

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

Lecture – 23
2D Random walks

(Refer Slide Time: 00:30)

The python and octave notebooks can

N	P(N)
0	25
1	20
2	35

Hello and welcome to this lecture in which we are going to begin with 2D Random walks.

(Refer Slide Time: 00:42)

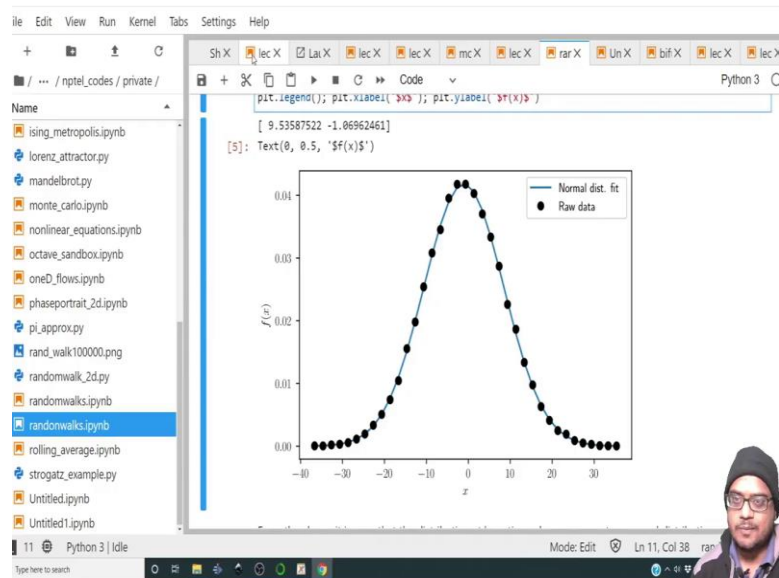
```
[5]: from scipy.optimize import curve_fit
def fitfunc(x, a, b):
    """Here the function to be fitted is written"""
    return 1/(2*np.pi*a**2)**0.5*np.exp(-(x-b)**2/(2*a**2))
# The values below are from the previous simulation
xvals = bins[0:-1]; yvals = counts;
opti_pt, _ = curve_fit(fitfunc, xvals, yvals);
print(opti_pt)

a = opti_pt[0]; b = opti_pt[1];
plt.plot(xvals, fitfunc(xvals, a, b), label='Normal dist. fit')
plt.plot(xvals, yvals, 'ok', label='Raw data');
plt.legend(); plt.xlabel('$x$'); plt.ylabel('$f(x)$')

[ 9.53587522 -1.06962461]
[5]: Text(0, 0.5, '$f(x)$')
```

And before going into 2D random walks, I just want to go back to our discussion and I think I had forgotten you to show how we can fit the normal distribution. I am just going to quickly browse through the program for that.

(Refer Slide Time: 01:15)



So, imagine you want to find out whether a given set of points. In this case, the black points they fit to a normal distribution. Well, how to go about doing that we know that a normal distribution has this particular form. So, the first thing we will do is create a function which will be the type of function that we are trying to fit, alright. So, it will take as an input x, it will take two parameters a and b.

So, the whole purpose of the fitting is to find out the appropriate parameters which will minimize the error between the fit and the raw data, alright. So, then this is the data we had from the histogram xvals and yvals. So, x values and y values is what we have.

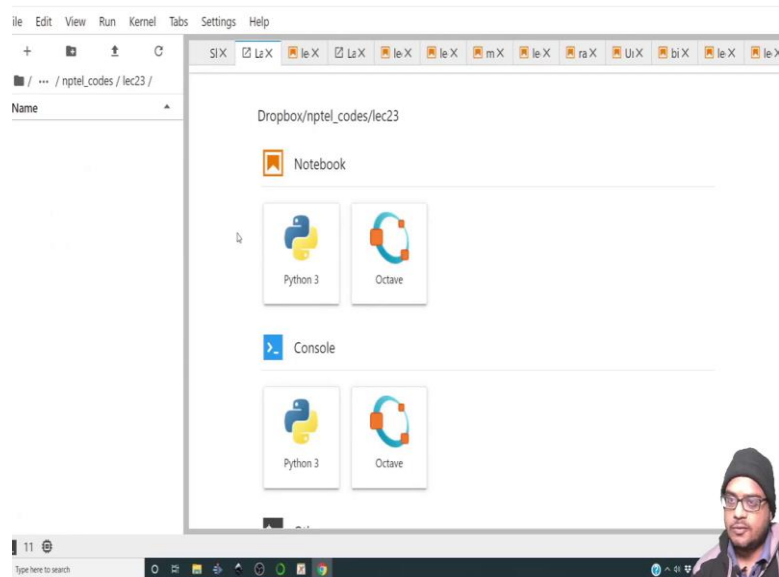
Then, we call the function curve fit which is inside scipy.optimize. We pass the function handle which will do the function evaluations. We will pass the data that is the x values and y values. The output is the set of parameters in the fitting function. The second output we do not worry about, alright.

Once we have that we can use the same function, but now we pass the output of the curve fit function as the two parameters. So, over here I am assigning a as the output 0 of

the output of the curve fit function, while I am assigning b as the output of the curve fit. This the second value of the output.

So, once I do that I plot it and I give it a label, and I also plot the raw data and after a lot of iterations or when you move ahead in time significantly, you will see that the distribution of locations does satisfy the normal distribution. There is something which I said I would discuss in the last class, but then I forgot about it. But anyway, it is not that difficult. It is there in the HTML that I have uploaded on my website and you can download it from there. The description is also there.

(Refer Slide Time: 03:50)



So, now let us move on to 2-dimensional random walks. So, let me create a new file, alright. So, what are we trying to do? So, in 1-dimensional random walk we had a random walk in which we had a walker or a particle and it would choose the step that it would take on the line, either it would sample from a step size distribution. So, $p(\Delta x)$ versus Δx , so it would sample from this distribution and then decide whether it has to go over here or here or how much it has to go over here.

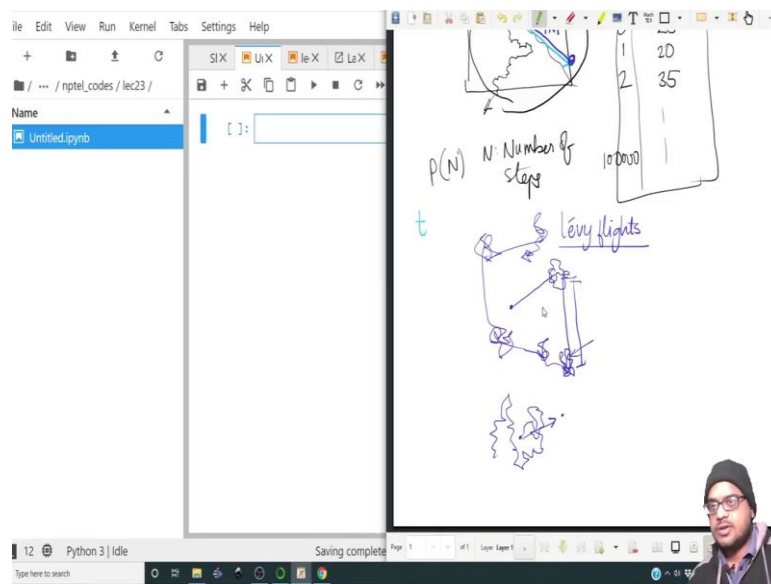
So, now, when we extend this idea to 2-dimensions, we obviously, are dealing with a 2-dimensional plane rather than a line and we can pose certain questions. For example, what are the number of steps taken before a given random walk exits a certain diameter. So, if I define this as the crossing boundary a Lakshman Rekha if you like. So, after how

many steps does the particle exit and I am interested to find out the total or the distribution of steps.

So, for example, I choose particle number 0 it maybe took 25 steps, particle 1 maybe it took less steps maybe it went straight ahead particle 2 maybe it take it took more steps. So, once I have say 10000 particles, I have a bunch of total steps the particle has taken, then I can find out the histogram of this to find out the pdf of N that is the number of steps.

Alternately, I could also ask the question after; so, here we are asking how many steps it took or we could equivalently ask that at a fixed time t what is the distribution of the locations that is the radial distance from the origin; so, if this is the origin at a given time t what is the distribution of locations of the particle. So, I am interested in the distance rather than the coordinates. So, these are some of the questions which are quite important from various physically naturally occurring phenomena like foraging.

(Refer Slide Time: 06:32)



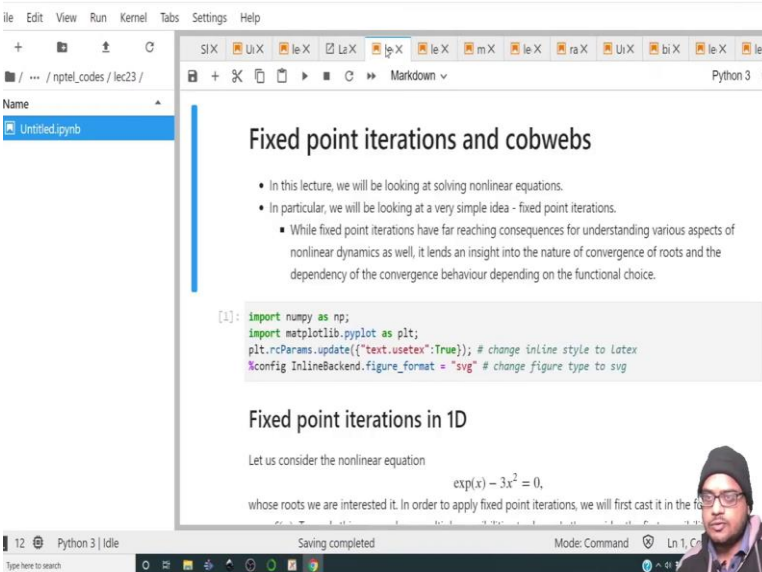
So, if you have heard of lions, and say you are in the savannah and you are hunting for food. So, what do predators do? They move a large distance, then they look around, they look around. Then, if they find something well and good, then they know that the area is depleted in resource and they will make a nice jump again and then they will again look around.

So, there are these large jumps and there are these small foraging, ok. So, this foraging behaviour does appear to have such a pattern. And as you will see such kinds of patterns are common for what are called as levy flights, ok.

But for example, if you are trying to figure out how an insect on the surface of a pond is moving you track the insect and you will see that it has all sorts of behaviour going on. But then what dictates that behaviour or what governs that motion of the insect? Is it the presence of some other insect? How does the motion get altered in the presence of other insects? How long how many steps?

Does it take for the insect to reach? For example, say a source of food. So, these are some of the questions and depending on the problem at hand. It is usually how you would define the random walk. There is no universality to how you would go ahead and define the random walk, ok.

(Refer Slide Time: 07:57)



The screenshot shows a Jupyter Notebook window with a slide titled "Fixed point iterations and cobwebs". The slide content includes:

- In this lecture, we will be looking at solving nonlinear equations.
- In particular, we will be looking at a very simple idea - fixed point iterations.
 - While fixed point iterations have far reaching consequences for understanding various aspects of nonlinear dynamics as well, it lends an insight into the nature of convergence of roots and the dependency of the convergence behaviour depending on the functional choice.

```
[1]: import numpy as np;
import matplotlib.pyplot as plt;
plt.rcParams.update({"text.usetex":True}); # change inline style to latex
%config InlineBackend.figure_format = "svg" # change figure type to svg
```

Fixed point iterations in 1D

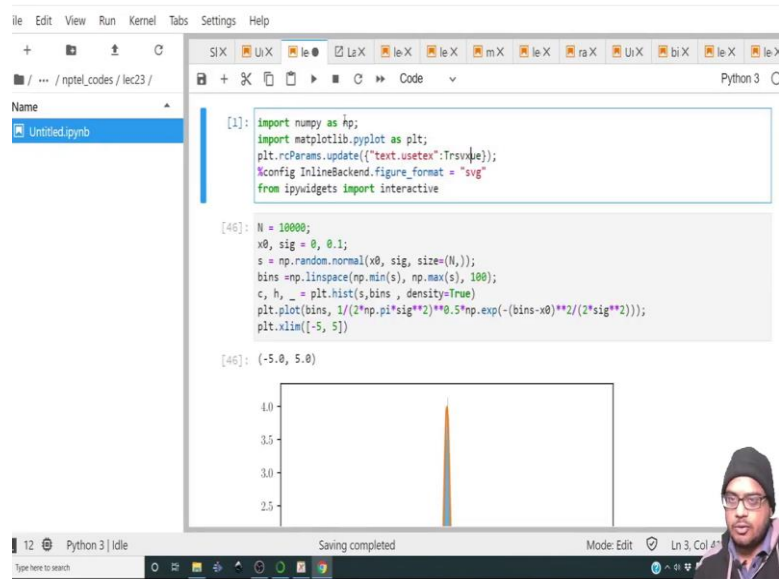
Let us consider the nonlinear equation

$$\exp(x) - 3x^2 = 0,$$

whose roots we are interested in. In order to apply fixed point iterations, we will first cast it in the form

The bottom of the screenshot shows a terminal window with the text "Saving completed" and "Mode: Command". A small video feed of a person is visible in the bottom right corner of the Jupyter interface.

(Refer Slide Time: 08:01)



The screenshot shows a Jupyter Notebook interface with a code cell and a plot. The code cell contains the following Python code:

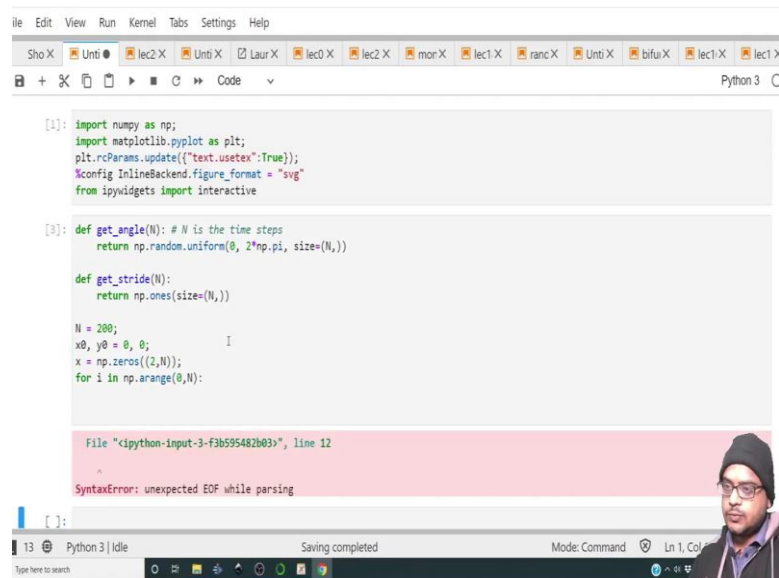
```
[1]: import numpy as np;
import matplotlib.pyplot as plt;
plt.rcParams.update({"text.usetex": True});
%config InlineBackend.figure_format = "svg"
from ipywidgets import interactive

[46]: N = 10000;
x0, sig = 0, 0.1;
s = np.random.normal(x0, sig, size=(N,));
bins = np.linspace(np.min(s), np.max(s), 100);
c, h, _ = plt.hist(s, bins, density=True);
plt.plot(bins, 1/(2*np.pi*sig**2)**0.5*np.exp(-(bins-x0)**2/(2*sig**2)));
plt.xlim([-5, 5])
```

The plot below the code shows a histogram of the generated data (blue bars) and a fitted normal distribution curve (orange line). The x-axis ranges from -5 to 5, and the y-axis ranges from 2.5 to 4.0. A small inset image of a person is visible in the bottom right corner of the notebook window.

So, with this background let us begin. So, we will need the opening bit of code and I will copy it from pardoning. I will copy it from one of the previous programs, alright.

(Refer Slide Time: 08:10)



The screenshot shows a Jupyter Notebook interface with a code cell and a syntax error message. The 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"
from ipywidgets import interactive

[3]: def get_angle(N): # N is the time steps
return np.random.uniform(0, 2*np.pi, size=(N,))

def get_stride(N):
return np.ones(size=(N,))

N = 200;
x0, y0 = 0, 0;
x = np.zeros((2,N));
for i in np.arange(0,N):
```

The error message below the code is: "File <ipython-input-3-f3b595482b03>, line 12: SyntaxError: unexpected EOF while parsing". A small inset image of a person is visible in the bottom right corner of the notebook window.

So, now, how do we go about this? Let us look at the simplest random walk what is called as the Pearson walk.

(Refer Slide Time: 08:23)

Python code in the notebook:

```
[1]: import numpy as np;
import matplotlib.pyplot
plt.rcParams.update({'te
%config InlineBackend.fi
from ipywidgets import
```

Handwritten notes:

Pearson walk

$\rightarrow r = l \leftarrow \text{fixed}$

$\theta = [0, 2\pi] \leftarrow \text{uniform}$

$p(r) = \delta(r-l)$

$c(r) = \int_0^r p(r') dr'$

$= \int_0^r \delta(r'-l) dr'$

$r \in (0, r)$

So, Pearson walk is you start from the origin, and you have a sort of uniform, not a uniform, but you have a fixed distance that you would take. So, each time a step is taken the length of the step r is fixed it is equal to l , but the direction can vary between 0 and 2π . So, the direction is uniformly distributed, but the step size is fixed. So, in terms of writing down the pdf of this if it is 1-dimension, so you would say $p(r) = \delta(r-l)$, right.

So, this is the delta function. And accordingly you can find the cumulative distribution of this function as well. So, the cumulative, so $c(r) = \int_0^r p(r') dr'$ and so this will become $\int_0^r \delta(r'-l) dr'$.

(Refer Slide Time: 09:57)

Python 3 | Idle

```
[1]: import numpy as np;
import matplotlib.pyplot
plt.rcParams.update({'te
%config InlineBackend.fi
from ipywidgets import I
```

Handwritten notes:

Pearson walk

$\int_{l_1}^{l_2} f(x') \delta(x'-a) dx' = f(a)$ if $a \in [l_1, l_2]$ else 0

$\delta(x-l)$

$p(x) = \delta(x-l)$

$C(r) = \int_0^r p(x') dx' = \int_0^r \delta(x'-l) dx'$

$l \in [0, r] \rightarrow 1$

$l \notin [0, r] \rightarrow 0$

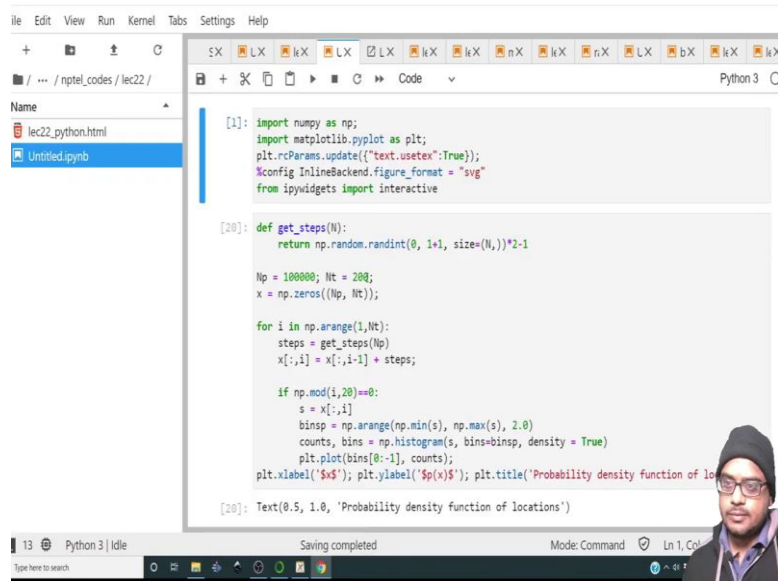
So, now, if the domain is 0 to r, right, if the domain of integration, so if l lies in 0 to r then this integral is going to be 1; if l it does not belong it is outside 0 to r, then the integration will be 0. So, the cumulative distribution for such function it will be 0 and then suddenly 1. So, now if you look back and on how to sample a certain random number from a given probability density function, you would uniformly you would draw uniform number between 0 and 1, project it on this curve and then project it back.

Because it is a vertical line all your samples will fall at $r = 1$ and obviously, these are properties of the Dirac function, Dirac delta function, ok. So, I have used the important property that $\int_{l_1}^{l_2} f(x') \delta(x'-a) dx'$, lower limit and upper limit or l_1 to u_1 , this is going to be $f(a)$, if a lies between l_1 and l_2 . If a does not lie in the domain then it will be 0. So, these are just some properties of delta function.

So, now, this is, ok. I mean this is quite easy to implement. You can imagine, you have a fixed increment r, so you are starting over here. You know you have a fixed increment, so your trajectory will lie somewhere on the circle at the next time instant, but as to which angle it will take it depends on the sampled value between 0 and 2π ok. So, if it is 0 it will move over here, if its π it will move over here, if it is $\pi/4$ it will go over here and so on. And after you raise this point you will again do this.

So, if it is after reaching this point if it is $\pi/2$ it will go something like this. So, let us try to implement this. So, first things first, we can borrow certain things from the previous lecture.

(Refer Slide Time: 12:22)



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

[20]: def get_steps(N):
return np.random.randint(0, 1+1, size=(N,))*2-1

Np = 100000; Nt = 200;
x = np.zeros((Np, Nt));

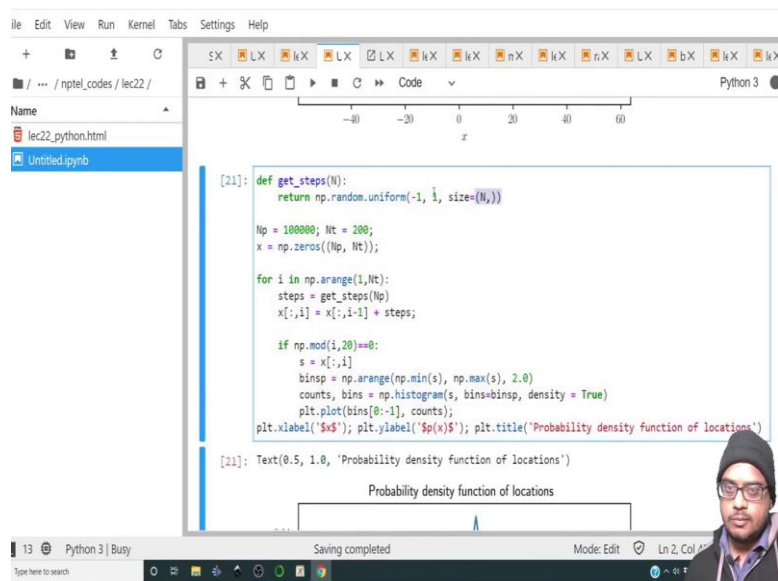
for i in np.arange(1,Nt):
steps = get_steps(Np)
x[:,i] = x[:,i-1] + steps;

if np.mod(i,20)==0:
s = x[:,i]
binsp = np.arange(np.min(s), np.max(s), 2.0)
counts, bins = np.histogram(s, bins=binsp, density = True)
plt.plot(bins[0:-1], counts);

plt.xlabel('$x$'); plt.ylabel('$p(x)$'); plt.title('Probability density function of locations')

[20]: Text(0.5, 1.0, 'Probability density function of locations')
```

(Refer Slide Time: 12:25)



```
[21]: def get_steps(N):
return np.random.uniform(-1, 1, size=(N,))

Np = 100000; Nt = 200;
x = np.zeros((Np, Nt));

for i in np.arange(1,Nt):
steps = get_steps(Np)
x[:,i] = x[:,i-1] + steps;

if np.mod(i,20)==0:
s = x[:,i]
binsp = np.arange(np.min(s), np.max(s), 2.0)
counts, bins = np.histogram(s, bins=binsp, density = True)
plt.plot(bins[0:-1], counts);

plt.xlabel('$x$'); plt.ylabel('$p(x)$'); plt.title('Probability density function of locations')

[21]: Text(0.5, 1.0, 'Probability density function of locations')
```

So, let me open that file as well. So, we will use the uniform distribution code. So, here we will say get angle and it will go from 0 to 2π . And so, what is the work flow that we are looking over here?

(Refer Slide Time: 12:53)

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

```
[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

[2]: def get_angle(N):
return np.random.uniform(0, 2*np.pi, size=(N,))
```

The whiteboard contains a plot of a step function with height 1 and width l . To the right of the plot is the integral equation: $\int_0^l \delta(x-l) dx = 1$. Below the plot is a table of initial conditions:

t	0	Δt
x	0	
y	0	
l	θ	

Below the table are the formulas: $\Delta x = l \cos \theta$ and $\Delta y = l \sin \theta$.

So, the work flow will be we initialize the vector that is x_0 , so this is at $t = 0$. So, this is time, this is x , this is y , alright. Then, we want to find an increment. So, we will we know that in this case the radius the step is fixed, so it is 1, we will sample theta and we will sample some θ , ok. So, θ belongs to a uniform distribution between 0 and 2π .

(Refer Slide Time: 13:36)

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

```
[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

[2]: def get_angle(N): # N is the time steps
return np.random.uniform(0, 2*np.pi, size=(N,))

def get_stride(N):
return np.ones(size=(N,))

N = 200;
x0, y0 = 0, 0;
for i in np.arange(0, N):
```

The whiteboard contains the same step function plot and integral equation as before. Below it is a table showing the sequence of time steps and displacement increments:

t	0	1	2	3	...	N
x	0	Δx_1	Δx_2	Δx_3	...	Δx_N
y	0	Δy_1	Δy_2	Δy_3	...	Δy_N

Below the table are the formulas for cumulative displacement: $\Delta x = l \cos \theta$ and $\Delta y = l \sin \theta$. To the right, the cumulative sums are given as: $0 + \Delta x_1 + \Delta x_2 + \Delta x_3$ and $0 + \Delta y_1 + \Delta y_2 + \Delta y_3$.

Now, based on this the $\Delta x = l \cos \theta$, $\Delta y = l \sin \theta$. So, at this; so at the first step, so instead of calling it Δt delta t I will just call it 1, it will be some $\Delta x_1, \Delta y_1$. Then at time 2, it will

be Δx_2 , Δy_2 , and so on for n , it will be Δx_n , Δy_n . So, at the end of this the trajectory will be the cumulative sum up until that point. So, suppose I want to find out the trajectory up until 3, so it will be simply $0 + \Delta x_1 + \Delta x_2 + \Delta x_3$.

This will be the x coordinate, the y coordinate will be $0 + \Delta y_1 + \Delta y_2 + \Delta y_3$. So, if I sort of plot the cumulative sum all the way, I should be able to find out the trajectory. Well, let us find out. So, over here what is N ? It is not the particle number. It is the time instances that we want, ok.

Unlike the previous problem where N was the number of particles that we were initializing here we are fixing our attention on one particle finding out all the mutually independent steps that it will take and then we make a cumulative sum, alright.

So, N over here is representing the time steps, ok. So, let me just put a comment, alright; `def get_stride(N)`, it should simply return `np.ones((N,))`, right. So, it has to just return a bunch of ones, alright. So, these are the two functions that we want well. Now, I can focus on one particle.

So, I will say how many time steps do you want. So, let us say we want 200 time steps, alright. So, then what do we do? For `i in np.arange(0,N)`, now we have to define the initial value or the initial location. So, $x_0, y_0 = 0, 0$, that is fine.

Now, what we will do is we will create the increment array. So, if we have 200 steps what will be; so, let us make the increment array to look something like this. Let this be the increment array, alright. So, the increment array we will say as just let us call `itx = np.zeros((2,N))`. So, now, it has to be of 2 rows and N columns, alright, so, yeah. So, we have created `x` which has 2 rows and N column. So, and obviously, we are initializing with 0, so x_0, y_0 it does not make sense.

(Refer Slide Time: 16:47)

```
def get_angle(N):  
    return np.random.uniform(0, 2*np.pi, size=(N,))  
  
def get_stride(N):  
    return np.ones(size=(N,))  
  
N = 200;  
x0, y0 = 0, 0;  
x = np.zeros((2,N));  
  
for i in np.arange(1,N):  
    l = get_stride(1);  
    th = get_angle(1);  
    deltax = l*np.cos(th); deltax = l*np.sin(th);  
    x[0,i] = deltax  
    x[1,i] = deltax
```

TypeError
Traceback (most recent call last)
10
11 for i in np.arange(1,N):
----> 12 l = get_stride(1);
13 th = get_angle(1);
14 deltax = l*np.cos(th); deltax = l*np.sin(th);

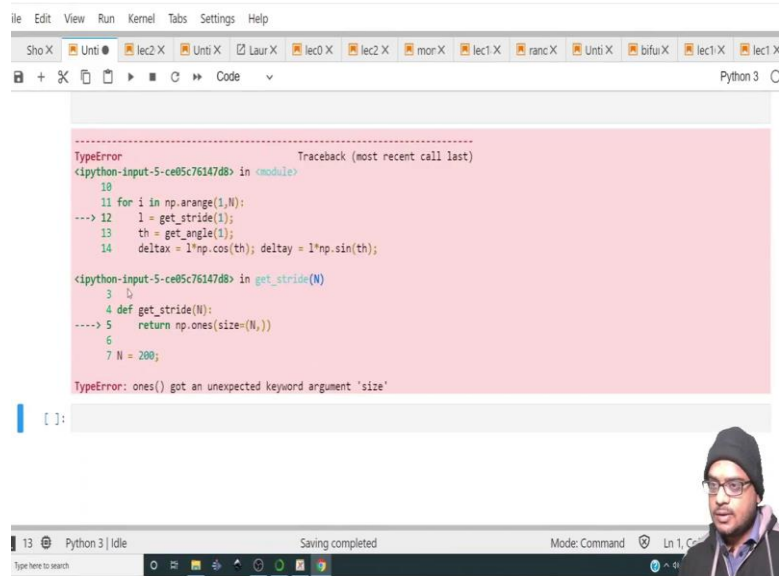
So, now we will do for i in $\text{np.arange}(1,N)$, because the first point is already the origin, so we do not need to worry about that. Now, what we will do is we will write down the logic. So, x , so for the i th column the first let me just do this. So, this will be Δx , so this has to be 0 and this has to be Δy .

Basically if I am over here the 0th row, i th column, this will be that particular increment and this will be this particular increment. So, the increment in y will have row number of 1, whereas, the increment in x will have a row number of 0. Now, $\text{deltax} = l * \text{np.cos}(th)$, and $\text{deltay} = l * \text{np.sin}(th)$. But now what is l and what is θ ?

So, l will be; in fact, we do not even need to run this in a loop we do not even need to run this in a loop, we can simply pass N and obtain all the steps that we need, fine, and then we can augment it with 0s, ok. So, well because I have written it so far, let me just, so we will just call one angle and one stride at a given time l equal to get stride one element and θ equal to get angle one element.

So, I am not vectorizing the code. I am asking it for each time step give me one stride and one length. And obviously, the stride will be one and the angle will also be one. So, I am not vectorizing anything.

(Refer Slide Time: 19:09)



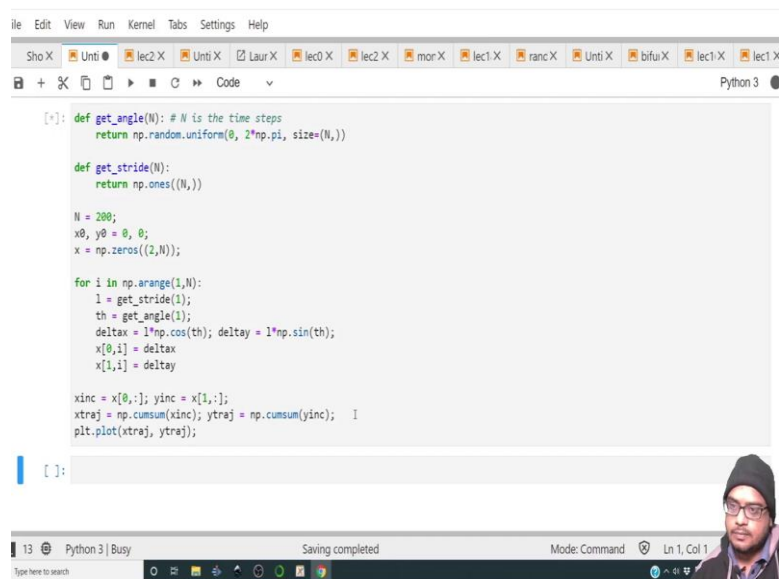
```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-5-ce05c76147d8> in <module>
    10
    11 for i in np.arange(1,N):
--> 12     l = get_stride(1);
    13     th = get_angle(1);
    14     deltax = l*np.cos(th); deltax = l*np.sin(th);

<ipython-input-5-ce05c76147d8> in get_stride(N)
     3     l
----> 4     def get_stride(N):
     5         return np.ones(size=(N,))
     6
     7     N = 200;

TypeError: ones() got an unexpected keyword argument 'size'
```

So, let me run this, and there is a error, sorry, ok.

(Refer Slide Time: 19:18)



```
[*]: def get_angle(N): # N is the time steps
      return np.random.uniform(0, 2*np.pi, size=(N,))

      def get_stride(N):
          return np.ones((N,))

      N = 200;
      x0, y0 = 0, 0;
      x = np.zeros((2,N));

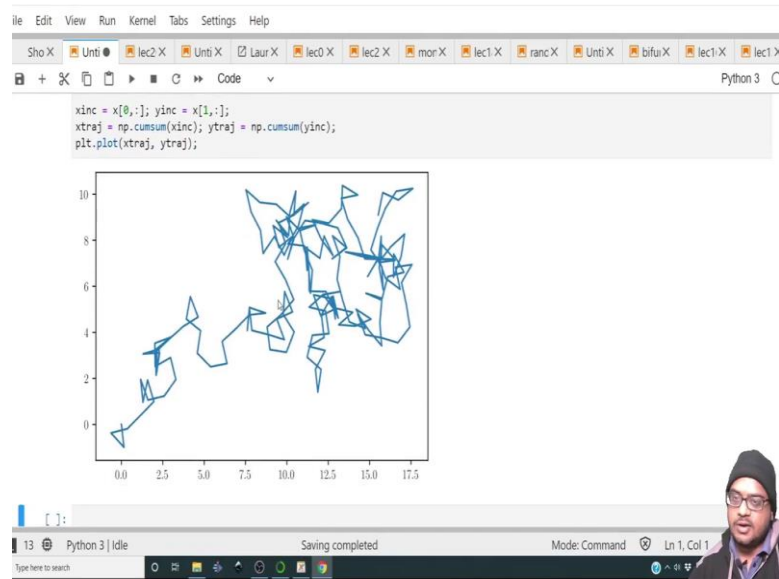
      for i in np.arange(1,N):
          l = get_stride(1);
          th = get_angle(1);
          deltax = l*np.cos(th); deltax = l*np.sin(th);
          x[0,i] = deltax
          x[1,i] = deltax

      xinc = x[0,:]; yinc = x[1,:];
      xtraj = np.cumsum(xinc); ytraj = np.cumsum(yinc);
      plt.plot(xtraj, ytraj);
```

So, now, let me show you. So, after this we have generated a series of x and y increments. So, let me do the following. Let me do a cumulative sum. So, xtraj or xinc = x[0,:] and yinc = x[1,:], alright. Now, we must do a cumulative sum.

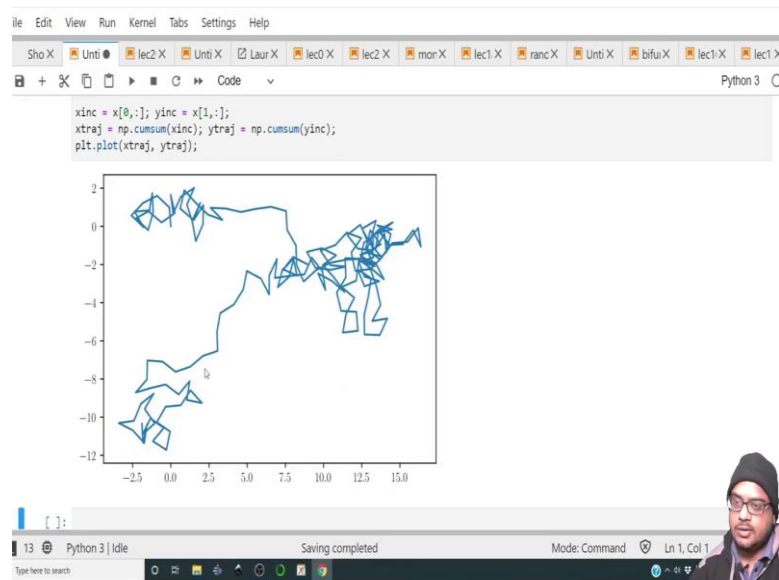
So, xtraj = np.cumsum(xinc) and ytraj = np.cumsum(yinc), alright, so far so good. Now, we will plot x and y and that should give us the trajectory for the single particle. So, plt.plot(xtraj, ytraj), ok.

(Refer Slide Time: 20:31)



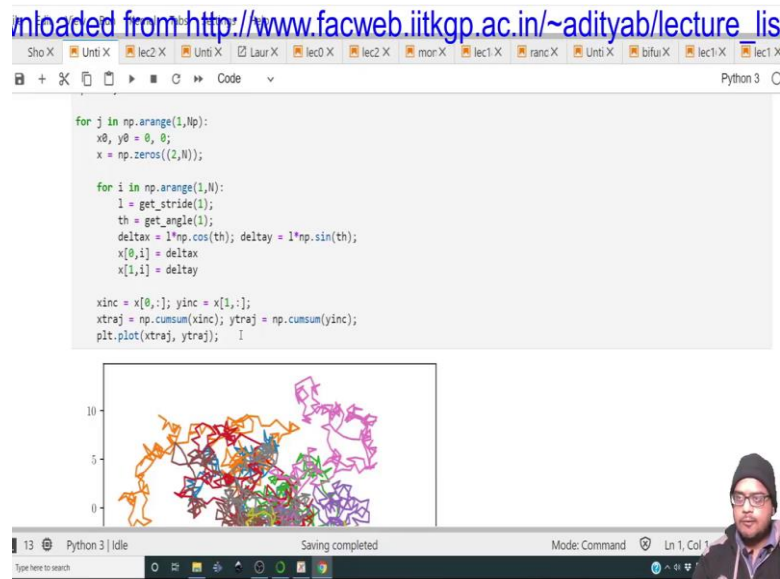
So, this is how the trajectory looks like. It started over here and it is doing this random walk, ok. So, if I run this again I will get a new random walk. So, let me because it is going to sample a new variable again, alright. So, let me do that, great. So, it is going in the other direction.

(Refer Slide Time: 20:55)

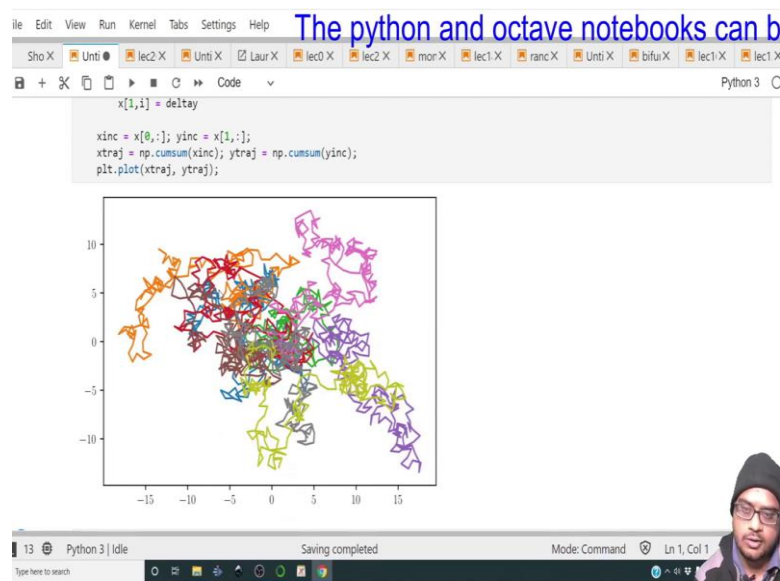


So, let us now wrap this entire program, so that we can run it for a few more particles.

(Refer Slide Time: 21:07)



(Refer Slide Time: 21:43)

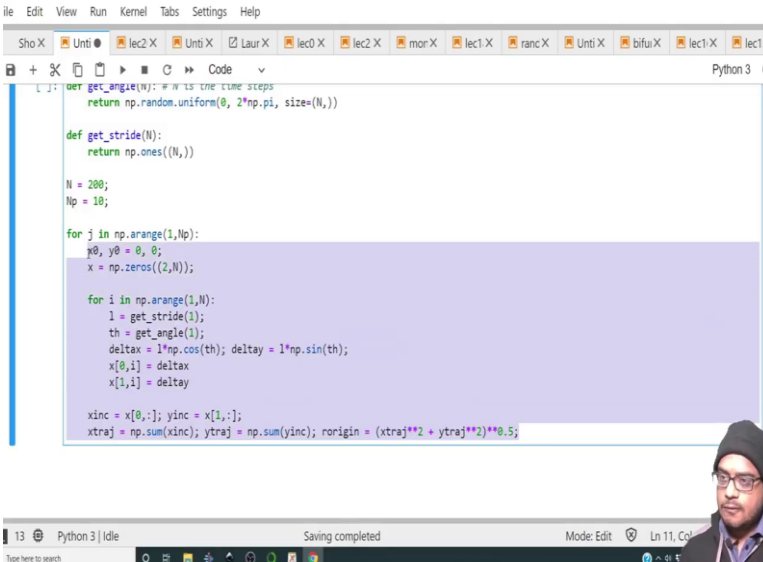


So, let me copy this. Let me go over here and let me, so let me do it for more particles. So, let me write $N_p = 10$ and over here I will write for j in $\text{np.arange}(1, N_p)$. So, I will do the whole initialization for each particle and eventually I will do the plot. So, now, let me run this and see what happens. This has to be comma.

So, great, we have 10 trajectories using the Pearson walk. But now I have this data, but now I really what I really want to know is I do not want to know the trajectory. I just want to know what radial location the particles are at the end of 200 steps. So, how do I

go about doing that? I do not want to find the trajectories. I do not want all these. So, that is quite easy.

(Refer Slide Time: 22:20)



```
def get_angle(N): # With time steps
    return np.random.uniform(0, 2*np.pi, size=(N,))

def get_stride(N):
    return np.ones((N,))

N = 200;
Np = 10;

for j in np.arange(1, Np):
    y0, y0 = 0, 0;
    x = np.zeros((2, N));

    for i in np.arange(1, N):
        l = get_stride(1);
        th = get_angle(1);
        deltax = l*np.cos(th); deltay = l*np.sin(th);
        x[0, i] = deltax
        x[1, i] = deltay

    xinc = x[0, :]; yinc = x[1, :];
    xtraj = np.sum(xinc); ytraj = np.sum(yinc); rorigin = (xtraj**2 + ytraj**2)**0.5;
```

So in fact, let me preserve this bit of program. Let me go over here. And so, we still have this. We still have number of particles and what we will do is we will wrap all of this. In fact, we do not need a cumsum because cumsum is required to find out the entire trajectory, we just need a sum and the sum will give me the final point, ok. The sum will give me the final point.

Once I have this, I will find out the rorigin = $(xtraj**2 + ytraj**2)**0.5$ it is the distance from the origin and I do not need to plot it. So, what I have over here is a need function which will give me the final distance from the origin. So, I will cut this. I do not need all that. I will define a function.

(Refer Slide Time: 23:22)


```
return np.ones((N,))

def get_final_loc(N):
    x0, y0 = 0, 0;
    x = np.zeros((2,N));

    for i in np.arange(1,N):
        l = get_stride(1);
        th = get_angle(1);
        deltax = l*np.cos(th); deltay = l*np.sin(th);
        x[0,i] = deltax
        x[1,i] = deltay

    xinc = x[0,:]; yinc = x[1,:];
    xtraj = np.sum(xinc); ytraj = np.sum(yinc); rorigin = (xtraj**2 + ytraj**2)**0.5;
    return rorigin

N = 200;
np = 10;
r = np.zeros((N,))

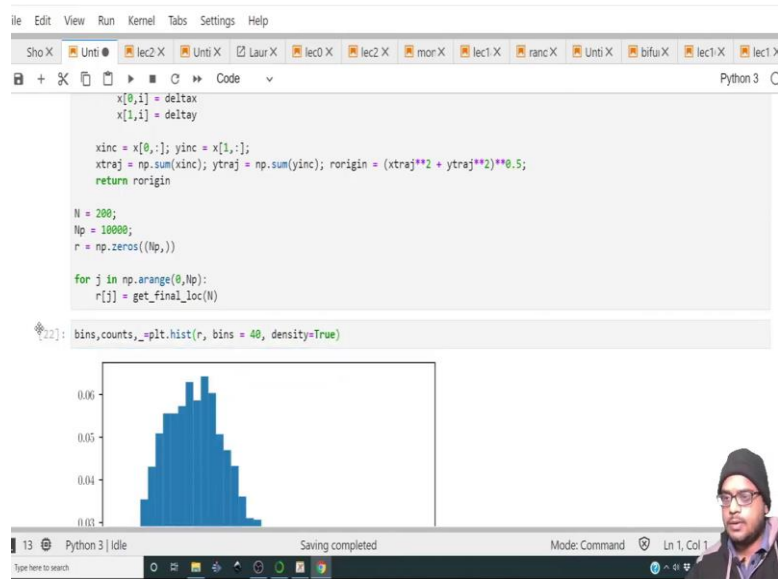
for j in np.arange(1,np):
    r[j] = get_final_loc(N)
```

So, I will define get final location and I will pass N, that is the total number of time steps. So, over here I have to do the same declarations. And eventually I must return rorigin, ok. So, it is abstracting that bit of code that you have to find the trajectory blah blah blah, but no I do not want to know all that. I just want the final distance. So, I will simply go over here and I will simply call the final get final location.

So, I will say get final location loc and I will pass N that is for a large number of particles np, I will pass that I want to know the final location for N number of steps that is 200 number of steps, I will assign it to an array r, alright and I will say r that is the final location is equal to np dot 0s and it will be of size, yeah N rows and it is just N rows it is a vector, ok.

(Refer Slide Time: 24:57)

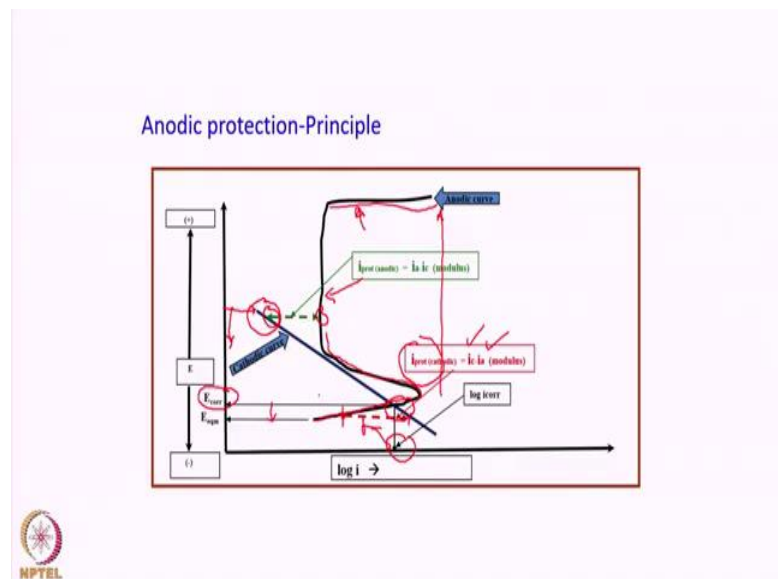
(Refer Slide Time: 25:47)



So, let me make it 10000. Let me make this as 200, so 10000 particles I should be able to then it will take a while to run because the code is not vectorized. And in fact, I will ask you to vectorize it as a part of your home task. It is not going to be an assignment, but you should go ahead and try to do that I will tell you what the logic will be.

Well, the program has run by now. Let me just it is a big array, so let me just plot it. So, `bins,counts,_=plt.hist(r, bins = 40, density=True)`. So, it gives me this kind of a distribution, ok.

(Refer Slide Time: 26:37)



If I want a better fit to this I will probably use more number of particles and it will show this kind of distribution. And it will be more illustrative when we do this in a log scale; on the x axis if I take it in a log it will be it will show some of some kind of a power law, this decaying tail may show some kind of a power law.

And that is for you to explore, once you start learning about random walks and the distributions that you get from multi-dimensional random walks. But this gives a very clear cut overview of what is going on, right and, ok. So, how do you go about vectorizing this? It is quite easy. Well, we have created the get angle and get stride function. So, and I will just give an outline. I am not going to implement this.

(Refer Slide Time: 27:33)

The screenshot shows a Python IDE with the following code:

```
[20]: def get_angle(N): # N is the time steps
      return np.random.uniform(0, 2*np.pi, size=(N,))

      def get_stride(N):
        return np.ones((N,))

      def get_final_loc(N):
        x0, y0 = 0, 0;
        x = np.zeros((2,N));

        for i in np.arange(1,N):
          l = get_stride(i);
          th = get_angle(i);
          deltax = l*np.cos(th); deltay = l*np.sin(th);
          x[0,i] = deltax
          x[1,i] = deltay

        xinc = x[0,:]; yinc = x[1,:];
        xtraj = np.sum(xinc); ytraj = np.sum(yinc); rorigin =
        return rorigin

      N = 200;
      Np = 10000;
```

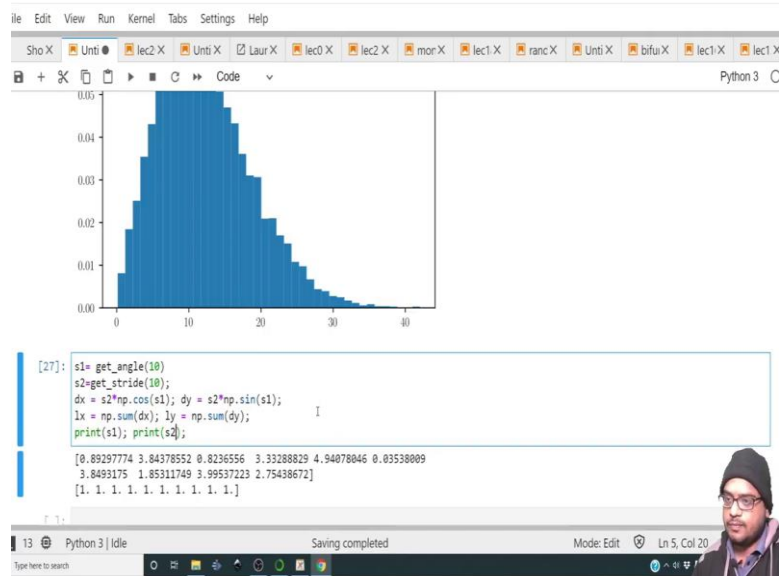
Hand-drawn diagram illustrating a 2D random walk:

- $N \rightarrow$ (Number of steps)
- θ : (Angle) shown as a horizontal line with arrows and 'x' marks.
- Δx : (Horizontal increment) shown as a horizontal line with arrows and 'x' marks.
- Δy : (Vertical increment) shown as a vertical line with arrows and 'x' marks.

So, because you know the number of steps, so then you will ask for get angle for N values, so you will get a bunch of angles ranging from 0 to π , ok. You have a bunch of angles, you have a bunch of lengths. And then using the formula $\Delta x = l \cos \theta$ $\Delta y = l \sin \theta$ you will have a bunch of increments, ok. So, then you augment this array with a 0 and then you first simply find the sum.

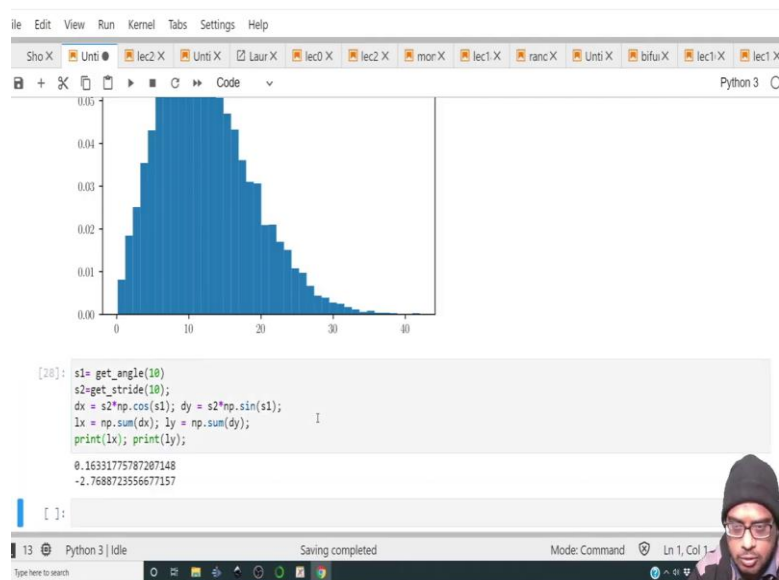
So, in case you are just trying to find the sum you do not even need to augment the error with 0. You just find the sum of these matrices, ok, and that is it. And let me just for completeness show you how I will do that.

(Refer Slide Time: 28:22)



So, let me say $s1 = \text{get_angle}(10)$. Let me just do it for 10, and $s2 = \text{get_stride}(10)$, right. So, let me print those. So, these are the angles and these are the strides. Then, you will say $dx = s2 * \text{np.cos}(s1)$, $dy = s2 * \text{np.sin}(s1)$ and so, then you will have dx and dy and then you will say $lx = \text{np.sum}(dx)$ and $ly = \text{np.sum}(dy)$ and that is it. I mean with the help of this you will have the final location, ok.

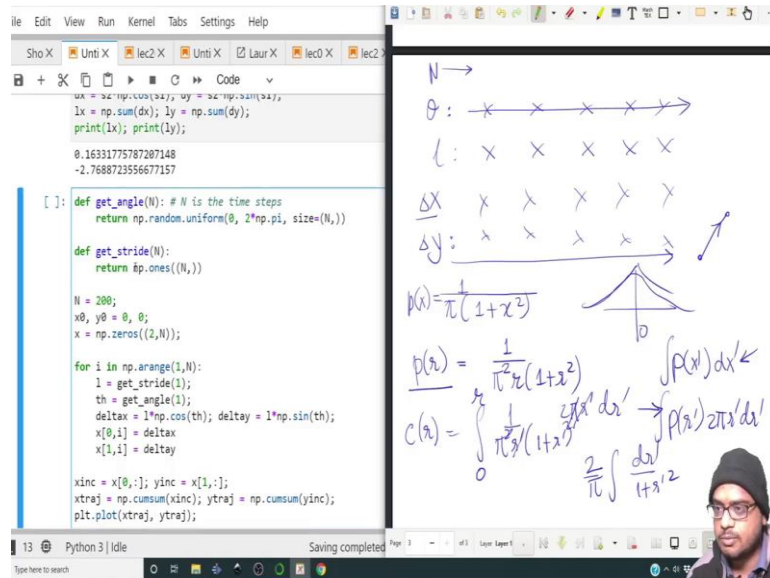
(Refer Slide Time: 29:33)



So, you do not need to run all those things in a loop, ok. And in fact, I will do the modification once we go to the levy walk the Cauchy distribution. But for now it gives a very clear overview of how you go about this. So, just you do not need you can avoid this loop all together.

Well, I will let you do that well. Now, that we have looked at a bounded distribution, we can turn our attention to a levy walk. Well, let me reuse this program. So, now, instead of having a unit length as the stride, let us do it for a Cauchy distribution.

(Refer Slide Time: 30:24)



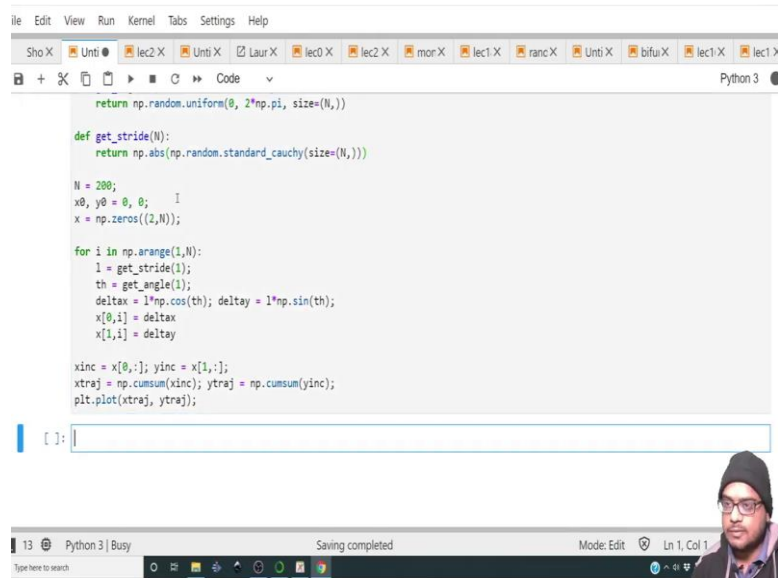
So, Cauchy distribution looks something like this. It is usually $\frac{1}{\pi(1+x^2)}$ and this is for 1-dimension. So, this is the $p(x)$ for a Cauchy distribution and it is a symmetric distribution about the origin, ok, about the origin it is a symmetric distribution. But now I am interested in the absolute value. So, we are more bothered with sampling the radial stride in a random walk using a Cauchy distribution. So, for that the distribution is actually $\frac{1}{\pi^2 r(1+r^2)}$.

And the reason is because once you try to find the cumulative of a step size distribution in 2-dimension it will not be simply $\int p(x') dx'$, but rather it will be $\int p(r') 2\pi r' dr'$, if everything is symmetric because it needs to find the area. So, this is the area in 1D, this is the area in 2D. So, then what is the cumulative of r ? It will be $\int_0^r \frac{1}{\pi^2 r'(1+r'^2)} 2\pi r' dr'$. So, this π cancels out.

So, it is r going from 0 to r' going from 0 to r . So, this becomes $\frac{2}{\pi} \int \frac{dr'}{1+r'^2}$. So, this is the same as finding out or a sampling a variable from the Cauchy distribution in 1-

dimension, ok. So, that gives us something to work around. So, instead of having `np.ones`, we can now have `np.random.Cauchy`.

(Refer Slide Time: 32:24)



```
return np.random.uniform(0, 2*np.pi, size=(N,))

def get_stride(N):
    return np.abs(np.random.standard_cauchy(size=(N,)))

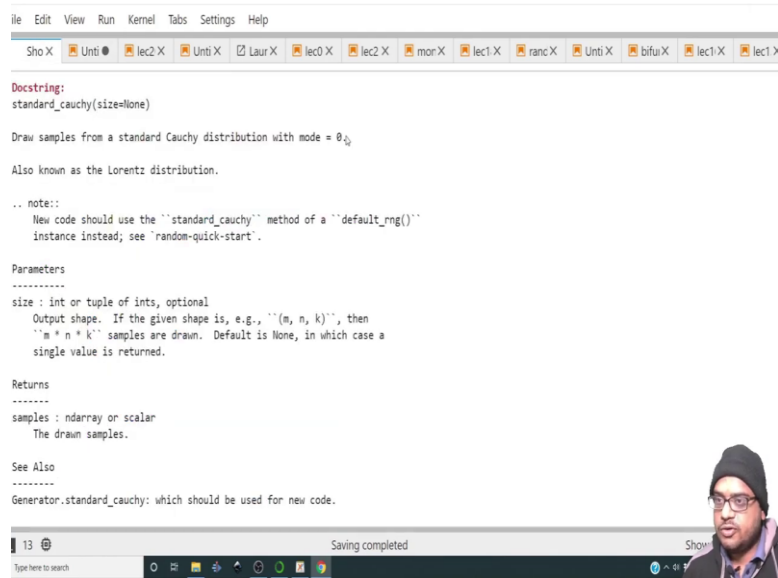
N = 200;
x0, y0 = 0, 0; i
x = np.zeros((2,N));

for i in np.arange(1,N):
    l = get_stride(i);
    th = get_angle(i);
    deltax = l*np.cos(th); deltay = l*np.sin(th);
    x[0,i] = deltax
    x[1,i] = deltay

xinc = x[0,:]; yinc = x[1,:];
xtraj = np.cumsum(xinc); ytraj = np.cumsum(yinc);
plt.plot(xtraj, ytraj);
```

So, let us look at what the syntax for standard Cauchy is.

(Refer Slide Time: 32:34)



```
Docstring:
standard_cauchy(size=None)

Draw samples from a standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

.. note::
    New code should use the ``standard_cauchy`` method of a ``default_rng()``
    instance instead; see ``random-quick-start``.

Parameters
-----
size : int or tuple of ints, optional
    Output shape. If the given shape is, e.g., ``(m, n, k)``, then
    ``m * n * k`` samples are drawn. Default is None, in which case a
    single value is returned.

Returns
-----
samples : ndarray or scalar
    The drawn samples.

See Also
-----
Generator.standard_cauchy: which should be used for new code.
```

(Refer Slide Time: 32:36)

```
file Edit View Run Kernel Tabs Settings Help
Sho X Unti X lec2 X Unti X Laur X lec0 X lec2 X mor X lec1 X ranc X Unti X bifuj X lec1 X lec1 X
size : int or tuple of ints, optional
Output shape. If the given shape is, e.g., ``(m, n, k)``, then
``m * n * k`` samples are drawn. Default is None, in which case a
single value is returned.

Returns
-----
samples : ndarray or scalar
The drawn samples.

See Also
-----
Generator.standard_cauchy: which should be used for new code.

Notes
-----
The probability density function for the full Cauchy distribution is
.. math:: P(x; x_0, \gamma) = \frac{1}{\pi} \frac{\gamma}{\gamma^2 + (x - x_0)^2}
and the Standard Cauchy distribution just sets :math:`x_0=0` and
:math:`\gamma=1`

The Cauchy distribution arises in the solution to the driven harmonic
oscillator problem, and also describes spectral line broadening. It
also describes the distribution of values at which a line tilted at
a random angle will cut the x axis.
```

So, mode 0, no problem, and this is the distribution, standard Cauchy is with $x_0 = 0$ and $\gamma = 1$, no problem. So, we need to just give the size, alright, to get. So, size=(N,) and let us save it. So, now, but wait; so, the Cauchy distribution will give you negative values as well. So, we need to x or return the absolute value of this, alright.

(Refer Slide Time: 33:18)

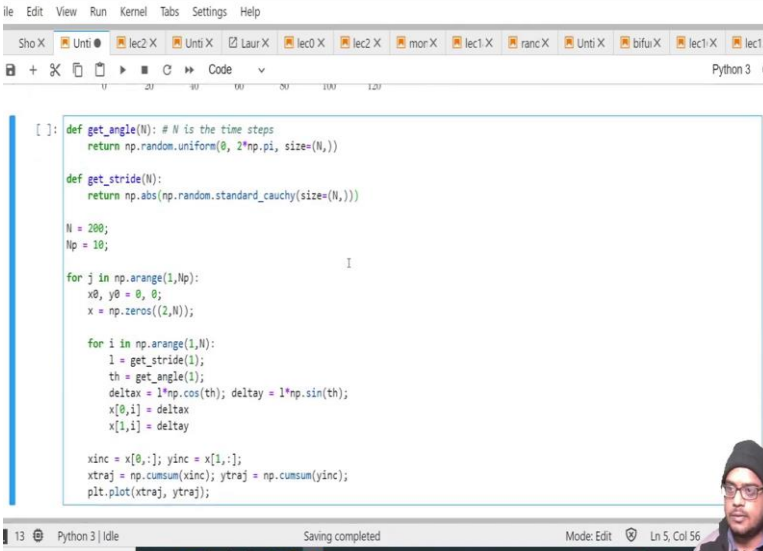
```
file Edit View Run Kernel Tabs Settings Help
Sho X Unti X lec2 X Unti X Laur X lec0 X lec2 X mor X lec1 X ranc X Unti X bifuj X lec1 X lec1 X
x[1,i] = deltay

xinc = x[0,:]; yinc = x[1,:];
xtraj = np.cumsum(xinc); ytraj = np.cumsum(yinc);
plt.plot(xtraj, ytraj);
```

So, we have returned the absolute value. Well, let us run this, let us see what happens. So, boom. Now, we have what is a levy walk. So, it starts over here, it makes huge

jumps, huge jumps then wanders around, his huge jump wanders around, huge jump wanders around, huge jump wanders around and things like that.

(Refer Slide Time: 34:01)



```
[ ]: def get_angle(N): # N is the time steps
      return np.random.uniform(0, 2*np.pi, size=(N,))

def get_stride(N):
    return np.abs(np.random.standard_cauchy(size=(N,)))

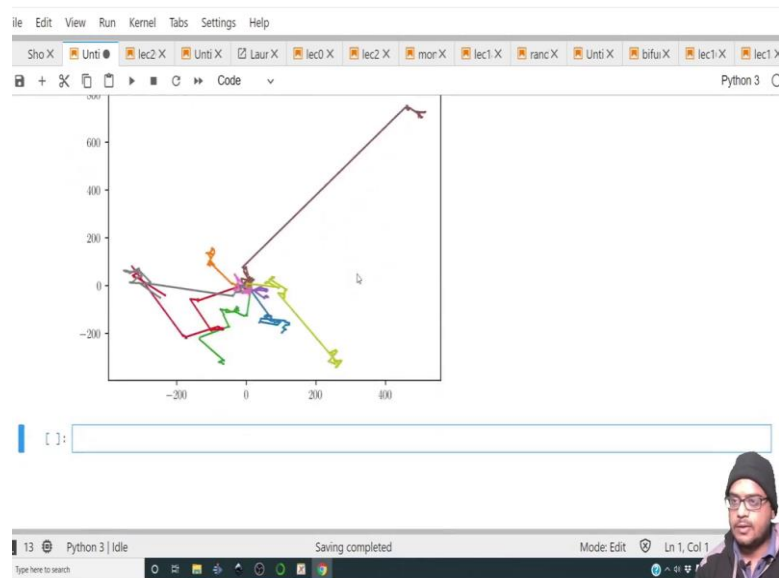
N = 200;
Np = 10;

for j in np.arange(1,Np):
    x0, y0 = 0, 0;
    x = np.zeros((2,N));

    for i in np.arange(1,N):
        l = get_stride(1);
        th = get_angle(1);
        deltax = l*np.cos(th); deltay = l*np.sin(th);
        x[0,i] = deltax
        x[1,i] = deltay

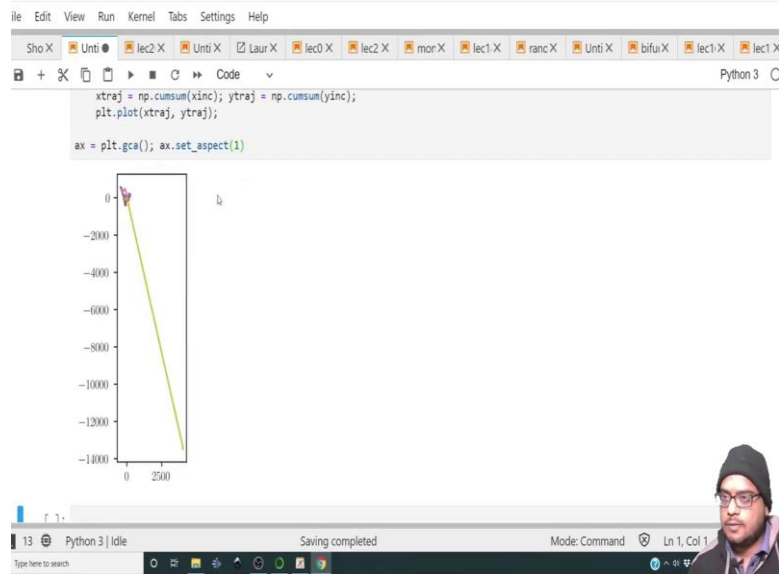
    xinc = x[0,:]; yinc = x[1,:];
    xtraj = np.cumsum(xinc); ytraj = np.cumsum(yinc);
    plt.plot(xtraj, ytraj);
```

(Refer Slide Time: 34:04)



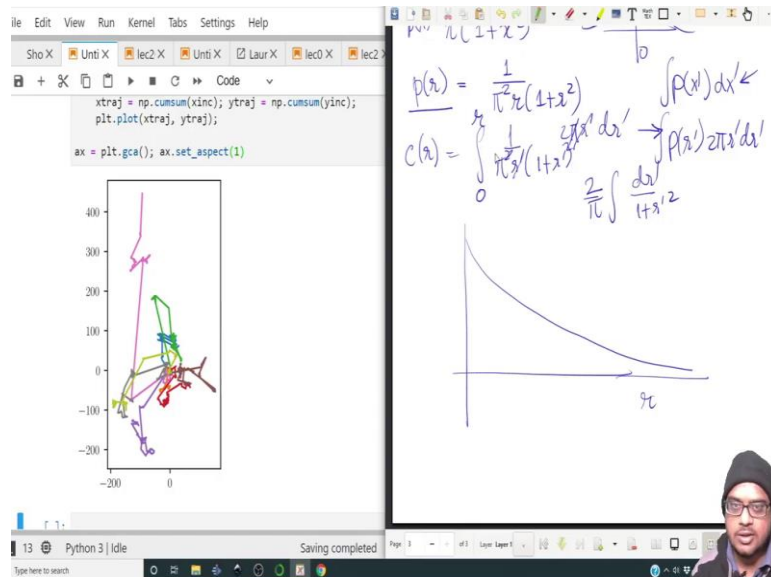
So, it is a classic foraging behaviour. In fact, let me now run this program for np number of particles.

(Refer Slide Time: 34:12)



So, let me grab hold of this program. So, the beauty is we can reuse program. I mean you do not need to be rigid about it. We just need to change how we sample the distribution, ok. So, now, let me run this. So, these are, so let me make the aspect ratio, correct. So, alright, so, one of the distributions really flew off.

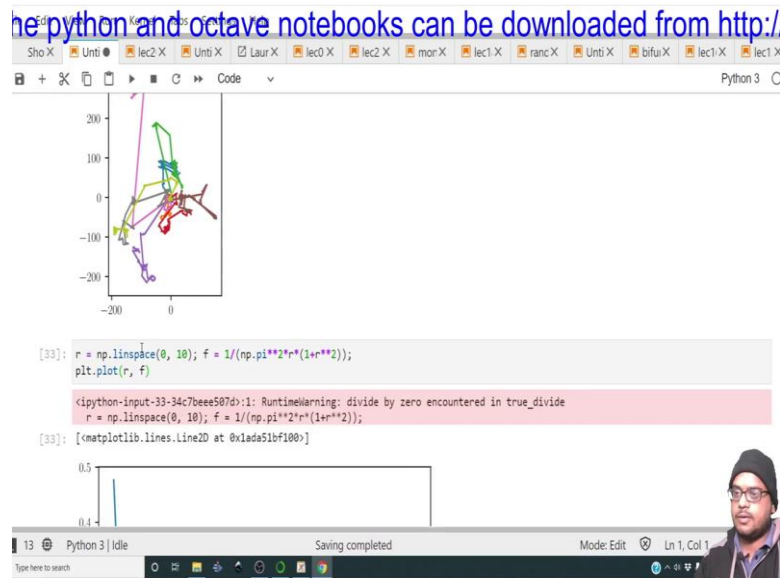
(Refer Slide Time: 34:28)



So, the thing is Cauchy distribution, ok. So, the Cauchy distribution has that kind of a fat tail that we had discussed in the previous lecture corresponding to a levy stable distribution or a Pareto distribution. So, it does have a tail in which there will be a small yet finite probability to get high jumps, so higher values of Δx or Δr in this case, ok. So,

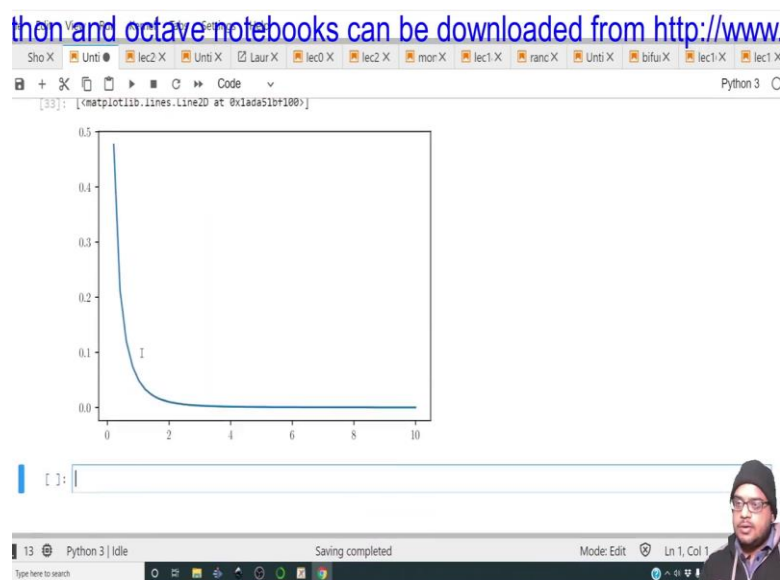
there will be a chance to get that. In fact, let me just show you, let me just plot the distribution function.

(Refer Slide Time: 35:06)



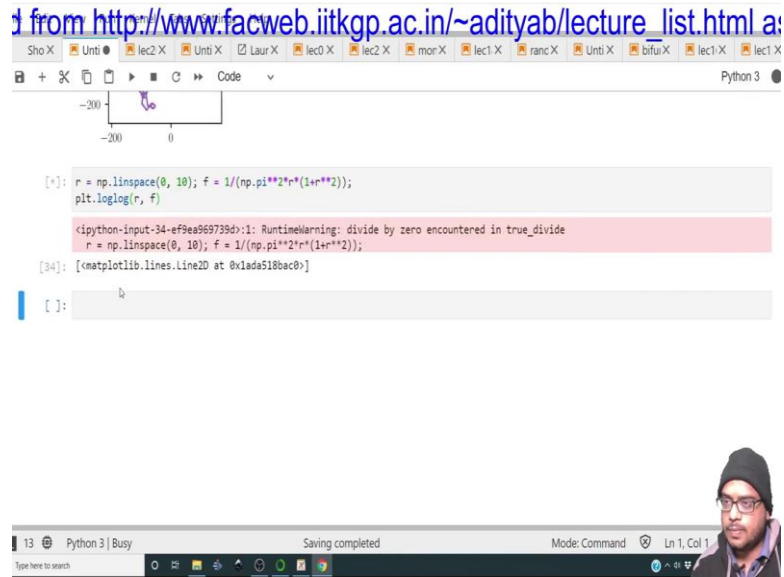
So, $r = np.linspace(0, 10)$ and $f = 1/(np.pi * r^2 * (1+r^2))$ and then we will do `plt.loglog(r, f)`, ok.

(Refer Slide Time: 35:36)

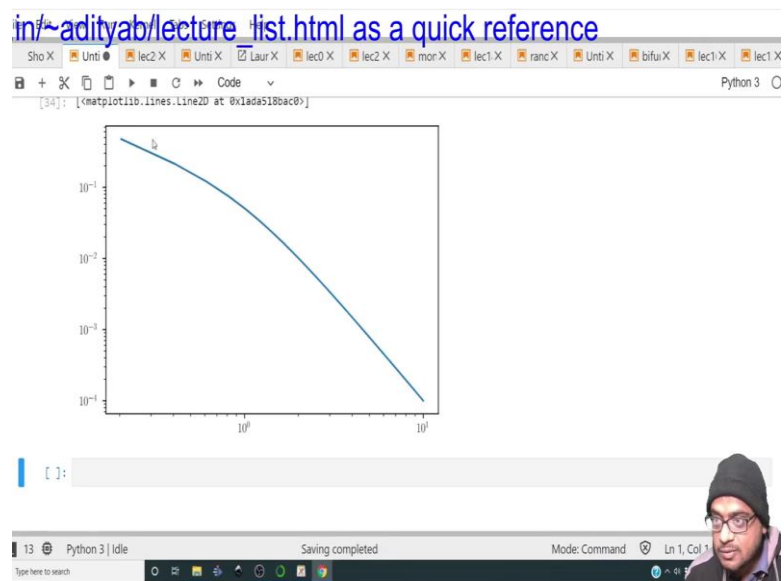


So, it looks something like this. So, the values are not 0 over here. They are small values, but they are not 0.

(Refer Slide Time: 35:46)



(Refer Slide Time: 35:51)



So, let me make it a log log, ok. So, it is small probability, but not 0. So, you will more often than not sample things which are in the higher probability zone. So, you will sample small steps. But every now and then you will sample that large step and that gives rise to this kind of a walk, ok. Let me run it one more time. So, each time you do it you will get a random walk, ok.

So, now, let me reuse one of our previous programs to find out the distribution of length for such a random walk. Let me copy this. Let me paste this for now. What I will do is I

will I will make this get final location vectorized, alright. I will make it vectorized. But before that let me just change the get stride function. I must replace it by this, alright.

(Refer Slide Time: 36:59)

```

[ ]: def get_angle(N): # N is the time steps
      return np.random.uniform(0, 2*np.pi, size=(N,))

def get_stride(N):
    return np.abs(np.random.standard_cauchy(size=(N,)))

def get_final_loc(N):
    dx = np.zeros((N,)); dy = np.zeros((N,)); # Initialization of the arrays
    l = get_stride(1);
    th = get_angle(1);
    dx =
        x[0,i] = deltax
        x[1,i] = deltay

    xinc = x[0,:]; yinc = x[1,:];
    xtraj = np.sum(xinc); ytraj = np.sum(yinc); rorigin = (xtraj**2 + ytraj**2)**0.5;
    return rorigin

N = 200;

```

So, I will remove all these, I will remove this, I will say dx is N, 1. I will say dy is also this and I will instead of calling it in a loop, I will get $\ln \theta$, I will remove the indentation. Once I have $\ln \theta$ then, so this is just initialization. So, then dx well you do not really need to initialize this, you do not need to really initialize this. It is sometimes good to initialize, ok.

(Refer Slide Time: 37:52)

```

[37]: def get_angle(N): # N is the time steps
      return np.random.uniform(0, 2*np.pi, size=(N,))

def get_stride(N):
    return np.abs(np.random.standard_cauchy(size=(N,)))

def get_final_loc(N):
    l = get_stride(N);
    th = get_angle(N);
    dx = l*np.cos(th); dy = l*np.sin(th);
    xtraj = np.sum(dx); y = np.sum(dy); rorigin = (xtraj**2 + ytraj**2)**0.5;
    print(rorigin)
    return rorigin

N = 200;
Np = 10000;
r = np.zeros((Np,))

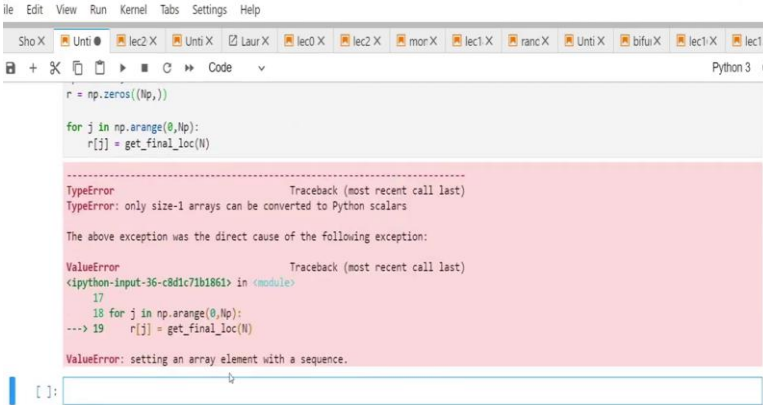
for j in np.arange(0,Np):
    r[j] = get_final_loc(N)

[1990.76031502 1990.76031805 1990.76190602 1990.76118116 1990.78744837
 1990.79000919 1990.79159643 1990.79606977 1990.80569284 1990.80398313
 1990.81908621 1990.82151448 1990.81761401 1990.82367142 1990.89022206
 1990.82661000 1990.82555501 1990.82555501 1990.82555501 1990.82555501

```

So, then dx. So, instead of getting only one stride we will get n strides. And $\text{deltax} = l * \text{np.cos}(\text{th})$ $\text{deltay} = l * \text{np.sin}(\text{th})$. We do not need this. We do not need any of this, alright; x trajectory will be simply these sum of all increments and the y trajectory will be simply the sum of all increments in the y direction. And yes, this that is it. This is the entire function. We have got it down from bunch of lines to only 5 lines, great.

(Refer Slide Time: 38:39)



```

r = np.zeros((Np,))

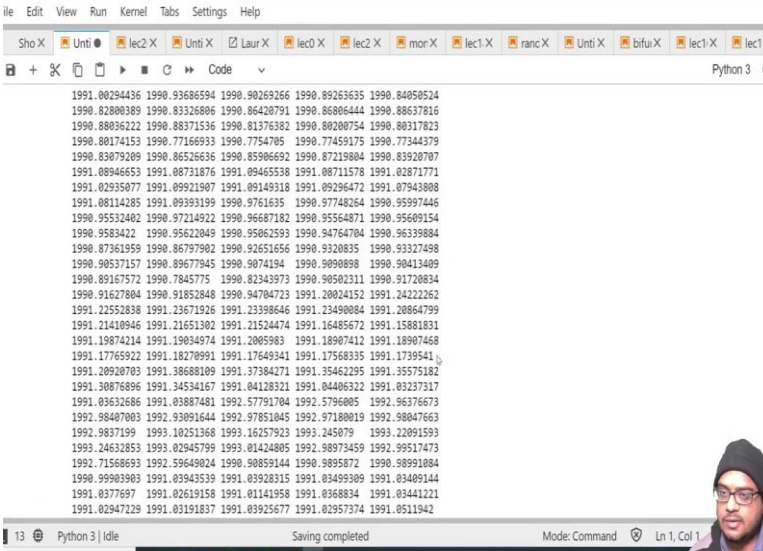
for j in np.arange(0,Np):
    r[j] = get_final_loc(N)

```

TypeError Traceback (most recent call last)
TypeError: only size-1 arrays can be converted to Python scalars
 The above exception was the direct cause of the following exception:
ValueError Traceback (most recent call last)
 <ipython-input-36-c8d1c71b1861> in <module>
 17
 18 for j in np.arange(0,Np):
 --> 19 r[j] = get_final_loc(N)
ValueError: setting an array element with a sequence.

So, let me run this and there is an error. Get final location. So, what is the problem? Let us let us see what is the problem; so, rorigin. So, let us print out, ok.

(Refer Slide Time: 39:02)



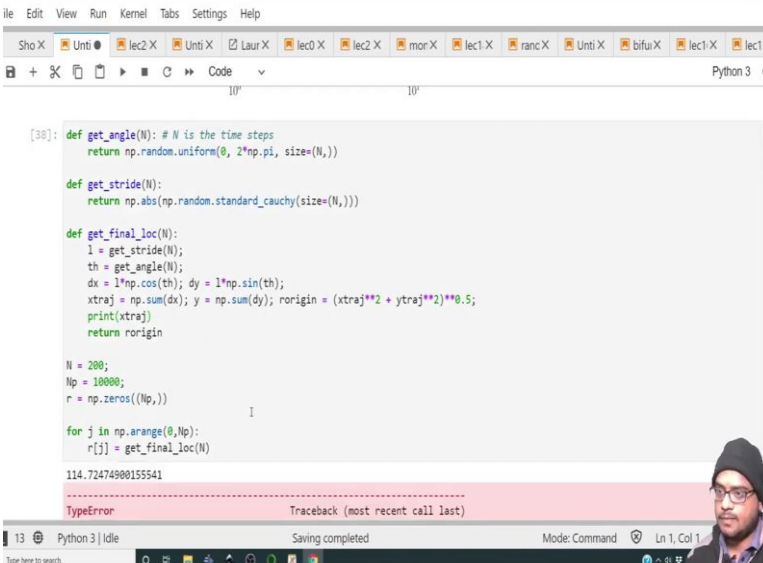
```

1991.00294436 1990.93686594 1990.90269266 1990.89263635 1990.84058524
1990.82800389 1990.83326806 1990.86420791 1990.86806444 1990.88637815
1990.88036222 1990.88371536 1990.81376362 1990.80200754 1990.80317823
1990.80174153 1990.77166933 1990.7754705 1990.77459175 1990.77344379
1990.83079209 1990.86526636 1990.85906692 1990.87219804 1990.83920787
1991.08946653 1991.08731876 1991.09465538 1991.08711578 1991.02871771
1991.02935077 1991.09921907 1991.09149318 1991.09296472 1991.07943808
1991.08114285 1991.09393199 1990.9761635 1990.97748264 1990.95997446
1990.95532482 1990.97214922 1990.96687182 1990.95564871 1990.95609154
1990.9583422 1990.95622049 1990.95062593 1990.94764704 1990.96339884
1990.87361959 1990.86797902 1990.92651656 1990.9320835 1990.93327498
1990.90537157 1990.89677945 1990.9074194 1990.9090898 1990.90413409
1990.89167572 1990.7845775 1990.82343973 1990.90502311 1990.91720834
1990.91627804 1990.91852848 1990.94704723 1991.20024152 1991.24222262
1991.22552838 1991.23671926 1991.23398646 1991.23490084 1991.20864799
1991.21410946 1991.21651302 1991.21524474 1991.16485672 1991.15881831
1991.19874214 1991.19034974 1991.2005983 1991.18907412 1991.18907468
1991.17765922 1991.18270991 1991.17649341 1991.17568335 1991.1739541
1991.20920703 1991.38688109 1991.37384271 1991.35462295 1991.35575182
1991.30876896 1991.34534167 1991.04128321 1991.04446322 1991.03237317
1991.03632686 1991.03887481 1992.57791704 1992.5796005 1992.96376673
1992.98407803 1992.93091644 1992.97851045 1992.97180019 1992.98047663
1992.9837199 1993.10251368 1993.16257923 1993.245079 1993.22091593
1993.24632853 1993.02945799 1993.01424805 1992.98973459 1992.99517473
1992.71568693 1992.59649024 1990.90859144 1990.9095072 1990.90991804
1990.99903903 1991.03943539 1991.03928315 1991.03499309 1991.03409144
1991.0377697 1991.02619158 1991.01141958 1991.0368834 1991.03441221
1991.02847229 1991.03191837 1991.03925677 1991.02957374 1991.0511942

```

It is a bunch of, ok.

(Refer Slide Time: 39:13)



```
file Edit View Run Kernel Tabs Settings Help
Sho X Unti X lec2 X Unti X Laur X lec0 X lec2 X mor X lec1 X ranc X Unti X bifuj X lec1 X lec1 X
Python 3

[30]: def get_angle(N): # N is the time steps
      return np.random.uniform(0, 2*np.pi, size=(N,))

      def get_stride(N):
        return np.abs(np.random.standard_cauchy(size=(N,)))

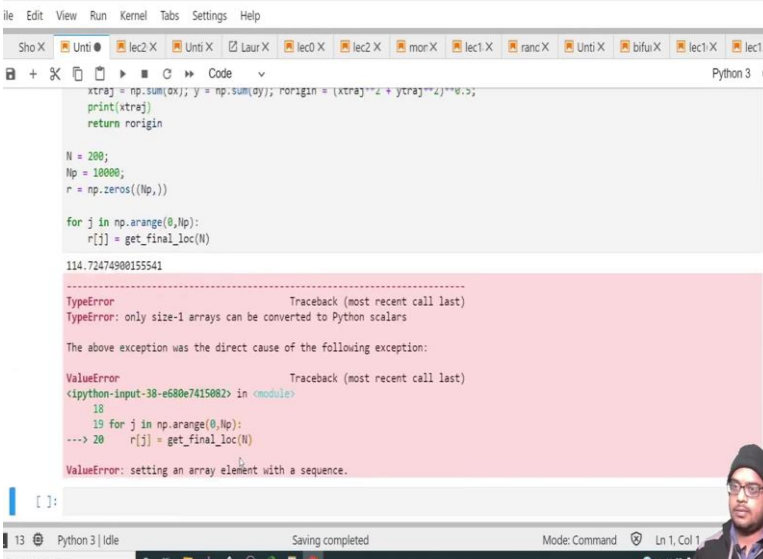
      def get_final_loc(N):
        l = get_stride(N);
        th = get_angle(N);
        dx = l*np.cos(th); dy = l*np.sin(th);
        xtraj = np.sum(dx); y = np.sum(dy); rorigin = (xtraj**2 + ytraj**2)**0.5;
        print(xtraj)
        return rorigin

      N = 200;
      Np = 10000;
      r = np.zeros((Np,))

      for j in np.arange(0,Np):
        r[j] = get_final_loc(N)

114.72474900155541
TypeError                                Traceback (most recent call last)
```

(Refer Slide Time: 39:14)



```
file Edit View Run Kernel Tabs Settings Help
Sho X Unti X lec2 X Unti X Laur X lec0 X lec2 X mor X lec1 X ranc X Unti X bifuj X lec1 X lec1 X
Python 3

xtraj = np.sum(dx); y = np.sum(dy); rorigin = (xtraj**2 + ytraj**2)**0.5;
print(xtraj)
return rorigin

N = 200;
Np = 10000;
r = np.zeros((Np,))

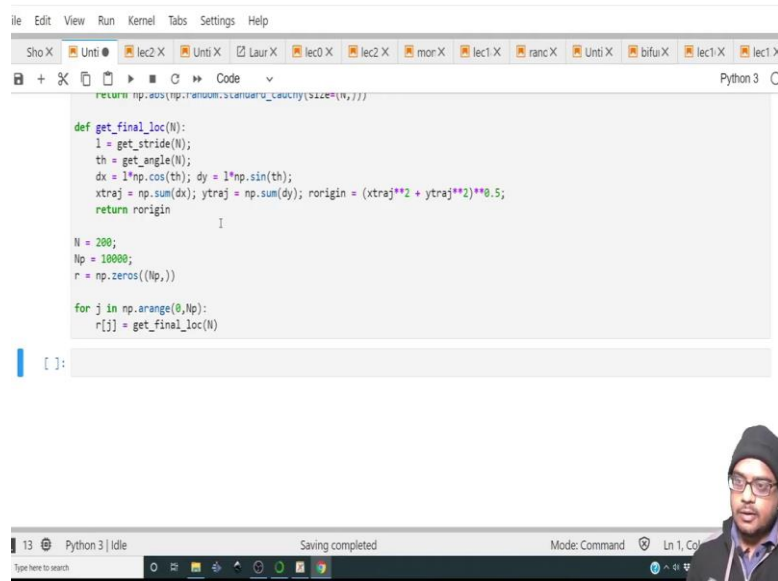
for j in np.arange(0,Np):
  r[j] = get_final_loc(N)

114.72474900155541
TypeError                                Traceback (most recent call last)
TypeError: only size-1 arrays can be converted to Python scalars

The above exception was the direct cause of the following exception:

ValueError                                Traceback (most recent call last)
<ipython-input-38-e680e7415082> in <module>
    18
    19 for j in np.arange(0,Np):
--> 20   r[j] = get_final_loc(N)
ValueError: setting an array element with a sequence.
```


(Refer Slide Time: 39:27)



```
return np.dot(np.random.randn(2), np.ones(N))

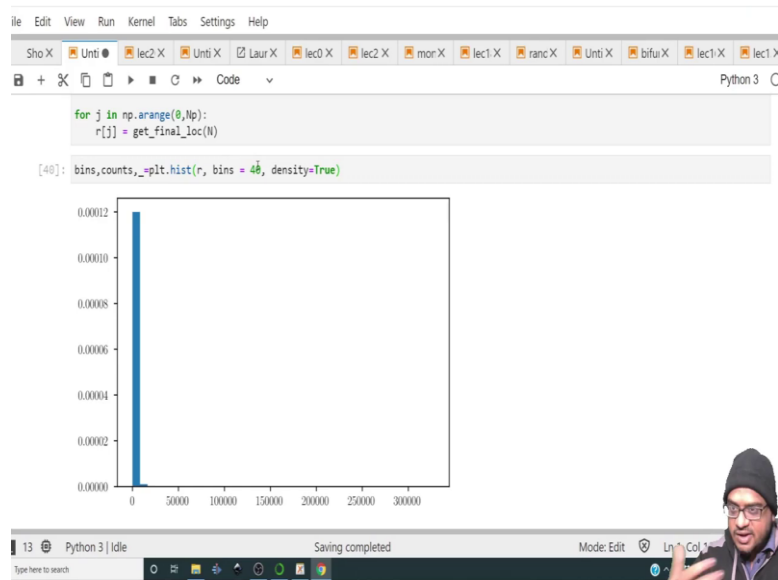
def get_final_loc(N):
    l = get_stride(N);
    th = get_angle(N);
    dx = l*np.cos(th); dy = l*np.sin(th);
    xtraj = np.sum(dx); ytraj = np.sum(dy); rorigin = (xtraj**2 + ytraj**2)**0.5;
    return rorigin

N = 200;
Np = 10000;
r = np.zeros((Np,))

for j in np.arange(0,Np):
    r[j] = get_final_loc(N)
```

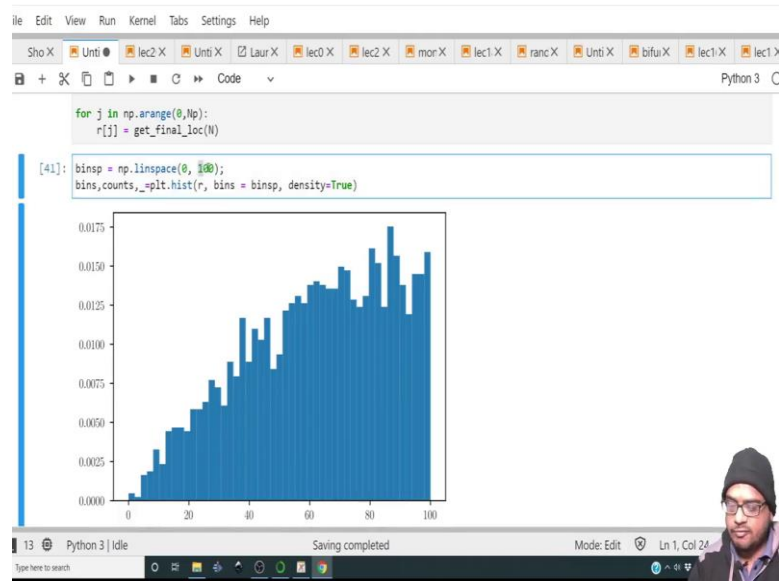
The issue is this has to be ytraj, ok. I do not know why I did that, ok. So, it is running the program for a bunch of particles. Let it run. It is already run I know, ok. So, it is already run. We need to do the pdf.

(Refer Slide Time: 40:06)

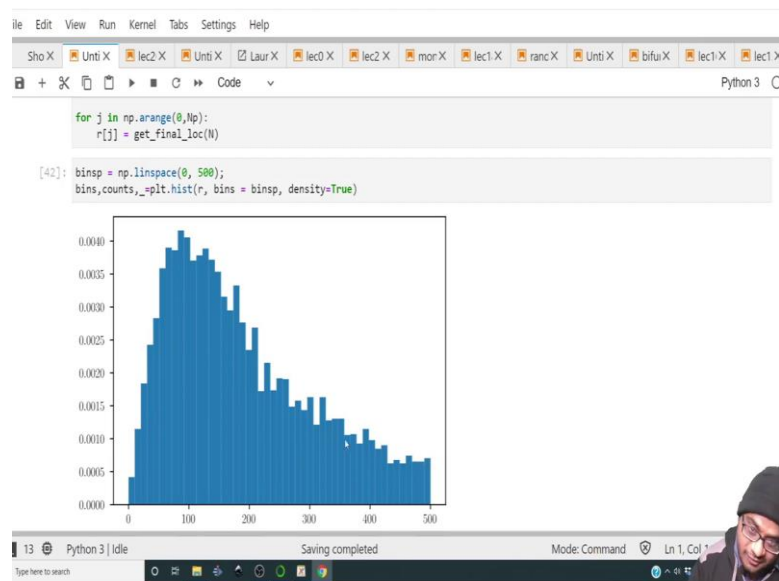


So, let me copy this just to find out the pdf, ok. So, obviously, there are some values over here, but most of them are still centred around the origin. In fact, let me decide what the bin locations will be.

(Refer Slide Time: 40:30)



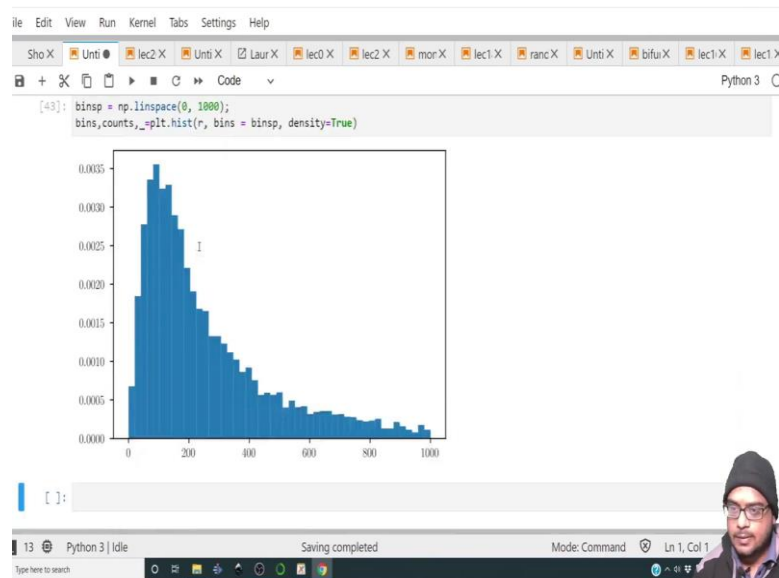
(Refer Slide Time: 40:50)



So, I will call it as bin space bin space equal to `np.linspace` and let it go from say 0 to 100. So, let us run this, ok. So, the distribution looks something like this. In fact, we can increase it to 500. Let us see what we have. So, we have something like this. So, it is a much fatter tail than before compared to the distribution over here. So, it decays down much faster, but over here it tends to decay much slower. So, at a given point in time there are majority of the particles over here.

So, the time it took was for 200 time steps. It is peaking around 100. What was the peak over here? It was something around 10, ok. So, the larger peak is indicative of larger steps taken because of the Cauchy distribution and the distribution appears to be tailing off very slowly and these are all highlights of the levy walk. So, if these distributions are usually a power law distributions, ok.

(Refer Slide Time: 41:42)



So, there are particles which have travelled such a large distances. They are not large in number, but regardless they are there, ok. So, I would like to conclude this particular week over here. And lot of things we have seen. And once you start doing a course on probability and statistics, I really hope you will find all these tools that we have discussed immensely useful and I hope you will make the best use of it.

With this I will see you again next week. Have a good week. Bye.