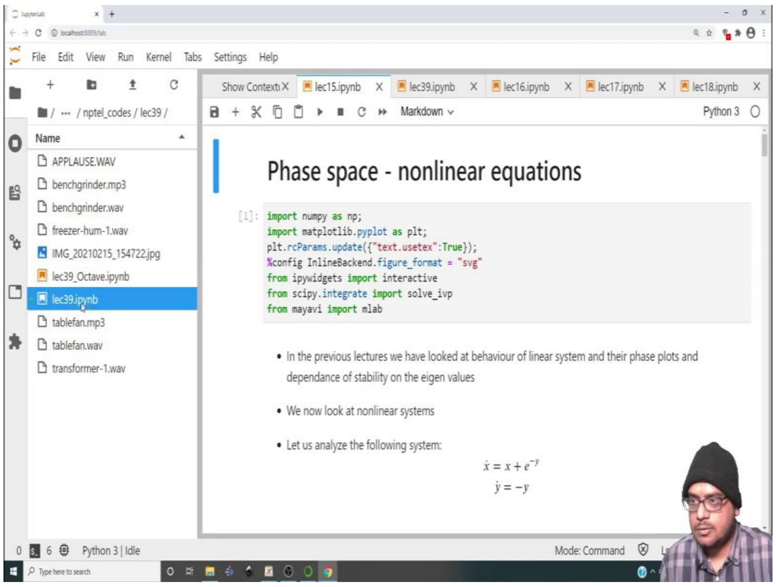
Tools in Scientific Computing

Prof. Aditya Bandopadhyay
Department of Mechanical Engineering
Indian Institute of Technology, Kharagpur

Lecture - 40
Spectrogram and Doppler shift

Hi everyone, in this lecture we are going to move ahead in our analysis of audio. So, in the previous lecture we had looked at the following work workflow.

(Refer Slide Time: 00:49)



Phase space - nonlinear equations

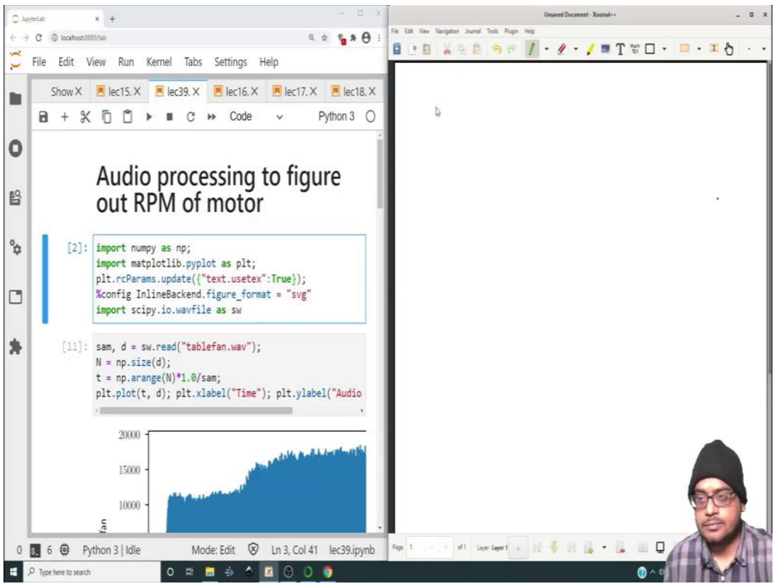
```
[1]: import numpy as np;
     import matplotlib.pyplot as plt;
     plt.rcParams.update({"text.usetex":True});
     %config InlineBackend.figure_format = "svg"
     from ipywidgets import interactive
     from scipy.integrate import solve_ivp
     from mayavi import mlab
```

- In the previous lectures we have looked at behaviour of linear system and their phase plots and dependance of stability on the eigen values
- We now look at nonlinear systems
- Let us analyze the following system:

$$\dot{x} = x + e^{-y}$$
$$\dot{y} = -y$$

So, what we had was ok (Refer Time: 00:46). So, let me open up the program it was alright so, what we did was analyze the signal we obtained from a motor or whatever.

(Refer Slide Time: 01:03)

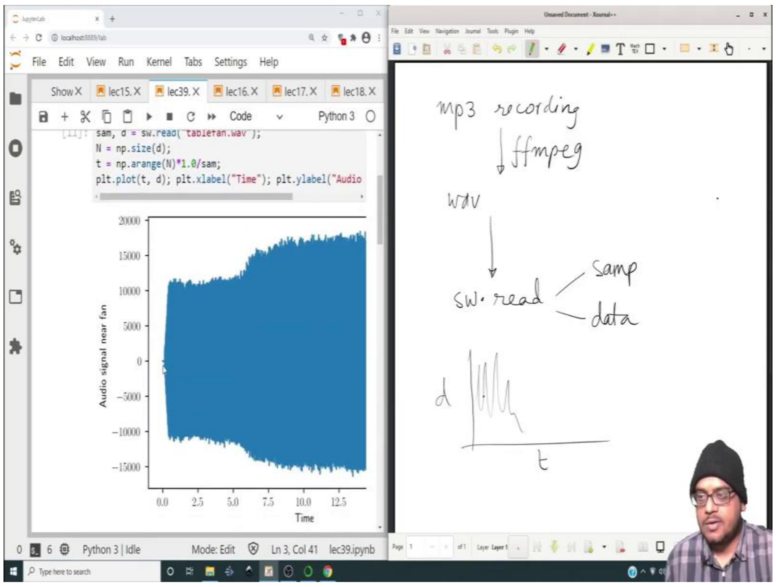


Audio processing to figure out RPM of motor

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

```
[11]: sam, d = sw.read("tablefan.wav");
      N = np.size(d);
      t = np.arange(N)*1.0/sam;
      plt.plot(t, d); plt.xlabel("Time"); plt.ylabel("Audio
```

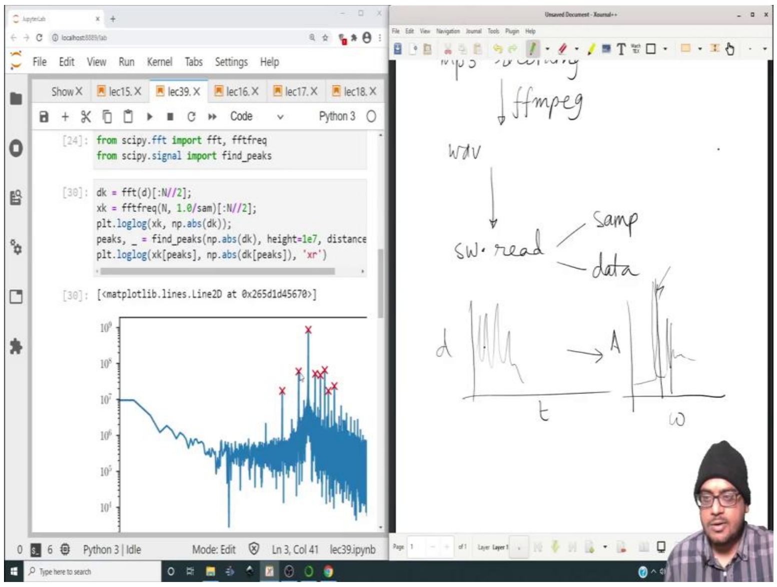
(Refer Slide Time: 01:18)



```
      N = np.size(d);
      t = np.arange(N)*1.0/sam;
      plt.plot(t, d); plt.xlabel("Time"); plt.ylabel("Audio
```

So, we took an mp3 recording, we converted into wave this was using ffmpeg. Once we had the wave file we did a SW dot wave read the SW dot read and that give us the sampling rate and that gave us the entire data and with the help of that we could plot the time series whatever wave form it is.

(Refer Slide Time: 02:06)

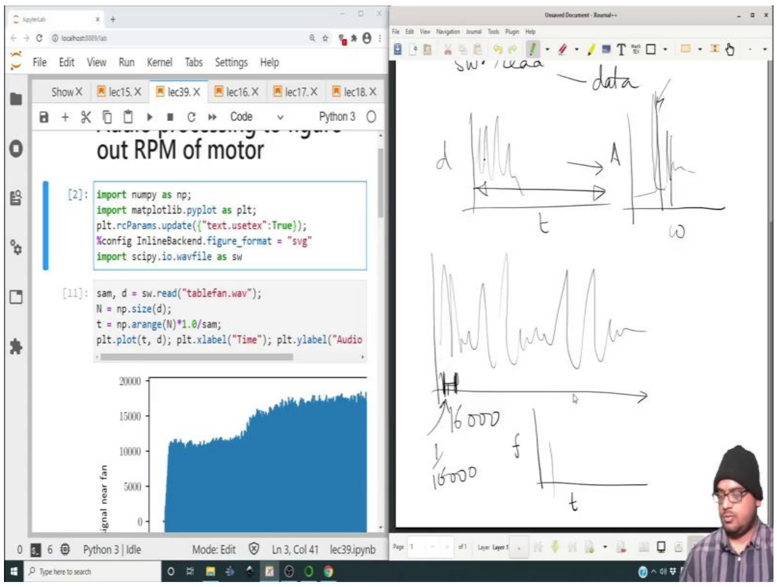


```
[24]: from scipy.fft import fft, fftfreq
      from scipy.signal import find_peaks

[30]: dk = fft(d)[:N//2];
      xk = fftfreq(N, 1.0/sam)[:N//2];
      plt.loglog(xk, np.abs(dk));
      peaks, _ = find_peaks(np.abs(dk), height=1e7, distance
      plt.loglog(xk[peaks], np.abs(dk[peaks]), 'xr')

[30]: [<matplotlib.lines.Line2D at 0x265d1d45670>]
```

So, we had something like this now, once we had that we took a Fourier transform of this time series to converted into frequency domain or omega domain and we plotted the amplitude. So, certain peaks were found and these were the dominant frequencies at which the energy is sort of contained. So, for example, in this it was at these frequencies for the case of a bench grinder it was at this frequency.

Remember this was a complete single recording meaning, we were working under the assumption that whatever signal we were getting that is periodic.

(Refer Slide Time: 02:49)



out RPM of motor

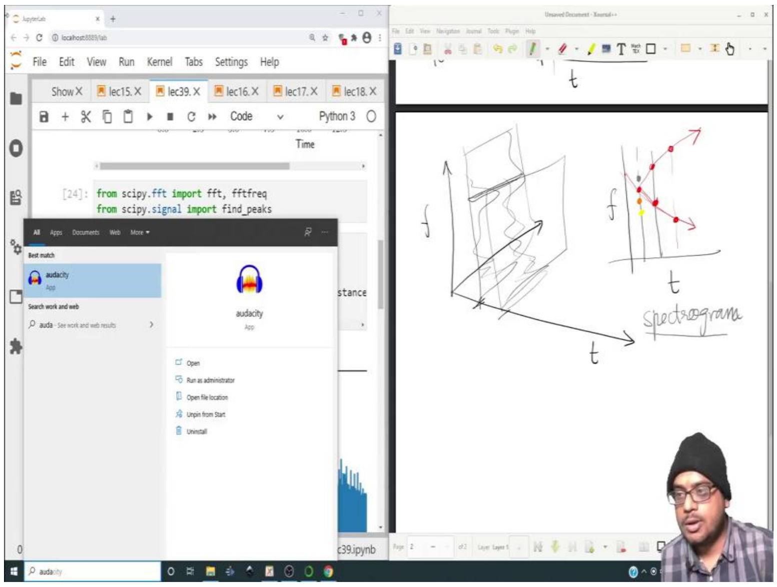
```
[2]: import numpy as np;
     import matplotlib.pyplot as plt;
     plt.rcParams.update({"text.usetex":True});
     %config InlineBackend.figure_format = "svg"
     import scipy.io.wavfile as sw

[11]: sam, d = sw.read("tablefan.wav");
      N = np.size(d);
      t = np.arange(N)*1.0/sam;
      plt.plot(t, d); plt.xlabel("Time"); plt.ylabel("Audio
```

And so we took a discrete Fourier transform of the entire domain however, what will happen if instead of this you have you want to do a sort of you want to analyze a speech for example. So, surely you cannot wait for the entire speech to finish you want to obviously, start figuring out which are the dominant frequencies in the audio. So, suppose this is the entire audio.

So, you know that you have 16000 samples per second of course, and change the bit rate you can change the number of samples per second do whatever you want, but essentially you are going to break down ok. So, the smallest Δt is 1/16000 and that corresponds to the largest frequency if you do not have to capture any higher frequency so, human audible range is 20 kilo Hertz beyond that there is no point.

So, you take a window like this find out the discrete Fourier transform and you assign it to this time. So, on this axis I have time and on this axis I am going to plot the frequency.

(Refer Slide Time: 04:15)



```
[24]: from scipy.fft import fft, fftfreq
      from scipy.signal import find_peaks
```

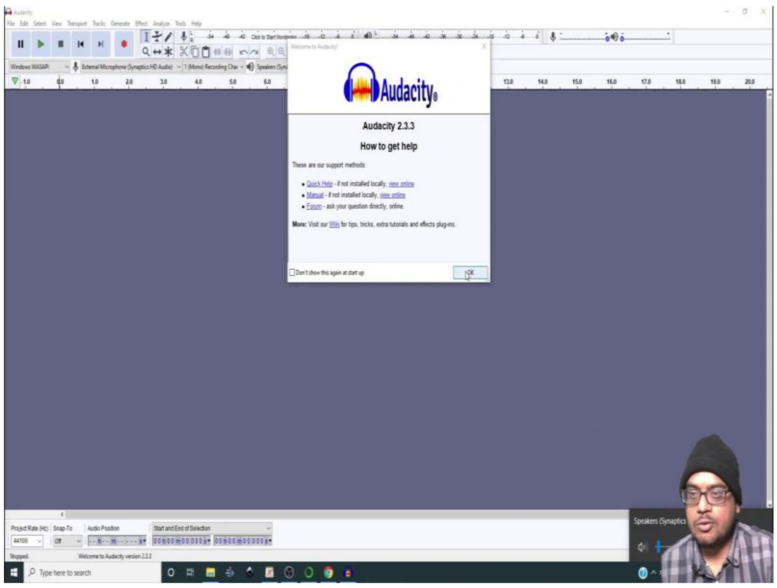
So, at a certain time, so if you can imagine you have a plot like this right. So, this is time this frequency so at a given time right you have all these plots at the next time you have all these plots. So, these are in the orthogonal plane to f t curve. So, in each plane you get that particular Fourier spectrum.

Now, because you do not want to make a 2D plot a 3D plot you take the amplitude at each frequency and you give it a color essentially you are still plotting in f and t, but now you make a color of this entire thing. So, wherever the frequency has the max amplitude to give it a red whenever it is medium you give it orange then yellow and so on. Meaning, at a given time for all the frequencies you have a different color and the colors they correspond to different amplitudes ok.

So, this is how you create what is called as a spectrogram and a spectrogram is a very useful tool to have because, it allows you to directly get a good visual feel of which frequencies at a certain time are dominating over the others. So, at the next time if you have the red appearing over here well and the next time you stand over here you know that as time increases the dominant frequency is reducing.

So, it will give a tone which is going down like it is going down, if instead it goes up and you know that the frequency is going up like something like that I will pardon my sound effects. So, spectrogram gives you that information in fact, on a spectrogram you will also sort of get an estimate of how the different harmonics are. So, let me do one thing, let me open a very popular tool it is called as audacity.

(Refer Slide Time: 06:51)



Welcome to Audacity!

Audacity 2.3.3

How to get help

These are our support methods:

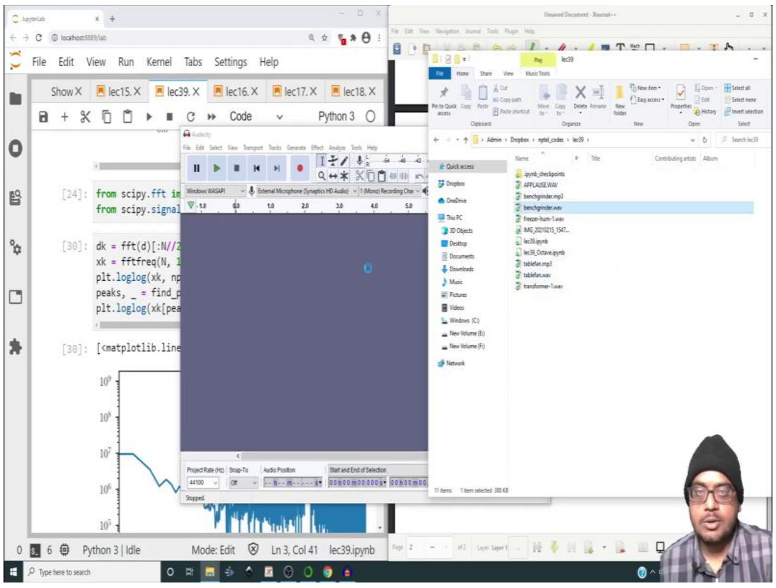
- Quick Help - if not installed locally, view online
- Manual - if not installed locally, view online
- Forum - ask your question directly, online.

More: Visit our Wiki for tips, tricks, extra tutorials and effects plug-ins.

Don't show this again at start up

Let me try to show you the spectrogram. So, audacity is once again a very it is an open source tool it is very robust and it can do a lot of things.

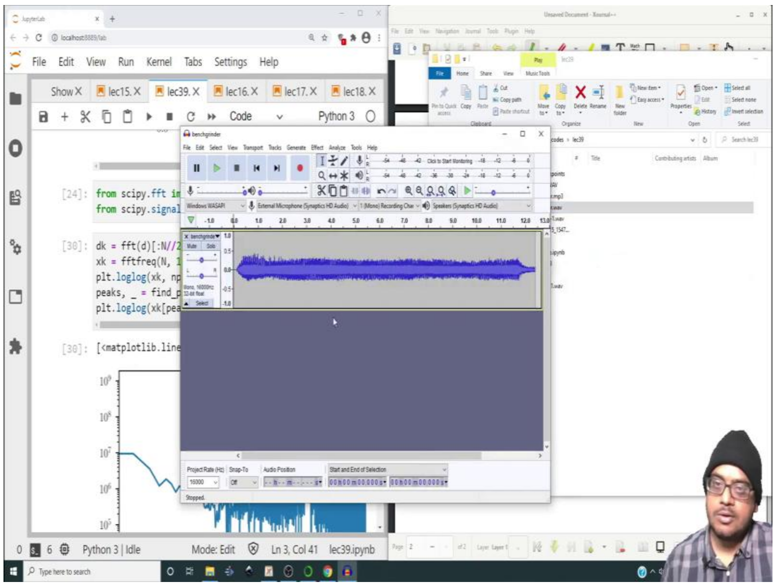
(Refer Slide Time: 06:59)



```
[24]: from scipy.fft im
      from scipy.signal

[30]: dk = fft(d)[:N//2
      xk = fftfreq(N, 1
      plt.loglog(xk, np
      peaks, _ = find_p
      plt.loglog(xk[pea

[30]: [<matplotlib.line
```

So, let me go to our previous folder let me open up the bench grinder video the bench grinder audio ok, this was the bench grinder audio.

(Refer Slide Time: 07:10)



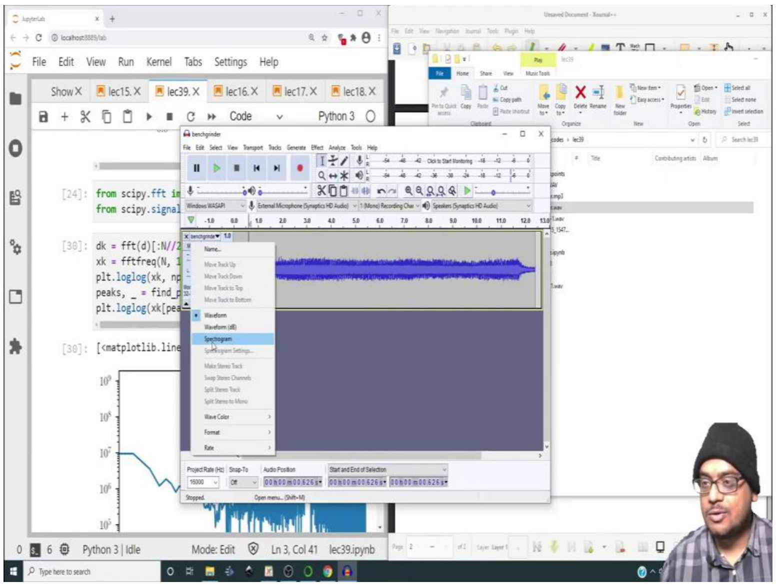
```
[24]: from scipy.fft im
      from scipy.signal

[30]: dk = fft(d)[:N//2
      xk = fftfreq(N, 1
      plt.loglog(xk, np
      peaks, _ = find_p
      plt.loglog(xk[pea

[30]: [<matplotlib.line
```

(Refer Slide Time: 07:20)

Audio of the sound source moving past detector

And so, I have turned on the volume in my quarter I do not want to wake up everyone, but see the beauty is we do not need to wake up everyone we can simply visualize the waveform.

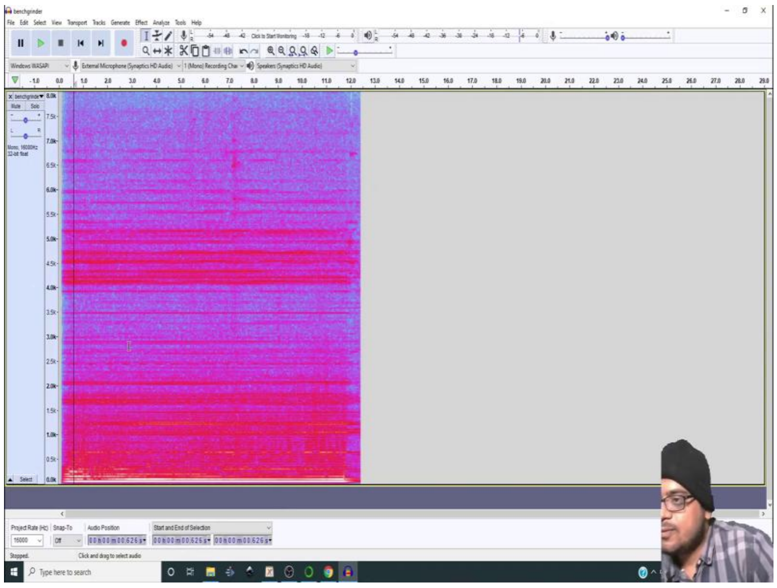
(Refer Slide Time: 08:01)



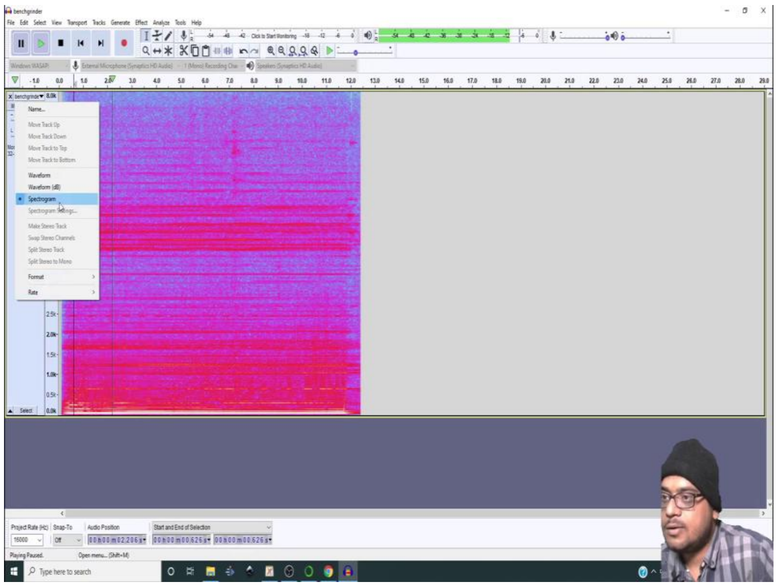
```
[24]: from scipy.fft in
      from scipy.signal

[30]: dk = fft(d)[:N//2
      xk = fftfreq(N, 1
      plt.loglog(xk, np
      peaks, _ = find_p
      plt.loglog(xk[pea

[30]: [<matplotlib.line
```

So, let me go over here let me switch on spectrogram ok. So, this is how the spectrogram looks like ok.

(Refer Slide Time: 08:05)

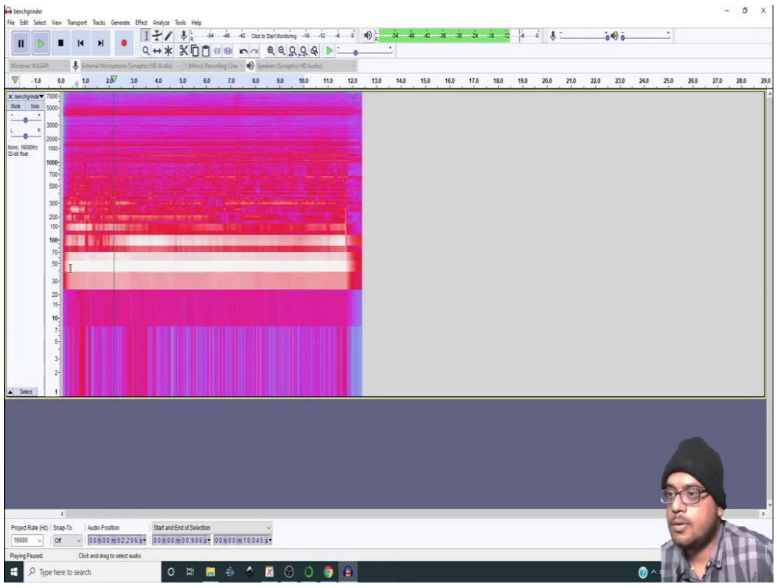


(Refer Slide Time: 08:26)



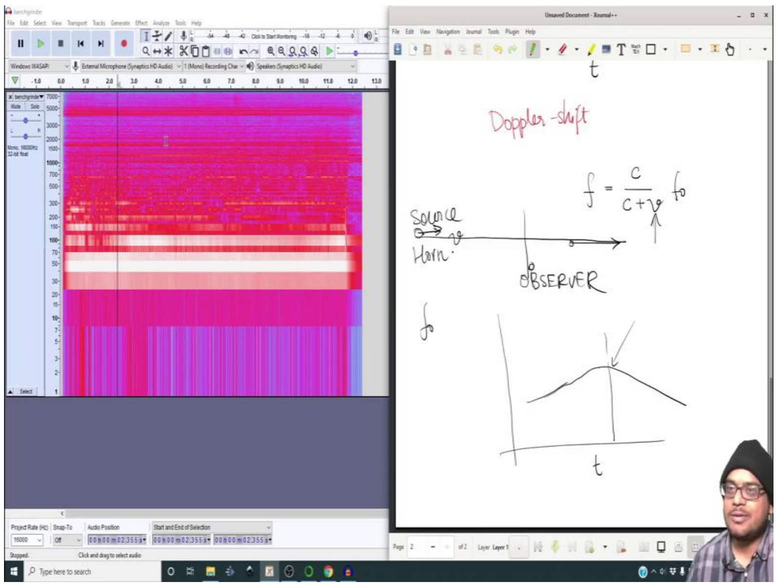
So, the spectrogram is peaking over here so, let me see if I can ok good.

(Refer Slide Time: 08:51)



So, I have taken a log of the y axis and look at where the peaks are formed so, it is 50 Hertz and at 100 Hertz 150 Hertz 200 Hertz 250 Hertz and so on. So, this spectrogram tells you that over the entire time of recording these are the dominant frequencies ok. The most dominant is the 50 Hertz frequency because, we know that the bench grinder was running at 3000 rpm and 3000 rpm corresponds to 50 Hertz.

(Refer Slide Time: 09:35)

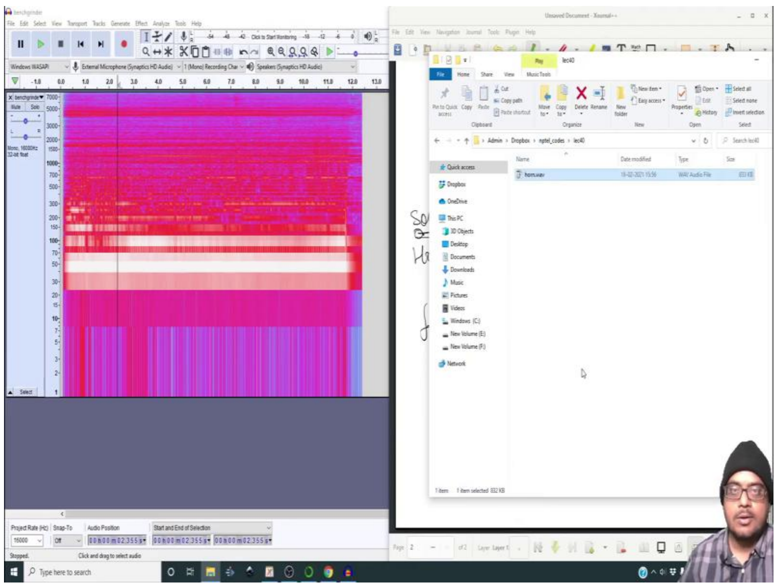


Now, in this particular lecture we are more interested in the Doppler shift. So, if you recall from elementary physics the Doppler shift corresponds to the apparent change of frequency. So, suppose this is an observer and this is a vehicle the moving source and the source has a horn the horn is blaring. So, if the horn is the source is steady you will perceive the horn to have a certain frequency say f naught.

But, once the horn on the source starts moving with the velocity v yeah the apparent frequency will be c upon c plus v times f naught, where if the source is approaching the observer the velocity will be negative meaning the observed frequency will be higher than the base frequency, but if the source is going away from the observer then you will perceive this as positive and so, the apparent frequency will reduce. So, what do you expect in a spectrogram.

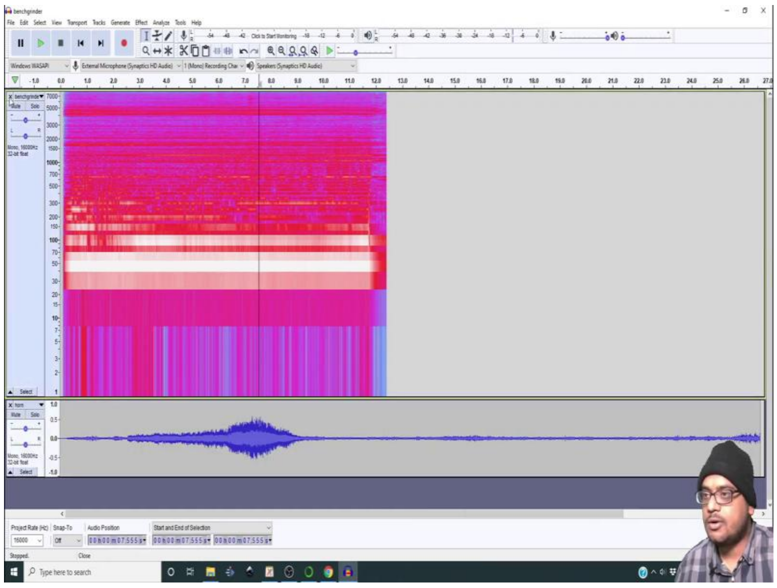
So, if this is time the peak that is this kind of a white color should sort of increase slightly and then fall off at this point you know that the source has passed you once you are standing over here. So, it is going like this. So, once it passes this point is no longer moving towards you if it is going away from you ok. So, then you know that you have passed well what I did was I went out to the street had a person do the recording.

(Refer Slide Time: 11:31)

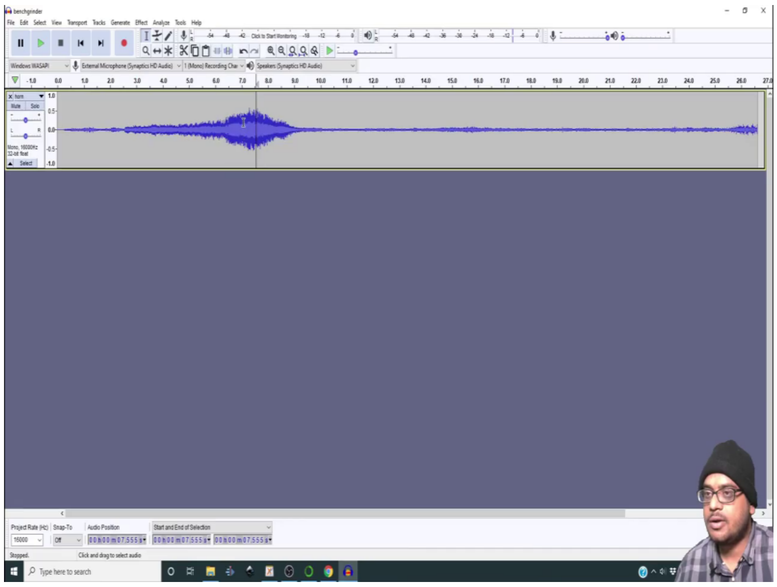


And the recording is saved in the wave file please let me just take it to alright. So, the recording is called as horn.wave.

(Refer Slide Time: 11:48)

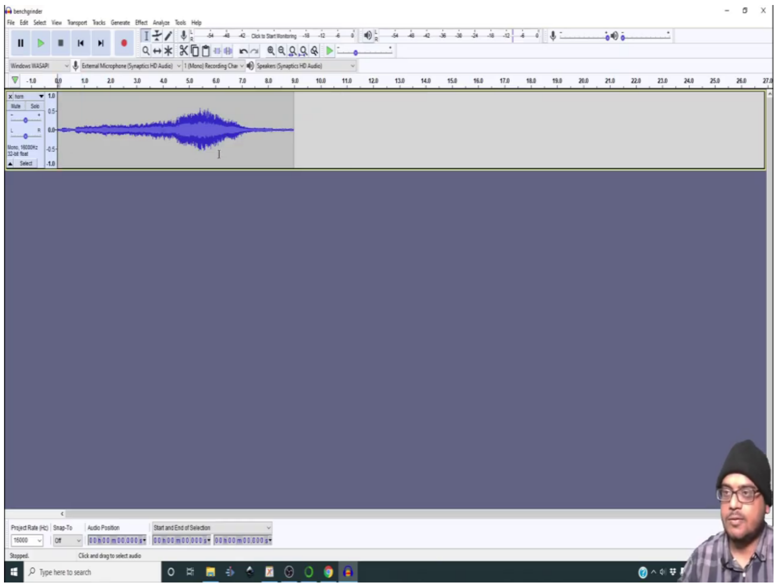


(Refer Slide Time: 11:51)

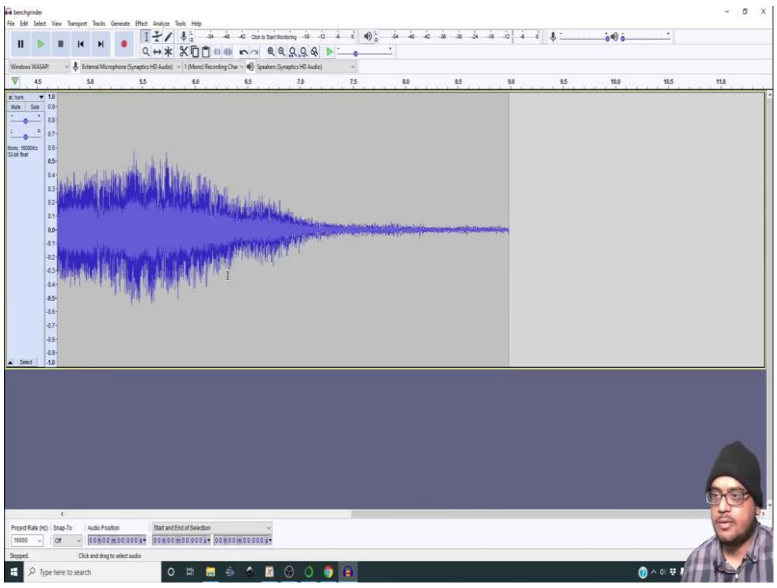


Let me load it let me get rid of this bench grinder, this is the time series for the horn not the horn, but so let me get rid of this ok.

(Refer Slide Time: 12:04)

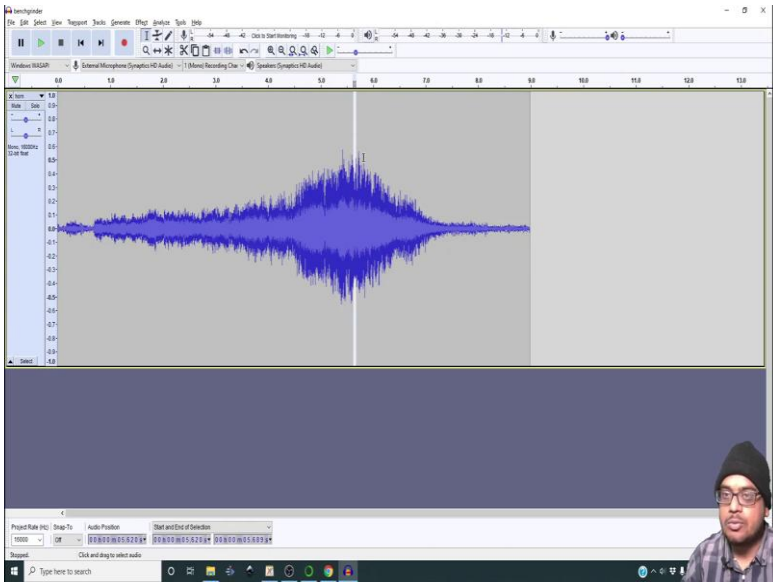


(Refer Slide Time: 12:10)



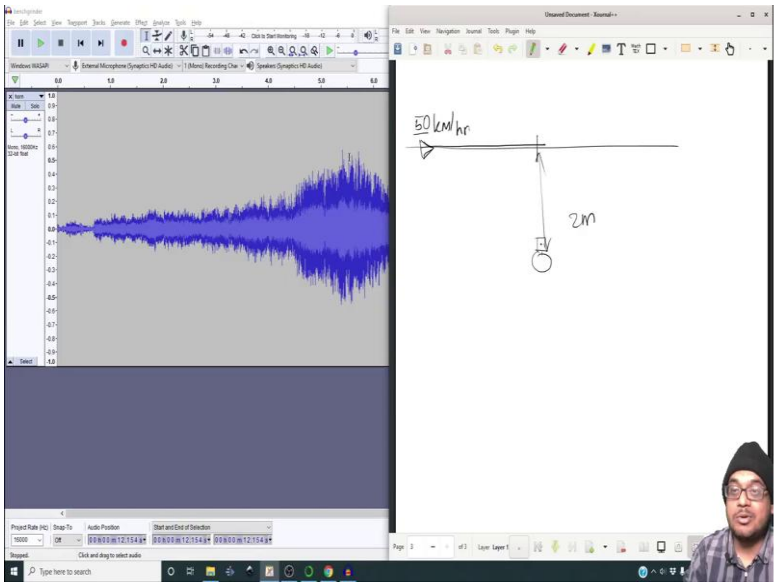
So, let me get rid of this as well ok. So, this is the recording I have ok.

(Refer Slide Time: 12:13)



So, I am interested in now ascertaining what is the shift in the frequency, well in this particular demonstration and the there was no control on what speed the observer was moving at we can sort of estimate maybe the observer was moving at 50 kilometers per hour and I leave it as a small task for you to figure it out.

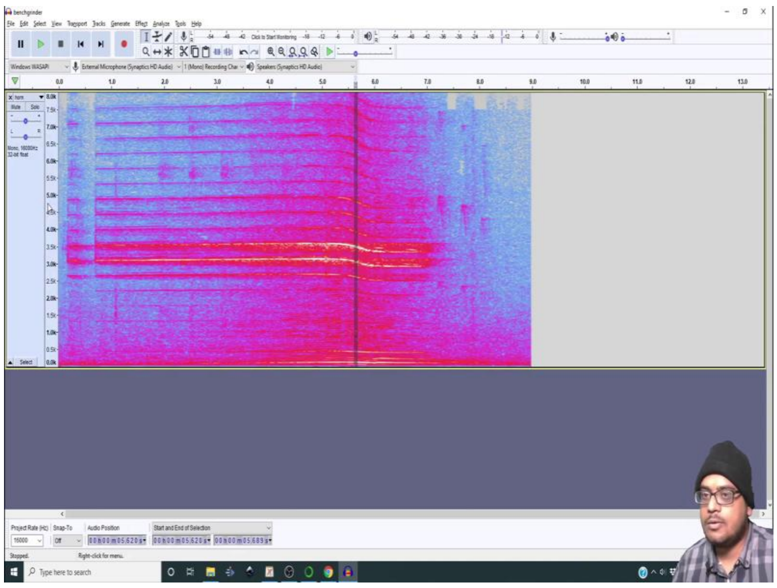
(Refer Slide Time: 12:48)



So, this was the observer I am showing the top view, that observer was holding the sound recording because it is a mobile phone; the distance between the moving vehicle or the street was 2 meter the vehicle was moving at a speed of 50 km/hr. I do not know what the frequency of the horn is but, you can sort of assume it to be whatever it is at far distances.

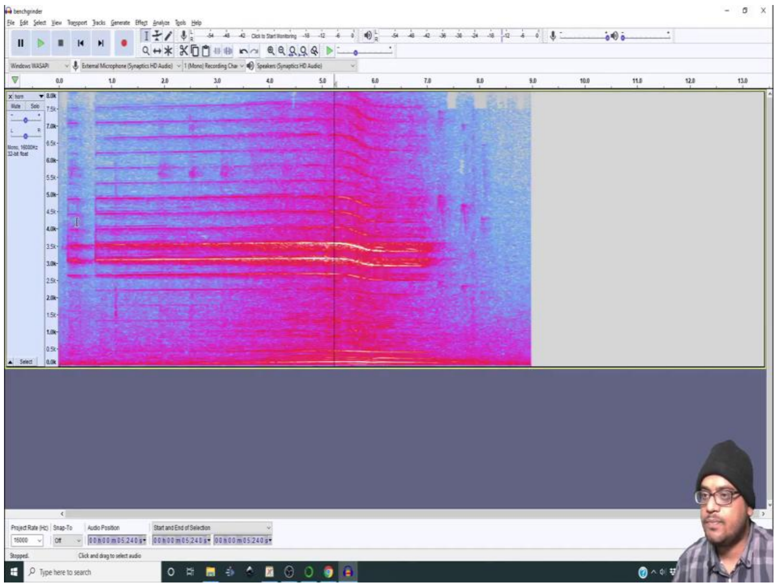
And once it starts moving you hear a high pitch sound and once the vehicle move moves by the frequency starts to reduce. And actually when the vehicle is moving by the amplitude of the sound also appears to increase and then decrease. So, around this point you think that the vehicle has moved by because, the amplitude also starts decreasing but we do not want to bother so much with the amplitude alright.

(Refer Slide Time: 13:50)

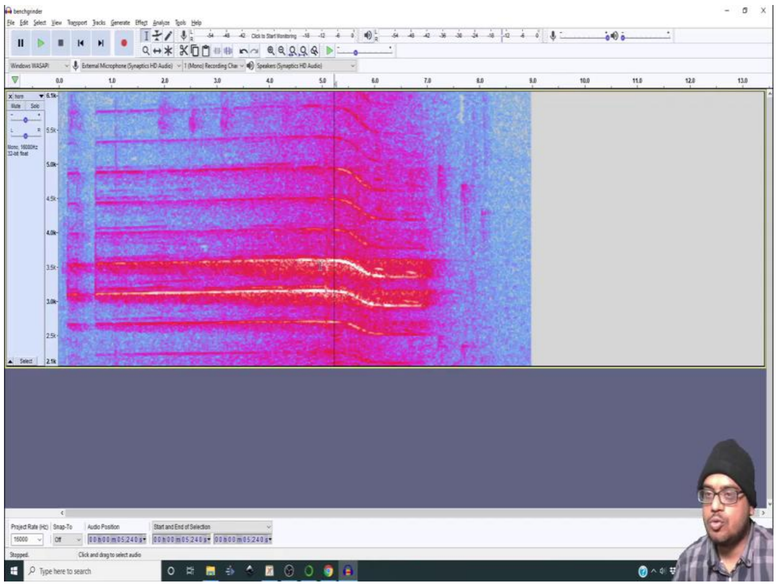


So now, let us look at the spectrogram, this gives us that information right at this point look this is increasing and then it decreases. In fact, let me make a logarithm of it actually the logarithm does not make things easy, let me zoom in alright.

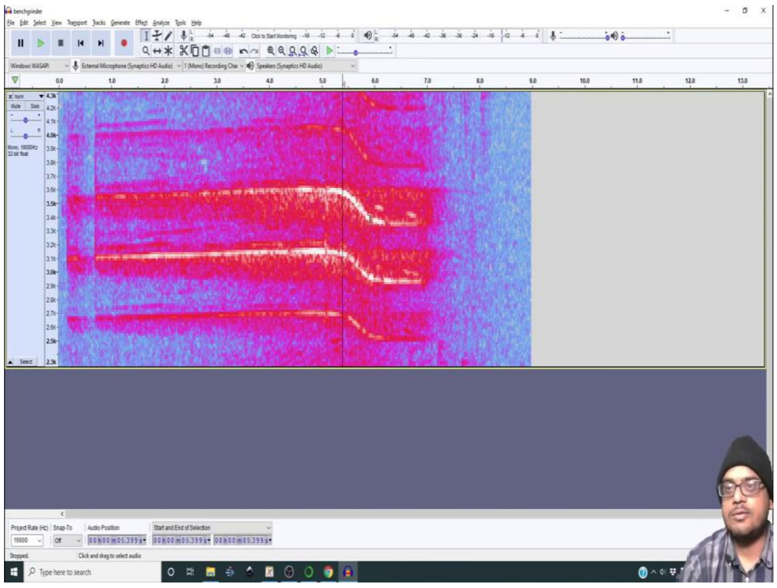
(Refer Slide Time: 14:10)



(Refer Slide Time: 14:14)

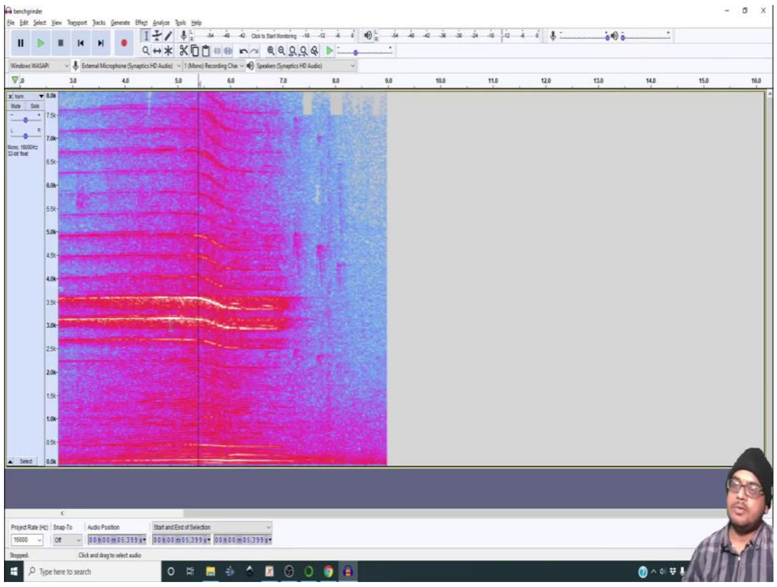


(Refer Slide Time: 14:19)



So, this is a slight increase in the frequency and then suddenly there is a fall over here there is a falling frequency and this is like the transition point where the vehicle has moved past us. So, now the question is all this is quite fine why are there multiple lines ok?

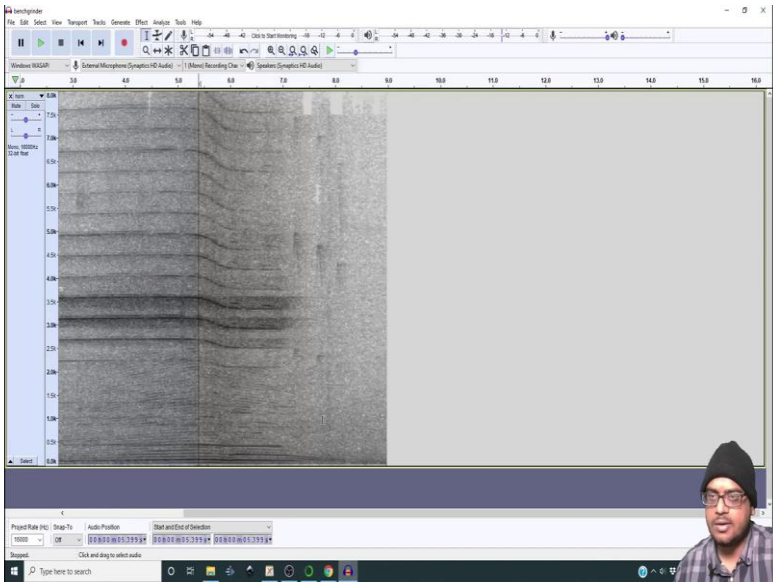
(Refer Slide Time: 14:51)



This is a very very valid question why are there multiple lines I mean there appear to be a bunch of lines this line over here, here, here, here, here, here it is like octaves all over the place. And the reason is quite simple the reason is the horn is not emitting a single frequency sound it is emitting a bunch of overtones, that is the reason why you have such a variety of peaks and apparently the horn on my vehicle peaks between these two frequencies with something like 3.2k and this 3.6k nice shrill sound.

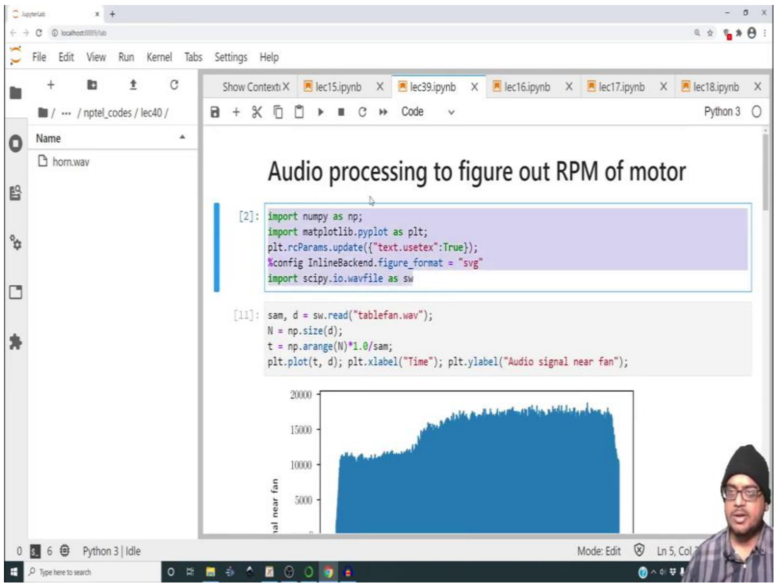
So, the horn is expected to have a nice shrill sound the shiller the sound the a faster people will move away from it that is why you get this multiple like peaks in between ok, let me see if I can change the color map.

(Refer Slide Time: 16:01)



The grayscale makes it look better, but anyway you get the point. So, let us see whether we can do all this in Python right.

(Refer Slide Time: 16:20)



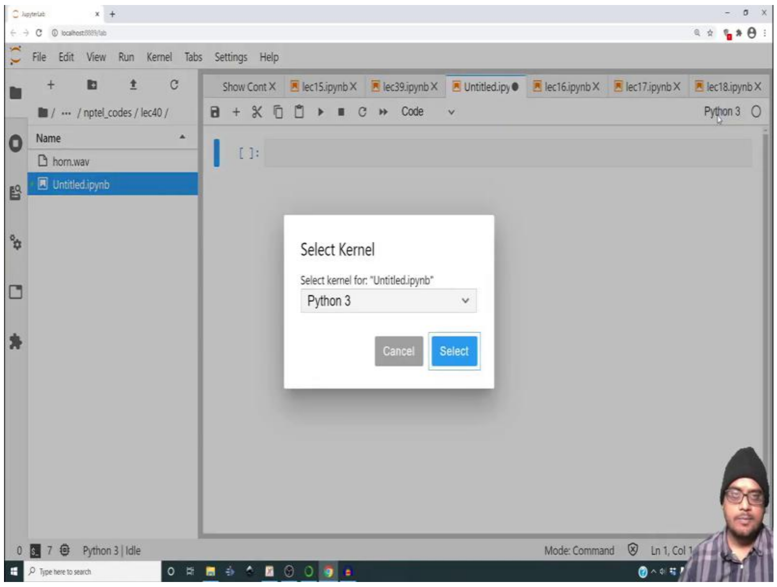
Audio processing to figure out RPM of motor

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

```
[11]: sam, d = sw.read("tablefan.wav");
      N = np.size(d);
      t = np.arange(N)*1.0/sam;
      plt.plot(t, d); plt.xlabel("Time"); plt.ylabel("Audio signal near fan");
```

So, let me go to Python let me go to the folder which contains horn dot wave so, let me copy this because we are going to need it anyway so, let me create a new file Python 3.

(Refer Slide Time: 16:30)

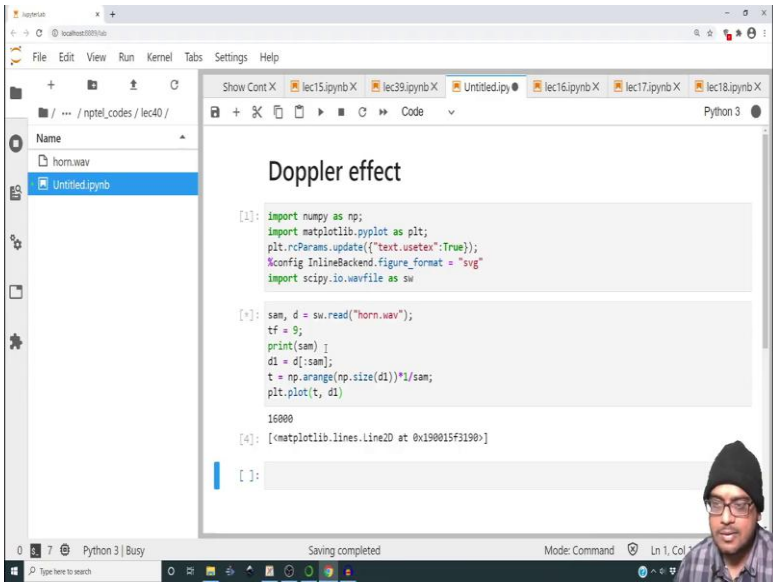


Select Kernel

Select kernel for: "Untitled.ipynb"

Python 3

(Refer Slide Time: 16:33)



Doppler effect

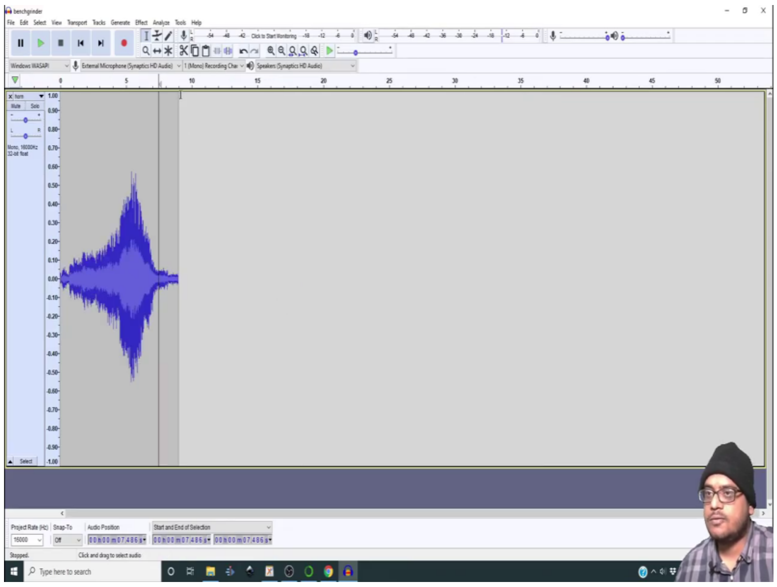
```
[1]: import numpy as np;
     import matplotlib.pyplot as plt;
     plt.rcParams.update({"text.usetex":True});
     %config InlineBackend.figure_format = "svg"
     import scipy.io.wavfile as sw
```

```
[*]: sam, d = sw.read("horn.wav");
     tf = 9;
     print(sam)
     d1 = d[:sam];
     t = np.arange(np.size(d1))*1/sam;
     plt.plot(t, d1)
```

```
16000
[4]: [<matplotlib.lines.Line2D at 0x190015f3190>]
```

So, let us declare it as Doppler effect so, let me execute this part of course because, we are going to need it anyway then we will have sam, d = sw.read("horn.wav") alright. So, let me plot the first 13 seconds of it so, let me go back to audacity let me go back to the waveform.

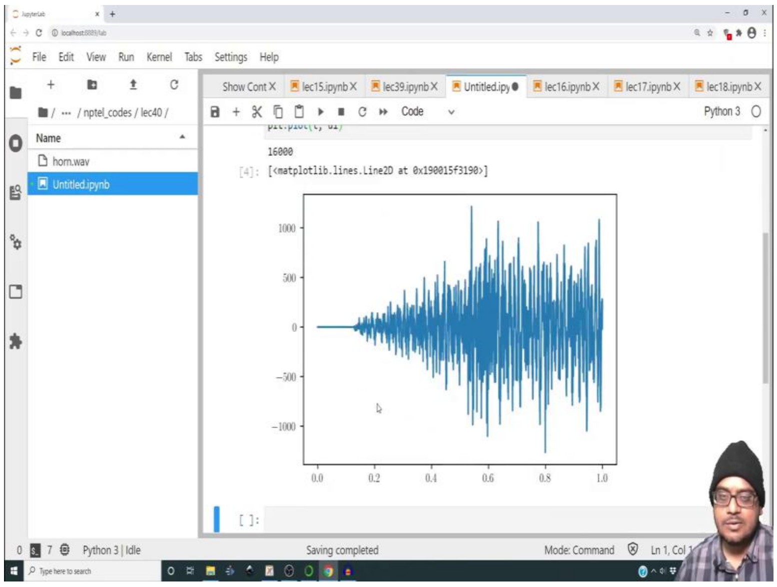
(Refer Slide Time: 17:14)



So, the waveform says that we are done with the entire process at around what is the time around 9 seconds. So, t_final will be 9, but if t_final is 9 how many point should we have look per second there is going to be 16,000 samples, how do I know that? Let me print sam because, sam is the number of samples per second so, it is 16,000. Meaning, in the data I need to take only data starting from 0 to 16,000 if I want to represent only.

1 second; meaning, if I say d1 = d[:sam]. Now, let me print let me plot plt.plot(t, d) where t = np.arange(np.size(d))*1/sam. So, this means, whatever d1 is that particular length I am creating an array of 0, 1, 2, 3 whatever and multiplying it by 1/sam because, in 1 second I have 16,000 samples. So, the Δt is going to be 1/16,000 alright. So, let me plot this so this is going to be a plot of 1 second.

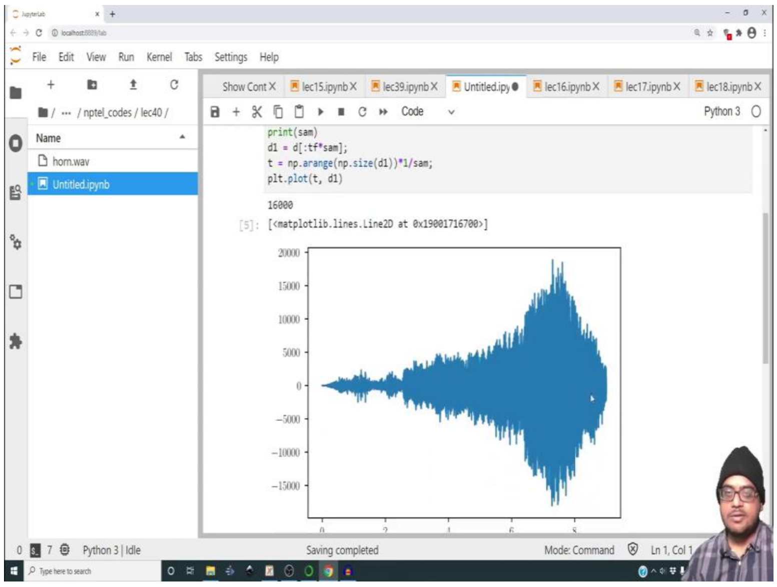
(Refer Slide Time: 19:08)



```
16000
[4]: [<matplotlib.lines.Line2D at 0x190015f3190>]
```

So, this is the plot of 1 second great. What I want to do is plot the first 9 seconds meaning,

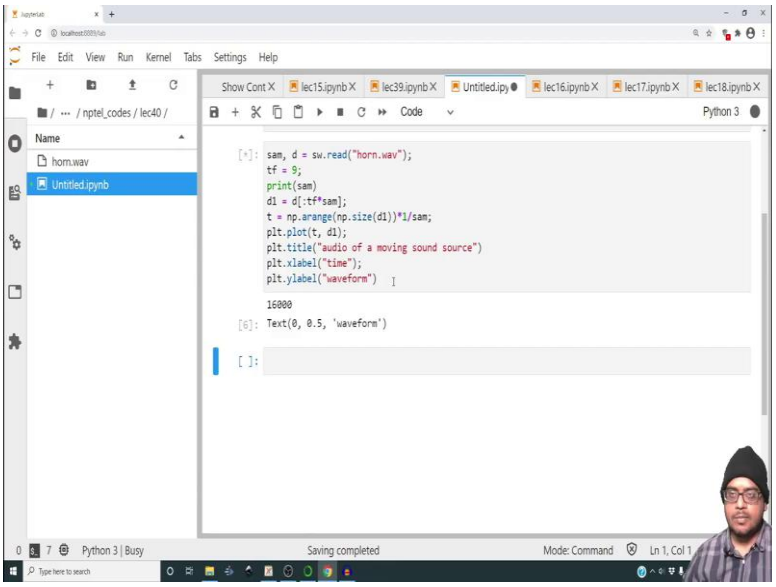
(Refer Slide Time: 19:20)



```
print(sam)
d1 = d[:tf*sam];
t = np.arange(np.size(d1))*1/sam;
plt.plot(t, d1)

16000
[5]: [<matplotlib.lines.Line2D at 0x19001716700>]
```

I am going to do 9 times or tf *sample and when I run this I should have the first 9 seconds of the video of the audio.

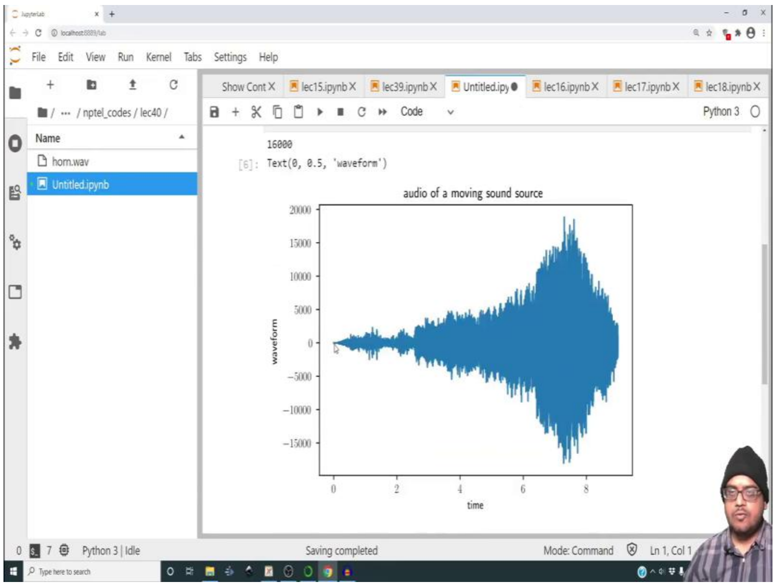
(Refer Slide Time: 19:33)



```
[*]: sam, d = sw.read("horn.wav");
     tf = 9;
     print(sam)
     d1 = d[:tf*sam];
     t = np.arange(np.size(d1))*1/sam;
     plt.plot(t, d1);
     plt.title("audio of a moving sound source")
     plt.xlabel("time");
     plt.ylabel("waveform")

     16000
[6]: Text(0, 0.5, 'waveform')

[ ]:
```

So, let me just give a title plt.title “audio of a moving sound source” alright, plt.xlabel("time"), plt.ylabel("waveform") alright.

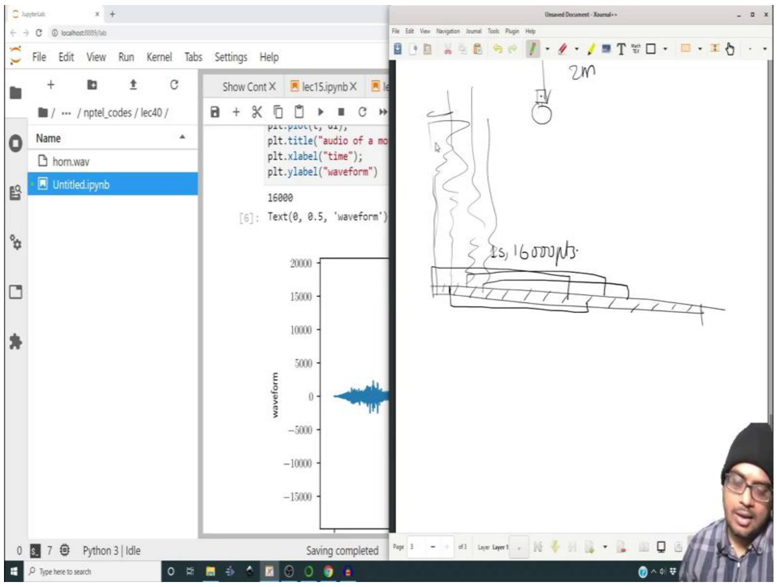
(Refer Slide Time: 20:10)



```
     16000
[6]: Text(0, 0.5, 'waveform')
```

So, what I would like to do now is to make a spectrogram out of it right. So, ideally I could have done this entire task of finding this out so, how would you do it if you were to do it by hand.

(Refer Slide Time: 20:34)



```
plt.plot(t, d1);
plt.title("audio of a mo
plt.xlabel("time");
plt.ylabel("waveform")
```

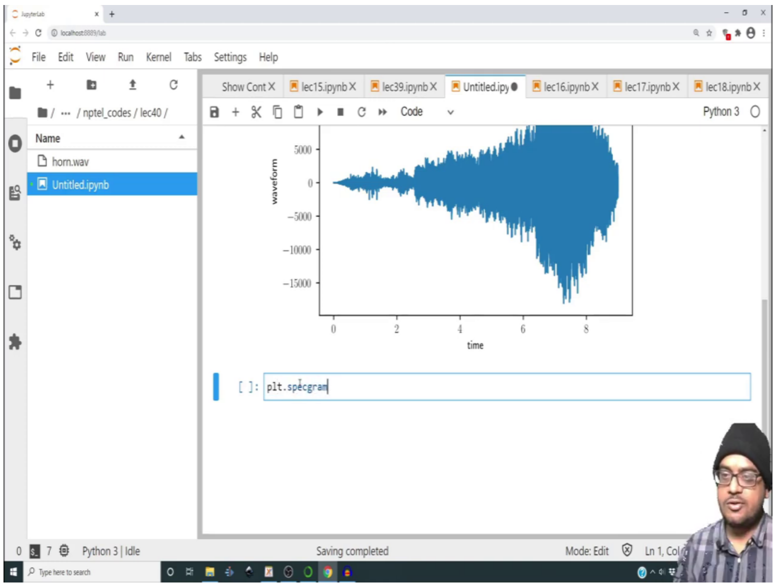
```
16000
```

```
[6]: Text(0, 0.5, 'waveform')
```

So, you know that you have a bunch of array points which represent your data and you take this point to this point say this is maybe over 1 second of sampling meaning your sampling 16,000 points. And then you take the Fourier transform it assign it to this time then, you take this to this then you assign it to this point, then this to this and assign it to this point, then this to this assign it to this point.

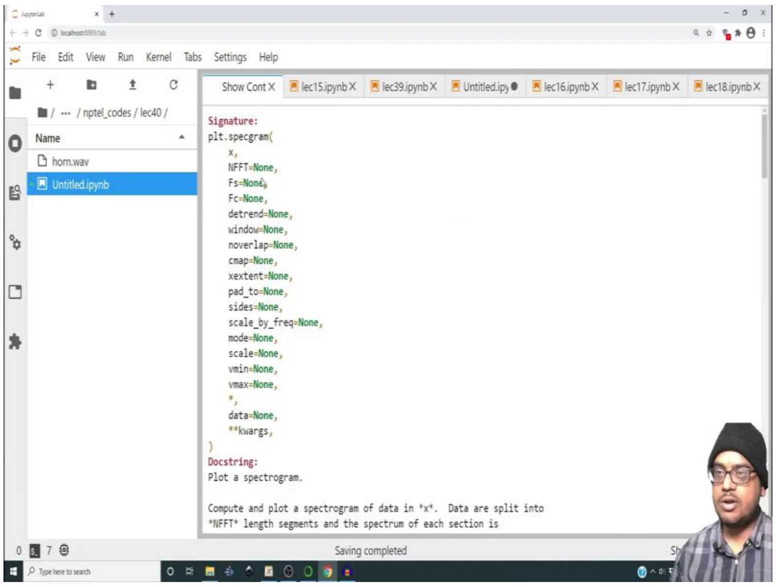
So, you are keeping the number of discrete points in the discrete Fourier transform fixed at 16,000 or whatever it is you can change; and then with the help of that you are finding out the discrete Fourier transform and assigning it to that time you could do that and then, you have that entire map based on the amplitude you assign that pixel or color and you can do it. But Python thankfully allows you to avoid doing all that and there is a inbuilt function called as spec specgram.

(Refer Slide Time: 21:45)



```
[ ]: plt.specgram
```

(Refer Slide Time: 21:50)

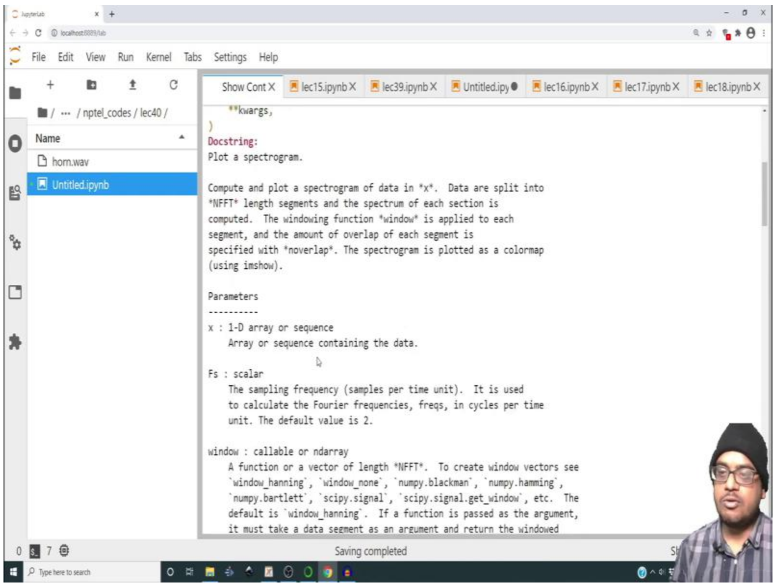


```
Signature:
plt.specgram(
    x,
    NFFT=None,
    Fs=None,
    Fc=None,
    detrend=None,
    window=None,
    noverlap=None,
    cmap=None,
    xextent=None,
    pad_to=None,
    sides=None,
    scale_by_freq=None,
    mode=None,
    scale=None,
    vmin=None,
    vmax=None,
    *,
    data=None,
    **kwargs,
)
Docstring:
Plot a spectrogram.

Compute and plot a spectrogram of data in *x*.  Data are split into
*NFFT* length segments and the spectrum of each section is
```

So, the name of the function is specgram. So, let me show you the contextual help.

Student: (Refer Time: 21:51).

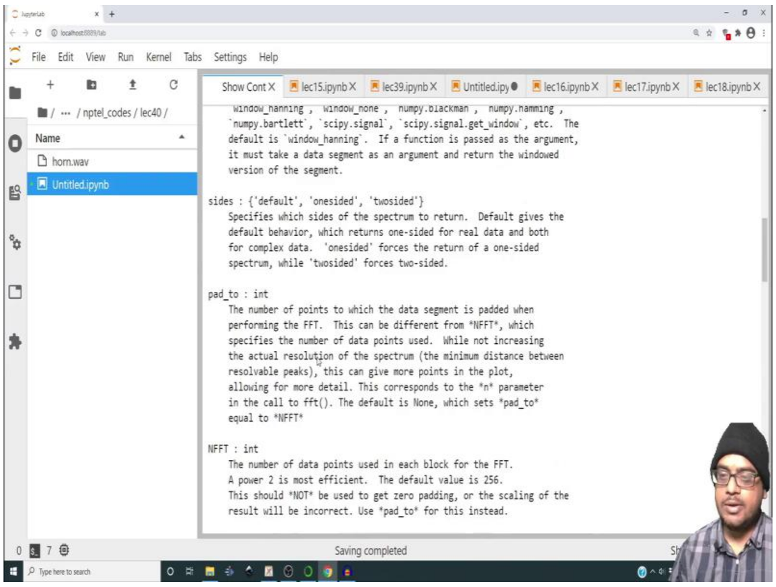
(Refer Slide Time: 21:58)



```
    **kwargs,
)
Docstring:
Plot a spectrogram.

Compute and plot a spectrogram of data in *x*.  Data are split into
*NFFT* length segments and the spectrum of each section is
computed.  The windowing function *window* is applied to each
segment, and the amount of overlap of each segment is
specified with *noverlap*. The spectrogram is plotted as a colormap
(using imshow).

Parameters
----------
x : 1-D array or sequence
    Array or sequence containing the data.

Fs : scalar
    The sampling frequency (samples per time unit).  It is used
    to calculate the Fourier frequencies, freqs, in cycles per time
    unit. The default value is 2.

window : callable or ndarray
    A function or a vector of length *NFFT*.  To create window vectors see
    `window_hanning`, `window_none`, `numpy.blackman`, `numpy.hamming`,
    `numpy.bartlett`, `scipy.signal`, `scipy.signal.get_window`, etc.  The
    default is `window_hanning`.  If a function is passed as the argument,
    it must take a data segment as an argument and return the windowed
```

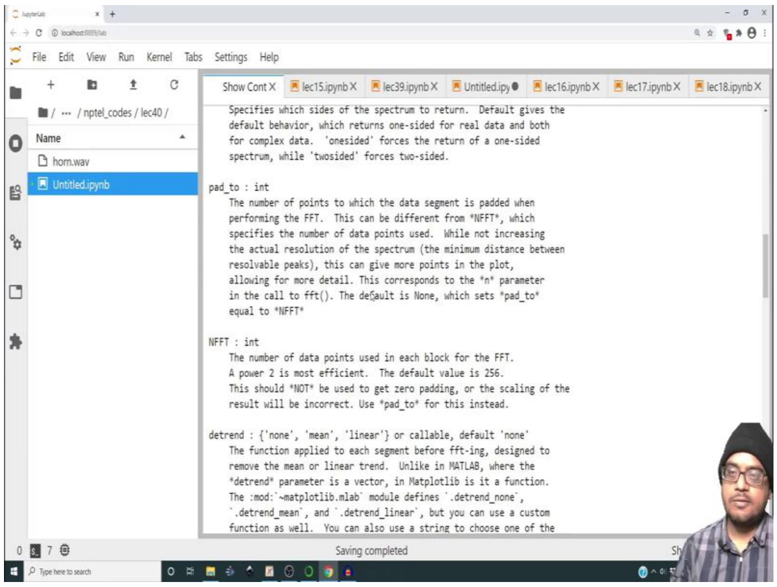
So, it is you given x that is the signal the NFFT you do not need to bother with that number of line segments then, you have the F s which stands for a sampling frequency in our case it is samples per time unit, which is going to be simply sam alright. And there is a whole lot of options, but you do not need to bother about all those for the simple case.

(Refer Slide Time: 22:18)



```
    `window_hanning`, `window_none`, `numpy.blackman`, `numpy.hamming`,
    `numpy.bartlett`, `scipy.signal`, `scipy.signal.get_window`, etc.  The
    default is `window_hanning`.  If a function is passed as the argument,
    it must take a data segment as an argument and return the windowed
    version of the segment.

sides : {'default', 'onesided', 'twosided'}
    Specifies which sides of the spectrum to return.  Default gives the
    default behavior, which returns one-sided for real data and both
    for complex data.  'onesided' forces the return of a one-sided
    spectrum, while 'twosided' forces two-sided.

pad_to : int
    The number of points to which the data segment is padded when
    performing the FFT.  This can be different from *NFFT*, which
    specifies the number of data points used.  While not increasing
    the actual resolution of the spectrum (the minimum distance between
    resolvable peaks), this can give more points in the plot,
    allowing for more detail. This corresponds to the *n* parameter
    in the call to fft(). The default is None, which sets *pad_to*
    equal to *NFFT*

NFFT : int
    The number of data points used in each block for the FFT.
    A power 2 is most efficient.  The default value is 256.
    This should *NOT* be used to get zero padding, or the scaling of the
    result will be incorrect. Use *pad_to* for this instead.
```

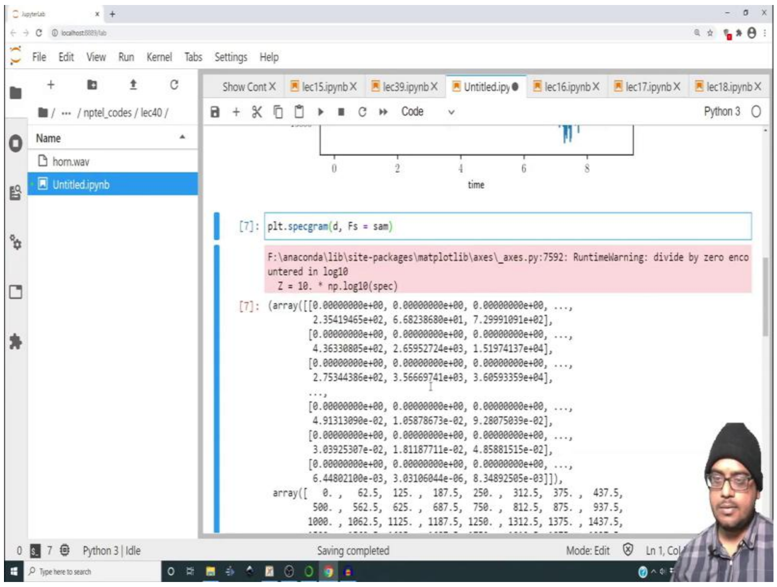
(Refer Slide Time: 22:22)



```
    Specifies which sides of the spectrum to return.  Default gives the
    default behavior, which returns one-sided for real data and both
    for complex data.  'onesided' forces the return of a one-sided
    spectrum, while 'twosided' forces two-sided.

pad_to : int
    The number of points to which the data segment is padded when
    performing the FFT.  This can be different from *NFFT*, which
    specifies the number of data points used.  While not increasing
    the actual resolution of the spectrum (the minimum distance between
    resolvable peaks), this can give more points in the plot,
    allowing for more detail. This corresponds to the *n* parameter
    in the call to fft(). The default is None, which sets *pad_to*
    equal to *NFFT*

NFFT : int
    The number of data points used in each block for the FFT.
    A power 2 is most efficient.  The default value is 256.
    This should *NOT* be used to get zero padding, or the scaling of the
    result will be incorrect. Use *pad_to* for this instead.

detrend : {'none', 'mean', 'linear'} or callable, default 'none'
    The function applied to each segment before fft-ing, designed to
    remove the mean or linear trend.  Unlike in MATLAB, where the
    *detrend* parameter is a vector, in Matplotlib is it a function.
    The :mod:`~matplotlib.mlab` module defines `.detrend_none`,
    `.detrend_mean`, and `.detrend_linear`, but you can use a custom
    function as well.  You can also use a string to choose one of the
```

Once you start doing complicated problems, you can sort of determine what these values are going to be for this particular case we will simply call it with x and Fs.

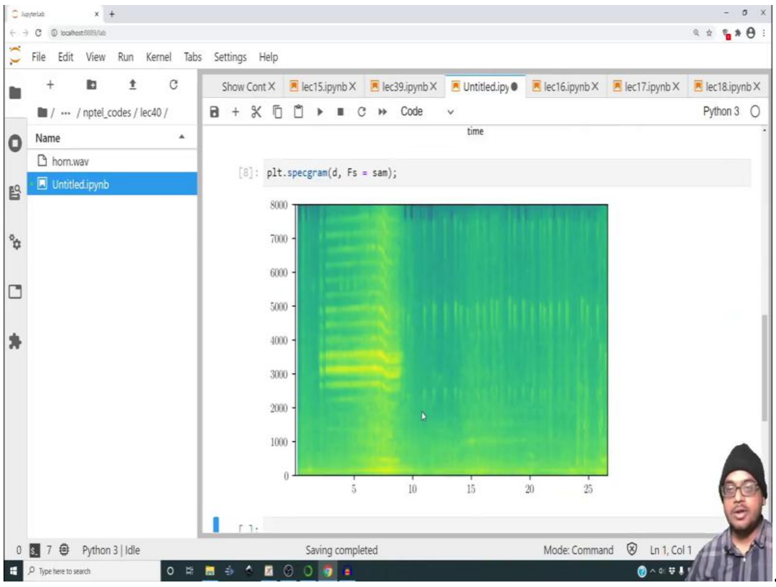
(Refer Slide Time: 22:34)



```
[7]: plt.specgram(d, Fs = sam)

F:\anaconda\lib\site-packages\matplotlib\axes\_axes.py:7592: RuntimeWarning: divide by zero enco
untered in log10
  Z = 10. * np.log10(spec)

[7]: (array([[0.00000000e+00, 0.00000000e+00, 0.00000000e+00, ...,
         2.35419465e+02, 6.68238680e+01, 7.29991091e+02],
        [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, ...,
         4.36330805e+02, 2.65952724e+03, 1.51974137e+04],
        [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, ...,
         2.75344386e+02, 3.56669741e+03, 3.60593359e+04],
        ...,
        [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, ...,
         4.91313090e-02, 1.05878673e-02, 9.28075039e-02],
        [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, ...,
         3.03925307e-02, 1.81187711e-02, 4.85881515e-02],
        [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, ...,
         6.44802100e-03, 3.03106044e-06, 8.34892505e-03]]),
  array([   0. ,   62.5,  125. ,  187.5,  250. ,  312.5,  375. ,  437.5,
          500. ,  562.5,  625. ,  687.5,  750. ,  812.5,  875. ,  937.5,
         1000. , 1062.5, 1125. , 1187.5, 1250. , 1312.5, 1375. , 1437.5,
```

So, specgram the data is going to be d Fs equal to sam and let see ok.

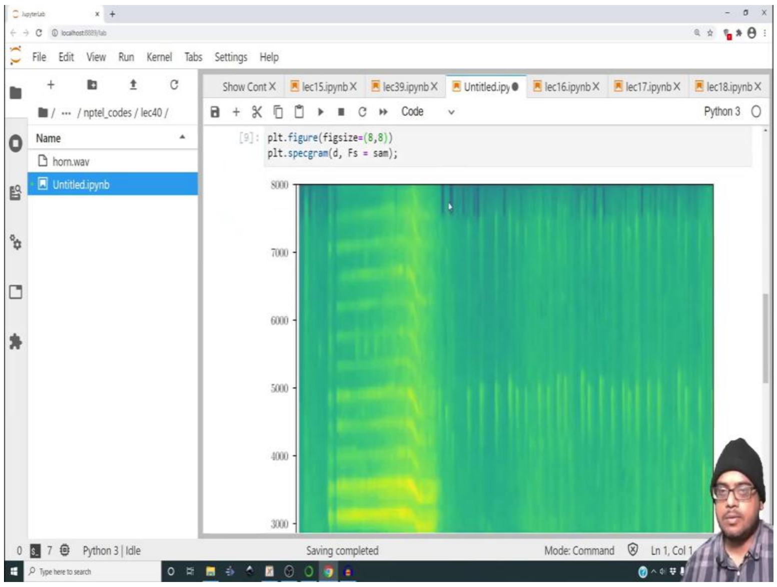
(Refer Slide Time: 22:48)



```
[8]: plt.specgram(d, Fs = sam);
```

There is a bunch of return values there you go this is the plot that we were looking for this is the exact plot we were able to make using audacity.

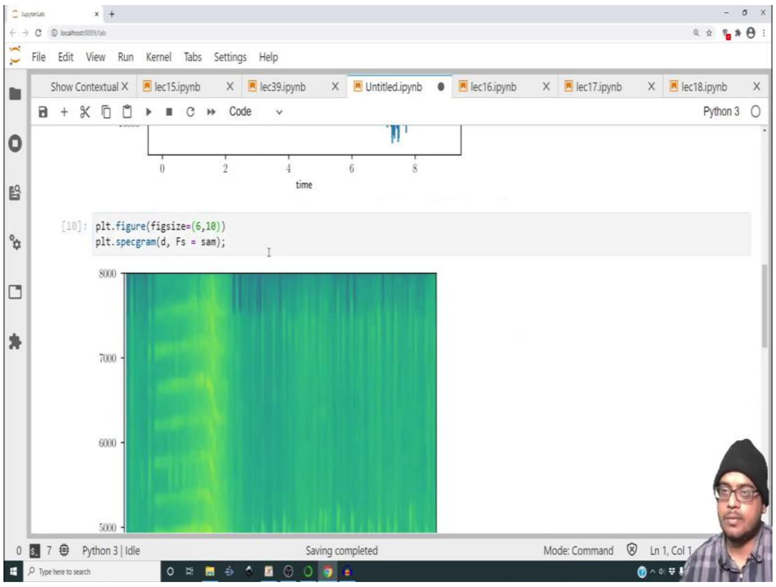
(Refer Slide Time: 23:12)



```
[9]: plt.figure(figsize=(8,8))
     plt.specgram(d, Fs = sam);
```

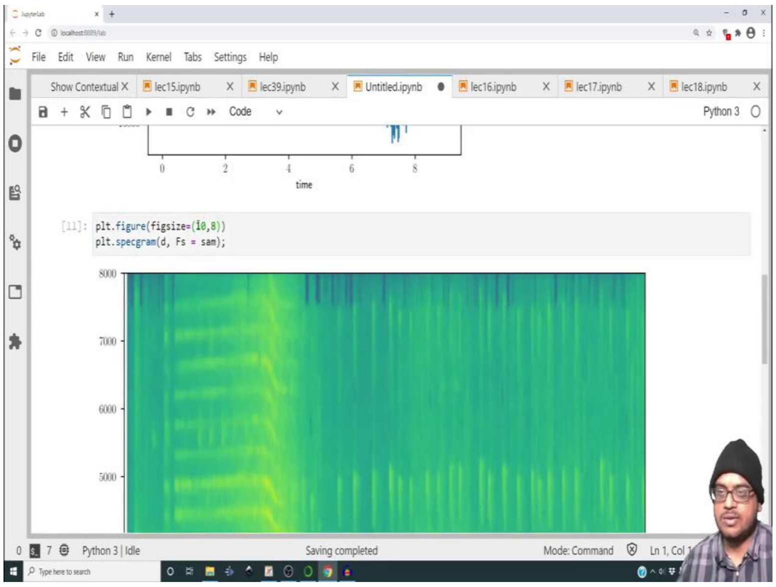
So, let me just increase the size of the figure maybe this is good alright.

(Refer Slide Time: 23:32)



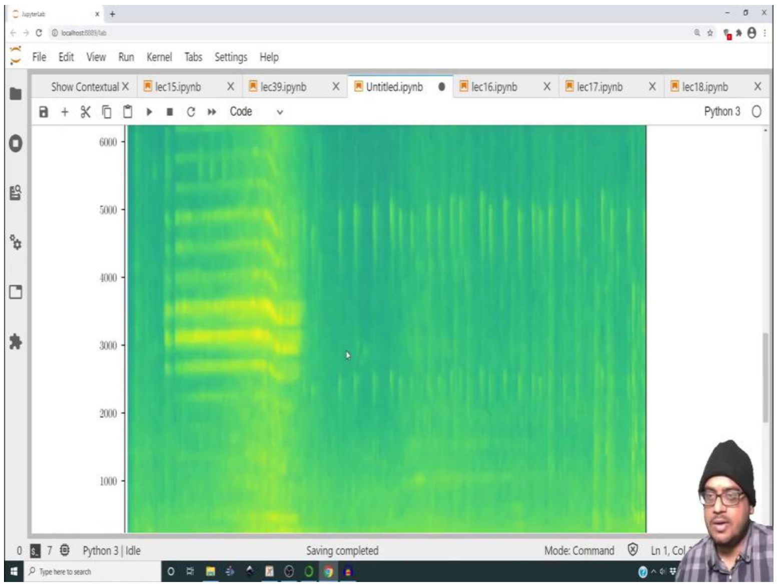
```
[10]: plt.figure(figsize=(6,10))
      plt.specgram(d, Fs = sam);
```

(Refer Slide Time: 23:38)



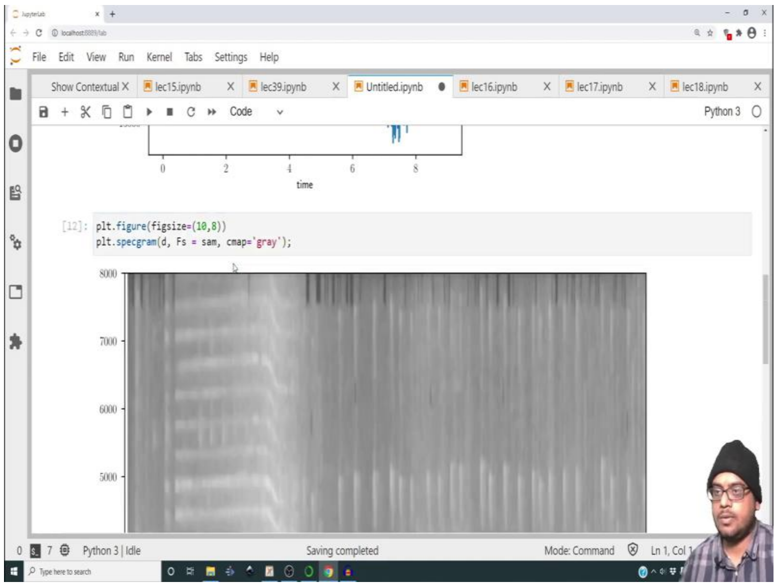
```
[11]: plt.figure(figsize=(10,8))
      plt.specgram(d, Fs = sam);
```

(Refer Slide Time: 23:43)



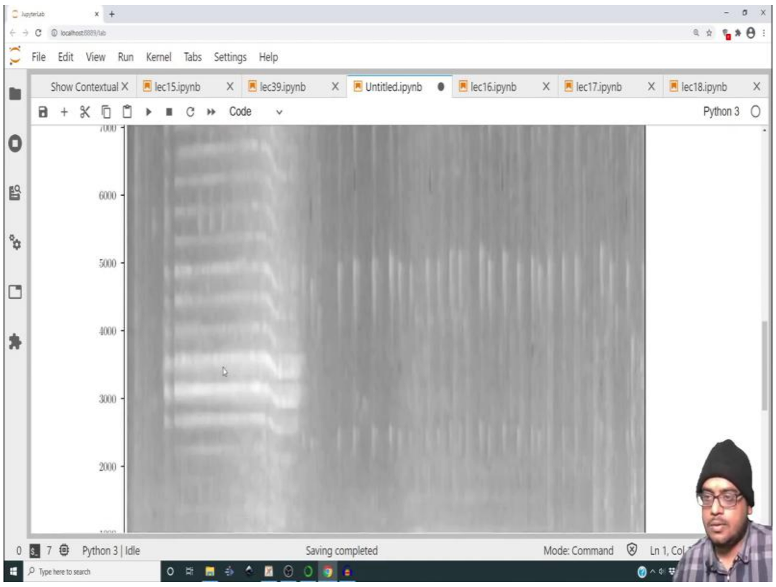
So, this is just adjusting the image size and let me make the color map as gray alright.

(Refer Slide Time: 23:47)



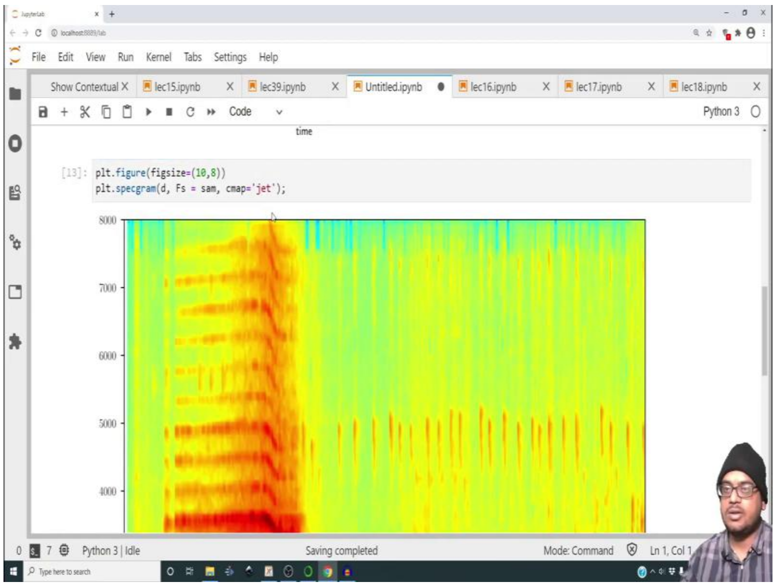
```
[12]: plt.figure(figsize=(10,8))
      plt.specgram(d, Fs = sam, cmap='gray');
```

(Refer Slide Time: 23:51)



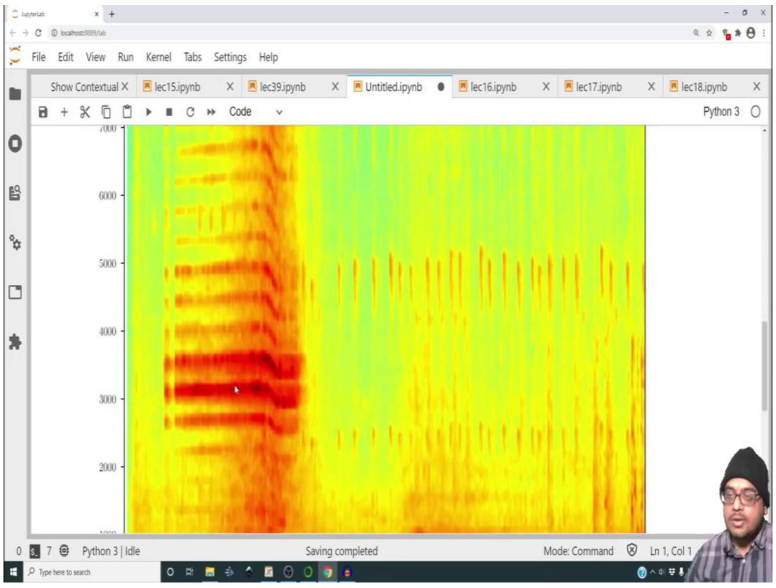
So, this is where that transition as well maybe gray is not the best we will make it jet alright.

(Refer Slide Time: 23:57)



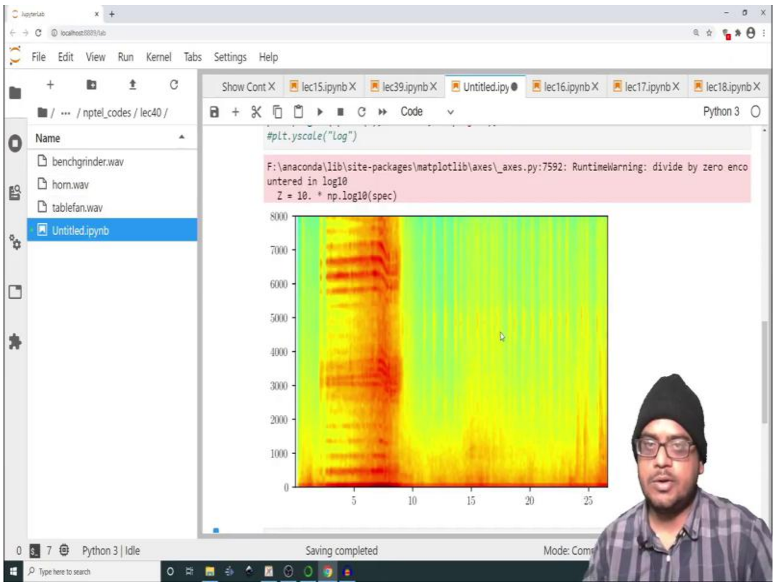
```
[13]: plt.figure(figsize=(10,8))
      plt.specgram(d, Fs = sam, cmap='jet');
```

(Refer Slide Time: 23:59)



So, this is where that transition occurs and it occurs at this particular frequency. It also shows the presence of multiple frequencies because, the horn is not just a single function not a single tone it is a mixture of multiple tones. In fact, let me drop the audio files that I had in the previous video the bench grinder and the table fan as well just to show how their spectrogram also look like so, let me change the ok.

(Refer Slide Time: 24:40)



```
#plt.yscale("log")
```

```
F:\anaconda\lib\site-packages\matplotlib\axes\_axes.py:7592: RuntimeWarning: divide by zero enco
untered in log10
  Z = 10. * np.log10(spec)
```

So, this is how the entire plot looks like and well I have I am hoping you have learned something new and this kind of things are incredibly useful and this is not just useful for analyzing audio files, but it is useful for analyzing any kind of signal that you may get maybe from hotwire anemometry, in case you are studying turbulence you put a probe somewhere and you measure the signal you can find out all sorts of information using the spectrogram.

At which a dominant frequencies you are achieving your signal that can give you a lot of idea in final spectrum and you can find a lot of things ok. This is very useful a very useful tool to know so, with this I conclude this particular lecture, and I will see you next time with a new topic on image processing until then have a nice day bye.