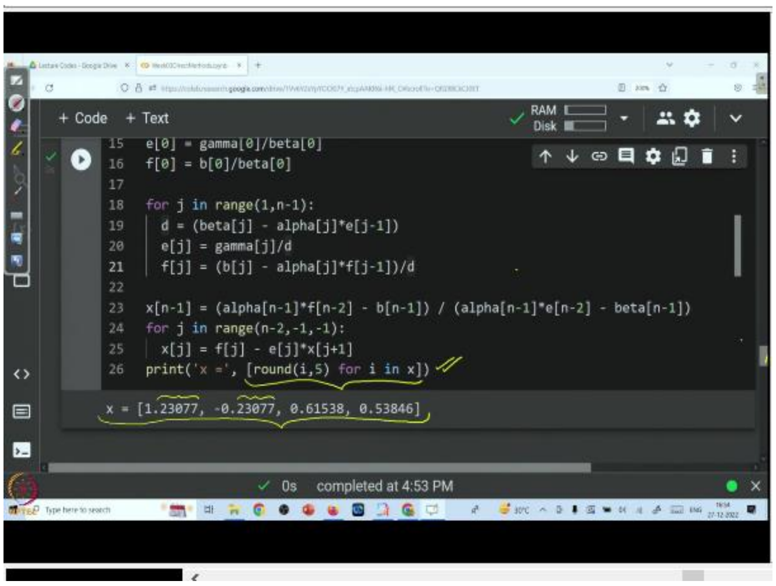
Here you can see that x_n can be written as this divided by this quantity. So that is what I am doing here x[n-1] which is nothing but our x_n is equal to the right-hand side divided by what was sitting in the left-hand side is now brought to the right-hand side. Once you have found x[n-1] then getting the other expressions are quite easy, you found which is mathematically x_n.

That you will put in the x_n minus first equation to get x_{n-1}. That is what I am doing here. Now I am creating a loop j ranging from n - 2 that is the first element of the range goes till 0. So, it has to go till 0 n - 3 and so on up to 0 it has to go that is why I put in range -1 and the increment is -1 here. It means it will keep on reducing the number till it reaches 0. So, that is what is meant by this command.

And now for each j what we are doing is $x_j = f_j - e_j x_{j+1}$. So, that is how the equation is going. Because if you recall we have written $x_j + e_j * x_{j+1} = f_j$. That was the equation and you know x_{j+1} because you know x_n from there you can get x_{n-1} and so on. So, this loop will go and do the back substitution process and finally it prints the vector x here. I have just put this command in order to get only the x values rounded to 5 decimal digits.

For that I have used this command. Now whatever the python prints from this command will be restricted to 5-digit rounding. Let us try to run this program and see the output.

(Refer Slide Time: 19:51)

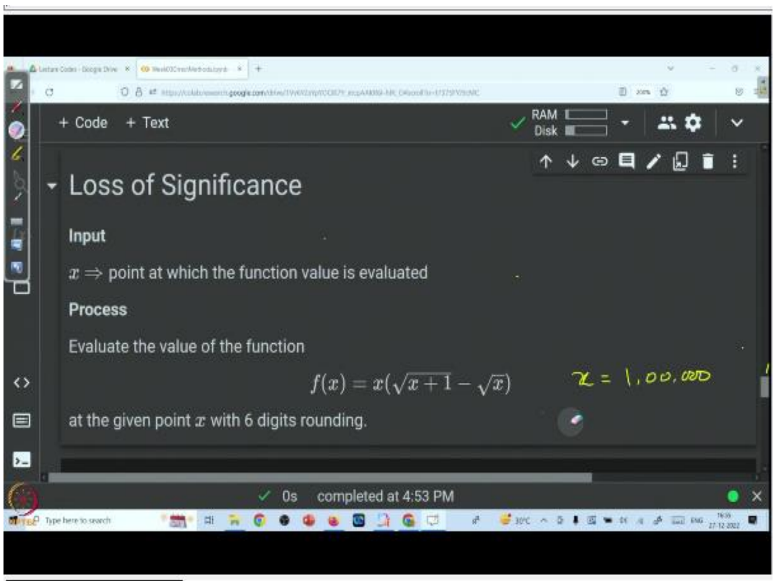


```
e[0] = gamma[0]/beta[0]
f[0] = b[0]/beta[0]

for j in range(1,n-1):
  d = (beta[j] - alpha[j]*e[j-1])
  e[j] = gamma[j]/d
  f[j] = (b[j] - alpha[j]*f[j-1])/d

x[n-1] = (alpha[n-1]*f[n-2] - b[n-1]) / (alpha[n-1]*e[n-2] - beta[n-1])
for j in range(n-2,-1,-1):
  x[j] = f[j] - e[j]*x[j+1]
print('x =', [round(i,5) for i in x])
```

```
x = [1.23077, -0.23077, 0.61538, 0.53846]
```

You can see that the program is executed and this command had printed the solution of the tridiagonal system as this, you can see that it is restricted to only 5 decimal digits in each component of x because of this command and that is the Thomas algorithm implementation.

(Refer Slide Time: 20:18)



Loss of Significance

Input

$x \Rightarrow$ point at which the function value is evaluated

Process

Evaluate the value of the function

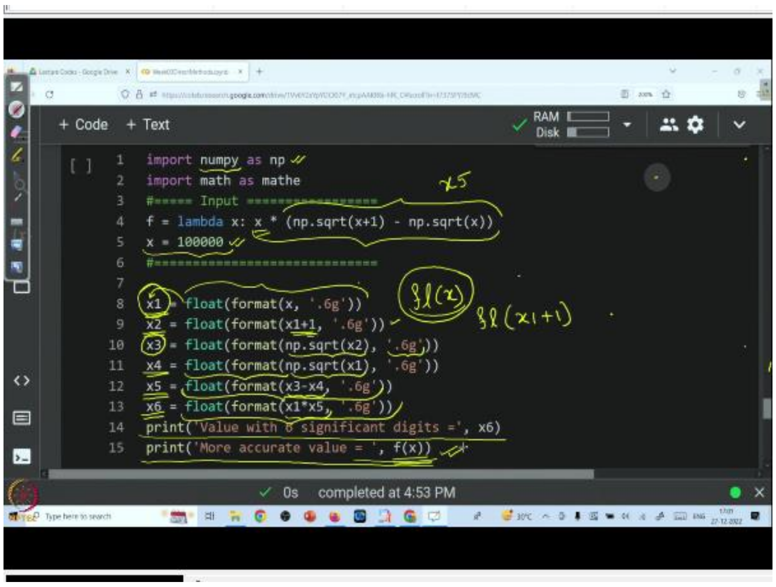
$$f(x) = x(\sqrt{x+1} - \sqrt{x})$$

at the given point x with 6 digits rounding.

Before ending this class let me also discuss an important program which I wanted to discuss in our previous python session, but because of the lack of time, I could not discuss it there but I would like to discuss it here. You can see that in one of our theory class, we have taken this function and we have calculated the value of this function at the point *x* = 100000 using 6-digit rounding.

Now how to implement this calculation as a python code? That is the question; let us try to understand how to implement this calculation on python.

(Refer Slide Time: 21:10)



```
import numpy as np
import math as mathe
#===== Input =================
f = lambda x: x * (np.sqrt(x+1) - np.sqrt(x))
x = 100000
#=============================

x1 = float(format(x, '.6g'))
x2 = float(format(x1+1, '.6g'))
x3 = float(format(np.sqrt(x2), '.6g'))
x4 = float(format(np.sqrt(x1), '.6g'))
x5 = float(format(x3-x4, '.6g'))
x6 = float(format(x1*x5, '.6g'))
print('Value with 6 significant digits =', x6)
print('More accurate value = ', f(x))
```

Well to define the function you can also use this command which will create a function f and assigns this expression symbolically and you can evaluate the function value by just plugging in $f(x)$ here. So, that can give you in one go the value of the function f which is equal to

$x\left(\sqrt{x+1}-\sqrt{x}\right)$. Remember square root is not a python command, it is inbuilt in one of the libraries or modules whatever you say and that module is numpy.

Therefore, I am calling numpy and then executing the square root here. And we want to find the value of this function at the point x = 100000, so I have also assigned the variable x = 100000. Now this line will get the value of f at 100000 with of course 52 digits in mantissa because generally python computes values in double precision, but what we want is to compute the value of f at x = 100000 using 6-digit rounding.

How to do that? Well, you can do it using this command. The command is float open brackets format open bracket whatever value you want to give and then this .6g. Suppose if you want to use 7-digit rounding then you have to use .7g or any n-digit rounding you have to use dot ng. That n value you have to plug in here. That is the format what we are doing is first we are finding fl(x).

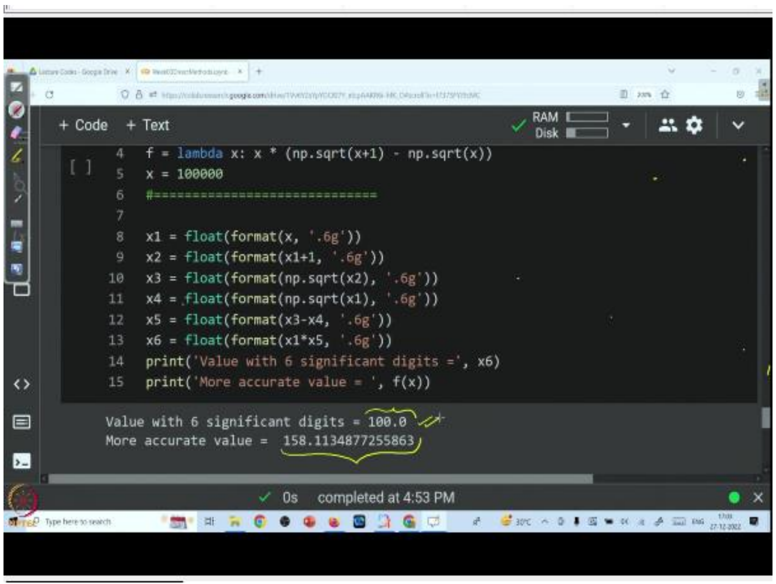
If you recall in one of our previous lectures, we have explained how to do arithmetic operation using n-digit rounding, you first have to round the number x and then you add that number with 1. To get this x + 1 expression, now what I am doing is, I am rounding x right to 6 decimal digits, storing that value in the variable x1 and then I am adding 1 to the value x1. Then I am again finding the 6-digit rounding of x1 + 1.

That is what I am doing here and that value is stored in x2 and then I am taking square root of x2. That is, I am computing this term here, after computing square root of x2, again I am taking 6-digit rounding of that and storing it in x3. Once you get x3 you keep it on one side and go to do this second term, remember the argument of the second term is fl(x) that is stored in x1.

Therefore, I am taking square root of x1 and then doing a 6-digit rounding of that and storing it in x4. Now x3 precisely has the 6-digit rounding of this term and x4 has 6-digits rounding of the second term. Now what I am doing is, I am subtracting these two terms and again taking the 6-digit rounding of that term and that is stored in x5. So, finally you have the complete 6-digit rounding calculation of this term stored in the variable x5. And then remember already fl(x) is stored in x1.

So, finally you are doing x1 * x5 and then doing the 6-digit rounding of that term and that is stored in x6. Therefore the final answer that we want, that is we want the value of this function at the point *x* = 100000 with 6-digit rounding calculation, is finally stored in x6 and that is what I am printing in this line. And I am also printing the value of the function f directly with 52 digit rounding because it is a double precision calculation and let us see the output of this program.

(Refer Slide Time: 26:16)



```python
f = lambda x: x * (np.sqrt(x+1) - np.sqrt(x))
x = 100000
#==============================

x1 = float(format(x, '.6g'))
x2 = float(format(x1+1, '.6g'))
x3 = float(format(np.sqrt(x2), '.6g'))
x4 = float(format(np.sqrt(x1), '.6g'))
x5 = float(format(x3-x4, '.6g'))
x6 = float(format(x1*x5, '.6g'))
print('Value with 6 significant digits =', x6)
print('More accurate value = ', f(x))
```

```
Value with 6 significant digits = 100.0
More accurate value =  158.1134877255863
```

The output of the function with 6-digit rounding is 100, whereas the more accurate answer is 158.113. If you go back to our class where we discuss this problem there also we have shown the output of this calculation as 100 which we have now calculated using a python code. With this we will end this class. Thank you for your attention.