

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

Lecture – 06

Nonlinear algebraic equations – system of equation and Newton’s basin of attraction

(Refer Slide Time: 00:27)

The image shows a Jupyter Notebook interface on the left and a blackboard with handwritten notes on the right. The Jupyter Notebook displays the title 'System of nonlinear equations and Newton's basins of attraction' and a code cell with the following Python code:

```
[1]: import numpy as np;
import matplotlib.pyplot as plt;
plt.rcParams.update({'text.useTex':True});
%config InlineBackend.figure_format = 'svg'
```

The blackboard contains handwritten chemical reactions and rate laws:

- At the top left, C_6H_6 is written with an arrow pointing to it from the text 'Benzene'.
- Below it, '1400F' is written.
- Two reversible reactions are shown:

$$C_6H_6 \xrightleftharpoons{r_1} C_{12}H_{10} + H_2$$

$$C_6H_6 + C_{12}H_{10} \xrightleftharpoons{r_2} C_{18}H_{14} + H_2$$
- Rate laws are written below the reactions:

$$r_1 = \frac{dC_B}{dt} = k_1 \exp\left(-\frac{15200}{T}\right) \left[P_B^2 - \frac{P_{B_{10}}}{k_1} \right]$$

$$r_2 = \frac{dC_B}{dt} = k_2 \exp\left(-\frac{15200}{T}\right) \left[P_B P_{B_{10}} - \frac{P_{B_{14}}}{k_2} \right]$$
- On the right side of the blackboard, three products are listed with arrows pointing to them: 'Biphenyl', 'Triphenyl', and 'Hydrogen'.

Hello everyone, I am Aditya Bandopadhyay and welcome to the 6th lecture, in which we are going to study system of Non-linear equations and Newton’s basins of attraction, ok. So, first let us try to motivate, how systems of non-linear equations can arise in your studies.

So, consider you are a chemical engineer and you are interested in a particular reaction. So, you have a tube like this and benzene that is C_6H_{12} , C_6H_6 rather; it comes in at a certain temperature. So, it comes in at 1400 Fahrenheit and it undergoes various decomposition reactions. So, the first reaction that it undergoes is C_6H_6 , it decomposes to $C_{12}H_{10}$ plus H_2 ; then $C_2C_6H_6$ plus $C_{12}H_{10}$ they react to give $C_{18}H_{14}$ and H_2 . So, this is called as biphenyl and this is called as triphenyl.

So, benzene it decomposes into a mixture actually of benzene into a mixture of biphenyl, triphenyl and hydrogen; these are the different products as a result of thermal decomposition. So, now let us try to write down the various reactions. So, we have two equations. So, let this be r_1 and this, this be r_2 .

So, r_1 is the rate at which benzene decomposes and it is equal to some constant times an Arrhenius factor, it looks something like this. So, it is Arrhenius factor which depends on temperature times the partial pressure of benzene square minus partial pressure of diphenyl, partial pressure of H₂ divided by K_1 .

Similarly, r_2 that is the rate of decomposition of benzene through this reaction ok; this is equal to some other constant. So, let me call this K_1 . So, this is K_2 times exponential of some another Arrhenius term actually times. So, this will be P_{hydrogen} or rather the forward reaction will be $P_{\text{benzene}} P_{\text{diphenyl}} - P_{\text{triphenyl}} P_{\text{hydrogen}}$ divided by k_2 ; this is how the reactions they look like. So, let us say that there are. So, let us consider only the mole fractions, ok.

(Refer Slide Time: 03:59)

So, considering that one mole fraction of benzene enters into the system. So, if x_1 moles of biphenyl and x_2 moles react from here and x_2 moles react from here. So, the moles of benzene, the mole fraction of benzene that will remain will be $1 - x_1 - x_2$ that of biphenyl will be $x_1 / 2$.

So, let us balance this reaction. So, there will be 2 over here. So, one mole of benzene would decompose into half a mole of biphenyl. So, that is why this $x_1/2$ occurs and eventually x_2 moles of benzene would react with biphenyl.

So, $x_1 - x_2$; this will be instead of a , it will be 1. So, these are the mole fractions of benzene that remain; these are the mole fractions of biphenyl. So, triphenyl you get x_2 moles, ok. So, triphenyl will be x_2 and hydrogen will be. So, over here it will be x_1 upon 2, so hydrogen it will be $x_1 / 2 + x_2$.

(Refer Slide Time: 05:43)

The image shows a JupyterLab interface on the left and a blackboard with handwritten equations on the right. The JupyterLab notebook has the title "System of nonlinear equations and Newton's basins of attraction" and contains the following code:

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

The blackboard contains the following handwritten text:

$$C_6H_6 + C_{12}H_{10} \rightleftharpoons C_{18}H_{14} + H_2$$

$$r_1 = \frac{dC_B}{dt} = k_1 \exp\left(-\frac{15200}{T}\right) \left[\frac{P_B^2}{K_1} - \frac{P_B P_H}{K_1} \right]$$

$$r_2 = \frac{dC_B}{dt} = k_2 \exp\left(-\frac{15200}{T}\right) \left[\frac{P_B P_H}{K_2} - \frac{P_B P_H}{K_2} \right]$$

Labels under the equations:

- Benzene: $1 - x_1 - x_2$
- Biphenyl: $\frac{x_1 - x_2}{2}$
- Triphenyl: x_2
- Hydrogen: $\frac{x_1}{2} + x_2$

At eqⁿ $r_1 = 0, r_2 = 0$

$$P_B^2 = \frac{P_D P_H}{K_1} \quad ; \quad P_B P_H = \frac{P_H P_H}{K_2}$$

A blue banner at the bottom of the image reads: "python and octave notebooks can be downloaded from http://..."

So, then at equilibrium what will happen is, the reaction rates will become 0. So, at equilibrium $r_1 = 0$ and $r_2 = 0$ and this implies $P_B^2 = P_D * P_H / K_1$.

(Refer Slide Time: 06:15)

The image shows a JupyterLab interface on the left and a blackboard with handwritten equations on the right. The JupyterLab notebook has the title "System of nonlinear equations and Newton's basins of attraction" and contains the following code:

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

The blackboard contains the following handwritten text:

$$r_2 = \frac{dC_B}{dt} = k_2 \exp\left(-\frac{15200}{T}\right) \left[\frac{P_B P_H}{K_2} - \frac{P_B P_H}{K_2} \right]$$

Labels under the equations:

- Benzene: $1 - x_1 - x_2$
- Biphenyl: $\frac{x_1 - x_2}{2}$
- Triphenyl: x_2
- Hydrogen: $\frac{x_1}{2} + x_2$

At eqⁿ $r_1 = 0, r_2 = 0$

$$P_B^2 = \frac{P_D P_H}{K_1} \quad ; \quad P_B P_H = \frac{P_H P_H}{K_2}$$

A blue banner at the bottom of the image reads: "www.facweb.iitkgp.ac.in/~adityab/lecture_list.html as a quick ref".

Similarly, $P_b \cdot P_d$ will be equal to $P_t \cdot P_h / K_2$, alright. So, this implies that. So, we make use of the fact that the partial pressure of benzene, diphenyl, hydrogen and triphenyl will be proportional to the mole fraction, ok.

So, this square will be proportional to $(1-x_1-x_2)^2$ and this will be equal to $(x_1/2-x_2) \cdot P_h$ where P_h will be $(x_1/2 + x_2)/k_1$. And the next reaction it will be P_b , so that is partial pressure of benzene times P_d . So, that will be equal to $x_1/2-x_2$ and this will be equal to x_2 multiplied by the partial pressure of hydrogen and all this divided by K_2 .

So, in order to find out how many moles in equilibrium that we will get, we need to solve this equation and this equation simultaneously. I am solving that simultaneously, requires you to solve this nasty looking non-linear equation. So, that is how these equations are quite relevant to any studies which involve a bunch of chemical reactions, sequence of chemical reactions, ok.

So, in fact let us proceed. So, let me execute this first cell; this first cell is something which is going to be common and so we are not going to sweat it a lot. So, it is import numpy and import matplotlib and update the rc Params of matplotlib and the inline rendering to be s v g, so that the plots in Jupiter Lab they look much better. So, let me execute this. So, let us try to solve the equation, the system of equation.

(Refer Slide Time: 08:47)

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

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

$$x_1 \exp(x_1+y) - 2 = 0$$
$$x_2 \exp(-x_2) = 0.5$$

[ ]:
```

On the right, a video feed shows a person with handwritten equations on a blackboard:

$$\text{At eq}^n \quad x_1 = 0, \quad x_2 = 0$$

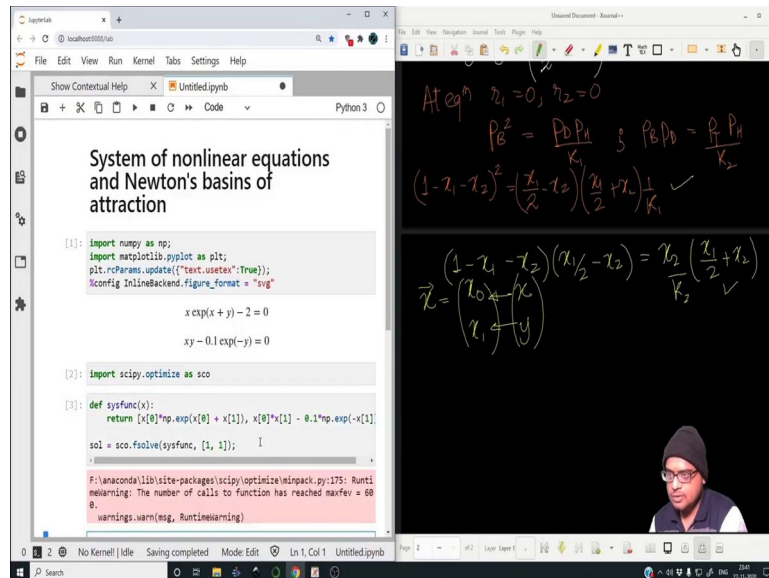
$$P_b^2 = \frac{P_b P_d}{K_1} \quad ; \quad P_b P_d = \frac{P_b P_d}{K_2}$$

$$(1-x_1-x_2)^2 = \left(\frac{x_1}{2}-x_2\right) \left(\frac{x_1}{2}+x_2\right) \frac{1}{K_1} \quad \checkmark$$

$$(1-x_1-x_2) \left(\frac{x_1}{2}-x_2\right) = \frac{x_2}{K_2} \left(\frac{x_1}{2}+x_2\right) \quad \checkmark$$

So, let me write it down. So, it will be $x \cdot e^{(x+y)} - 2$ and $xy - 0.1 \cdot e^{-y}$. So, this could be a model for some set of reactions as well.

(Refer Slide Time: 09:15)



But essentially we want to find out the roots of this, so both will be equal to 0, ok. Let us first make use of the library functions in python and later on we will see how we can encode this and we will try to see how we can get some insight, when we write the code ourselves, ok. So, let us import `scipy.optimize as sco`, ok. Let us define the functions.

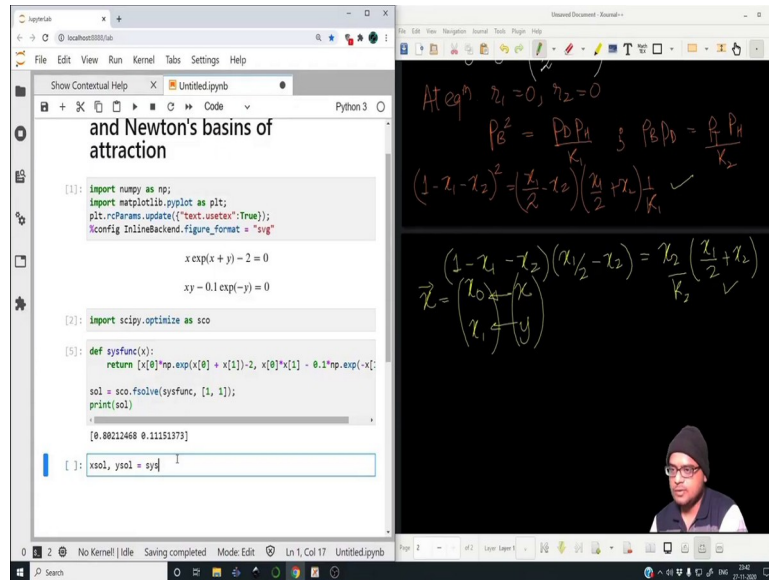
So, define system of `sysfunc`, it will take as an input `x`. So, here `x` will be a vector and that vector will comprise of `x[0]` and `x[1]`, ok. Whenever we are doing sequence of series of equations, we will give as an input `x`; but here `x` will not be a scalar, rather `x` will be a vector and the vector components will be `x[0]` and `x[1]`, because the indexing in python starts with 0 with 0 ok, that is the same with `c` as well.

So, now we must return an array ok, you must return an array. So, the first element of the array will be this `x[0]*np.exp(x[0]+x[1])`; because `y=x[1]` and `x[0]=x`. So, over here this is equal to `x` and this is equal to `y` ok; this is something which we will keep in mind. Apart from this it will return the second set of equation as well, which is `x[0]*x[1]-0.1*np.exp(-x[1])`, alright.

So, once we have this, let us say `sol= sco.fsolve`; we will pass the function handle that is `sysfunc` and we will give some guess values. So, the guess values will be in this case `(1,1)`,

ok. So, let me execute this. So, these are runtime error it says, the number of calls to the function has exceeded 60, ok. So, let us see whether we have input something incorrectly; we have forgotten to put - 2 over here.

(Refer Slide Time: 12:03)



So, there was an offset and once I added the correct equation; I have forgotten; I had forgotten to write this - 2 anyway. So, it executes, let us see what, let us print out what sol is. So, print sol, let us print out. So, sol = 0.8021 whatever it is and 0.111. So, we can verify whether this is the solution by substituting this array into system of equations, ok. So, let us find out.

(Refer Slide Time: 12:54)

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

```

and Newton's basins of attraction

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

x exp(x+y) - 2 = 0
xy - 0.1 exp(-y) = 0

[2]: import scipy.optimize as sco

[5]: def sjfunc(x):
return [x[0]*np.exp(x[0] + x[1]) - 2, x[0]*x[1] - 0.1*np.exp(-x[1]]
sol = sco.fsolve(sjfunc, [1, 1]);
print(sol)

[7]: re1, re2 = sjfunc(sol);
print(re1, re2)

4.884981398350609e-15 3.788636071533347e-15

```

The blackboard contains the following handwritten notes:

At eqn $x_1 = 0, x_2 = 0$

$$P_B^2 = \frac{P_D P_H}{K_1} \quad ; \quad P_B D_0 = \frac{P_H P_H}{K_2}$$

$$(1 - x_1 - x_2)^2 = \left(\frac{x_1}{2} - x_2\right) \left(\frac{x_1}{2} + x_2\right) \checkmark$$

$$(1 - x_1 - x_2)(x_1/2 - x_2) = x_2 \left(\frac{x_1}{2} + x_2\right) \checkmark$$

$$\vec{x} = \begin{pmatrix} x_0 + x_1 \\ x_1 + y \end{pmatrix}$$

So, xsol ,ysol =sys or return value of x[0]; return value of equation 1 return value of equation 2 equal to sysfunc of sol. Let us print what the return values are re1 ,re2. So, it is 10^{-15} .

So, obviously whatever we have over here is a solution of the system of non-linear equations. So, this is how you can sort of pass on the function handle to the f solve solver, which is inside the sub module optimize of scipy; it is a very easy thing to do. But now let us try to encode this particular program our self. So, we will do it using a fixed point iteration and let us first devise how we can form this particular fixed point iteration.

(Refer Slide Time: 13:50)

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

```

[2]: import scipy.optimize as sco

[5]: def sjfunc(x):
return [x[0]*np.exp(x[0] + x[1]) - 2, x[0]*x[1] - 0.1*np.exp(-x[1]]
sol = sco.fsolve(sjfunc, [1, 1]);
print(sol)

[7]: re1, re2 = sjfunc(sol);
print(re1, re2)

4.884981398350609e-15 3.788636071533347e-15

Perform fixed point iterations to solve the same problem

[1]: def g(x):
return [2*np.exp(-x[0] + x[1]), 0.1/x[0]*np.exp(-x[1])];

x = np.linspace(0, 1);
y = np.linspace(0, 1);
[X, Y] = np.meshgrid(x, y);

```

The blackboard contains the following handwritten notes:

$$(1 - x_1 - x_2)(x_1/2 - x_2) = x_2 \left(\frac{x_1}{2} + x_2\right) \checkmark$$

$$\vec{x} = \begin{pmatrix} x_0 + x_1 \\ x_1 + y \end{pmatrix}$$

$$x e^{xy} = 2 \quad ; \quad xy = 0.1 \exp(-y)$$

$$x = \frac{2}{e^{xy}} \quad ; \quad y = \frac{0.1 \exp(-y)}{x}$$

A plot is shown on the blackboard with the intersection point labeled as the root.

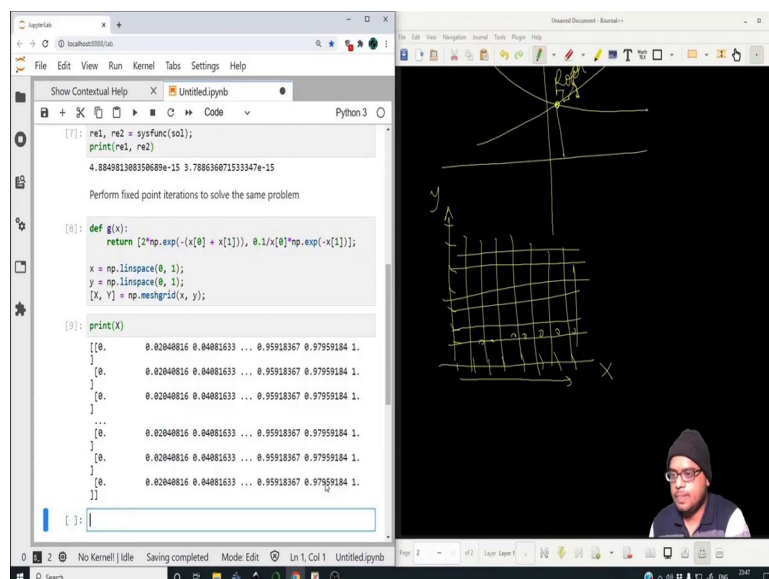
So, over here, we have $x * e^{(x+y)} = 2$ and the other is $x * y = 0.1 e^{-y}$. So, the first equation that we can form is $x = 2 / e^{(x+y)}$ and the second equation will be $y = 0.1 e^{-y} / x$. So, this seems like a fair approximation. So, let us define, ok. So, let us define the right hand side of this particular function. So, let us define $g(x)$; let us just call it g , ok. So, it will return two values.

So, the first will be $2 * np.exp(x)$; so $-2 * np.exp(-(x_0 + x_1))$, ok. So, this is the first value that we will return; second value will be $0.1 / x[0] * np.exp(-x[1])$. So, these will be the two return values, ok. So, we are done with the function; before trying to solve all this, let us first plot the two functions.

So, how can we plot these two functions? So, keep in mind, these are not explicit functions; these are rather implicit functions ok, we want to find out say the root of this. Essentially there will be some curve which will be defined by this equation; let us say it is this curve, some curve which will be which will be defined by this equation and wherever they intersect that is the root.

But let us first try to plot these things; but because they are not explicit functions on their own, you must find out an implicit way of plotting it, ok. This is something which is quite useful and it will be quite useful for many other works. Let us define $x = np.linspace(0,1)$ and $y = np.linspace(0,1)$. Then what we will do is, we will define the mesh grid. So, $[X,Y] = np.meshgrid(x,y)$.

(Refer Slide Time: 17:02)

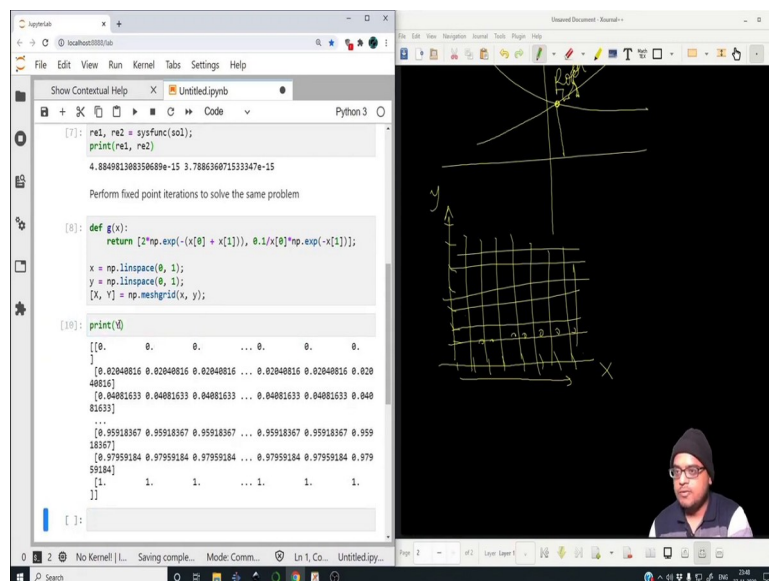


So, what it does is, if we have a linspace like this; so suppose this is x ok, these are all values of x and we have a array like this. So, these are all the values in y. So, the mesh grid; so capital X and capital Y will contain all the inside intersection points ok, it will contain all the inside intersection points.

So, this is what all the inside intersection points will be assigned to x. So, all the x coordinates of all these points will be inside capital X and all the y coordinates of all these intersection points will be in capital Y. So, in fact let me just run this much and show you what x and y are.

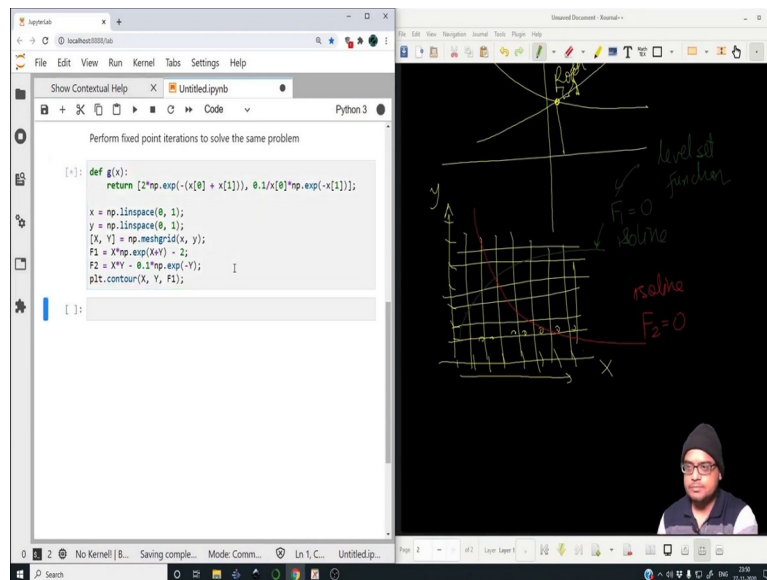
So, look, across as we move along columns, the x increases; but as we move along rows, the x does not change. So, it means that when you move along rows, it has the same x value. But if we if we print y, we will see that as we move along rows the y value will change; but as we move along columns, the x value remains, the y value remains constant.

(Refer Slide Time: 18:05)



So, I hope this gives you an idea of what a meshgrid structure looks like. So, it gives you an output x and y. Let me delete this cell. So, once we are done with this, we will try to create an implicit function of these two things; meaning we will create a level set, this is called as creating a level set.

(Refer Slide Time: 18:34)

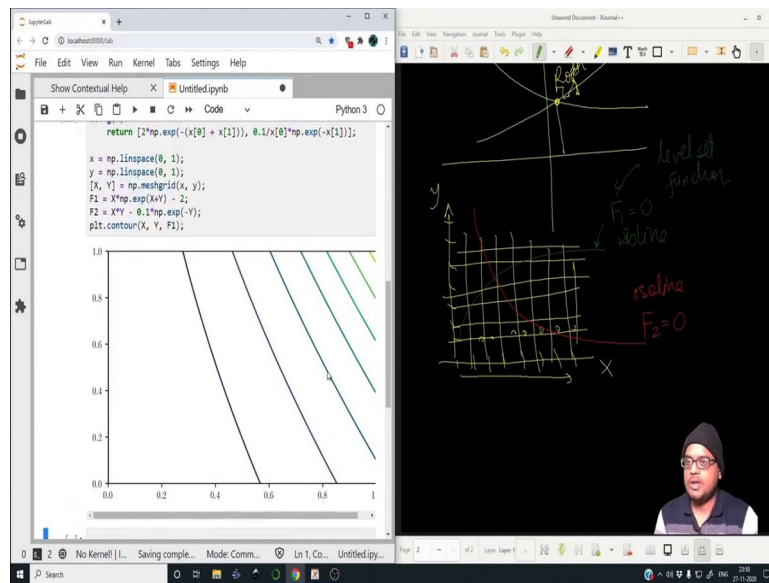


So, let me create the first level set, it will be clear what it means. So, $F1 = X * np.exp(X+Y) - 2$
 $F2 = X * Y - 0.1 * np.exp(-Y)$. So, these are two levels, ok. So, capital F and capital Y are two functions.

So, depending on the value of F. So, suppose F were to be 0; if F were to be 0, if I try to plot the iso line of F1, where all the points along this curve correspond to $F1 = 0$. So, this is an iso line, along this curve all the values of F1 will be 0. And if all the values of F1 are 0 along this iso line; it means that it is the curve that we are looking for. So, you do an implicit declaration all over the mesh and try to find the iso line of that particular level set, F1 is called as a level set function.

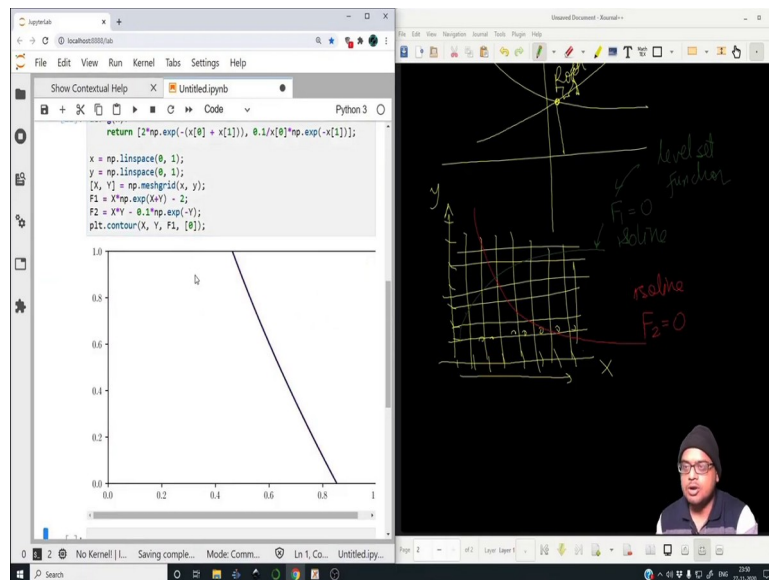
Similarly, we will have another level set function say like this. So, this will be the iso line of all the points on the curve, where $F2 = 0$, ok. So, let us do that; let us do the iso line. So, the way to do it is; `plt . contour`. So, `X ,Y,F1`; if I simply do this contour, it will give me a family of curves for F1.

(Refer Slide Time: 20:29)



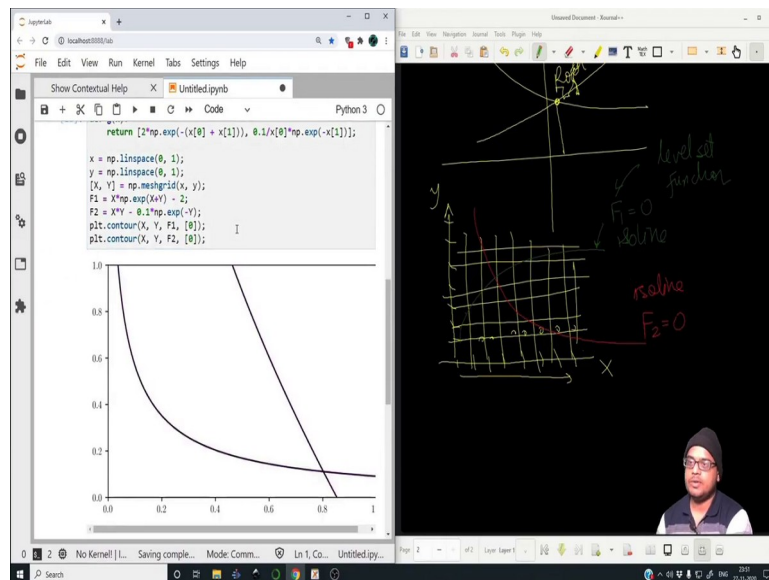
It will give me a family of curves for F_1 ; meaning I will have all the different family of curves for different values of F_1 . So, this iso line is for some particular value of F_1 , this iso line is for some particular value of F_1 and so on; but I am more interested in finding out the iso line for F_1 when it is 0.

(Refer Slide Time: 20:45)



So, by specifying that I only want that iso line, I can plot only that single curve, alright.

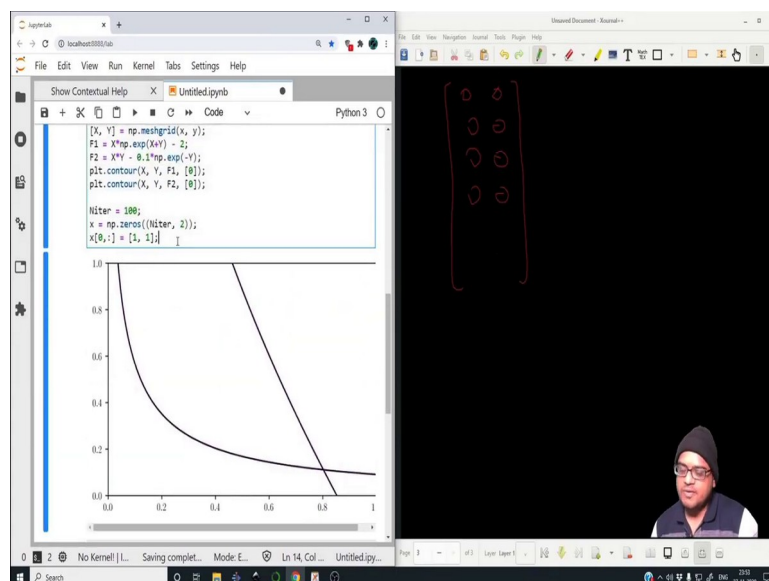
(Refer Slide Time: 20:58)



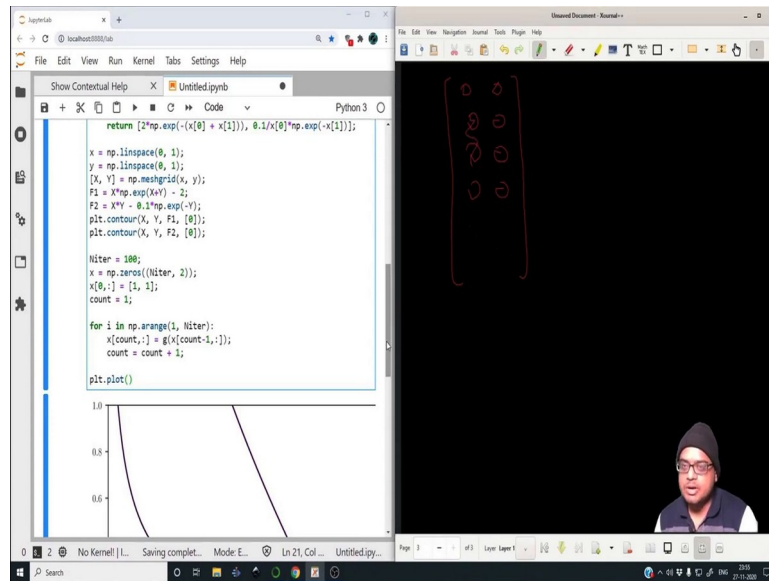
Similarly, I would like to find out the iso line of X, Y, F_2 and the 0 curve, alright. So, over here the intersection point is very close to 0.8 and the that is fair enough; the first intersection point was in fact near 0.8 and the y value is 0.11.

Let us see great, the y value also seems to be something close to 0.11. So, the intersection of these two curves in the plane gives us the root; the root is the pair x, y where the two curves intersect, alright. Let us now write down the fixed point iterations as we have defined over here, excellent.

(Refer Slide Time: 21:43)



(Refer Slide Time: 23:39)

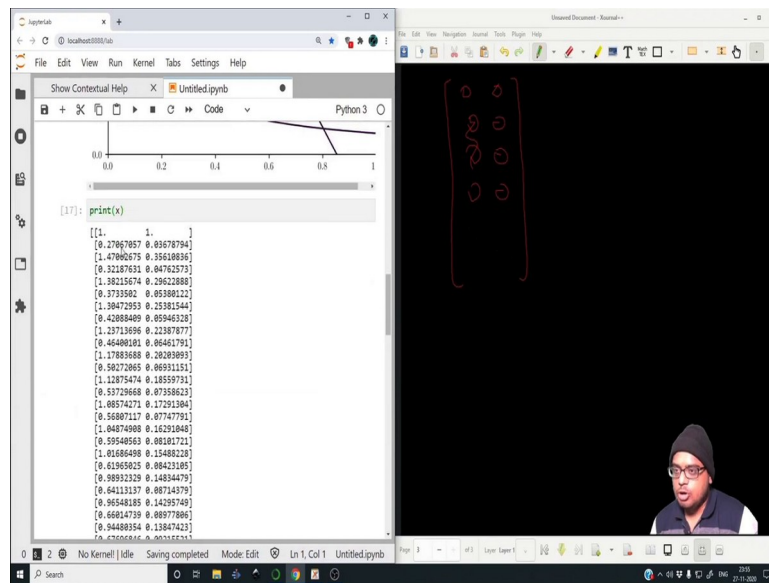


So, we have defined the guess values, we will defined count=1 for i in np.arange(1,Niter); we will say $x[\text{count},:] = g(x[\text{count} - 1,:])$, ok. We are selecting one entire row, passing it to the g function that we have defined earlier, ok.

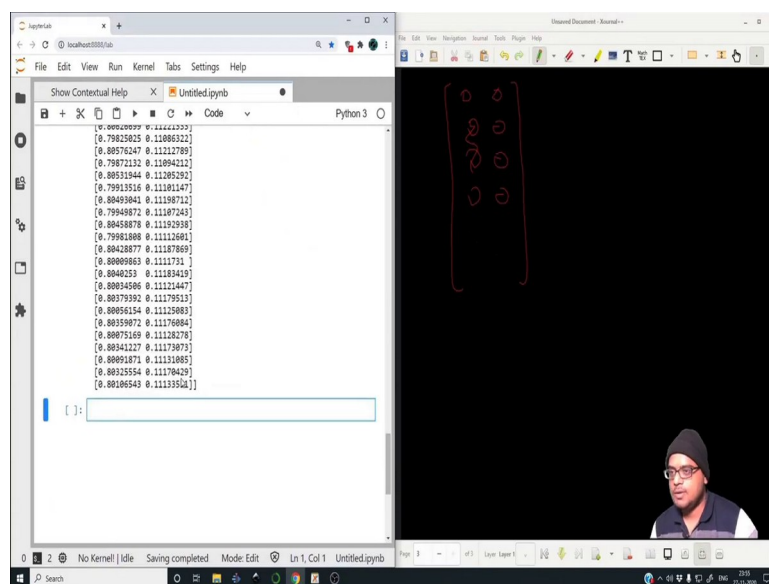
So, essentially in the very first loop 1,1 gets passed to this function, the return value is assigned to this. So, once we are done with this computation, we have to update count. So, count will become count +1, ok.

Let me execute this and in the end on top of this plot, let us do plt.plot. So, we must plot all the x y values, ok. So, x the first entire row contains all the x iterates and x the second entire; the first column contains all the values of x, basically this will have some values, in fact let me print out what x is to show you.

(Refer Slide Time: 25:09)

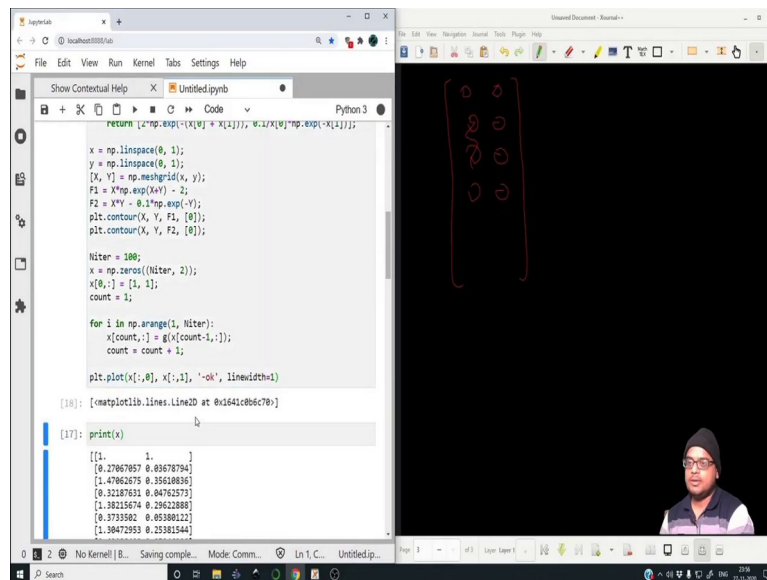


(Refer Slide Time: 25:11)



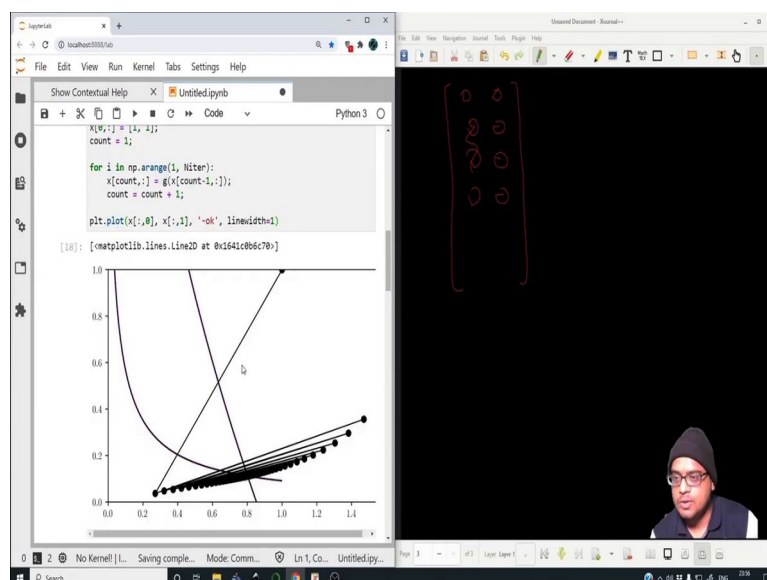
So, print x, alright. So, the first column it contains all the iterations that have happened to the values of x; the second column contains all the iterations that have happened to the value of y. So, when we plot these points; this is the x coordinate, this is the y coordinate. So, we must plot all the columns which are corresponding to x and all the columns which are corresponding to y.

(Refer Slide Time: 25:38)



So, we must do over here, all rows of the 0th column x all rows of the 1st column. So, this particular splicing we have discussed it in the very previous lecture, this is called as splicing. So, just giving a colon(:) means, it is selecting all the rows, comma 0 means of the 0th column; similarly over here all the rows comma first column. And let us mark them by a black something like this by black markers.

(Refer Slide Time: 26:15)

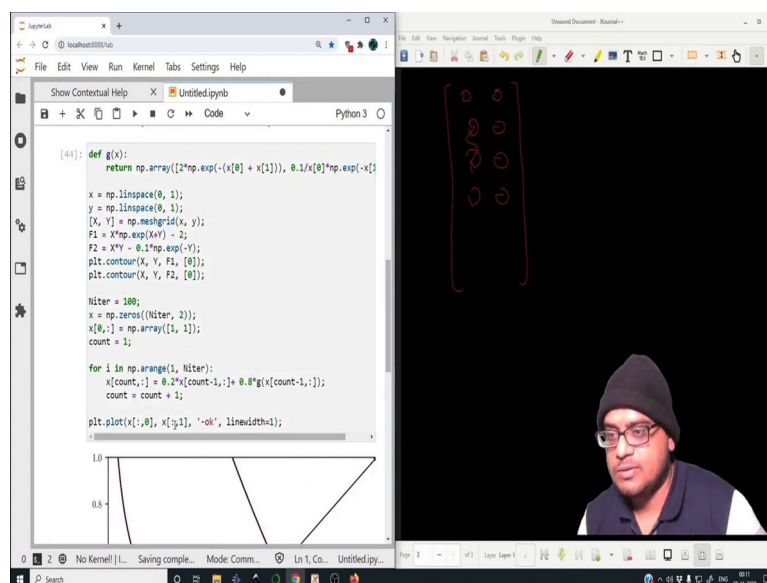


Let us set the line width to be equal to 1; let us see what happens, ok. So, we have started our guess over here; this point corresponds to 1, 1. At the end of the first iteration, it has swung to

this point; second iteration swung to this point, third over here, ok. So, it is zigzagging its way towards the final root, alright.

So, we see this is the nature of convergence for a system of equations, ok. You can try this out for a variety of system of equations; you have the basic outline of the program with you, try to go wild with it. So, one little point that I would like to impress over here is a small subtlety; actually we are returning over here a list, this particular thing in python is a list. Ideally you would like to cast it to a numpy array before passing it out of a list.

(Refer Slide Time: 27:19)

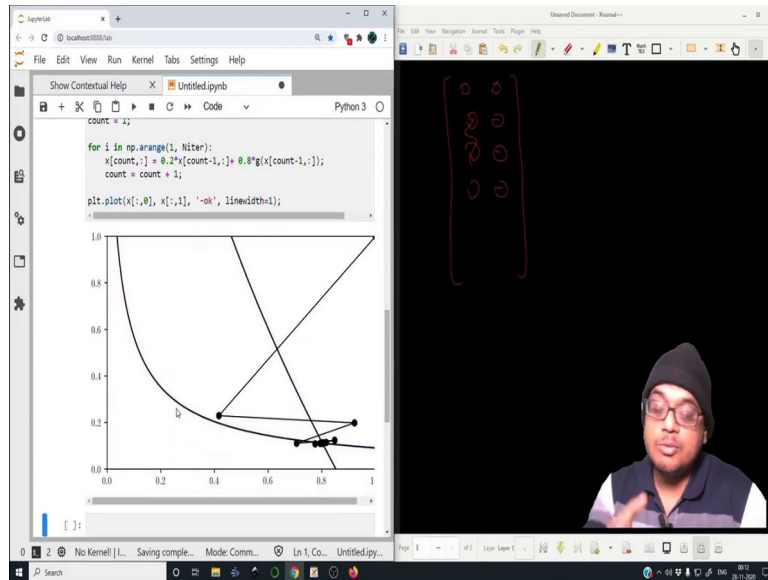


So, rather than having this return value, we rather do it np.array of this entire thing. So, before returning it from the function, it would convert it to a numpy array nothing else, it does not change anything; but instead of returning a list, it would return a numpy array. And lists in python have very weird behavior, very syntactically different behavior than numpy arrays, ok.

So, we have this solution; I had written down something over here, this was the initial the original equation. And we can modify the sequence to sort of dampen out the oscillations. So, how to dampen out the oscillations? So, let us update the value of x partially with the old value of x itself and partially with the new value that we found out using the g function, ok.

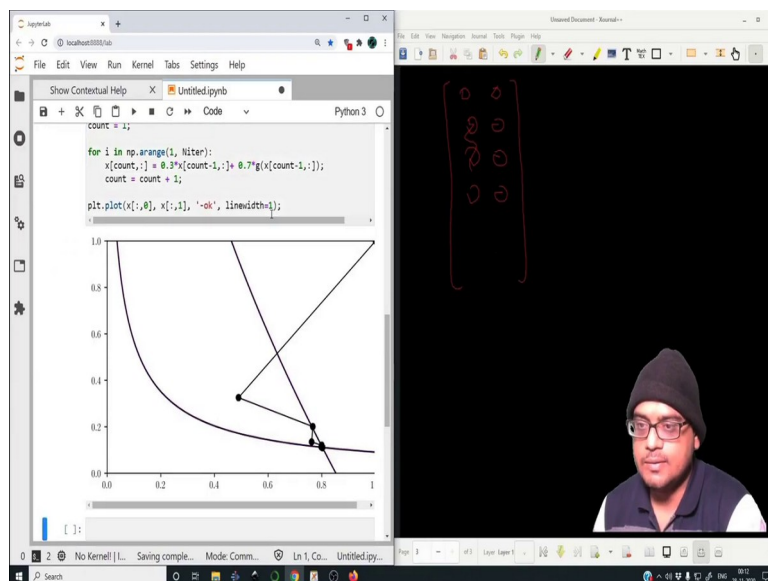
So, let me do this, let me do $x[\text{count}-1, :]$ plus this. Let me multiply it by a factor say, suppose 0.2 and correspondingly let me multiply this by the 1 minus that factor. So, it is essentially a weighted sum of these two terms; let us see what happens.

(Refer Slide Time: 28:42)



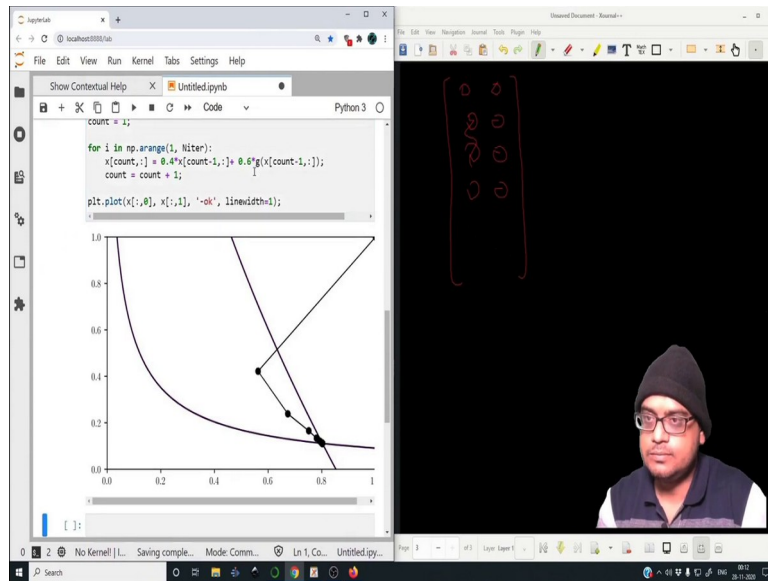
Because the number of the oscillations have died down quite quickly. So, damping the update value allows us to reduce oscillations; in fact let me do this 0.3, 0.7.

(Refer Slide Time: 28:54)

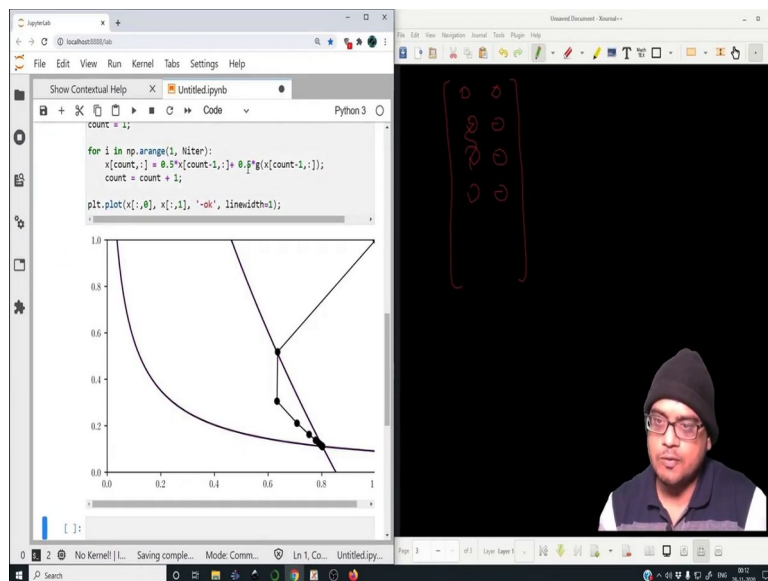


The oscillations have dampened down even further; they are converging to the root quite fast.

(Refer Slide Time: 29:05)

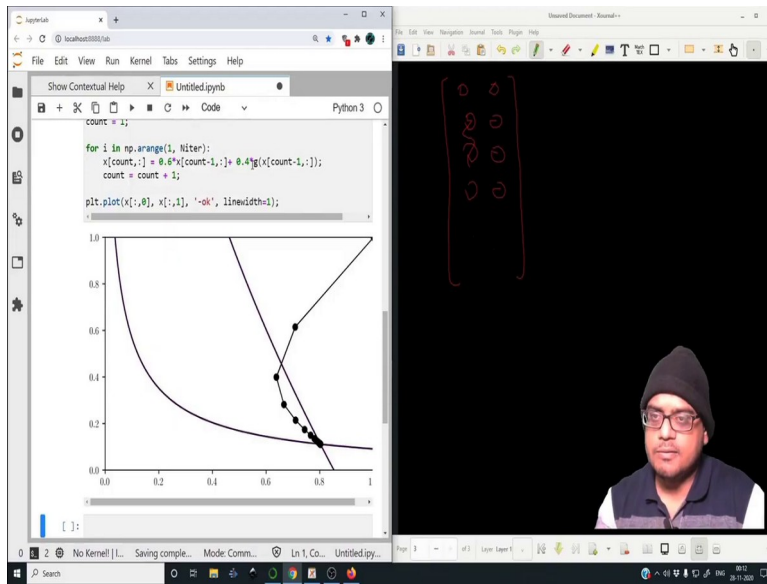


(Refer Slide Time: 29:13)



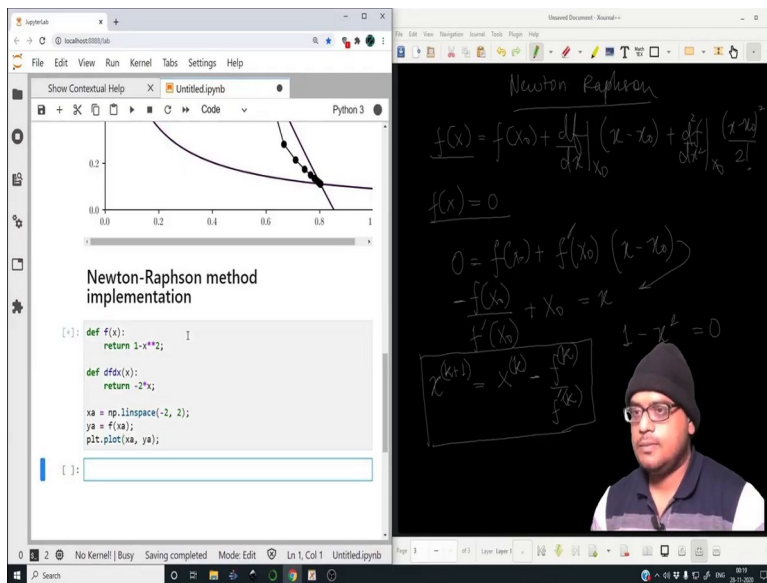
Let me do this, ok. So, it is converging rather fast. So, this is how you can take a weighted sum between the old value and the new value, alright.

(Refer Slide Time: 29:19)



So, have a go at this; try to figure out why the behavior changes like this. Try to apply your mind to it, try to find out the mathematics behind it, ok.

(Refer Slide Time: 29:41)



So, let us proceed to Newton's, Newton Raphson iteration are. So, what is the idea behind the Newton Raphson iteration? So, we have a $f(x)$. So, suppose we write $f(x)$ in the neighborhood of a point x_0 . So, we have $f(x_0) + \frac{df}{dx}\bigg|_{x_0} (x - x_0) + \frac{d^2f}{dx^2}\bigg|_{x_0} \frac{(x - x_0)^2}{2!} + \dots$ and so on.

So, now, when we approach the root; when we approach the root for the non-linear equation the; so basically we are trying to solve this particular equation. So, we have made a guess point around the root. So, if x is a root, so we will have 0 is equal to $f(x_0) + f'(x_0)(x-x_0)$.

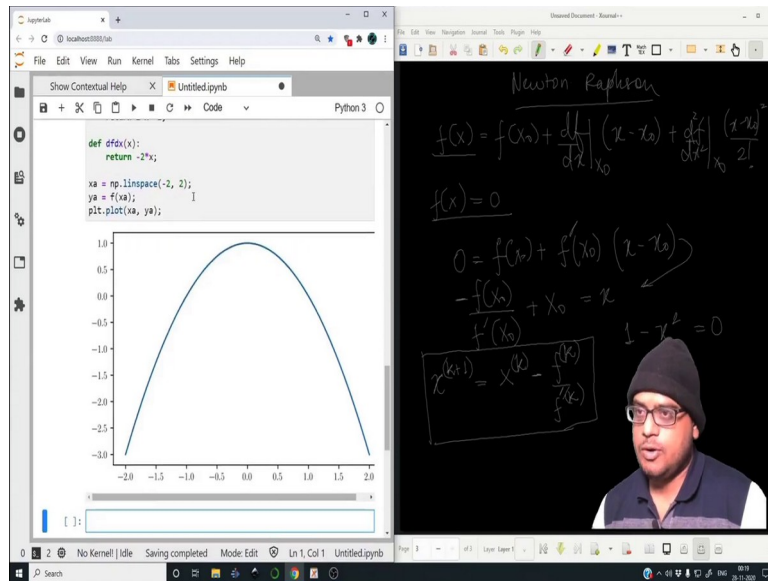
So, now if we rearrange this, we can find out an expression for the root. So, we have $-f(x_0)/f'(x_0)+x_0$ and this will be equal to x ; I have just rearranged all of this. So, then x at the k th iteration or rather x at the $(k+1)$ th iteration will be equal to $x[k] - f/f'$. So, f and f' are both evaluated at the k th iteration.

So, this gives us a very easy way of sort of trying to find out a root with an initial guess and like I have shown in the previous lecture; you would also need to supply the derivative of x to the function. But in this lecture, we are going to write it down our self; because later on we are going to use certain divergent properties as well to make an entire phase plot. So, it is worth our time to writing it down and it is a really simple code, it is this is the code; it is just one update, ok.

So, let us go over here, let me write down what we are trying to do; Newton Raphson method implementation ok, we have moved ahead of fixed point iterations, ok. So, let us define the function. So, let us make a simple function. So, let the function be $1 - x^2$. So, we want to find out the roots; so obviously the roots are $+1$ and -1 .

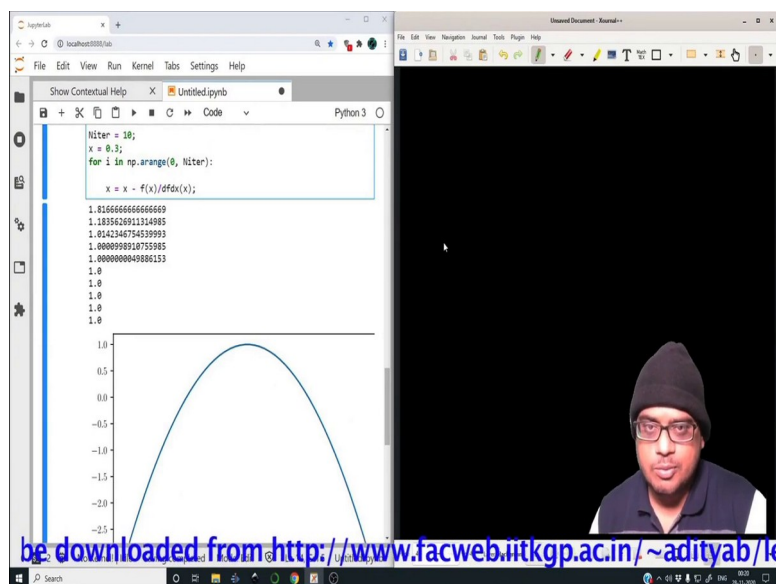
So, let us go over here, let us define f which takes an input x and the return value will be $1 - x^2$. Let us also define $dfdx$; it takes an input x and let us return $-2*x$, fine. Let us define the x array. So, x array will be `np.linspace(-2,2)`, y array will be $f(x)$. Let us plot this, ok.

(Refer Slide Time: 33:12)



So, this is the plot quite obviously. So, now, let us do, let us write down the Newton Raphson iterations on top of this.

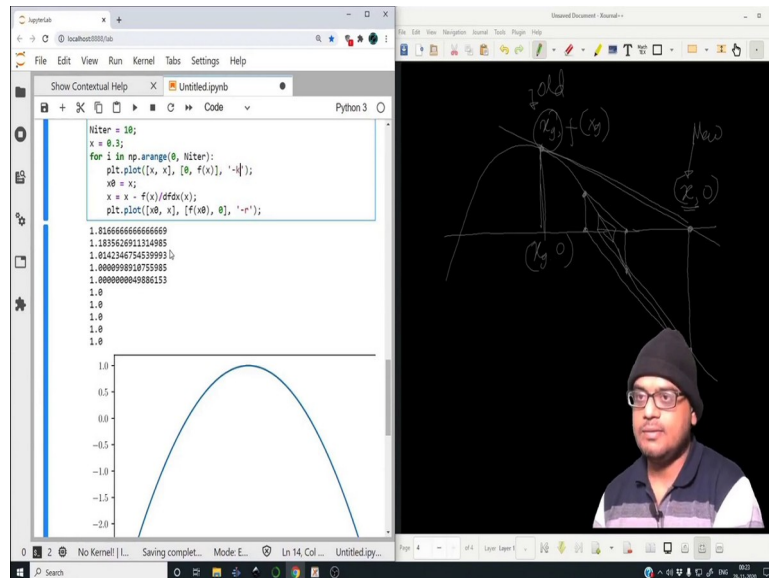
(Refer Slide Time: 33:24)



So, let us define Niter as 10, let us say the guess value is 0.3, alright. For i in $np.arange(0, Niter)$; let us do $x = x - f(x) / dfdx(x)$ that is it. I mean if we print out the values of x . So, print x ; so this gives us the values it goes through. So, first point is 0.3, after that it shifts to 1.81; then 1.18 and then it eventually converges to 1, it converges right rather quickly to the value of the root, ok.

So, let us try to pictorially also draw this. So, once we have the guess value, what should we do? I mean what is actually happening graphically; this is you should be able to tell this.

(Refer Slide Time: 34:35)



You should be able to understand what is happening graphically. So, this is the curve, you are making; suppose this is the guess value. So, essentially you are trying to find out the tangent over here; wherever this intersects the axis, this is the function. So, then you drop the function on this; then you make a tangent over here, you end up over here, you drop a vertical over here, you draw a tangent over here, drop a vertical to the function, tangent and then something like this happen.

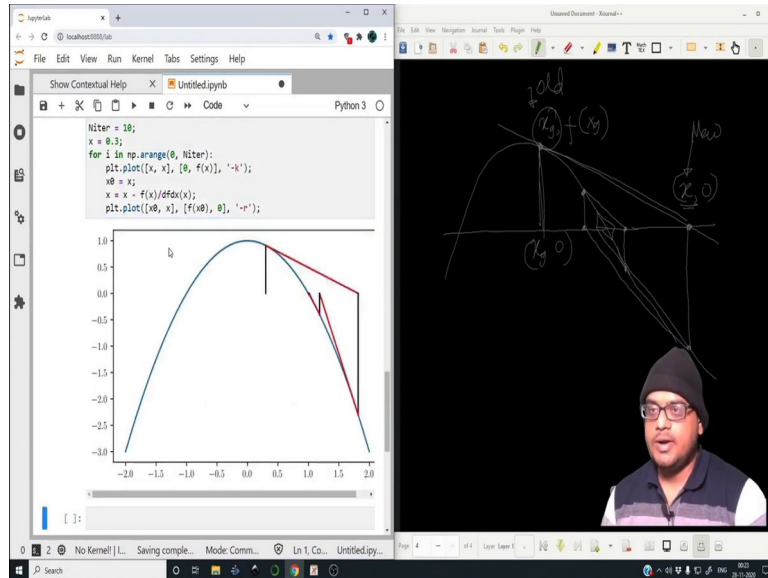
So, let us see, let us try to plot this entire sequence. So, the first point that we will draw is the. So, the first point is obviously the guess value and f of the guess value. And from that we draw whatever we have updated comma 0 ok; it says quite simple. So, let us do this. So, let us plot, so `plt.plot`. So, the first thing that we will plot or rather let us draw the line directly.

So, the x coordinates. So, let us first draw this particular line, let us draw this line. So, the coordinate will be $x_0, f(x_0)$ to $x_1, 0$. So, in order to plot, we will first give the x coordinate. So, it will be x, x ; the y coordinates will be $0, f(x)$, alright. Once the updated value is obtained, let me save it to x_1 ; let me save the old value in x_0 .

So, now we will do `plt.plot`. So, it is going from this old value to the new value. So, this is the new value, this is the old value ok. So, the x coordinates will be x_{old}, x and the y

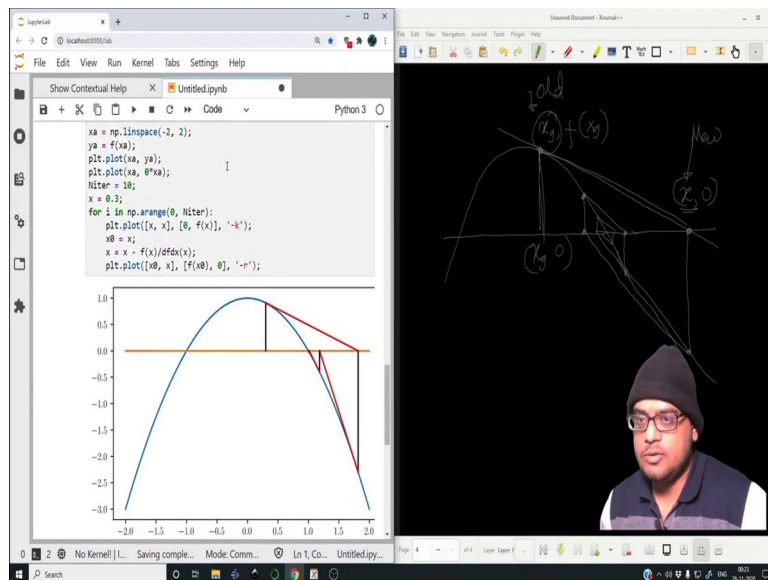
coordinates will be $f(x_0), 0$ alright. So, let me make this tangent to be a red line; let me make the vertical lines to be black lines, alright.

(Refer Slide Time: 37:10)



So, this is how it looks. So, this was the initial guess; we go over here, make a tangent.

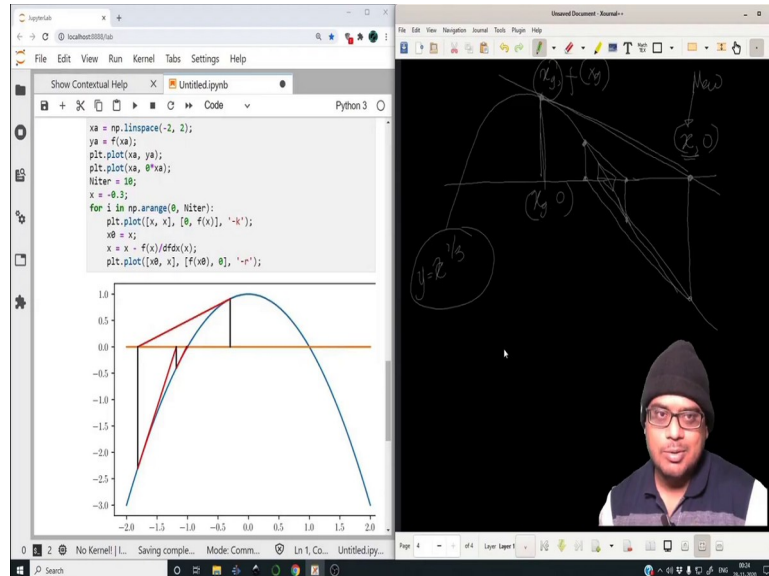
(Refer Slide Time: 37:21)



So, let me also draw the x axis. So, this makes things much easier. So, if this is the guess value, then you draw a tangent, you drop the vertical onto the curve, draw a tangent, drop the vertical onto the curve again. So, eventually you converge very quickly to the plot. Let us

change the guess value; let us change the guess value to - 0.3 for example and it obviously converges to the other root.

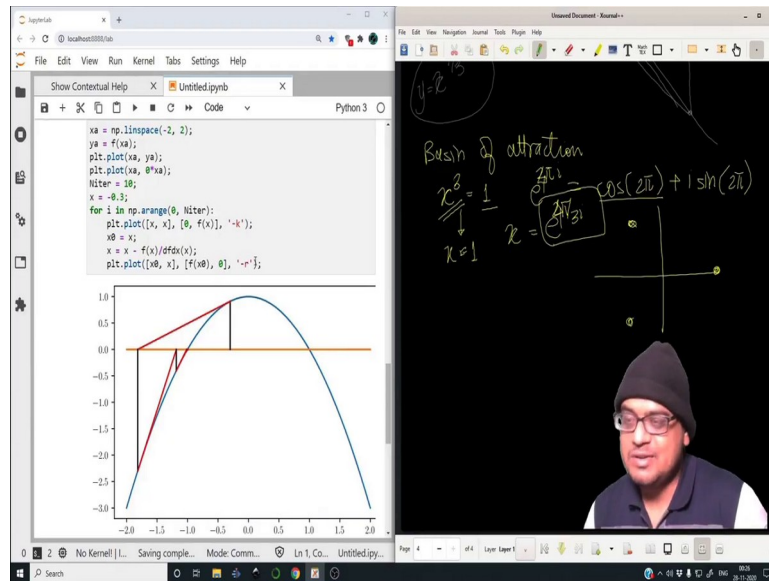
(Refer Slide Time: 37:52)



So, it all depends on where you guess, ok. Now, you may think that great, Newton Raphson iterations they converge for all guess values regardless of what is going on; but obviously, that is incorrect, it does not converge, you need a good guess value also. Not for this particular simple case; you can go ahead and plot, you can go ahead and use this function to draw a bunch of complicated curves.

In particular there is a very famous case x ; so $y = x^{(1/3)}$. So, this particular thing, the convergence using Newton iterations is quite difficult, ok. I ask you to have a go to this; have a go on this sometime later on.

(Refer Slide Time: 38:57)



So, now this is fine; but let us now move on to a slightly more interesting topic, it is called as basins of attraction. So, we know that if we have a n th order polynomial, there will be n roots. So, there is a multiplicity of roots, ok.

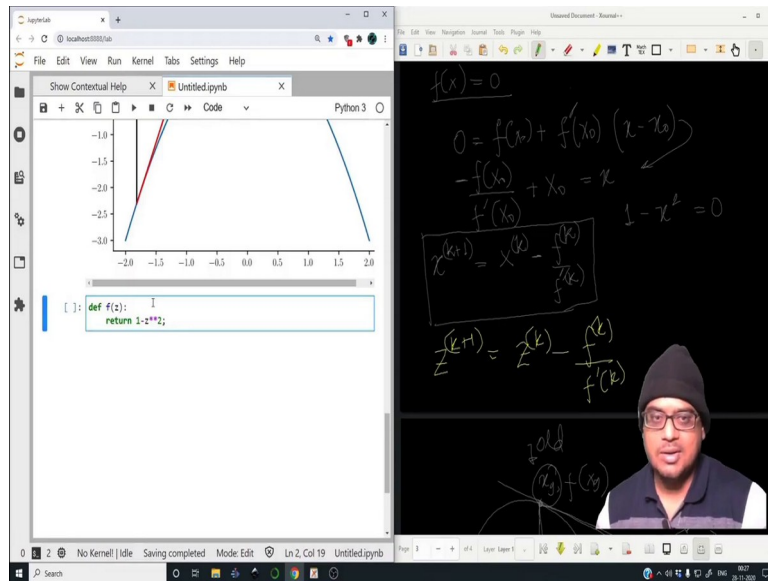
So, if we consider just the equation x to the power say $x^3=1$, suppose this equation. So, what are the roots? So, immediately you may say $x=1$ is your root and that is fine $x=1$ is indeed a root; but in the complex plane there are two more roots, ok. So, 1 is $e^{(2\pi i)}$; so $2\pi i$, ok.

This is what 1 is in the complex plane ok; because e to the power this is $\cos(2\pi) + i\sin(2\pi)$, ok. So, right over here you see that this is 1 , this is 0 . So, this is the representation of 1 .

So, when you take a third root, x becomes $e^{(2\pi i/3)}$, so obviously these are some, this has some coordinates; I mean in the argand plane that is the complex plane, there are points which correspond to this. So, one root is this, one root will be this and one root will be this. So, there will be three roots; the third root I mean, this is also a solution, ok.

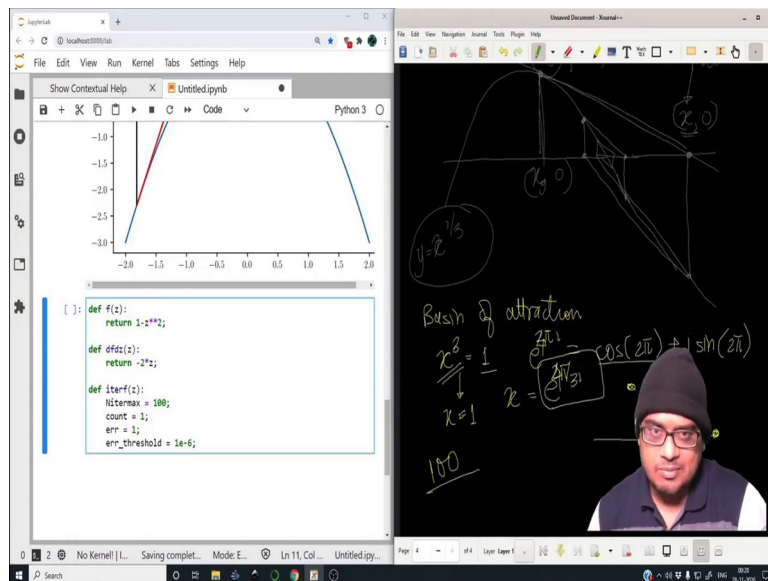
So, you can find out $n\pi$ and then divide and then you get multiplicity of roots. So, this is something which you might have studied in class 11 or 12. So, now, let us see how we can represent all this in the complex plane. So, the same technique can also be used to find out the roots in the complex plane.

(Refer Slide Time: 41:00)



So, simply $Z(k + 1) = Z(k) - f(k)/f'(k)$. So, it is just a matter of replacing everything by the complex function, nothing else.

(Refer Slide Time: 41:25)

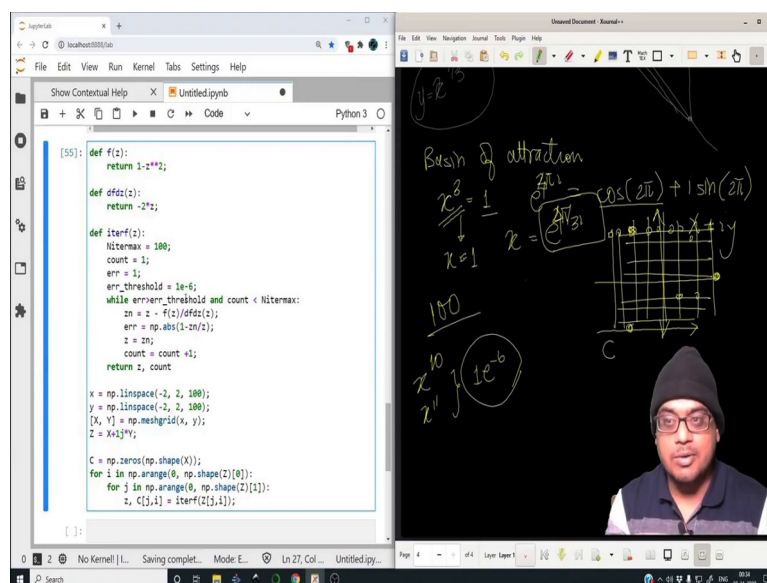


So, let us do that, let us find out the following. Let us make a general function; let us define $f(z)$ and it will return $1 - z^2$. I mean it is just an easy case, we know that this root will still be plus - 1, we do not need to worry so much; but I mean depending on what this return value will be, we will obtain a bunch of complex solutions as well.

Let us define dfdz it will take an input z; let it return $-2*z$. Let us define the iteration. So, define iterate or iterf and it will take as an input z that is the initial guess. So, once it enters with an initial guess Nitermax = 100, count =1, error =1, err_threshold =1e-6.

So, the reason why I am defining an error threshold is because, I am giving a very large number of iterations; I am giving 100 iterations. So, suppose like in the previous problem; in the previous problem, we give 10 iterations, but we see that even after 4 iterations it had converged, right. So, we do not want to waste iterations to keep, to make it keep converging.

(Refer Slide Time: 43:02)



Rather once it reaches a condition where suppose x_{10} and x_{11} ; so the difference between these two is $1e-6$ that is the threshold that we defined. If the difference between these two falls below this; we say that, the convergence has happened, we do not need to iterate anymore. That is why I define a bound on the error, so that we can break the loop; we do not need to keep on looping till 100 iterations.

So, the way to do it is, while error is greater than error threshold ok; what should we do? So, z_n will be equal to $z - f(z)/dfdz(z)$. The error that will be calculated will be $np.abs()$; so we will find out the relative error, $1 - z_n/z$, ok. So, it is updating the value of z and also assigning it to z_n , so that we still have the old value z; we are finding out the absolute, the relative error between z_n and z and then we can update z. So, then z equal to z_n and we can update count as well

So, once we do this, we can return the value of count; we can return the value of z and count, ok. So, we will return z ,count. So, not only will we have. Actually so, Niter max does not make sense, Niter max in this program does not make sense; but we can say if the error is greater than error threshold and count is less than Niter max.

So, it there can be a case where you are continuously iterating; but you have not reached the root ok, you will keep on iterating till infinity, but you want the program to stop somewhere. So, you say that, unless you still not reach the error threshold; but wait you have done sufficiently large number of iterations, so you have to stop somewhere.

So, that is the bit the program bit which will do that. So, it will check these two conditions; if error is greater than error threshold and the count is less than the maximum iterations, ok. So, it will return the value of z and the number of iterations it took, ok. So, let me execute this and see there is no error fine, everything is fine.

So, now let us define the complex plane, this is the complex plane; a point on the complex plane is x plus i y, ok. So, in that case this will be `x = np.linspace(-2,2)` let say 100 points, `y = np.linspace(-2,2)` and say 100 points, `X,Y = np. Meshgrid(x,y)`.

Then we will define the z matrix as `X + 1 j * Y`; remember that X and Y are two matrices which contain all the x coordinates and y coordinates of all the intersection points, then we assign it to z. So, now, let us define c as just a color array; it will sort of show the number of iterations it took for each point in that entire grid.

So, `np.zeros(np.shape(X))`. So, it is going to create a c array. So, corresponding to this entire grid, we will have a bunch of x values for all of this, a bunch of y values of all of this.

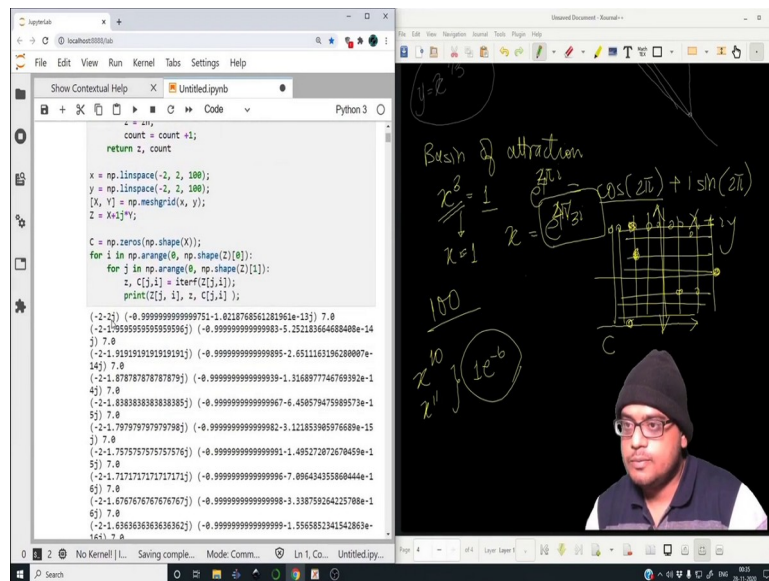
And for all the points on the grid, we will define a zero array c and eventually that c will contain the total number of counts or the total number of iterations it took to reach that particular threshold that we have defined, alright. So, c is this for i in `np.arange(0,np.shape(z)[0])` its first value; for j in `np.arange(0, np.shape(z)[1])` the first value, `c[j,]`. So, `z,c[j,i]=iterf(z[j,i])`.

So, we are looping over all the values in this grid. Note it is j,i, it is not i,j and think about it, why it is that way ok; because the rows and columns are swapped. If you move along

columns, you are actually incrementing x that is the reason why it is j,i. And I request you to think about it; unless you think deeply about it, it will never be clear.

So, $c[j,i]$ is equal to. So, you are taking one point on the argand plane, passing it to the function iterf, doing all the iterations, finding out how many iterations it took for a point, for that particular point to converge to a root; the value of the root will be z. So, once this is done. In fact, let me print out the value as well.

(Refer Slide Time: 49:18)



So, let me print out z and $c[j,i]$. And in fact, let me print out the initial or the point in the argand plane as well, where we have started. So, it will give us the point where we have started the converged point and the number of iterations it took. Let me execute this, ok.

So, the first point $-2 - 2j$, it converge to $-0.99 + 10^{(-13)}$. So, it converge to a value of -1; essentially this is -1, it took 7 iterations. In fact, this is a very odd way of representing it; I do not want so many digits.

(Refer Slide Time: 50:10)

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

```

x = np.linspace(-2, 2, 100);
y = np.linspace(-2, 2, 100);
[X, Y] = np.meshgrid(x, y);
Z = X+1j*Y;

C = np.zeros(np.shape(X));
for i in np.arange(0, np.shape(Z)[0]):
    for j in np.arange(0, np.shape(Z)[1]):
        z, c[j,i] = iterf(z[i,j]);
        print(z[i,j], np.round(z, 1), c[j,i]);
    
```

The output of the code is a list of complex numbers and their rounded values, such as:

```

(-2-2j) (-1-0j) 7.0
(-2-1.9595959595959595j) (-1-0j) 7.0
(-2-1.9191919191919191j) (-1-0j) 7.0
(-2-1.8787878787878787j) (-1-0j) 7.0
(-2-1.8383838383838383j) (-1-0j) 7.0
(-2-1.7979797979797979j) (-1-0j) 7.0
(-2-1.7575757575757575j) (-1-0j) 7.0
(-2-1.7171717171717171j) (-1-0j) 7.0
(-2-1.6767676767676767j) (-1-0j) 7.0
(-2-1.6363636363636362j) (-1-0j) 7.0
(-2-1.5959595959595959j) (-1-0j) 7.0
(-2-1.5555555555555555j) (-1-0j) 7.0
(-2-1.5151515151515151j) (-1-0j) 7.0
(-2-1.4747474747474747j) (-1-0j) 7.0
(-2-1.4343434343434343j) (-1-0j) 7.0
(-2-1.3939393939393939j) (-1-0j) 7.0
(-2-1.3535353535353535j) (-1-0j) 7.0
(-2-1.3131313131313131j) (-1-0j) 7.0
(-2-1.2727272727272727j) (-1-0j) 7.0
(-2-1.2323232323232323j) (-1-0j) 7.0
(-2-1.1919191919191919j) (-1-0j) 7.0
(-2-1.1515151515151515j) (-1-0j) 7.0
(-2-1.1111111111111111j) (-1-0j) 7.0
    
```

So, let me just round this off to one decimal. So, $\text{np.round}(z,1)$, oh much better. So, we have started with $-2 - 2j$, it has converged to a value of $-1 + 0j$, it took 7 iterations; then we took this value, it also converged to -1 , 7 iterations so on and so forth.

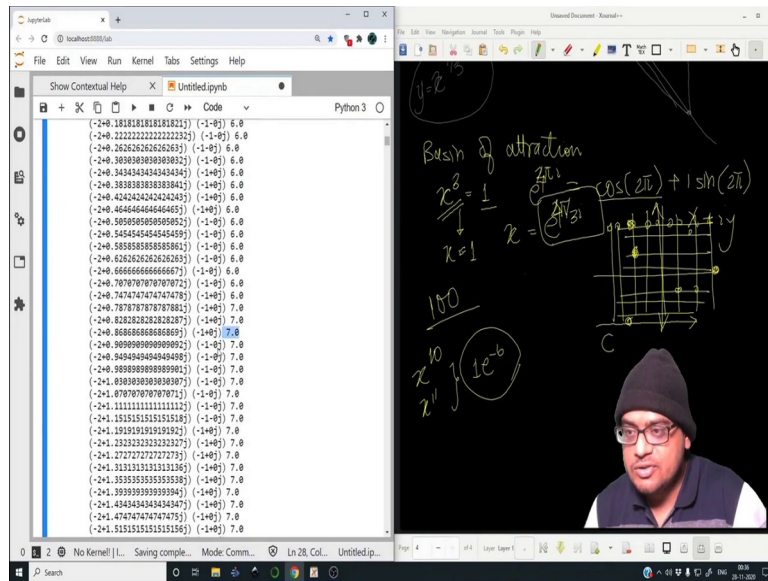
(Refer Slide Time: 50:32)

The screenshot shows the same Jupyter Notebook with the code from the previous slide. The output is now rounded to one decimal place:

```

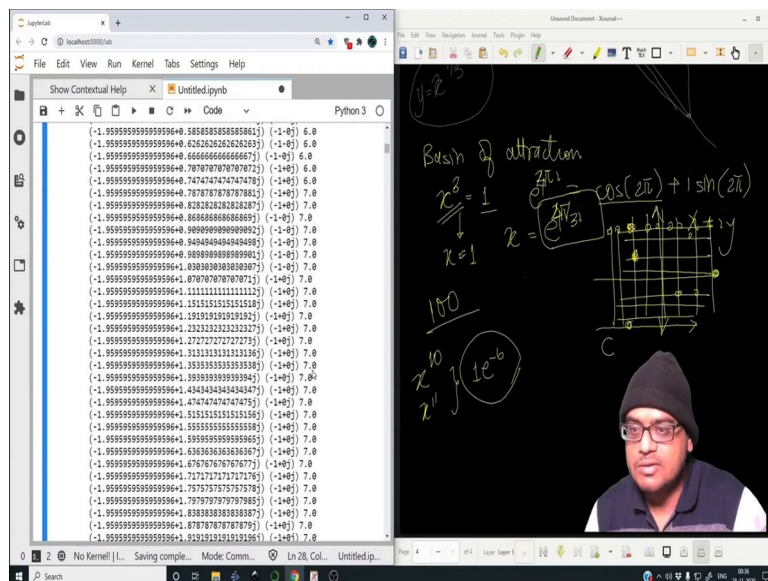
(-2-2j) (-1+0j) 6.0
(-2-1.96j) (-1+0j) 6.0
(-2-1.91j) (-1+0j) 6.0
(-2-1.87j) (-1+0j) 6.0
(-2-1.84j) (-1+0j) 6.0
(-2-1.8j) (-1+0j) 6.0
(-2-1.76j) (-1+0j) 6.0
(-2-1.72j) (-1+0j) 6.0
(-2-1.68j) (-1+0j) 6.0
(-2-1.64j) (-1+0j) 6.0
(-2-1.6j) (-1+0j) 6.0
(-2-1.56j) (-1+0j) 6.0
(-2-1.52j) (-1+0j) 6.0
(-2-1.48j) (-1+0j) 6.0
(-2-1.44j) (-1+0j) 6.0
(-2-1.4j) (-1+0j) 6.0
(-2-1.36j) (-1+0j) 6.0
(-2-1.32j) (-1+0j) 6.0
(-2-1.28j) (-1+0j) 6.0
(-2-1.24j) (-1+0j) 6.0
(-2-1.2j) (-1+0j) 6.0
(-2-1.16j) (-1+0j) 6.0
(-2-1.12j) (-1+0j) 6.0
(-2-1.08j) (-1+0j) 6.0
(-2-1.04j) (-1+0j) 6.0
(-2-1j) (-1+0j) 6.0
    
```

(Refer Slide Time: 50:36)



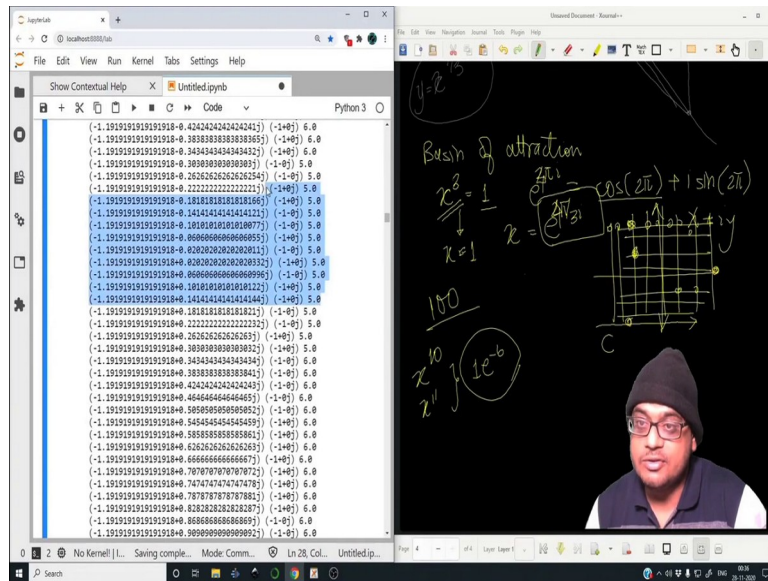
Some of them took 6 iterations; some of them took 7 iterations ok, some of them 6 iterations.

(Refer Slide Time: 50:42)



Let us scroll through this 7 iterations, 6 iterations; to reach that error threshold, it has taken 6 iterations.

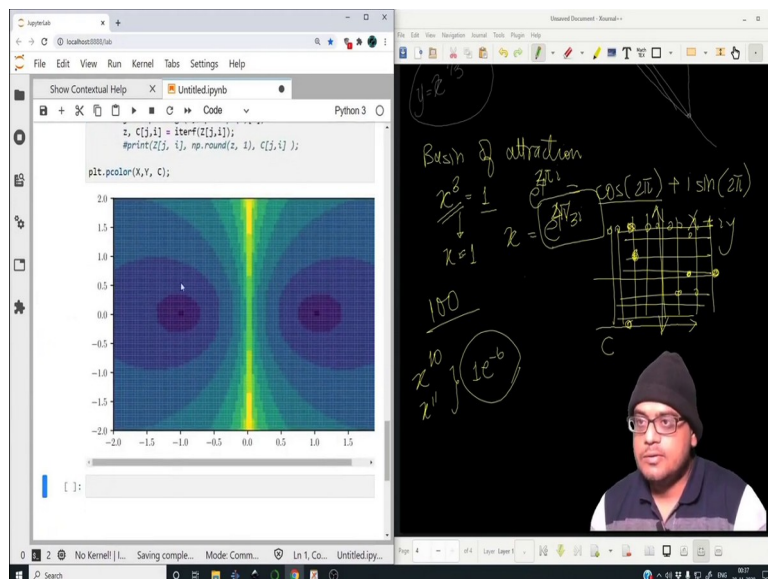
(Refer Slide Time: 50:57)



There are some zones with 5 iterations excellent; but obviously there will be points in the argand plane where the root would have been exactly that point, it would have taken no iterations, some points have taken 8 iterations, ok.

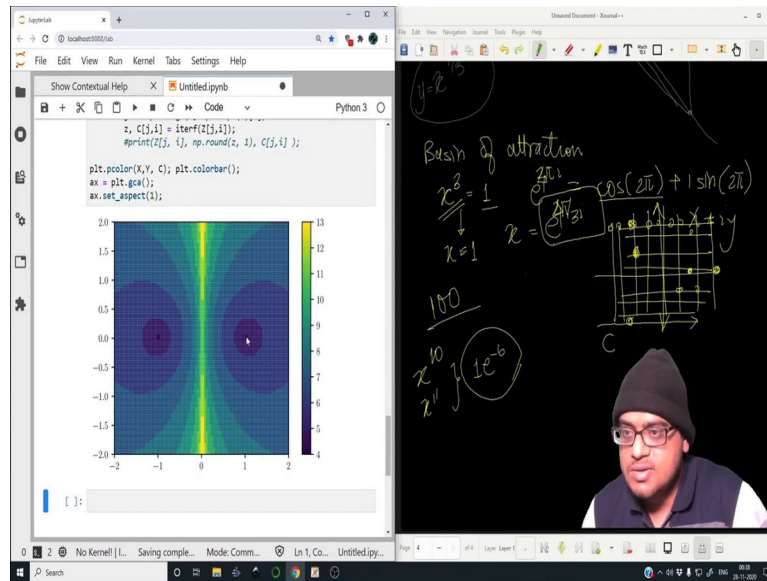
So, rather than scrolling this list indefinitely; let us in fact plot for each corresponding point on the argand plane, the number of iterations it took. Let me comment out this line; once we exit the loop, let me plot it.

(Refer Slide Time: 51:32)



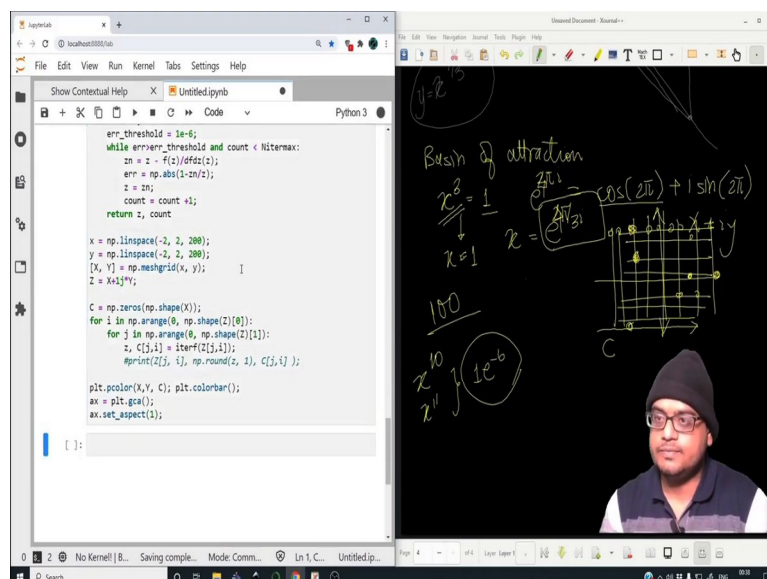
So, plt.pcolor; so pcolor stands for a pseudo color plot, ok. So, it is a pseudo color plot; it tells it, it gives you an information about the 2 D data that you have. So, x,y,c; let us do this, ok.

(Refer Slide Time: 51:54)



So, let me add a color bar. So, plt.colorbar and let me set the aspect ratio to 1. So, ax=plt.gca and then we will do ax.set_aspect(1), alright. So, in this region there are points which took only 4 iterations; but over here they took 13 iterations.

(Refer Slide Time: 52:30)



In fact if we resolve this further, let me resolve this further; it will take a bit more time to run, it should not take too much time, I am running a fairly decently powered computer, ok.

(Refer Slide Time: 52:41)

The screenshot shows a Jupyter Notebook interface. On the left, a Python script is visible:

```

C = np.zeros(np.shape(X));
for i in np.arange(0, np.shape(z)[0]):
    for j in np.arange(0, np.shape(z)[1]):
        z, C[i,j] = iterf(z[i,j]);
        #print(z[i,j], np.round(z, 2), C[i,j]);

plt.pcolorm(X, Y, C); plt.colorbar();
ax = plt.gca();
ax.set_aspect(1);

```

Below the code is a heatmap plot showing iteration counts. The x and y axes range from -2 to 2. The plot shows two purple regions (low iteration) and a yellow/green region (high iteration) in the center.

On the right, there is a video feed of a person and a blackboard with handwritten notes. The notes include:

- $z^2 = 1$
- Basin of attraction
- $z^2 = 1 \Rightarrow z = 1$
- $z = \frac{2\pi i}{2\pi} = \cos(2\pi) + i \sin(2\pi)$
- A grid diagram with points and arrows.
- Handwritten notes: z^{10} , z^{11} , 10^{-6} .

So, these are the zones where it takes a lot of iterations; these are the zones where it takes moderate number of iterations, these are the zones where it takes minimum number of iterations. And it takes minimum number of iterations; because look it is very close to the roots; it is very close to the roots, in fact this is for $z^2 = 1$. So, what is the equation; $1 - z^2 = 0$, so essentially $z^2 = 1$.

(Refer Slide Time: 53:14)

The screenshot shows a Jupyter Notebook interface. On the left, a Python script is visible:

```

[61]: def f(z):
        return 1-z**3;

        def dfdz(z):
            return -3*z**2;

        def iterf(z):
            Niternax = 100;
            count = 1;
            err = 1;
            err_threshold = 1e-6;
            while err > err_threshold and count < Niternax:
                zn = z - f(z)/dfdz(z);
                err = np.abs(1-zn/z);
                z = zn;
                count = count + 1;
            return z, count;

x = np.linspace(-2, 2, 200);
y = np.linspace(-2, 2, 200);
[X, Y] = np.meshgrid(x, y);
Z = X+1j*Y;

C = np.zeros(np.shape(X));
for i in np.arange(0, np.shape(z)[0]):
    for j in np.arange(0, np.shape(z)[1]):
        z, C[i,j] = iterf(z[i,j]);
        #print(z[i,j], np.round(z, 2), C[i,j]);

```

Below the code is a heatmap plot showing iteration counts. The x and y axes range from -2 to 2. The plot shows three distinct regions of low iteration (purple) and high iteration (yellow/green).

On the right, there is a video feed of a person and a blackboard with handwritten notes. The notes include:

- $z^3 = 1$
- Basin of attraction
- $z^3 = 1 \Rightarrow z = 1$
- $z = \frac{2\pi i}{3} = \cos(2\pi/3) + i \sin(2\pi/3)$
- A grid diagram with points and arrows.
- Handwritten notes: z^{10} , z^{11} , 10^{-6} .

Let me make it cube and the derivative will be $-3*z^2$; let us see what happens, ok.

(Refer Slide Time: 54:36)

```
err_threshold = 1e-6;
while err < err_threshold and count < Nitermax:
    zn = z - f(z)/dfdz(z);
    err = np.abs(1-zn/z);
    z = zn;
    count = count + 1;
return z, count

x = np.linspace(0, 1, 200);
y = np.linspace(0, 1, 200);
[X, Y] = np.meshgrid(x, y);
Z = X+1j*Y;

C = np.zeros(np.shape(X));
for i in np.arange(0, np.shape(Z)[0]):
    for j in np.arange(0, np.shape(Z)[1]):
        z, C[i,j] = iterf(Z[i,j]);
        #print(Z[i, j], np.round(z, 2), C[i,j]);

plt.pcolor(X, Y, C); plt.colorbar();
ax = plt.gca();
ax.set_aspect(1);
```

Basin of attraction
 $z^2 = 1$
 $z = 1$
 $z = -1$
 $z = \cos(2\pi) + j \sin(2\pi)$
 $z = 1$
 $z = -1$
 $z = 10^{-6}$

(Refer Slide Time: 54:47)

```
err_threshold = 1e-6;
while err < err_threshold and count < Nitermax:
    zn = z - f(z)/dfdz(z);
    err = np.abs(1-zn/z);
    z = zn;
    count = count + 1;
return z, count

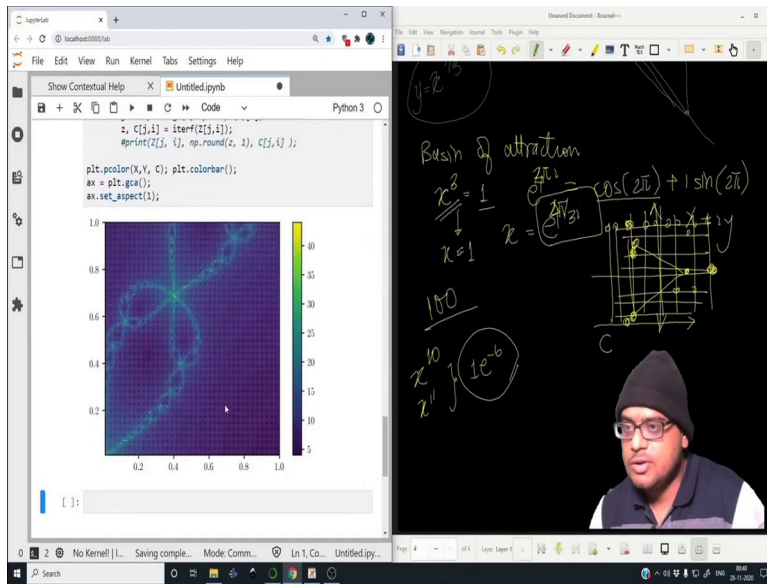
x = np.linspace(0.01, 1, 200);
y = np.linspace(0.01, 1, 200);
[X, Y] = np.meshgrid(x, y);
Z = X+1j*Y;

C = np.zeros(np.shape(X));
for i in np.arange(0, np.shape(Z)[0]):
    for j in np.arange(0, np.shape(Z)[1]):
        z, C[i,j] = iterf(Z[i,j]);
        #print(Z[i, j], np.round(z, 2), C[i,j]);

plt.pcolor(X, Y, C); plt.colorbar();
ax = plt.gca();
ax.set_aspect(1);
```

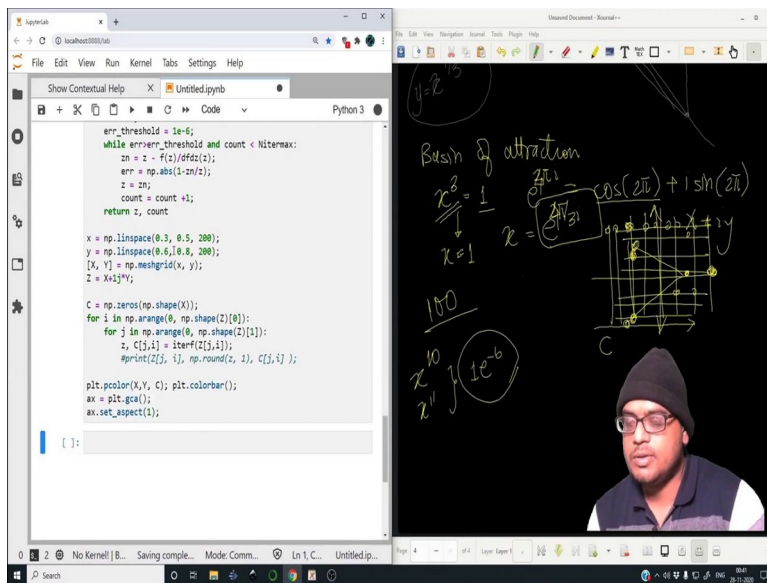
Let me run this cell again, ok. So, I should take 0.01 and 0.01; because having 0 that d f d x causes some errors. So, once we run this, we will see a magnified portion, ok.

(Refer Slide Time: 55:02)



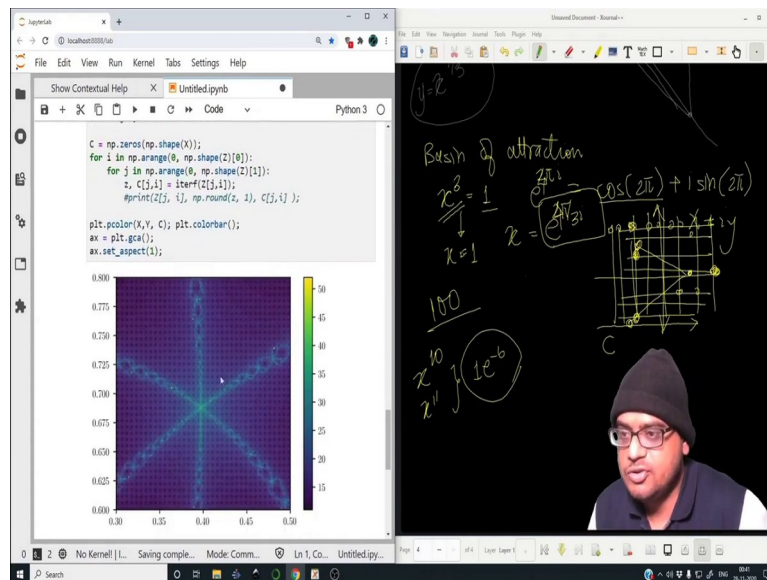
So, I do not know if you can see this in the recorded resolution; there is plenty of things that are going on.

(Refer Slide Time: 55:30)



In fact let us try to zoom in on this particular point; so this particular point is approximately say it is a tile of 0.6 to 0.8 in the y direction and 0.3 to 0.5 in the x direction ok, 0.3 to 0.5 in the x direction and 0.8 and 0.6. Let me run this. So, it will do its calculation no problem.

(Refer Slide Time: 55:55)

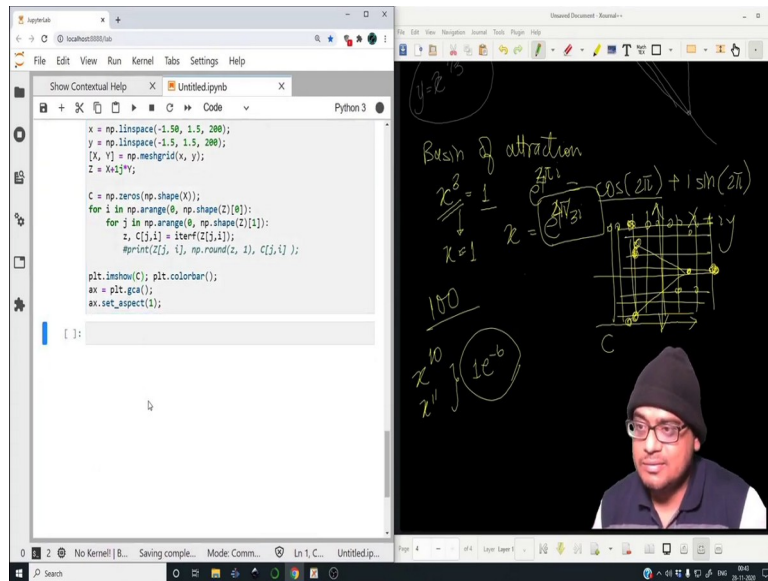


It takes a bit of time; because I have given a significantly large, ok. So, it shows a very; I mean despite being, despite zooming into this further and further, we seem to have that braided structure and it is appearing out of nowhere. I mean you could not anticipate from this that, there will be these zones in and otherwise purple looking domain.

So, the purple is zones which do not require lot of iterations to converge; but inside them there are these link like structures and if you even zoom in further, you will keep on seeing this link like structures.

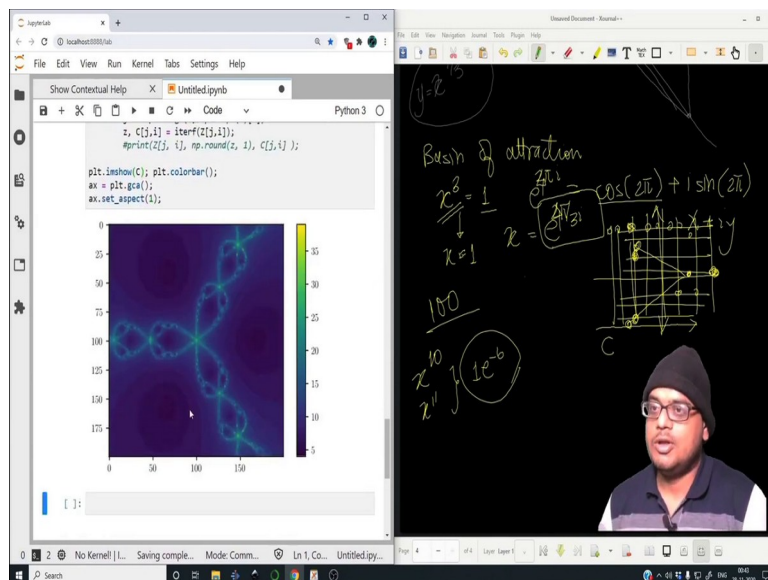
And the orange, the purple zone is called as the zone of attraction; these blue links they are the zones where there is no, I mean the convergence takes a lot of time. In fact, for a very complicated form of the function that we are trying to find; there may be zones where there is no convergence at all, ok. There may be zones where there is no convergence; in fact let me go back to the default value.

(Refer Slide Time: 57:02)



Let me go back to - 1.5 to 1.5. In fact, instead of pcolor; pcolor that rendering takes a bit more time. Let me just show the; let me show the array c as an image, that does not take so much time, ok. So, the computation ok; let me just tell you the computation here does not take much time, it is the pcolor function that takes lot of time.

(Refer Slide Time: 57:44)



Now, this should run quite fast; at least relatively fast ok, there you go. So, these are the zones where the roots are; these are also the zones where you have fast convergence, ok.

Inside all those links and lobes, you have zones of fast convergence, ok. It is a very fascinating diagram; it is not at all trivial to predict a priori.

(Refer Slide Time: 58:12)

The screenshot shows a Jupyter Notebook interface on the left and a video feed on the right. The notebook code defines a function $f(z) = 1 - z^4$ and its derivative $dfdz(z) = -4z^3$. It then implements an iteration function $iterf(z)$ that updates z to $z - f(z)/dfdz(z)$ for 100 iterations. The iteration points are stored in a grid C . The video feed shows handwritten notes on a blackboard. The notes include the title "Basin of attraction", the equation $z^4 = 1$, and the polar form $z = r(\cos(\theta) + i\sin(\theta))$. A grid is drawn with points z^k and z^{k+1} connected by arrows, illustrating the iteration process. The video feed also shows a person's face in the bottom right corner.

(Refer Slide Time: 58:26)

The screenshot shows a Jupyter Notebook interface on the left and a video feed on the right. The notebook code plots the iteration points C using `plt.imshow(C); plt.colorbar();`. The resulting plot is a heatmap showing the distribution of iteration points, with a color scale from 0 to 100. The video feed shows a diagram of the complex plane with a grid and arrows indicating the iteration process. The video feed also shows a person's face in the bottom right corner.

In fact, let me make this z^4 . So, z^4 , you have 4 roots ok, you have 4 roots over z to the power 4; this is the first root, second root, third root, fourth root, ok.

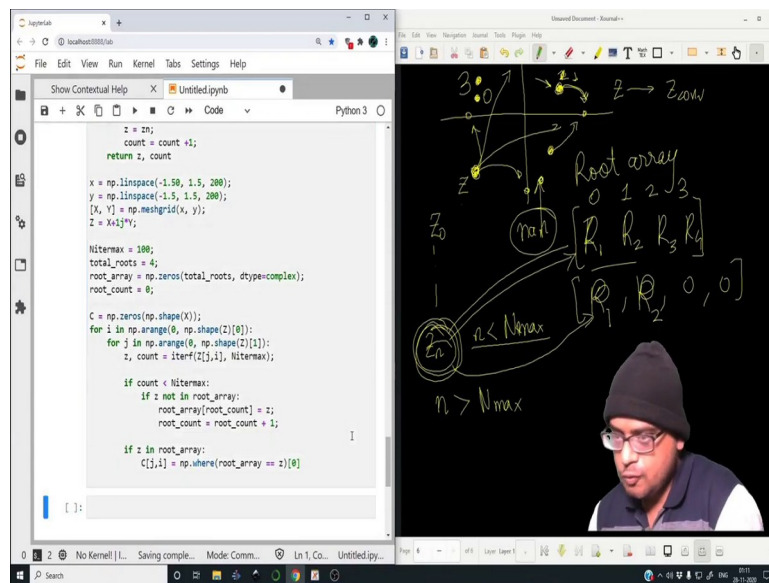
So, let us see; let us not guess around and let us see what happens, ok. So, 100 iterations have been hit at several points, ok. So, these are the, these diagonal lines are the zones where you do not really have a very easy conversions; but these locations are the points where you have fast convergence. These gaps between these weird links, they are also having good

convergence. Even on these lines, you have zones of fast convergence and slow convergence; it is quite bizarre ok; it is quite bizarre.

So, lastly we move on to the last topic, in which we want to classify this entire zone as to which root it actually converges to, ok. Right now, we just know that ok, this point takes less iterations to converge, this point takes less iteration, these points take less iterations to converge; but there is still the question whether this guess over here?

So, essentially we are guessing this point and does it converge to this root, does it converge to this root, does it converge to this root or does it converge to this root? A very non trivial question to answer as well and the answer is not at all straightforward ok; the answer is not at all straightforward. So, let us make a program to see that, ok.

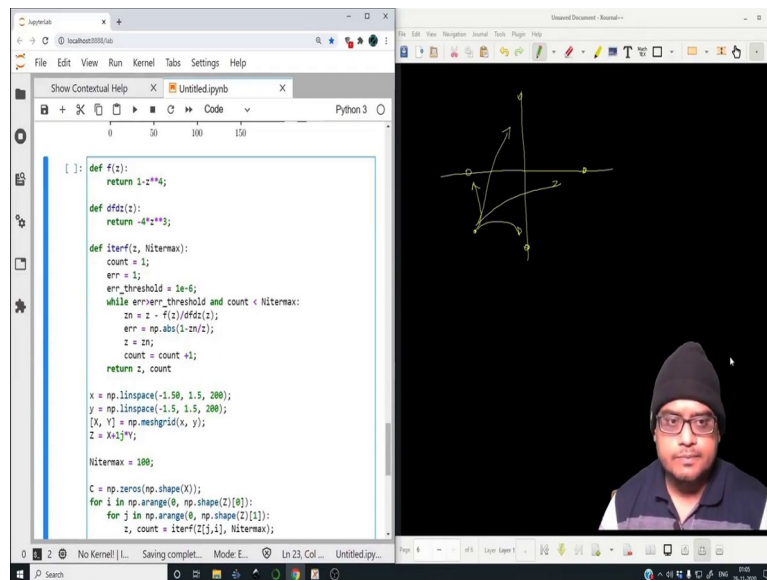
(Refer Slide Time: 60:13)



So, now let us look at the basins of attraction that is, if we have an argand plane and if we know that these are the roots, we want to know whether when we take this point on the argand plane and then when we iterate it; whether this root converges to this root or this root or this root or this root, essentially that is what we want to find out.

So, let us make use, let us make use of the same code; let us reuse this code. So, we do not need to plot anything and in fact, rather than passing the number of iteration, rather than hard coding the number of iterations; let us pass it into the function, ok.

(Refer Slide Time: 61:01)



```
[ ]: def f(z):  
    return 1-z**4;  
  
    def dfdz(z):  
        return -4*z**3;  
  
    def iterf(z, Nitermax):  
        count = 1;  
        err = 1;  
        err_threshold = 1e-6;  
        while err > err_threshold and count < Nitermax:  
            zn = z - f(z)/dfdz(z);  
            err = np.abs(1-zn/z);  
            z = zn;  
            count = count + 1;  
        return z, count  
  
    x = np.linspace(-1.50, 1.5, 200);  
    y = np.linspace(-1.5, 1.5, 200);  
    [X, Y] = np.meshgrid(x, y);  
    Z = X+1j*Y;  
  
    Nitermax = 100;  
  
    C = np.zeros(np.shape(X));  
    for i in np.arange(0, np.shape(Z)[0]):  
        for j in np.arange(0, np.shape(Z)[1]):  
            z, count = iterf(Z[i,j], Nitermax);
```

So, let us take as an output the count; we want to take the count into picture, we do not want this. So, this is essentially what the code is and we have to pass the count the Nitermax as well.

So, we have to define the Nitermax over here. So, let say it is 100. So, as the maximum number of iterations we will allow. So, what we want to do is, essentially take this value of z in the argand plane, iterate it. So, when it is iterated; so this is z_0 , then suppose you said z_n . So, if this converged value with this error threshold is reached in n less than Nitermax, then we found a root.

So, we must take this z_n and we must insert it into; we must keep in mind that it is a root. So, how do we keep in mind it is a root? We must create a root array. So, the root array will contain the different roots. So, root 1, root 2, root 3; so if it is a fourth order equation, there will be 4 roots and so on, ok. So, we have to store this. So, because we do not know the roots of priory; if we knew the roots of priory, why would be doing this in the first place? So, we do not know the roots.

So, we need to define certain things. So, we will define total roots, so it will be 4. So, root array will be `np . zeros` and this will be total roots. And remember the roots will be complex; so we have to change the data type to complex; this is how you do it.

In fact, the total roots if, even if we do not know if it is a complicated function; we can define the total roots to be much more than 4, it can be some large number. The fact that we are initializing everything to zero; so it makes things much simple, we can take the total roots to be much larger, other things will all be zero, ok.

So, each time you find a root, you insert it; you insert a root if the root is not already in the root array. So, initially the root array is all zeros; suppose you take this particular point, it converges to root 1. So, you take whatever this converged value will be and you insert it over here. So, this will become a root 1.

Suppose this particular point, it converges to a different value; so then you take it and insert into this. Suppose this point it converges to the same value; so then root 2 is already existing. So, do not need to insert anything to root array. So, you take all the points in the argand plane, iterate all of them, see their convergence and if they converge to a new root, you add it to the root array; if it does not, then you do something else that is what the idea is behind the root array.

So, root array is this, root count =0; then we have called this, this is z is the root and count is the number of iterations. So, if count is less than Nitermax; it means that, if the number of iterations has exceeded the maximum number of iterations, we have not converged anywhere within this particular error threshold.

It means that, that particular point in the argand plane took way more iterations and it probably would not converge; this means that we will assign this argand plane point a value of nan ok, nan stands for not a number, it does not stand for a cheese naan or a butter naan, it is not a number.

So, if you do not achieve convergence; you will put it as nan, so that in the plot it will appear as a white space. So, if count is less than Nitermax; what will we do? If z not in root array. So, if the converged value is not inside the root array; then what do you do? You insert it into the root array. So, root array, root count will be equal to z ; then root count is equal to root count plus 1 ok, this is what you do.

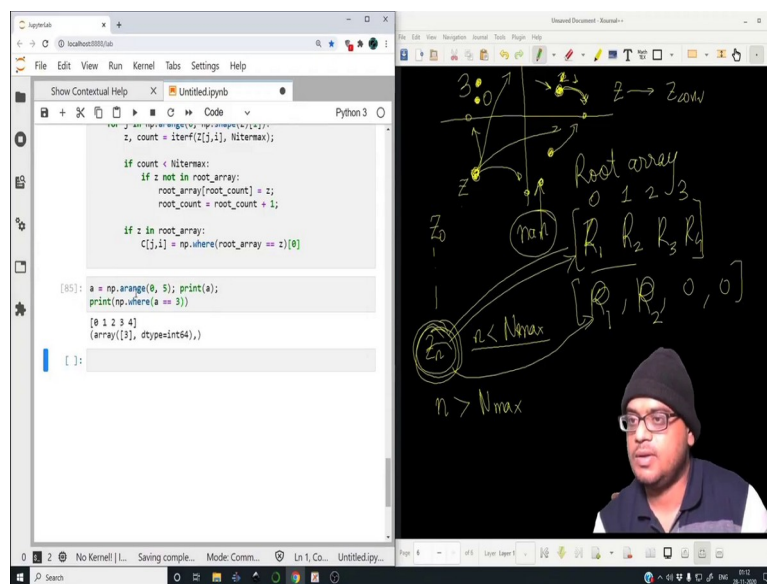
Now, if root or if z is actually in the root array, ok. So, if z in root array; then what do you do? So, for this particular point to be converged to this root and this the converged value for this. So, the converged value, if it is R_2 ; then we will assign this point, this particular index.

So, the index will be 0, 1, 2, 3. So, if it converges to second root, we will assign it a value of 2, assign it a value of 1; if this value converges to the third root, we will assign it a value of 3; if this point converges to the first root, we will assign it a value of 0.

So, basically we are taking the entire argand plane; for each point on the argand plane we are assigning it a value 0, 1, 2, 3 depending on to which root it converges, right. And if it does not converge, we set it to a value of nan; when we plot this depending on the color of the zone, we should be able to predict or not predict, we should be able to tell that this point will converge to this value ok, that is what we are targeting.

So, if z is in the root array; then what should we do? $C[j, i]$ will receive the index. So, $np.where$ root array equal to equal to z ok; it checks where root array is equal to z and the 0th entry of this.

(Refer Slide Time: 67:47)



And let me show you what the where function does just to recall your memory. So, $a = np.linspace(0, 5)$, let me print a . Now, not $linspace$, $arange$. So, now let me print $np.where(a == 3)$ or. So, it says a is 3 at the third point 0 1 2 3; in fact let me make this 2 to 10.

(Refer Slide Time: 68:27)

The screenshot shows a Jupyter Notebook interface on the left and a blackboard with handwritten notes on the right. The notebook code is as follows:

```
from math import pi; import numpy as np; z = 1 + 1j; z, count = iterF(2[j,i], Nitermax);  
  
if count < Nitermax:  
    if z not in root_array:  
        root_array[root_count] = z;  
        root_count = root_count + 1;  
  
if z in root_array:  
    c[j,i] = np.where(root_array == z)[0]
```

The notebook output shows:

```
[86]: a = np.arange(2, 10); print(a);  
print(np.where(a == 3))  
[2 3 4 5 6 7 8 9]  
(array([1]), dtype=int64,)
```

The blackboard contains handwritten notes: a complex plane diagram with points z_0, z_1, z_2, z_3 and a root array $[R_1, R_2, R_3, R_4]$. It also includes the expression $n < N_{max}$ and $n > N_{max}$.

(Refer Slide Time: 68:37)

The screenshot shows a Jupyter Notebook interface on the left and a blackboard with handwritten notes on the right. The notebook code is as follows:

```
from math import pi; import numpy as np; z = 1 + 1j; z, count = iterF(2[j,i], Nitermax);  
  
if count < Nitermax:  
    if z not in root_array:  
        root_array[root_count] = z;  
        root_count = root_count + 1;  
  
if z in root_array:  
    c[j,i] = np.where(root_array == z)[0];  
else:  
    c[j,i] = np.nan;
```

The notebook output shows an error:

```
-----  
IndexError                                Traceback (most recent c  
all last)  
<ipython-input-88-5674d3881fc> in <module>  
33     if count < Nitermax:  
34         if z not in root_array:  
--> 35             root_array[root_count] = z;  
36             root_count = root_count + 1;  
37  
IndexError: index 4 is out of bounds for axis 0 with size 4
```

The notebook output also shows:

```
[87]: a = np.arange(2, 10); print(a);  
print(np.where(a == 3)[0])  
[2 3 4 5 6 7 8 9]  
[1]
```

The blackboard contains handwritten notes: a complex plane diagram with points z_0, z_1, z_2, z_3 and a root array $[R_1, R_2, R_3, R_4]$. It also includes the expression $n < N_{max}$ and $n > N_{max}$.

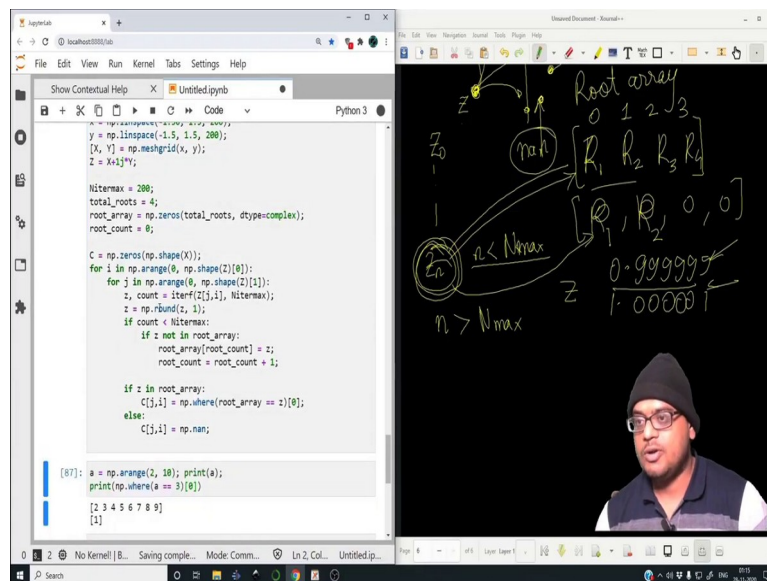
So, $a = 3$ happens at 1, so this is the solution. And we do not require the data type; so we need to take the 0th element of this output. So, `np.where` outputs it has a tuple, because there may be matrices.

So, we need the 0th element of this output, this function output; that is why this 0 appears. These are just syntax, I mean the logic should be clear; syntax follows any time, you can always Google the syntax, but it should be clear what the logic is. So, `c` should take the value

of the location of the root. If it is not the case, if z is not inside the root array; then what do you do?

If z is not inside the root array. So, else $c[j, i]$ has to be assigned a value of nan, there you go. So, this is it, this is the entire program; let me run this ok, there is a small error. Let us see what has happened, index 4 is out of bounds; let us see what is the issue, ok.

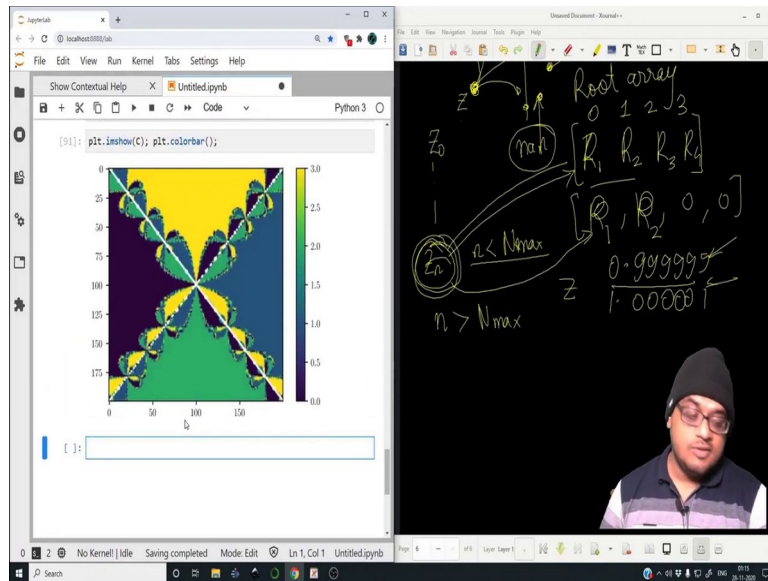
(Refer Slide Time: 69:51)



So, here is the issue; the value of z that we obtain over here. So, there will be roots which look something like this versus there will be roots which look something like this. Well these are essentially the same root, but the fact that the computer uses a floating point representation; we have to really reduce the number of digits in order to make these roots appear the same.

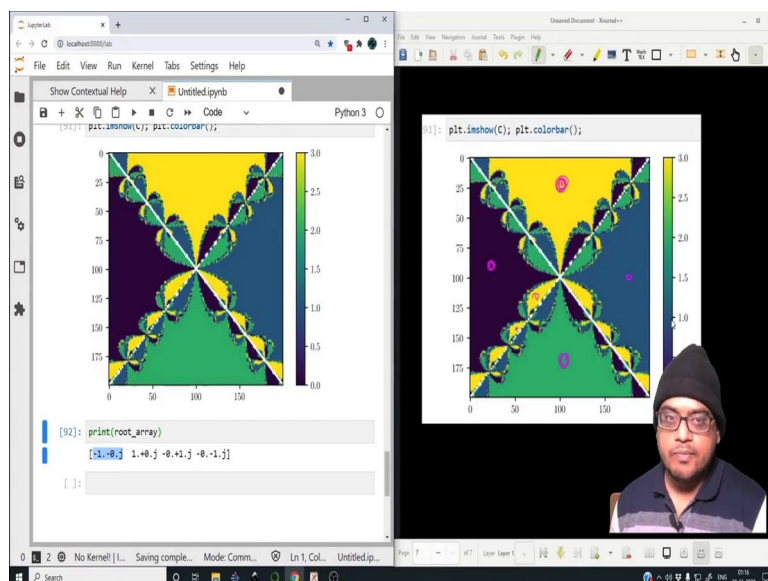
So, z , so essentially it is what is happening; because of this it is interpreting this and this as two different roots. So, it is creating much more roots than is desired. So, this is a classic mistake that this code attracts. So, z should be rounded. So, $np.round(z)$ and we have to round it to the first decimal place. This should now work excellent; it works let me remove this particular cell, we do not need it.

(Refer Slide Time: 70:51)



Let me now plot this. So, `plt.imshow(c)` and `plt.colorbar`; wow this looks fantastic. So, what is happening? We have four colors; we have four colors over here 3, 2, 1 and 0. In fact, let me take a screenshot of it and try to explain it in this.

(Refer Slide Time: 71:28)

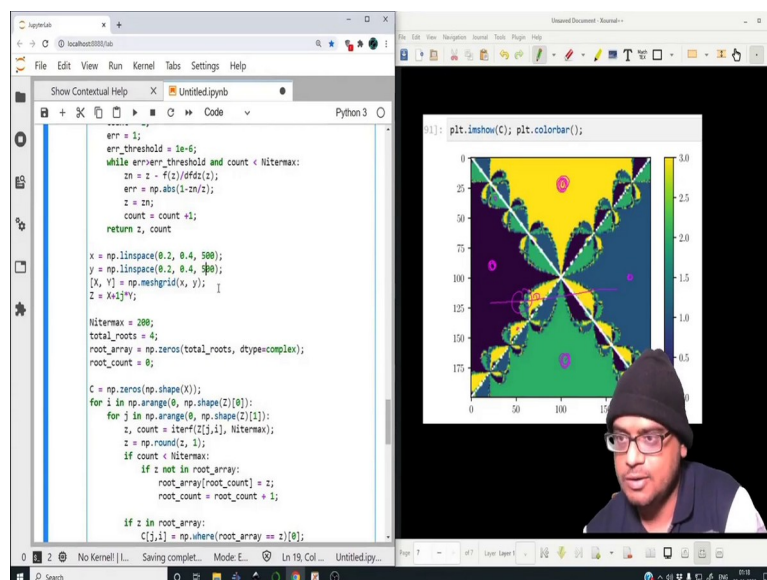


So, what is happening? Let us see. So, all the points; all the points that originate in this purple sector, they will converge to a certain root; all the points that originate in the yellow sector, they will converge to some different root; all the points that originate in the green sector, in the blue sector will converge to their each different roots.

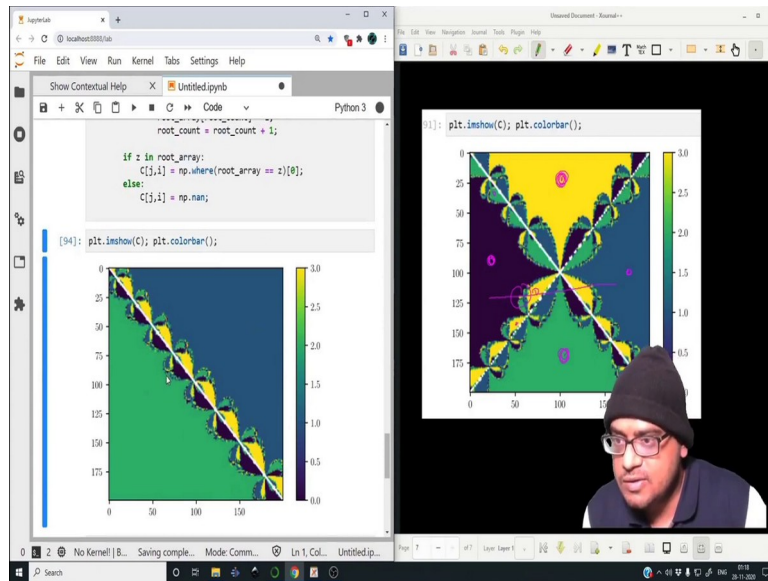
So, in case you want to know which root it converges to, we simply need to print out root array. So, the purple ones they converge to - 1; the blue ones they converge to 1; the green one they converge to i and the yellow one they converge to $-i$. So, now look there are pockets, there are small pockets which have different convergence properties; if you traverse across this line, you will see that, ok.

If you are at this boundary, it is very difficult for you to start off at a certain point and end up in a different in one of the particular roots; because you would expect this entire region to be converging to a certain root, but there are small zones. In fact, let me run this program and let us zoom into a certain section; let us try to zoom in to a certain section. So, say 0.2 to 0.4 and the same over here.

(Refer Slide Time: 73:14)

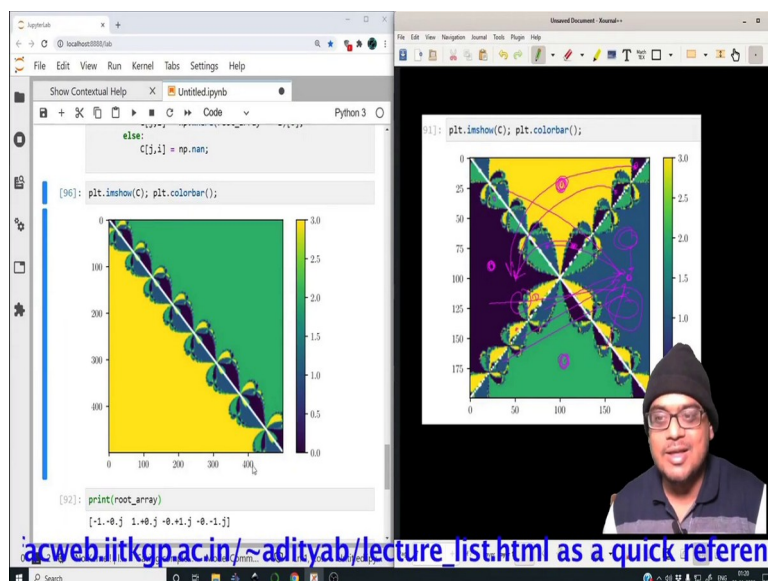


(Refer Slide Time: 73:31)



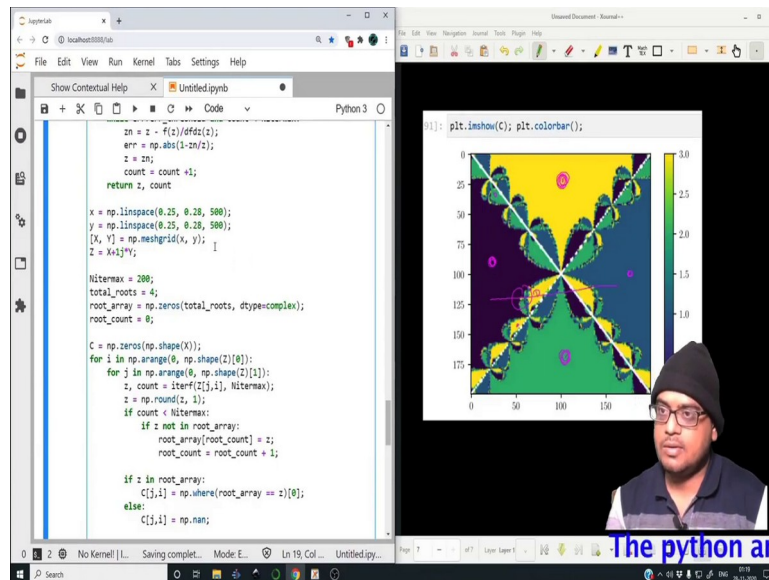
So, look if you focus over here very closely; let me increase the resolution maybe that helps 500 by 500 ok, this may take a while, but it is worth it. So, in that in the meantime let us see. So, there is green, then there is some pockets of multi colored stuff going on; then there is purple, then there is white. So, the white corresponds to zones where there is no convergence, ok. So, the white these two lines you have nans; that is why they are white, they have no color, much better.

(Refer Slide Time: 74:12)



So, there is purple over here; then again blue, then again green, yellow. So, there are there is some multi, I mean I do not want to call it a fractal right now; but the more you zoom in, the more these structures they will reveal themselves, ok.

(Refer Slide Time: 74:36)

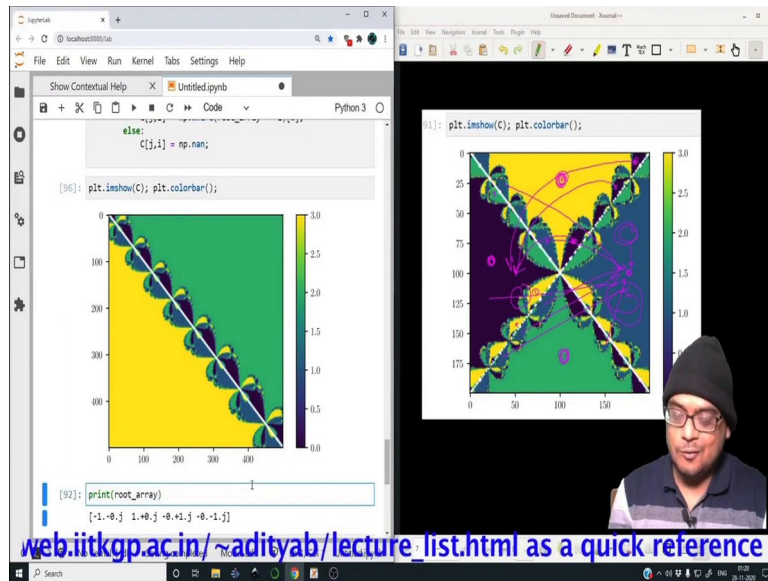


In fact, let us zoom in even more; let us see whether we get something from this, ok. So, understanding the convergence is not at all trivial is what I want to say and these things are called as Newton's basins of attraction.

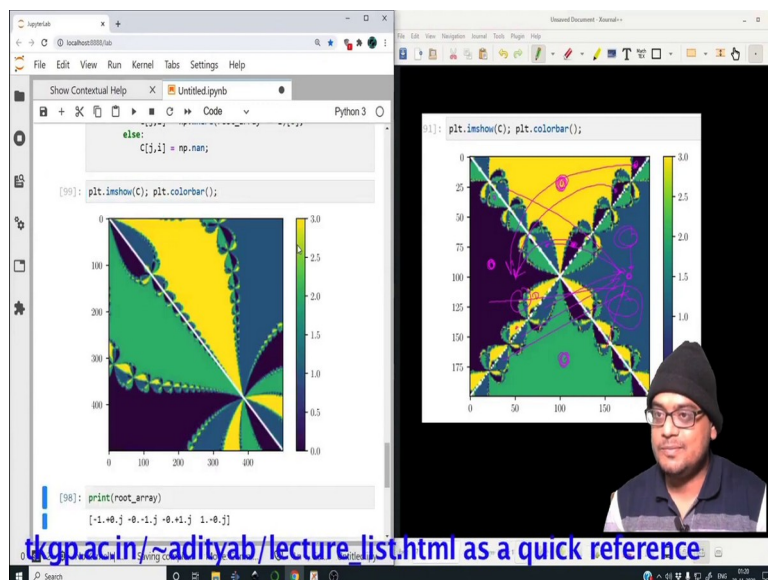
So, it is not at all trivial to tell whether this point in the argand plane will converge to that particular root; what you have is, this purple value over here, it is still converging to - 1, this thing is still converging to - 1, this thing is also still converging to -1 ok, the green one was converging to 1.

So, this thing is converging to 1, these all regions are converging to 1 no doubt, this is converging to 1, this is converging to 1, this is converging to 1, this is converging this is converging to 1 and there are small zones inside this as well; the more you zoom in, there will be even more zones of small very small pockets of green as well, they will also converge to 1.

(Refer Slide Time: 75:45)



(Refer Slide Time: 75:47)



Look the structure becomes even more bizarre. So, over here, there is another series of colors that will happen and the more you keep on zooming in; the more this bizarre structure will begin revealing itself. This white line is the zone where not even a single value will converge; all the points on the diagonals, they will diverge, ok. So, if you guess on this line, you will not get anything.

So, yeah this is what I wanted to discuss, I hope you have learned something new; if not, you have seen how to create such complicated plots with the help of very easy programming. And I will see you next time with another lecture on non-linear dynamics; until then it is goodbye from me, have a nice day, bye.