

Tools in Scientific Computing
Prof. Aditya Bandopadhyay
Department of Mechanical Engineering
Indian Institute of Technology, Kharagpur

Lecture - 24
Boundary Value Problems

(Refer Slide Time: 00:35)

The screenshot shows a Jupyter Notebook interface. The main content area displays a slide titled "Boundary Value Problems". The slide features a diagram of a string fixed at both ends, represented by a horizontal line with vertical arrows at each end pointing upwards. Below the diagram, the following text is written: $\frac{d^2f}{dx^2} = f$ [linear ODE], $x \in [0, 2]$, and at $x=0, f(0) =$. The Jupyter Notebook interface includes a file browser on the left, a central area with icons for Notebook, Python 3, and Console, and a bottom status bar. A small video feed of the professor is visible in the bottom right corner of the notebook window.

Hello everyone in this particular week we are going to study about Boundary Value Problems. So, far we have considered equations which evolved in time and they are classified as initial value problems; because the initial conditions are known and then the system state and things such as plotting the state in the phase diagram, they are the evolution of the parameters in accordance to how they are governed and their initial value problems.

But in a different category of problems we can have a domain such as a string and then you can impose certain conditions. For example, we can have a displacement of the string like this and this end of the string is left free things like that. So, things which we are familiar with things pertaining to vibrations and all these things or displacements and they all fall within the purview of boundary value problems.

So, without any delay let us look at a particular very simple boundary value problem $\frac{d^2f}{dx^2} = f$. So, it is a linear ODE and we have to define over what domain is this

particular equation valid. So, let us take it over the domain 0 to 2 and at $x = 0$; let us say $f(x)$ is equal to 1 mean directly right. The first boundary condition as $f(0) = 1$, second condition could be $f'(2) = 0$.

(Refer Slide Time: 02:37)

So, this in a solid mechanics viewpoint this would be a prescribed displacement and this would be a free or unconstrained member. So, something which maintains a slope, but it does not define what the displacement will be ok.

So, it has to just maintain a slope it could go from something like this also I mean not this much, it could go something like this all it has to do is maintain this displacement there is no prescription for it. And maybe if we have to find that out as a part of the solution who knows.

So, these are the two conditions and this condition is known as the Dirichlet condition and this particular condition is known as the Neumann boundary condition. So, here the value of the function is specified at the end point one of the endpoints and over here the derivative is specified. And so, this is how you distinguish between a Dirichlet and a Neumann.

Alternately you could have the situation where you have a mixture of both that is something like $f'(2) + 2f(2) = a$, or something. So, it is a mixture of first derivative and

the specified value of the function and this kind of boundary condition is known as the Robin boundary condition ok.

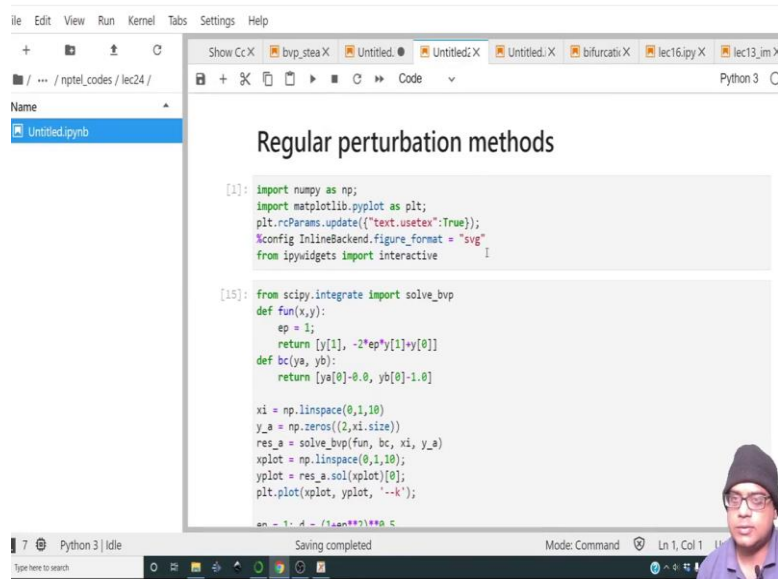
So, how do we go over approaching this problem? Well, first thing we could do is obtain the analytical solution. So, what is the solution for this? So, $f = A \cosh x + B \sinh x$ and that is the straightforward solution. I mean you could write the solution for this as $f = Ae^x + Be^{-x}$, that is a linear combination of e^x and e^{-x} where the constants A and B are determined using the boundary condition.

So in fact, let us use this hyperbolic form. So, we have $f = A \cosh x + B \sinh x$ using the first boundary condition $f(0) = 1$. So, 1 so when once you substitute $x = 0$ this term will become 1; so, this will be A. And this particular term will become 0, because \sinh of 0 is 0. So, A becomes equal to 1.

And similarly using the Neumann boundary condition so what do we have? $0 = A \sinh x|_{x=2} + B \cosh x|_{x=2}$; this becomes $A \sinh 2 + B \cosh 2$ and A is already equal to 1 and this implies $B = \frac{-\sinh 2}{\cosh 2}$.

So, combining this so this is the solution ok where A is this and $B = \frac{-\sinh 2}{\cosh 2}$. So, we could write down the expression by substituting all this over here, but because we are going to plot it in python anyway or octave as you like. So, we are not going to bother much about that.

(Refer Slide Time: 07:13)



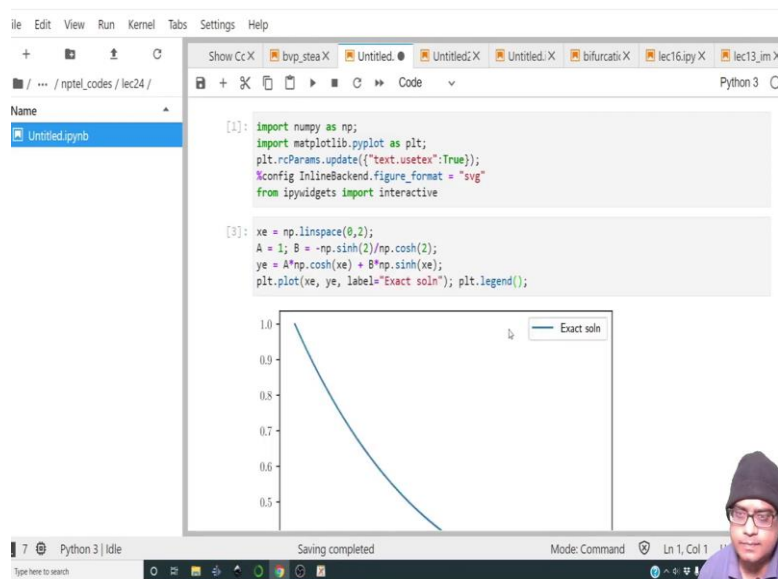
```
file Edit View Run Kernel Tabs Settings Help
+ Show Cc X bvp_stea X Untitled: X Untitled: X Untitled: X bifurcati X lec16.ipyn X lec13_im X
Python 3
Name
  Untitled.ipynb
Regular perturbation methods

[1]: import numpy as np;
import matplotlib.pyplot as plt;
plt.rcParams.update({"text.usetex":True});
%config InlineBackend.figure_format = "svg"
from ipynbwidgets import interactive

[15]: from scipy.integrate import solve_bvp
def fun(x,y):
    ep = 1;
    return [y[1], -2*ep*y[1]+y[0]]
def bc(ya, yb):
    return [ya[0]-0.0, yb[0]-1.0]

xi = np.linspace(0,1,10)
ya = np.zeros((2,xi.size))
res_a = solve_bvp(fun, bc, xi, ya)
xplot = np.linspace(0,1,10);
yplot = res_a.sol(xplot)[0];
plt.plot(xplot, yplot, '--k');
```

(Refer Slide Time: 07:20)



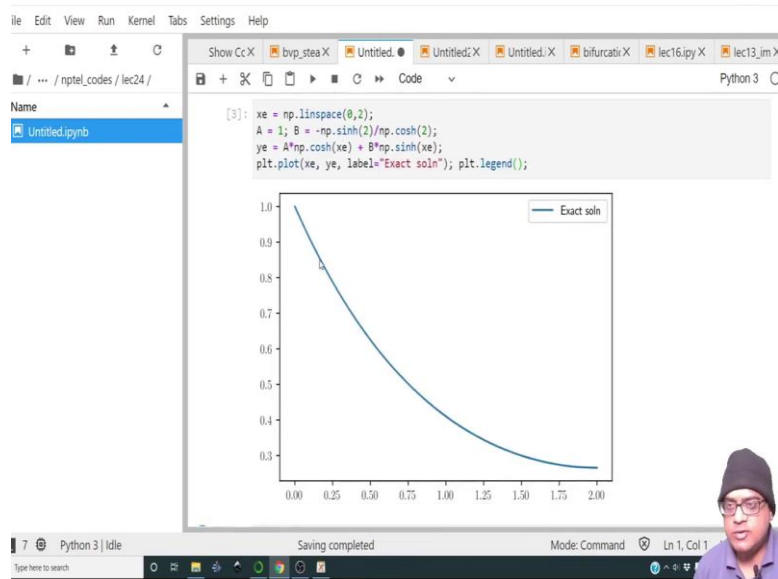
```
file Edit View Run Kernel Tabs Settings Help
+ Show Cc X bvp_stea X Untitled: X Untitled: X Untitled: X bifurcati X lec16.ipyn X lec13_im X
Python 3
Name
  Untitled.ipynb
[1]: import numpy as np;
import matplotlib.pyplot as plt;
plt.rcParams.update({"text.usetex":True});
%config InlineBackend.figure_format = "svg"
from ipynbwidgets import interactive

[3]: xe = np.linspace(0,2);
A = 1; B = -np.sinh(2)/np.cosh(2);
ye = A*np.cosh(xe) + B*np.sinh(xe);
plt.plot(xe, ye, label="Exact soln"); plt.legend();
```

So, let us go and create a new file. So, let me copy this bit of code alright. So, let me then create a linspace between 0 and 2. And let me in fact, let me call it xe this is for x exact and y exact is $ye = A \cdot \text{np.cosh}(xe) + B \cdot \text{np.sinh}(xe)$ we can send this to the next line because we need to define the constants A and B.

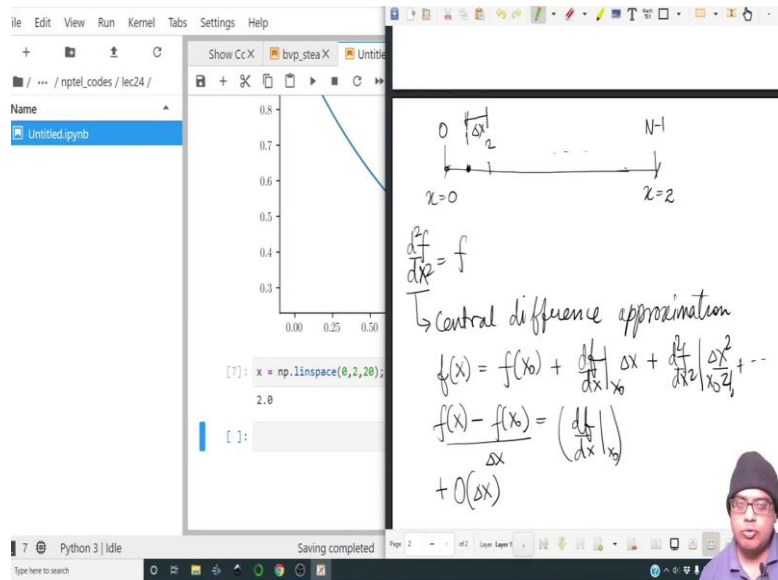
So, A is 1 while $B = -\text{np.sinh}(2)/\text{np.cosh}(2)$. Alternately you would have written it as $-\text{np.tanh}(2)$, but anyway this is fine. So, with this let us do a plot.

(Refer Slide Time: 08:31)



So, this is how the exact solution looks like and the value of the function at the other endpoint that is $x = 2$ appears to be something which is near to 0.3, but less than 0.3. So, this is what we have from the exact same, but now we are interested to solve this numerically. So, how do we go about solving it numerically? Let us see.

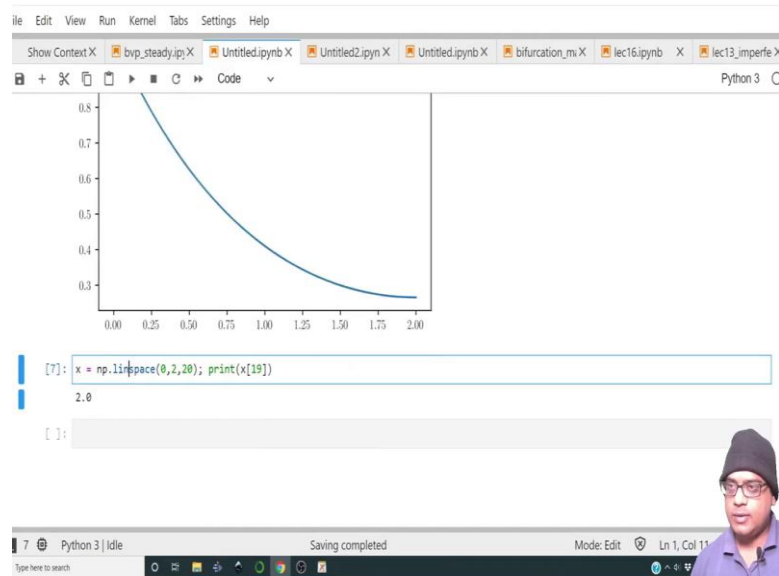
(Refer Slide Time: 09:08)



So, the first thing that we do the first very first thing that we go over doing is to discretize the domain into a bunch of points. So, this is $x = 0$ and this is $x = 2$. This is

going to be so, we make a linspace out of this. So, this will be 0, 1, 2 all the way to N-1. So, if we have a linspace which has an elements. So, let us do that and see.

(Refer Slide Time: 09:42)



So and let us keep this code over here and let us write this here. So, $x = \text{np.linspace}(0,2,20)$ and let us say we take 20 points. And yeah so let us print x just for good measure. So, this is how x looks like the 0th point is 0. And the last point so 0 will be 0 whereas, 19 will be 2. So, it is going all the way from 0 to $N - 1$ alright.

So, now our task is to discretize this particular equation over on this grid. So, we have what $\frac{d^2 f}{dx^2} = f$. So, how do we discretize a function at a point? So, I must write the function in terms of the neighboring points. So, the very easy way of doing it is writing a central difference approximation alright.

So, I mean I whether or not it is clear how to do this I will just give you a gist of how it is done. So, we have $f(x) = f(x_0) + \frac{df}{dx} \Big|_{x_0} \Delta x + \frac{d^2 f}{dx^2} \Big|_{x_0} \frac{\Delta x^2}{2!} + \dots$. So, now $\frac{f(x) - f(x_0)}{\Delta x}$ gives us a good estimate of what $\frac{df}{dx} \Big|_{x_0}$ is going to be. But we have neglected the other terms, but the magnitude of the other term. So, I have divided everything by Δx . So, this will be $+O(\Delta x)$.

Meaning whatever I am neglecting is going to be proportional to the Δx that I have over the grid, if Δx is small then this approximation is also good if Δx is large this approximation is not going to be good. So, how do you go about finding the central difference approximation? Well you have on one hand the approximation for f of near x_0 .

(Refer Slide Time: 12:37)

And similarly, you can write an approximation for f . So, what we have? This is x_0 this is I mean well. So, I just write an approximation for $f(-x)$ that is the point on the other side.

$$\text{So, } f(-x) = f(x_0) - \frac{df}{dx} \Big|_{x_0} \Delta x + \frac{df^2}{dx^2} \Big|_{x_0} \frac{\Delta x^2}{2!} - \dots$$

So, as you can imagine if I add these 2 equations the odd terms will cancel out because Δx^3 term will be there and here it will be $-\Delta x^3$. So, all these odd terms will cancel out.

$$\text{So, finally, we have } f(x) + f(-x) = 2f(x_0) + \frac{df^2}{dx^2} \Big|_{x_0} \Delta x^2 + O(\Delta x^4) \text{ and so on.}$$

$$\text{And so, this gives us a good measure of the second derivative as } \frac{f(x) - 2f(x_0) + f(-x)}{\Delta x^2}.$$

And so, the error will now be order Δx^2 because I have divided everything by Δx^2 . So, this also is gets divided by Δx .

So, such an approximation is said to be second order accurate because as Δx becomes small, by say a factor of half the error is expected to reduce by a factor of one-fourth

which is in stark contrast to the forward difference approximation for a single derivative. Well I have rub that now.

But similarly, I could do what we had earlier $\frac{df}{dx} \Big|_{x_0} = \frac{f(x) - f(x_0)}{\Delta x} + O(\Delta x)$. And now I can choose whatever x and $-x$ is to be a point coinciding with the neighboring grid. So, now x becomes x_1 and $-x$ becomes $x-1$. Because $f(-x)$ is simply the $-$ first node and this x will become the first node, if this node is the 0th node.

(Refer Slide Time: 15:24)

The image shows a Python IDE window with a plot and handwritten mathematical derivations. The plot shows a curve on a grid. The handwritten notes include:

$$f(x) = f(x_0) + \frac{df}{dx} \Big|_{x_0} \Delta x + \frac{d^2f}{dx^2} \Big|_{x_0} \frac{\Delta x^2}{2} + O(\Delta x^3)$$

$$f(-x) = f(x_0) - \frac{df}{dx} \Big|_{x_0} \Delta x + \frac{d^2f}{dx^2} \Big|_{x_0} \frac{\Delta x^2}{2} - O(\Delta x^3)$$

$$f(x) + f(-x) = 2f(x_0) + \frac{d^2f}{dx^2} \Big|_{x_0} \Delta x^2 + O(\Delta x^4)$$

$$\frac{d^2f}{dx^2} \Big|_{x_0} = \frac{f(x) - 2f(x_0) + f(-x)}{\Delta x^2} + O(\Delta x^2)$$

$$\frac{df}{dx} \Big|_{x_0} = \frac{f(x) - f(x_0)}{\Delta x} + O(\Delta x)$$

$$\frac{df}{dx} \Big|_{x_i} = \frac{f(x_{i+1}) - f(x_{i-1}))}{2\Delta x}$$

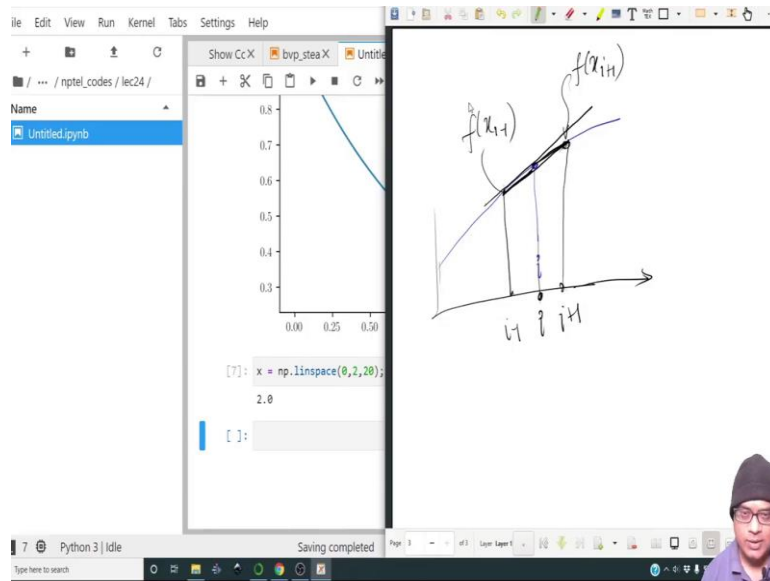
The IDE window shows a plot of a function and the following code in the console:

```
[7]: x = np.linspace(0,2,20);
      2.0
[ ]:
```

So, this approximation was first order accurate, but you can actually evaluate a second order accurate derivative and the way to do it is; is going to be $f(i+1)$. So, if I call

$$\text{this } \frac{df}{dx} \Big|_{x_i} = \frac{f(x_{i+1}) - f(x_{i-1}))}{2\Delta x}.$$

(Refer Slide Time: 15:53)



Essentially it is saying and if this is a function if this is 1 point i . So, if you want to approximate the slope at this point which is something like this, I can discretize the grid over which it is defined. So, this is i this is $i + 1$, this is $i - 1$. So, this point is $f(x_{i-1})$ this point is $f(x_{i+1})$. So, I can approximate it by taking a difference between these two points that is the central difference I can approximate it between these two points this is the finite the forward difference.

I can in fact, approximately using these two points as well that is the backward difference. So, depending on the scheme that you use 1 will arrive at an approximation to that particular derivative which will have some order of accuracy. And as to what that accuracy will be is a matter of what your choice is alright.

(Refer Slide Time: 17:15)

So, let us get rolling with this so yeah. So, $\frac{d^2 f}{dx^2} = f$. So, I can do a finite difference approximation at the i th node. So, what this will be? Will be $\frac{f_{i-1} + 2f_i + f_{i+1}}{\Delta x^2} = f_i$. So obviously, the left hand side is order Δx^2 accurate. Now, let me collect the terms containing I mean let me segregate the terms f_{i-1} , f_i and f_{i+1} .

So, I have f_{i-1} ok. So, f_{i-1} is obviously, referring to f at x_{i-1} ok. So, this is just a small shorthand for writing $f_{i-1} + f_i(-2 - \Delta x^2) + f_{i+1} = 0$. So, you have many nodes and here is 0 and you have many nodes and here you have $N - 1$ ok here you have $N - 1$.

So, now this particular discretization let me call it as A. So, this is valid for the interior nodes because we must treat the boundary conditions a bit a bit differently. In the sense that I know that $f(0) = 1$ and $\left. \frac{df}{dx} \right|_{x=2} = 0$.

(Refer Slide Time: 19:15)

Valid for interior node

$$f(0) = 1, \quad \left. \frac{df}{dx} \right|_{x=2} = 0$$

$$f_0 = 1, \quad \frac{f_{N-1} - f_{N-2}}{\Delta x} = 0 \quad N-1$$

$i=0 \quad f_0 = 1$
 $i=1 \quad f_0 + f_1 \alpha + f_2 = 0$
 $i=2 \quad f_1 + f_2 \alpha + f_3 = 0$
 \vdots
 \vdots
 $i=N-1 \quad f_{N-1} - f_{N-2} = 0$

Python 3 | Idle Saving completed

Now, while we are added let us have sort of discretization for this fellow as well. So, this is quite trivial this is simply going to be $f(0)$ is going to be 1 this is going to be. So, let me take a backward difference at the final node. So, backward difference at the final node will involve $N - 1$ and $N - 2$ fine. So, this will be $\frac{f_{N-1} - f_{N-2}}{\Delta x} = 0$, alright.

So, this is for node number 0 this is for node number $N - 1$, but what about the inner nodes. Let me substitute $i = 1$ over here. So, what do I have? I have $f(0) + f(1)$. So, let me call this as α alpha because; it is not going to change over the equations because we will have a regular grid meaning Δx is not changing across various locations on the grid. So, that kind of a grid is called as a uniform grid or a regular grid no problem.

So, its going to be $f_1 \alpha + f_2 = 0$. Now, what I can do is I can write it for $i = 2, 3$ and so on. So, this is for $i = 1, i = 2$ i will have $f_1 + f_2 \alpha + f_3 = 0$ and so on. Why $= 0$; obviously, i have $f(0) = 1$ and for f sorry or $i = N - 1$ i have $f_{N-1} - f_{N-2} = 0$ because Δx can go on the right hand side and it will be; obviously, 0.

(Refer Slide Time: 21:14)

The python and octave notebooks

$i=0$ $f_0 = 1$

$i=1$ $f_0 + f_1 \alpha + f_2 = 0$

$i=2$ $f_1 + f_2 \alpha + f_3 = 0$

\vdots

$i=N-1$ $f_{N-1} - f_{N-2} = 0$

$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ \alpha & 1 & 0 & \dots & 0 \\ 0 & 1 & \alpha & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{N-1} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$

$[7]: x = \text{np.linspace}(0, 2, 20);$

2.0

$[]:$

Now, what I can do is write down these equations. So, for N equations I have N unknowns. The N unknowns are f_0, f_1, f_2 all the way to f_{N-1} . And how many equations do we have? We do have an equation starting from $1 = 0$ to $i = N - 1$.

So, its a set of linear algebraic equations. So, we have converted the ordinary differential equation into a set of linear equations alright. So, if I can go ahead and write this in a matrix form how will this particular set of equations look? So, over here I have f_0, f_1, f_2 all the way to f_{N-1} . So, f_{N-2} comes on top and this will be equal to something.

(Refer Slide Time: 22:08)

$i=N-1$ $f_{N-1} - f_{N-2} = 0$

$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ \alpha & 1 & 0 & \dots & 0 \\ 0 & 1 & \alpha & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{N-1} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$

$A \vec{x} = \vec{b}$

\vec{x}

$[8]: N = 20;$

$x = \text{np.linspace}(0, 2, N);$

$A = \text{np.zeros}(N, N); b = \text{np.zeros}(N, 1);$

$[]:$

So, this kind of equation will always come out in the form $Ax = b$. Where A is like the mass matrix I mean it borrows its nomenclature from solid mechanics or structural mechanics its stiffness matrix ok. So, b this is the right hand side ok let us not alert to that no increase or not. So, we simply obtain this set of equations say $Ax = b$ forgotten most of structural mechanics.

But yeah you do get an equation of this particular form over there as well alright. So, this is a matrix, this is a vector, this is a vector. So, let us populate those vectors. So, what do we have. So, f_0 so for the first node $f_0 = 1$. So, this will be $1 \ 0 \ 0 \ 0 \ 0 = 1$. So, this is $1 \times f_1 = 1$ over here we have $1 \ \alpha \ 1$ then all 0s this is going to be equal to 0 over here we have.

So, there is no f_0 over here. So, this becomes 0 is α this is 1 is $\alpha \ 1$ and everything else is going to be 0. Again, 1 sorry this is $0 \ 1 \ \alpha \ 1 \ 0$ so, this continues on. So, apart from the first row and the last row this keeps on continuing this tridiagonal set keeps on continuing up until the last node upon which we have a backward difference to take care of alright.

So, the backward difference in this case is quite simple it simply implies $f_{N-1} = f_{N-2}$. So, this is -1 over here and this is going to be 1 and this is equal to 0. Why is it -1 ? So, all these are going to be 0 is $-1 \times f_{N-2} +$ so, basically this particular term $+1 \times f_{N-1}$ is equal to 0 alright great.

So, now our task is to construct the matrix A and the vector b and then solve for x ok. So, solution x I mean over here I am writing it as x , but it will be really f great. So, let us go over here in fact, yeah. So, we do have an expression for x . So, now, what will be the size of the matrix A ? So, A has to be a square matrix having the same size as x .

So, it will be np dot zeros and the size will be. So, let me soft code it as N ; where N will be the number of points. So, $N = 20$ alright great. So, A has been initialized to 0s. Now what will be b ? So, b will be np dot zeros. So, its N rows and 1 column alright. So, A has been initialized b has been initialized. Now it is a matter of constructing the appropriate structure for A . So, what we could do is run a loop from $i = 0$ to $N - 1$ and based on this particular logic we can assign all the different members of the matrix.

(Refer Slide Time: 26:06)

The image shows a Jupyter Notebook interface with a Python code cell and a plot. The code defines a 1D Poisson problem on a grid with $N=20$ nodes. The plot shows a smooth curve representing the solution. To the right of the code, there are handwritten mathematical notes in blue and black ink. These notes describe the finite difference discretization of the second-order differential equation, showing the stencil for interior nodes ($i=1$ to $i=N-1$) and the boundary conditions at $i=0$ and $i=N-1$. The boundary conditions are given as $f_0 = 1$ and $f_{N-1} - f_{N-2} = 0$. A matrix equation $Ax = b$ is also shown, with the matrix A being tridiagonal and the vector b containing the source term and boundary values.

So, we are not going to do it like that, but I am just going to outline how to do it in case you are doing it in c or something where vectorization is not possible this is how you would do it for $i = 1$ sorry for i in 1 np.arrange $0, N$.

So, A so what do we have? So, if $i = 0$, then you will say $A_{0, 0} = 1$ else; $i = N - 1$ or rather yeah $= n - 1$, then what will we do? We will say a $N - 1, N - 2 = -1$ $A_{N - 1, N - 1} = -1$ rather 1.

So, these two if else statements take care of the boundary conditions rather the boundary nodes; if they are not the boundary nodes then we continue as normal else A_N or rather $A_{i, i}$. Well I could have written this in terms of i as well $i - 1$, I think it is better to write it in terms of i .

(Refer Slide Time: 27:32)

The image shows a Python Jupyter notebook on the left and handwritten mathematical derivations on the right. The notebook code is as follows:

```

[9]: N = 20;
x = np.linspace(0,2,N); dx = x[1] - x[0];
A = np.zeros((N, N)); b = np.zeros((N,1));
alpha = -2 - dx**2;

for i in np.arange(0,N):
    if i == 0:
        A[i,i] = 1;
    elif i == N-1:
        A[i,i-1] = -1; A[i,i] = 1;
    else:
        A[i,i] = alpha; A[i,i-1] = 1; A[i,i+1] = 1;

[10]: print(A)

```

The output of the print statement is:

```

[[ 1.  0.  0.  0.  0.
  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.
  1. -2.01108033  1.  0.  0.]

```

The handwritten notes on the right show the following derivations:

$$f_{i-1} = f(x_{i-1})$$

$$f_{i-1} - 2f_i + f_{i+1} = f_i$$

$$\frac{f_{i-1} + f_i(-2 - \alpha) + f_{i+1}}{\Delta x^2} = 0 \quad \text{A}$$

Valid for interior node

$$f(0) = 1, \quad \left. \frac{df}{dx} \right|_{x=2} = 0$$

$$f_0 = 1, \quad \frac{f_{N-1} - f_{N-2}}{\Delta x} = 0 \quad N-1$$

Handwritten equations for nodes:

$$i=0 \quad f_0 = 1$$

$$i-1 \quad f_0 + f_1 \alpha + f_2 = 0$$

$$i-2 \quad f_1 + f_2 \alpha + f_3 = 0$$

A small inset in the bottom right corner shows a person's face.

Because its already doing the check whether i is = N - 1. So, I do not need to write it in terms of N - 1 i can simply write it in terms of i yeah. I think that is much better. So, A i , i will be = α A oops i , i - 1 is going to be 1 A i , i + 1 is also going to be 1. So, essentially it is assigning the three values and you can figure it out whether how this is going to work.

So, let me define what α is. So, alpha we had defined as $-2 - \Delta x^2$. So, $-2 \times -dx^2$. So, what is going dx going to b?e dx is going to be $x_1 - x_0$ its simply the difference between 2 nodes alright. So, let me run this and let me show you how A looks like. In fact, let me reduce the number to something manageable like 5 alright. So, this is how A looks like.

(Refer Slide Time: 28:47)

Python code in the notebook:

```

[11]: N = 5;
x = np.linspace(0,2,N); dx = x[1] - x[0];
A = np.zeros((N, N)); b = np.zeros((N,1));
alpha = -2 - dx**2;

for i in np.arange(0,N):
    if i == 0:
        A[i,i] = 1;
    elif i == N-1:
        A[i,i-1] = -1; A[i,i] = 1;
    else:
        A[i,i] = alpha; A[i,i-1] = 1; A[i,i+1] = 1;

[12]: print(A)

[[[ 1.  0.  0.  0.  0. ]
 [ 1. -2.25 1.  0.  0. ]
 [ 0.  1. -2.25 1.  0. ]
 [ 0.  0.  1. -2.25 1. ]
 [ 0.  0.  0. -1.  1. ]]]

```

Handwritten notes:

$f_{i-1} = f(x_{i-1})$

$$\frac{f_{i-1} - 2f_i + f_{i+1}}{\Delta x^2} = f_i$$

$$f_{i-1} + f_i \frac{\alpha}{(-2 - \Delta x^2)} + f_{i+1} = 0 \quad \text{A}$$
 Valid for interior node
 $f(0) = 1, \quad \frac{df}{dx}|_{x=2} = 0$
 $f_0 = 1, \quad \frac{f_{N-1} - f_{N-2}}{\Delta x} = 0 \quad N-1$

Matrix equations for nodes:

$$\begin{cases} i=0 & f_0 = 1 \\ i=1 & f_0 + f_1 \alpha + f_2 = 0 \\ i=2 & f_1 + f_2 \alpha + f_3 = 0 \end{cases} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \end{bmatrix}$$

So, look its 1 over here and all 0s and 1 - 2.25 1, 1 - 2.25 1 and the boundary nodes are different, but everything else is like a tridiagonal matrix.

(Refer Slide Time: 29:06)

Python code in the notebook:

```

[13]: N = 7;
x = np.linspace(0,2,N); dx = x[1] - x[0];
A = np.zeros((N, N)); b = np.zeros((N,1));
alpha = -2 - dx**2;

for i in np.arange(0,N):
    if i == 0:
        A[i,i] = 1;
    elif i == N-1:
        A[i,i-1] = -1; A[i,i] = 1;
    else:
        A[i,i] = alpha; A[i,i-1] = 1; A[i,i+1] = 1;

[12]: print(A)

[[[ 1.  0.  0.  0.  0.  0.  0. ]
 [ 1. -2.25 1.  0.  0.  0.  0. ]
 [ 0.  1. -2.25 1.  0.  0.  0. ]
 [ 0.  0.  1. -2.25 1.  0.  0. ]
 [ 0.  0.  0.  1. -2.25 1.  0. ]
 [ 0.  0.  0.  0. -1.  1.  0. ]
 [ 0.  0.  0.  0.  0. -1.  1. ]]]

```

Handwritten notes:

$f_{N-1} - f_{N-2} = 0$

Matrix equation:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \alpha & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & \alpha & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & \alpha & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & \alpha & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$\underline{\underline{A}} \underline{\underline{x}} = \underline{\underline{b}}$

(Refer Slide Time: 29:11)

```
file Edit View Run Kernel Tabs Settings Help
Show Context X bvp_steady.ipyn X Untitled.ipynb X Untitled2.ipyn X Untitled.ipynb X bifurcation_m X lec16.ipynb X lec13_imperfe X
Python 3
A[1,1] = alpha; A[1,1-1] = 1; A[1,1+1] = 1;

[14]: print(A)

[[ 1.  0.  0.  0.  0.  0.
  0.  ]
 [ 1. -2.111111111 1.  0.  0.  0.
  0.  ]
 [ 0.  1. -2.111111111 1.  0.  0.
  0.  ]
 [ 0.  0.  1. -2.111111111 1.  0.
  0.  ]
 [ 0.  0.  0.  1. -2.111111111 1.
  0.  ]
 [ 0.  0.  0.  0.  1. -2.111111111
  1.  ]
 [ 0.  0.  0.  0.  0.  0. -1.
  1.  ]]
```

And actually, boundary value problems where you are discretizing with the nearest neighbor it does resemble a tridiagonal matrix alright ok. So, yeah this is what you can do to construct A, but its actually looping over all the elements and doing and we can alternately take a great shortcut. Let me comment [FL] I cannot comment over here ok. But let me split cells and keep it for later. So, what we will do is we will use the np dot diag function. So, what is the np dot diag function?

(Refer Slide Time: 29:59)

```
file Edit View Run Kernel Tabs Settings Help
Python 3
dv = alpha*np.ones((N,));
dv1

[13]: for i in np.arange(0,N):
      if i == 0:
          A[i,i] = 1;
      elif i == N-1:
          A[i,i-1] = -1; A[i,i] = 1;
      else:
          A[i,i] = alpha; A[i,i-1] = 1; A[i,i+1] = 1;

[19]: a = [1, 2, 3]; A = np.diag(a,0); print(A)
      b = [5, 6]; B = np.diag(b, 1); print(B)

[[1 0 0]
 [0 2 0]
 [0 0 3]]
[[0 5 0]
 [0 0 6]
 [0 0 0]]
```

Handwritten diagram showing a matrix equation $Ax = b$. The matrix A is tridiagonal with 1s on the main diagonal and α on the sub and super diagonals. The vector x is $[x_0, x_1, x_2, \dots, x_{N-1}]^T$ and the vector b is $[b_0, b_1, b_2, \dots, b_{N-1}]^T$. The equation is written as $Ax = b$ with A and x underlined.

So, let me show you. So, let some small a be 1 2 3 right. So, $A = \text{np.diag}(a, 0)$. So, now, let me sorry let me show what A is. So, it will take a vector and it will push all the vector elements to the diagonal elements. So, I have to pass a vector which is having the same size as the length of the square matrix alright. So, over here it is 3 and so on. So, this gives us an idea. So, because all of them are going to be α except the zeroth node and the $N - 1$ node I can handle them easily ok.

So, let me write this. So, diagonal vector is going to be α times np.ones the size is going to be N rows and its just a vector $dv1$ is the offset ok. So, $dv1$ is the offset meaning if b is = 5, 6 and I write $B = \text{np.diag}(b, 1)$ then let me print B its going to take the vectors and it is going to put the vectors in the shifted location to the diagonal.

The shift is specified by this. So, over it is inserting b the elements of b into the matrix capital B and having a shift of 1. So obviously, it has to have 1 less element the more you shift the lesser elements you should have great that gives us a different idea.

(Refer Slide Time: 31:55)

The image shows a video lecture interface. On the left is a Jupyter Notebook with the following code:

```
[20]: N = 5;
x = np.linspace(0,2,N); dx = x[1] - x[0];
A = np.zeros((N, N)); b = np.zeros((N,1));
alpha = -2 - dx**2;

dv = alpha*np.ones((N,));
dv1 = np.ones((N-1,));
A = np.diag(dv) + np.diag(dv1,1) + np.diag(dv1,-1);
print(A)
```

The output of the code is a 5x5 matrix:

```
[[-2.11111111 1. 0. 0. 0.
  0. 0. -2.11111111 1. 0.
  0. 0. 0. -2.11111111 1.
  0. 0. 0. 0. -2.11111111
  0. 0. 0. 0. 0. 1.]]
```

On the right is a hand-drawn diagram illustrating the matrix structure. It shows a tridiagonal matrix A multiplied by a vector x equals a vector b . The matrix has 1s on the main diagonal and α on the diagonals shifted by 1. The vector x has elements x_0, x_1, x_2, x_3, x_4 and the vector b has elements b_0, b_1, b_2, b_3, b_4 . A note says $f_{N-1} - f_{N-2} = 0$. A small inset shows the speaker's face.

So, this simply going to be $\text{np.ones}(N - 1)$. So, yeah that is it. So, now, a will be $\text{np.diag}(dv) + \text{np.diag}(dv1, 1) + \text{np.diag}(dv1, -1)$. Where I have used the fact that the upper shifted diagonal and the lower shifted diagonal have the same value to be inserted ok.

(Refer Slide Time: 32:32)

The screenshot shows a Jupyter Notebook with the following Python code:

```

[13]: 7;
      = np.linspace(0,2,N); dx = x[1] - x[0];
      = np.zeros((N, N)); b = np.zeros((N,1));
      pha = -2 - dx**2;

      = alpha*np.ones((N,));
      1 = np.ones((N-1));
      = np.diag(dv) + np.diag(dv1,1) + np.diag(dv1,-1)

[18]: for i in np.arange(0,N):
      if i == 0:
          A[i,i] = 1;
      elif i == N-1:
          A[i,i-1] = -1; A[i,i] = 1;
      else:
          A[i,i] = alpha; A[i,i-1] = 1; A[i,i+1] = 1

[19]: a = [1, 2, 3]; A = np.diag(a,0); print(A)
      b = [5, 6]; B = np.diag(b, 1); print(B)

[[1 0 0]]
  
```

Handwritten notes on the right side of the notebook show the matrix structure for $i = N-1$:

$$f_{N-1} - f_{N-2} = 0$$

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ \alpha & 1 & 0 & \dots & 0 \\ 0 & 1 & \alpha & 1 & \dots & 0 \\ 0 & 0 & 1 & \alpha & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \dots & 1 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{N-1} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Below the matrix, it is noted that $\underline{Ax} = \underline{b}$ and \underline{x} .

(Refer Slide Time: 32:43)

The screenshot shows a Jupyter Notebook with the following Python code:

```

[21]: N = 5;
      x = np.linspace(0,2,N); dx = x[1] - x[0];
      A = np.zeros((N, N)); b = np.zeros((N,1));
      alpha = -2 - dx**2;

      dv = alpha*np.ones((N,));
      dv1 = np.ones((N-1));
      A = np.diag(dv) + np.diag(dv1,1) + np.diag(dv1,-1);
      print(A)

[[-2.25  1.   0.   0.   0. ]
 [ 1.  -2.25  1.   0.   0. ]
 [ 0.   1.  -2.25  1.   0. ]
 [ 0.   0.   1.  -2.25  1. ]
 [ 0.   0.   0.   1.  -2.25]]

[13]: for i in np.arange(0,N):
      if i == 0:
          A[i,i] = 1;
      elif i == N-1:
          A[i,i-1] = -1; A[i,i] = 1;
      else:
          A[i,i] = alpha; A[i,i-1] = 1; A[i,i+1] = 1
  
```

Handwritten notes on the right side of the notebook show the matrix structure for $i = N-1$:

$$f_{N-1} - f_{N-2} = 0$$

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ \alpha & 1 & 0 & \dots & 0 \\ 0 & 1 & \alpha & 1 & \dots & 0 \\ 0 & 0 & 1 & \alpha & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \dots & 1 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{N-1} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Below the matrix, it is noted that $\underline{Ax} = \underline{b}$ and \underline{x} .

So, having done this let us print what A is. Let me reduce this to 5. So, great we have the tridiagonal matrix as we want. Now, I will unset the boundary nodes to be zeros.

(Refer Slide Time: 32:54)

The screenshot shows a Python Jupyter notebook on the left and a whiteboard on the right. The notebook code is as follows:

```
[22]: N = 5;
x = np.linspace(0,2,N); dx = x[1] - x[0];
A = np.zeros((N, N)); b = np.zeros((N,1));
alpha = -2 - dx**2;

dv = alpha*np.ones((N,));
dv1 = np.ones((N-1,));
A = np.diag(dv) + np.diag(dv1,1) + np.diag(dv1,-1);
A[0,:] = 0; A[N-1,:] = 0; print(A)
```

The output of the code is a 5x5 matrix:

```
[[ 0.  0.  0.  0.  0. ]
 [ 1. -2.25  1.  0.  0. ]
 [ 0.  1. -2.25  1.  0. ]
 [ 0.  0.  1. -2.25  1. ]
 [ 0.  0.  0.  0.  0. ]]
```

The whiteboard on the right shows the equation $f_{N-1} - f_{N-2} = 0$ and a matrix equation $Ax = b$. The matrix A is shown as a 5x5 matrix with a diagonal of 1s and a super-diagonal of 1s. The vector b is shown as a 5x1 vector with a 1 in the first row and 0s elsewhere.

So, $A_{0,0}$ all is going to be 0; $A_{N-1, N-1}$ all is also going to be 0. Let me print and show that we have indeed unset these 0 and the last node great.

(Refer Slide Time: 33:14)

The screenshot shows a Python Jupyter notebook on the left and a whiteboard on the right. The notebook code is as follows:

```
[22]: N = 5;
x = np.linspace(0,2,N); dx = x[1] - x[0];
A = np.zeros((N, N)); b = np.zeros((N,1));
alpha = -2 - dx**2;

dv = alpha*np.ones((N,));
dv1 = np.ones((N-1,));
A = np.diag(dv) + np.diag(dv1,1) + np.diag(dv1,-1);
A[0,:] = 0; A[N-1,:] = 0;

A[0,0] = 1;
A[N-2,N-1] = -1; A[N-1,N-1]
```

The output of the code is a 5x5 matrix:

```
[[ 0.  0.  0.  0.  0. ]
 [ 1. -2.25  1.  0.  0. ]
 [ 0.  1. -2.25  1.  0. ]
 [ 0.  0.  1. -2.25  1. ]
 [ 0.  0.  0.  0.  0. ]]
```

The whiteboard on the right shows the equation $f_{N-1} - f_{N-2} = 0$ and a matrix equation $Ax = b$. The matrix A is shown as a 5x5 matrix with a diagonal of 1s and a super-diagonal of 1s. The vector b is shown as a 5x1 vector with a 1 in the first row and 0s elsewhere.

So, now we can simply incorporate the boundary condition. So, $A_{0,0}$ is going to be 1 and the other boundary condition is going to be $A_{N-2, N-1} = -1$ $A_{N-1, N-1}$ or we do not need to write $N-1$. I am just being silly over here.

(Refer Slide Time: 33:34)

The screenshot shows a Jupyter Notebook on the left and a whiteboard on the right. The notebook code is as follows:

```

[23]: N = 5;
x = np.linspace(0,2,N); dx = x[1] - x[0];
A = np.zeros((N, N)); b = np.zeros((N,1));
alpha = -2 - dx**2;

dv = alpha*np.ones((N,));
dv1 = np.ones((N-1,));
A = np.diag(dv) + np.diag(dv1,1) + np.diag(dv1,-1);
A[0,:] = 0; A[N-1,:] = 0;

A[0,0] = 1;
A[-1,-2] = -1; A[-1,-1] = 1;
b[0] = 1;
print(A)

[[ 1.  0.  0.  0.  0. ]
 [ 1. -2.25  1.  0.  0. ]
 [ 0.  1. -2.25  1.  0. ]
 [ 0.  0.  1. -2.25  1. ]
 [ 0.  0.  0. -1.  1. ]]

[13]: for i in np.arange(0,N):

```

The whiteboard shows a handwritten matrix equation $A\vec{x} = \vec{b}$ and a diagram of a tridiagonal matrix structure. The matrix is shown as:

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ \alpha & 1 & 0 & \dots & 0 \\ 0 & 1 & \alpha & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{N-1} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

The whiteboard also shows the equation $f_{N-1} - f_{N-2} = 0$ and a diagram of a tridiagonal matrix structure.

Because we can simply write $A - 1, -2$ is -1 and $A - 1, -1 = 1$. And we have seen this in the very beginning that -1 is simply same as $N - 1$. Let me in fact, print A and convince you.

So, $A_{0,0}$ has been set to 1 ; $A_{-1,-2}$ has been set to -1 and $A_{-1,-1}$ has been set to 1 . So, now, I must also set the b matrix. So, b_0 is going to be 1 . So, b_0 is simply going to be 1 that is it. So, now, we do have the system of equations say $Ax = b$ and its just a matter of solving it.

(Refer Slide Time: 34:33)

The screenshot shows a Jupyter Notebook on the left and a whiteboard on the right. The notebook code is as follows:

```

dv = alpha*np.ones((N,));
dv1 = np.ones((N-1,));
A = np.diag(dv) + np.diag(dv1,1) + np.diag(dv1,-1);
A[0,:] = 0; A[N-1,:] = 0;

A[0,0] = 1;
A[-1,-2] = -1; A[-1,-1] = 1;
b[0] = 1;

f = np.dot(np.linalg.pinv(A), b);

```

The whiteboard shows a handwritten matrix equation $A\vec{x} = \vec{b}$ and a diagram of a tridiagonal matrix structure. The matrix is shown as:

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ \alpha & 1 & 0 & \dots & 0 \\ 0 & 1 & \alpha & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{N-1} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

The whiteboard also shows the equation $f_{N-1} - f_{N-2} = 0$ and a diagram of a tridiagonal matrix structure.

So, how do you solve? I mean because its a very trivial case we take an inverse of A and multiply it. So, $A^{-1}A f = A^{-1} b$ and but $A^{-1}A$ is going to be the identity matrix. So, $\vec{f} = A^{-1}b$ and $\vec{f} = \vec{f}$. So, the solution $\vec{f} = A^{-1}b$.

So in fact, let us do it I mean its not the most efficient way of doing it most way, but let us do it regardless. So, f is going to be np dot inv A and now we have to do a matrix multiplication.

(Refer Slide Time: 35:22)

The screenshot shows a Jupyter Notebook interface. On the left, the code cell contains the following Python code:

```

13 b[0] = 1;
14
15 f = np.dot(np.inv(A), b);

```

Below the code, an `AttributeError` is displayed, indicating that the `numpy` module does not have an `inv` attribute. The error message is:

```

AttributeError: module 'numpy' has no attribute 'inv'

```

On the right side of the notebook, there is a handwritten slide with mathematical derivations. The slide shows a matrix equation:

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ \alpha & 1 & 0 & \dots & 0 \\ 0 & \alpha & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{N-1} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

The slide also shows the derivation:

$$\vec{f} = A^{-1}b$$

and the matrix equation:

$$A \vec{f} = b$$

The slide also shows the derivation:

$$\vec{f} = \vec{I} \vec{f} = A^{-1}b$$

(Refer Slide Time: 35:25)

The screenshot shows a Jupyter Notebook interface. On the left, the code cell contains the following Python code:

```

dv = alpha*np.ones((N,));
dv1 = np.ones((N-1,));
A = np.diag(dv) + np.diag(dv1,1) + np.diag(dv1,
A[0,:] = 0; A[N-1,:] = 0;
A[0,0] = 1;
A[-1,-2] = -1; A[-1,-1] = 1;
b[0] = 1;
f = np.dot(np.linalg.pinv(A), b);

```

Below the code, an `AttributeError` is displayed, indicating that the `numpy` module does not have an `inv` attribute. The error message is:

```

AttributeError: module 'numpy' has no attribute 'inv'

```

On the right side of the notebook, there is a handwritten slide with mathematical derivations. The slide shows a matrix equation:

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ \alpha & 1 & 0 & \dots & 0 \\ 0 & \alpha & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{N-1} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

The slide also shows the derivation:

$$\vec{f} = A^{-1}b$$

and the matrix equation:

$$A \vec{f} = b$$

The slide also shows the derivation:

$$\vec{f} = \vec{I} \vec{f} = A^{-1}b$$

(Refer Slide Time: 35:36)

```
file Edit View Run Kernel Tabs Settings Help
Show Context X bvp_steady.ipyn X Untitled.ipynb X Untitled2.ipyn X Untitled.ipynb X bifurcation_m X lec16.ipynb X lec13_imperfe X
Python 3
x = np.linspace(0,2,N); dx = x[1] - x[0];
A = np.zeros((N, N)); b = np.zeros((N,1));
alpha = -2 - dx**2;

dv = alpha*np.ones((N,));
dv1 = np.ones((N-1,));
A = np.diag(dv) + np.diag(dv1,1) + np.diag(dv1,-1);
A[0,:] = 0; A[N-1,:] = 0;

A[0,0] = 1;
A[-1,-2] = -1; A[-1,-1] = 1;
b[0] = 1;

f = np.dot(np.linalg.inv(A), b);

[13]: for i in np.arange(0,N):
      if i == 0:
          A[i,i] = 1;
      elif i == N-1:
          A[i,i-1] = -1; A[i,i] = 1;
      else:
          A[i,i] = alpha; A[i,i-1] = 1; A[i,i+1] = 1;

[10]: a = [1, 2, 3]; A = np.diag(a,0); print(A)
      b = [5, 6]; B = np.diag(b, 1); print(B)
```

So, np.dot A * b there is there appears to be an error np. inv. So, that the function inv actually lies inside the linalg sub module yeah.

(Refer Slide Time: 35:43)

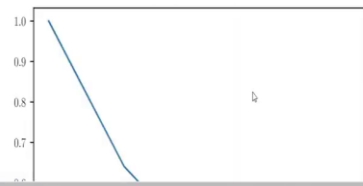
```
file Edit View Run Kernel Tabs Settings Help
Show Context X bvp_steady.ipyn X Untitled.ipynb X Untitled2.ipyn X Untitled.ipynb X bifurcation_m X lec16.ipynb X lec13_imperfe X
Python 3
alpha = -2 - dx**2;

dv = alpha*np.ones((N,));
dv1 = np.ones((N-1,));
A = np.diag(dv) + np.diag(dv1,1) + np.diag(dv1,-1);
A[0,:] = 0; A[N-1,:] = 0;

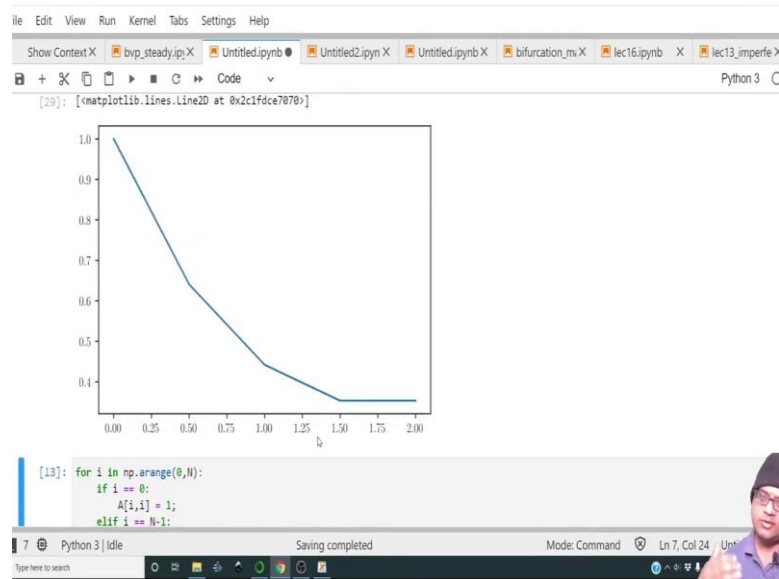
A[0,0] = 1;
A[-1,-2] = -1; A[-1,-1] = 1;
b[0] = 1;

f = np.dot(np.linalg.inv(A), b);
plt.plot(x, f)

[20]: [ <matplotlib.lines.Line2D at 0x2c1f4ce7070> ]
```



(Refer Slide Time: 35:47)



So, now that we have done this we can plot it. So, `plt.plot(x, f)`. Well let us try to superpose this with the earlier solution what was it x_e and y_e .

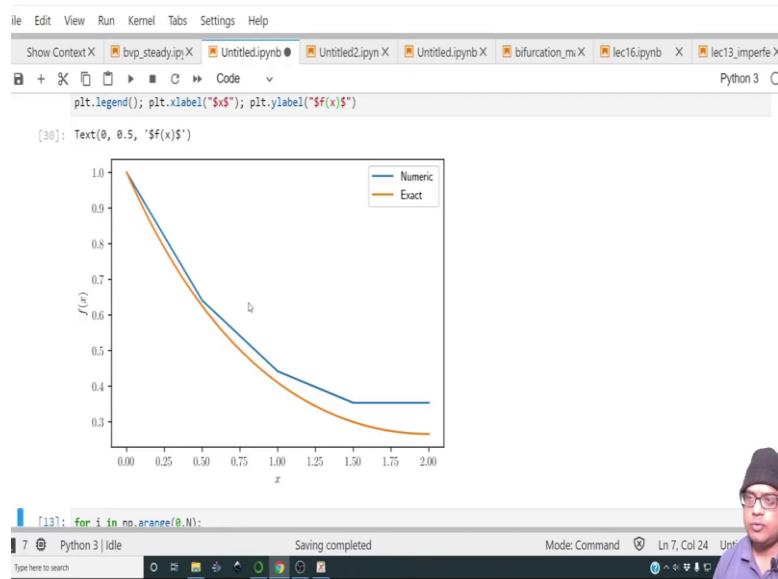
(Refer Slide Time: 35:56)

The screenshot shows a Jupyter Notebook interface with the following code:

```
A = np.diag(dv) + np.diag(dv1,1) + np.diag(dv1,-1);  
A[0,:] = 0; A[N-1,:] = 0;  
  
A[0,0] = 1;  
A[-1,-2] = -1; A[-1,-1] = 1;  
b[0] = 1;  
  
f = np.dot(np.linalg.inv(A), b);  
plt.plot(x, f, label="Numeric");  
plt.plot(xe, ye, label="Exact");  
plt.legend(); plt.xlabel("$x$"); plt.ylabel("$f(x)$")  
  
[30]: Text(0, 0.5, "$f(x)$")  
  
[13]: for i in np.arange(0,N):  
      if i == 0:  
          A[i,i] = 1;  
      elif i == N-1:  
          A[i,i] = -1; A[i,i] = 1;  
      else:  
          A[i,i] = alpha; A[i,i-1] = 1; A[i,i+1] = 1;  
  
[10]: a = [1, 2, 3]; A = np.diag(a,0); print(A)  
      b = [5, 6]; B = np.diag(b, 1); print(B)  
      print(A, B)
```

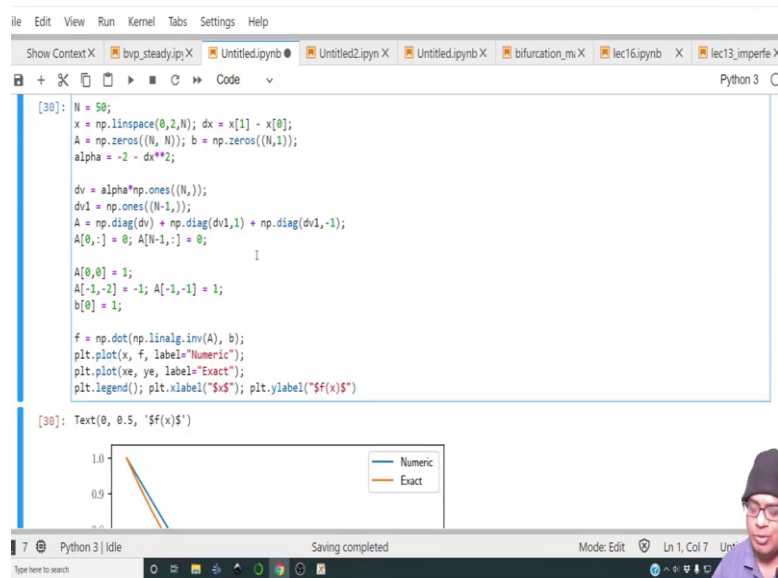
So, let me give this a label of approximate or numeric `plt.plot(xe, ye, 'ok', label="Exact"); plt.legend(); plt.xlabel("x"); plt.ylabel("$f(x)$").`

(Refer Slide Time: 36:43)

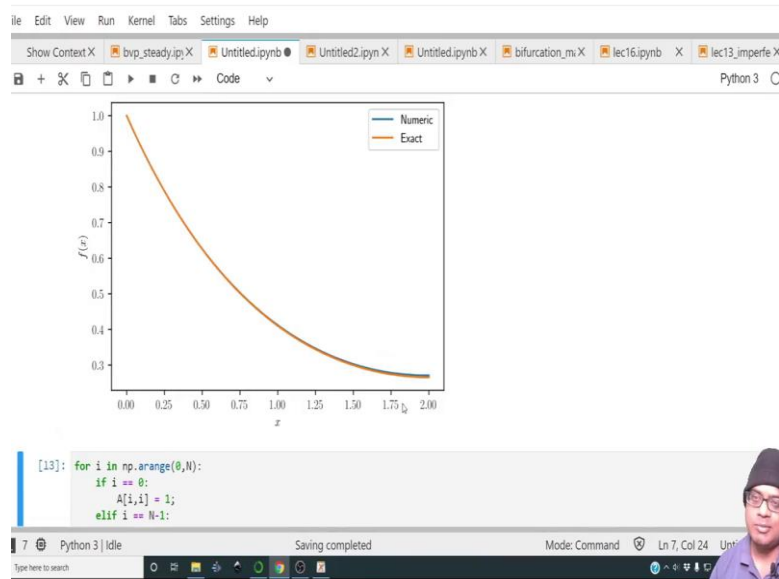


So, there appears to be some error; I mean not some error, but it is exact at the end, but here there is a great deal of error. In fact, oh 1 reason could be the very ridiculously low number of grid point.

(Refer Slide Time: 36:58)

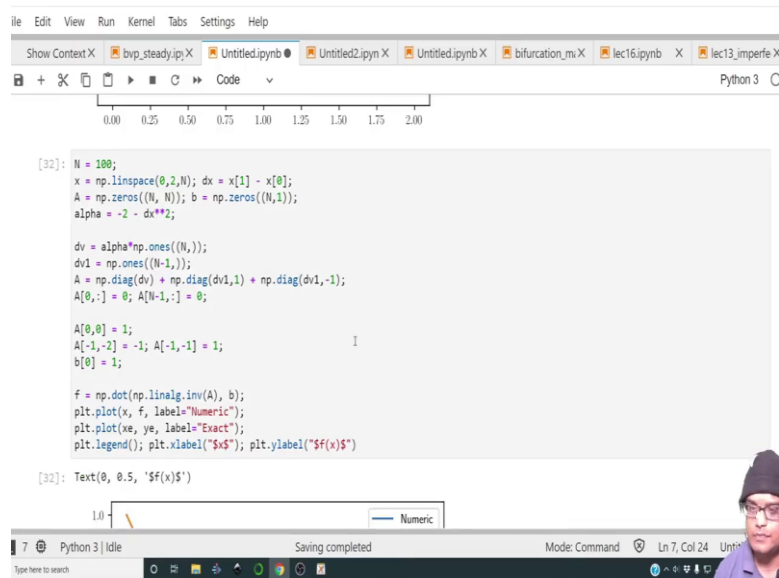


(Refer Slide Time: 37:00)

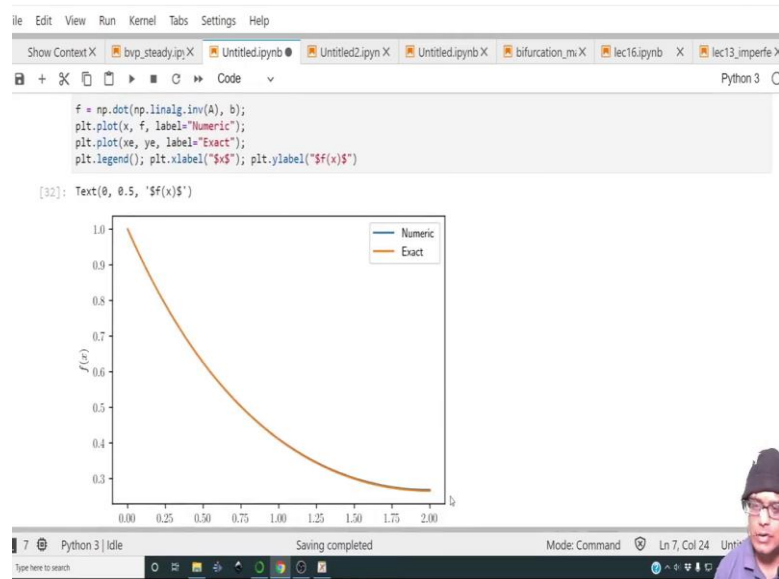


So, let me take 50 grid points maybe it makes things better it makes things much better the numeric and the exact solution do a does appear to match, but at the end point there is still something its not really matching exactly.

(Refer Slide Time: 37:16)

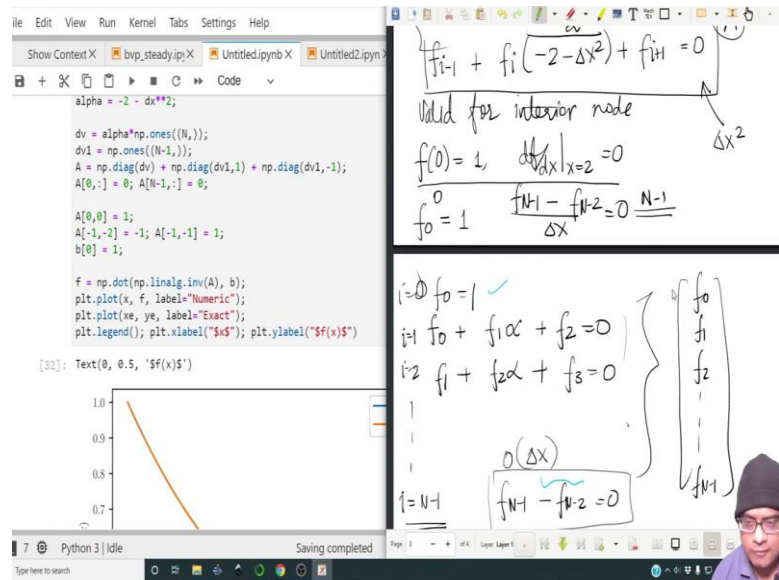


(Refer Slide Time: 37:18)



Well maybe if I make it a 100 points its much much better, but well why is does it not match as good as the other points? So, over here for example, you can hardly discern that there is some error, but why is this I mean let us go back to our workout working. So, the backward difference this has a truncation error of Δx of order Δx .

(Refer Slide Time: 37:52)



Whereas the central differencing that we have used for the interior nodes it is order Δx^2 the truncation order is order Δx^2 . So, we must now come up with an approximation

of the boundary condition at the end node at $N - 1$ th node which is order Δx^2 and the way to do it is again a central difference scheme.

(Refer Slide Time: 38:19)

The screenshot shows a Python IDE with the following code:

```

alpha = -2 - dx**2;

dv = alpha*np.ones((N,));
dv1 = np.ones((N-1,));
A = np.diag(dv) + np.diag(dv1,1) + np.diag(dv1,-1);
A[0,:] = 0; A[N-1,:] = 0;

A[0,0] = 1;
A[-1,-2] = -1; A[-1,-1] = 1;
b[0] = 1;

f = np.dot(np.linalg.inv(A), b);
plt.plot(x, f, label="Numeric");
plt.plot(xe, ye, label="Exact");
plt.legend(); plt.xlabel("$x$"); plt.ylabel("$f(x)$")

```

The plot shows a numerical solution (orange line) and an exact solution (blue line) for a boundary value problem. The x-axis ranges from 0 to 1, and the y-axis ranges from 0.7 to 1.0. The numerical solution is slightly lower than the exact solution near the boundary.

Handwritten notes on the right side of the slide:

- A diagram showing nodes $N-2$, $N-1$, and N . Node N is marked with a checkmark and an arrow pointing to it from the text $\frac{df}{dx}|_{N-1} = 0$.
- The equation $\frac{f_N - f_{N-2}}{\Delta x} = 0$ is written, with a note $O(\Delta x^2)$ below it.
- The equation $f_{N-2} = -2f_{N-1}$ is written below.

So, you have the $N - 1$ th node, you have the $N - 2$ th node. Now, we have to imagine a ghost node say the N th node. So, this is actually not a part of the domain its a extended ghost node that you sort of have in order to do something with it. So, now, the boundary condition was $\frac{df}{dx}$ at $N - 1$ or the $N - 1$ th node is $= 0$.

So, $\frac{f_N - f_{N-2}}{\Delta x} = 0$, but because its using the N th node N th - second node its order

Δx^2 accurate alright. The truncation order is of order Δx^2 , but what do I gain with this. I now have introduced an extra node. So, I can write down the equation that I had for the interior nodes for the boundary node as well.

(Refer Slide Time: 39:32)

The screenshot shows a Python IDE with the following code:

```

alpha = -2 - dx**2;

dv = alpha*np.ones((N,));
dv1 = np.ones((N-1,));
A = np.diag(dv) + np.diag(dv1,1) + np.diag(dv1,-1);
A[0,:] = 0; A[N-1,:] = 0;

A[0,0] = 1;
A[-1,-2] = -1; A[-1,-1] = 1;
b[0] = 1;

f = np.dot(np.linalg.inv(A), b);
plt.plot(x, f, label="Numeric");
plt.plot(xe, ye, label="Exact");
plt.legend(); plt.xlabel("$x$"); plt.ylabel("$f(x)$")

```

The plot shows a smooth curve starting at (0, 1) and decreasing towards 0. Handwritten notes on the right illustrate the ghost node concept:

- Nodes are labeled N-2, N-1, and N. Node N is marked as a "ghost".
- The equation $\frac{df}{dx} = 0$ at node N-1 is used to derive $f_N = f_{N-2}$.
- The finite difference equation $\frac{f_N - f_{N-2}}{\Delta x} = 0$ is shown to be $O(\Delta x^2)$.
- The resulting equation is $f_{N-2} + \alpha f_{N-1} + f_N = 0$.
- Substituting $f_N = f_{N-2}$ yields the final equation: $2f_N + \alpha f_{N-1} = 0$.

So, now I can have $f_{N-2} + \alpha f_{N-1} + f_N = 0$. So, I can do this, I can use the equation that I had formulated in the interior, because I have extended the domain by 1 ghost node. So, this sort of becomes the interior this is the ghost node very spooky.

So, now I have an equation in terms of f_N , but from this equation I have what? $f_N = f_{N-2}$. So, substituting this over here I will have $2f_N + \alpha f_{N-1} = 0$. And now this is the last equation that I should modify everything to, alright.

(Refer Slide Time: 40:36)

The screenshot shows a Python IDE with the following code:

```

N = 100;
x = np.linspace(0,2,N); dx = x[1] - x[0];
A1 = np.zeros((N, N)); b = np.zeros((N,1));
A2 = np.zeros((N, N));
alpha = -2 - dx**2;

dv = alpha*np.ones((N,));
dv1 = np.ones((N-1,));
A1 = np.diag(dv) + np.diag(dv1,1) + np.diag(dv1,-1);
A1[0,:] = 0; A1[N-1,:] = 0;
A2 = np.diag(dv) + np.diag(dv1,1) + np.diag(dv1,-1);
A2[0,:] = 0; A2[N-1,:] = 0;

A1[0,0] = 1; A2[0,0] = 1;
A1[-1,-2] = -1; A1[-1,-1] = 1;
A2[-1,-1] = alpha; A2[-1,-2] = 2;
b[0] = 1;

f = np.dot(np.linalg.inv(A1), b);
g = np.dot(np.linalg.inv(A2), b);
plt.plot(x, f, label="Numeric");
plt.plot(x, g, label="Second order");
plt.plot(xe, ye, label="Exact");
plt.legend(); plt.xlabel("$x$"); plt.ylabel("$f(x)$")

```

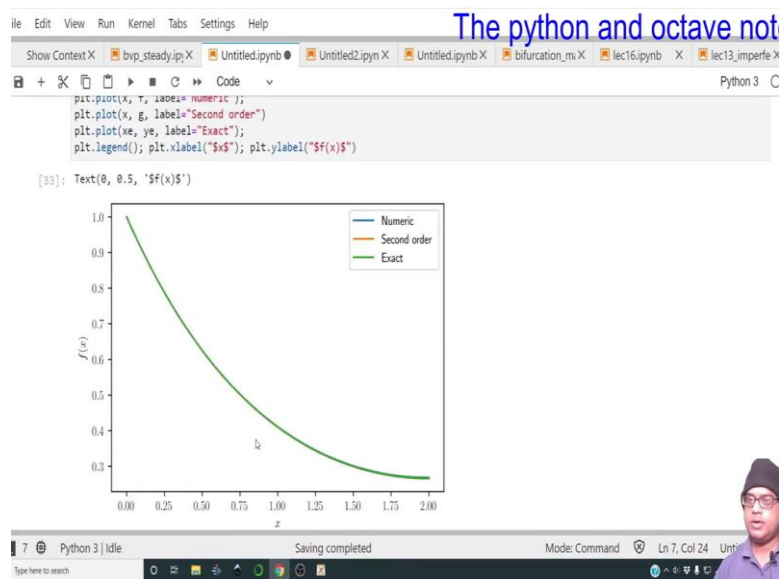
The title "The python and octave" is visible at the top right of the IDE window.

So, let me make a copy of this or in fact, let me change it to A1 and I will make the other thing as A2. So, that we can plot everything in N 1. So, everything is same for A1 and A2 up until the last point. In fact, we could have used np dot copy, but anyway we have doing we have done it the hard way no problem. We could have used np dot copy that is copy make a copy of A1 and assign it to A2 but anyway.

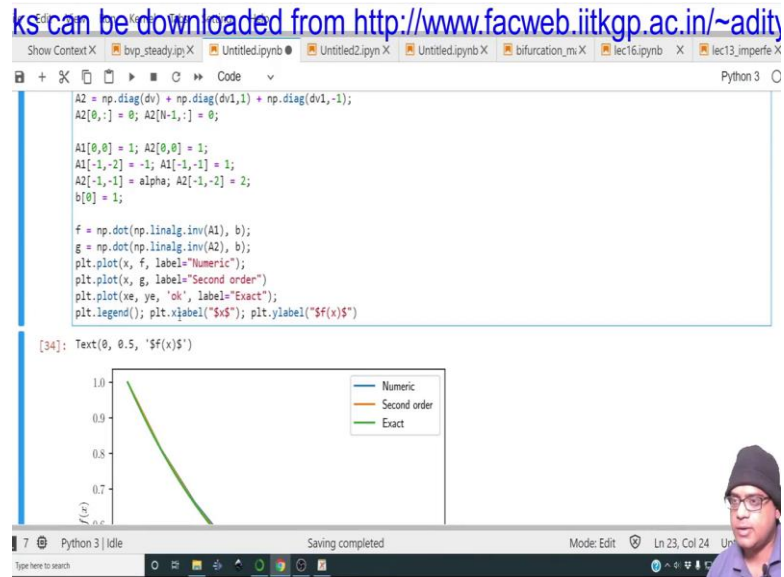
So, now what changes is over here. So, this does not change this is still 1; I mean 1, but over here now what do we have? So, this is α alpha, and this is 2 ok; this is going to be alpha sorry this has to be written in terms of f_{N-2} . So, I will have $A2 - 1, - 1 = \alpha$ and $A2 - 1, - 2$ and that is going to be $= 2$ ok yeah that is pretty much it.

Now, let me make another solution let me call it g this will be the inverse of A2. And we expect the solution from A2 to be more accurate than A1 let me also plot, alright.

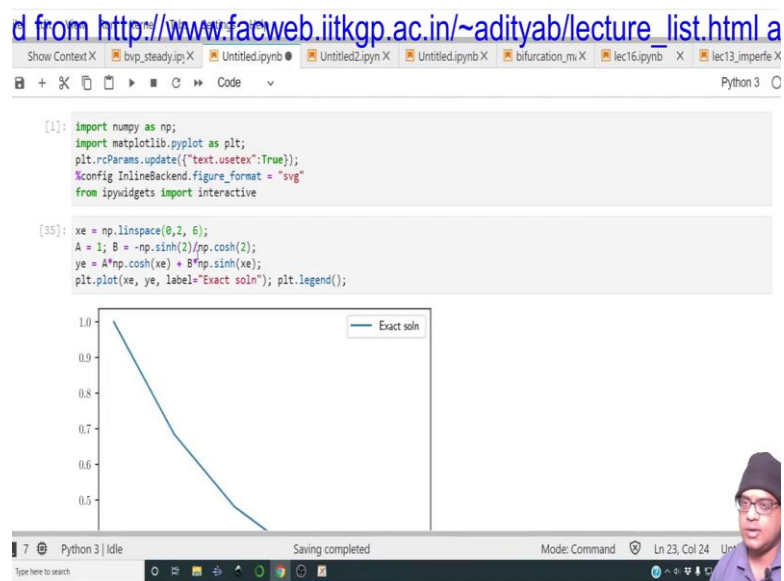
(Refer Slide Time: 42:32)



(Refer Slide Time: 42:51)

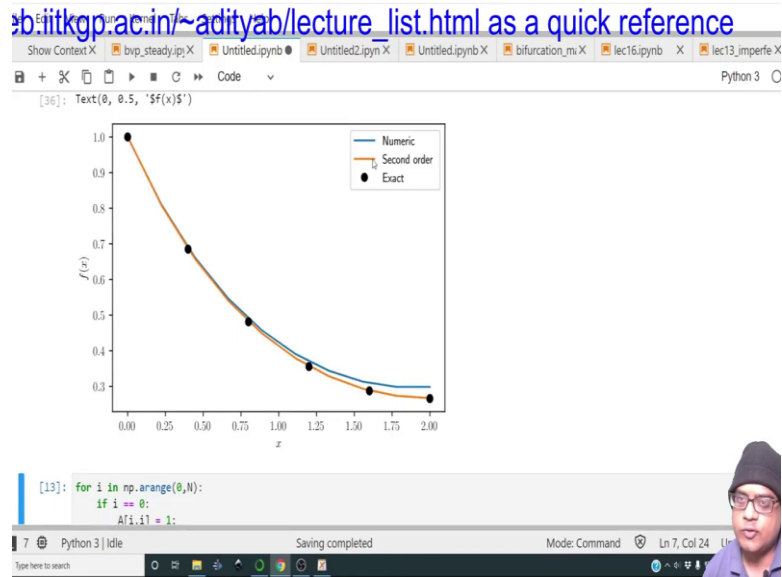


(Refer Slide Time: 42:58)

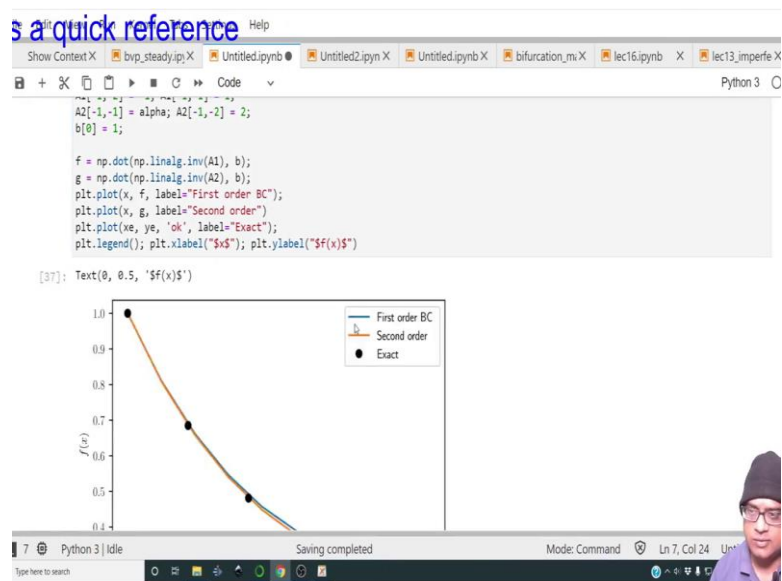


We can put as a marker and for the exact solution we can take maybe only 6 points. So, because it is an exact solution we know that the markers will be exactly at the correct point.

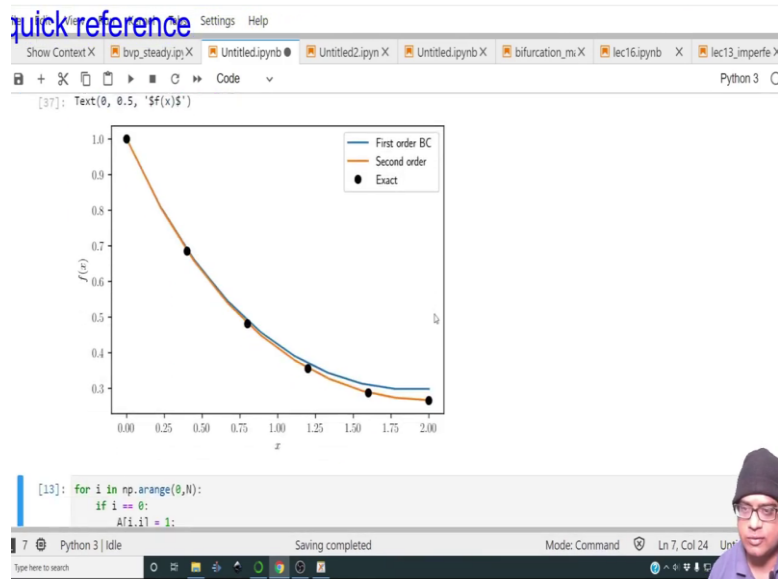
(Refer Slide Time: 45:05)



(Refer Slide Time: 43:14)



(Refer Slide Time: 43:19)



So, look the second order solution is far more accurate than the first order solution. In fact, let me make it first order. So, just to say that its the boundary condition is of the first order ok, the second order boundary condition matches quite well.

So, what you can do is give a go of this technique that you learned the ghost node. You can do it for any you can open up a textbook and find out any equation that you have you can try to solve it using this technique ok.

(Refer Slide Time: 43:45)

```
file Edit View Run Kernel Tabs Settings Help  
Show Context X bvp_steady.ipyn X Untitled.ipynb X Untitled2.ipyn X Untitled.ipynb X bifurcation_m X lec16.ipynb X lec13_imperfe X  
Python 3
```

```
[13]: for i in np.arange(0,N):  
      if i == 0:  
          A[i,i] = 1;  
      elif i == N-1:  
          A[i,i-1] = -1; A[i,i] = 1;  
      else:  
          A[i,i] = alpha; A[i,i-1] = 1; A[i,i+1] = 1;
```

```
[19]: a = [1, 2, 3]; A = np.diag(a,0); print(A)  
      b = [5, 6]; B = np.diag(b, 1); print(B)
```

```
[[1 0 0  
  0 2 0  
  0 0 3]]  
[[0 5 0  
  0 0 6  
  0 0 0]]
```

```
[ ]:
```

```
Python 3 | Idle Saving completed Mode: Edit Ln 2, Col 40
```

And I have shown it for a linear equation, but you can extend all this calculation for non-linear as well. And you now know the tips and tricks to how to implement all of this in any language really. We are not dependent all we have used is the inverse finding trick over here. And actually, in the next class before beginning I will discuss a bit about various other algorithms.

So, you do not really go about finding an inverse these are direct methods. So, to say, but do it iteratively. I will just make a small note on what techniques do exist and what is being used because once you have large systems finding an inverse of a sparse matrix is not that easy ok.

So, go ahead and try different problems from your textbook you have analytical solutions as well; you can try to plot them to convince yourself everything works ok. So, with this we end this in the next class we will see how to use some of the inbuilt libraries in python to do all this thing in a very small I mean wrapped way ok. So, we will be using the solve BVP ,nd and I will discuss about some solvers. So, with this we conclude this lecture I will see you next time bye.