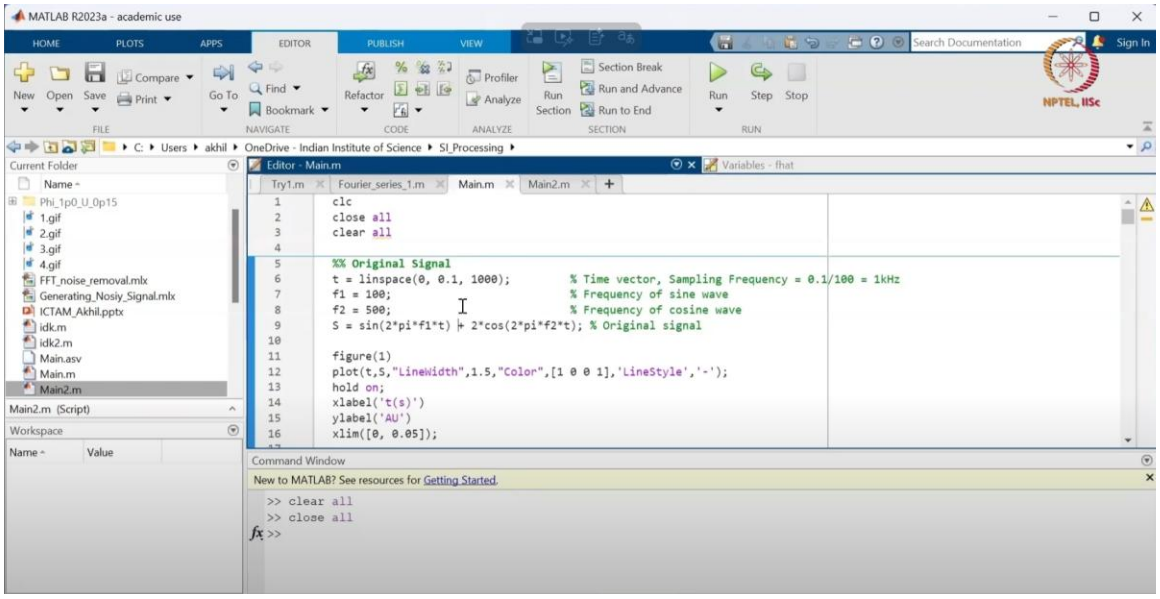
Advanced Measurement Techniques in Fluid Mechanics and Heat Transfer
Prof. Saptarshi Basu
Department of Mechanical Engineering
Indian Institute of Science, Bengaluru
Week – 04
Lecture - 20

Experimental Signal Processing in MATLAB

So, this is a demonstration to show how to use tools such as FFT to process our experimental data. Since this is a demonstration, I won't be using an actual experimental signal since we want to compare our process data with the ground truth. So we'll first generate a synthetic ground truth and then add noise to it to create simple experimental data. And then we'll proceed from there. So to start off, this is the signal I'm going to generate. I'm going to assume that this is my ground truth.

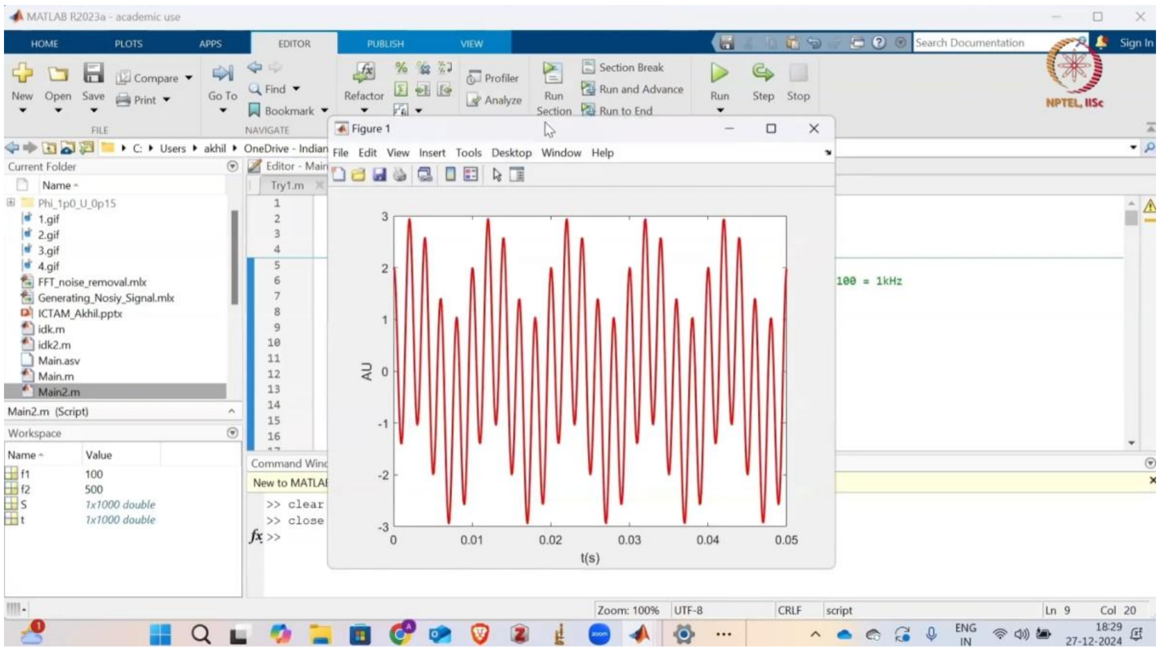
So, this is basically a combination of a sine wave and a cosine wave. The sine signal has a frequency of 100 Hz (F1), and the cosine signal has a frequency of 500 Hz. The sine signal has an amplitude of 1, and the cosine signal has an amplitude of 2. So, let us look at how this ground truth, how the actual signal looks.



```
clc
close all
clear all

%% Original Signal
t = linspace(0, 0.1, 1000);          % Time vector, Sampling Frequency = 0.1/100 = 1kHz
f1 = 100;                            % Frequency of sine wave
f2 = 500;                            % Frequency of cosine wave
S = sin(2*pi*f1*t) + 2*cos(2*pi*f2*t); % Original signal

figure(1)
plot(t,S,"LineWidth",1.5,"Color",[1 0 0 1],'LineStyle','-');
hold on;
xlabel('t(s)')
ylabel('AU')
xlim([0, 0.05]);
```

```
>> clear all
>> close all
>>
```

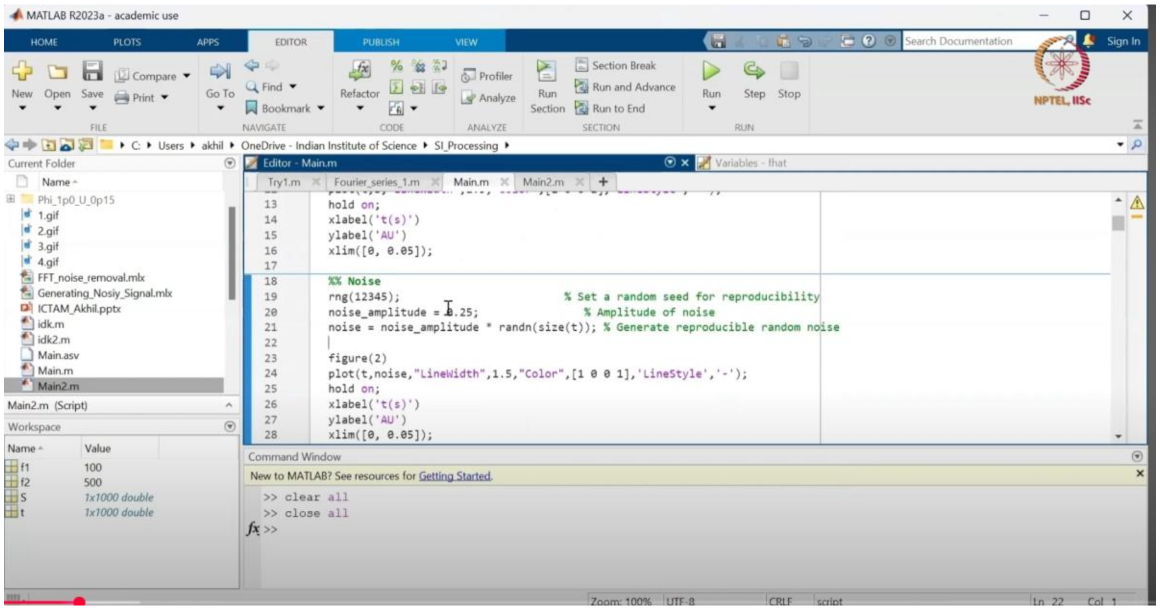
Let us run this section, and this is what we get. So this is how the ground truth looks. So this is the original signal. So this can be a representative of anything. It can represent how pressure variations may be occurring at a particular point in the flow field.



```
>> clear
>> close
```

It might be representative of a velocity. It might be a representation of the position of an object that you're tracking. It can be anything. To sense these parameters, we can use different visualization techniques or sensing techniques to capture the data. And we'll again end up with a time series signal, as we saw before.

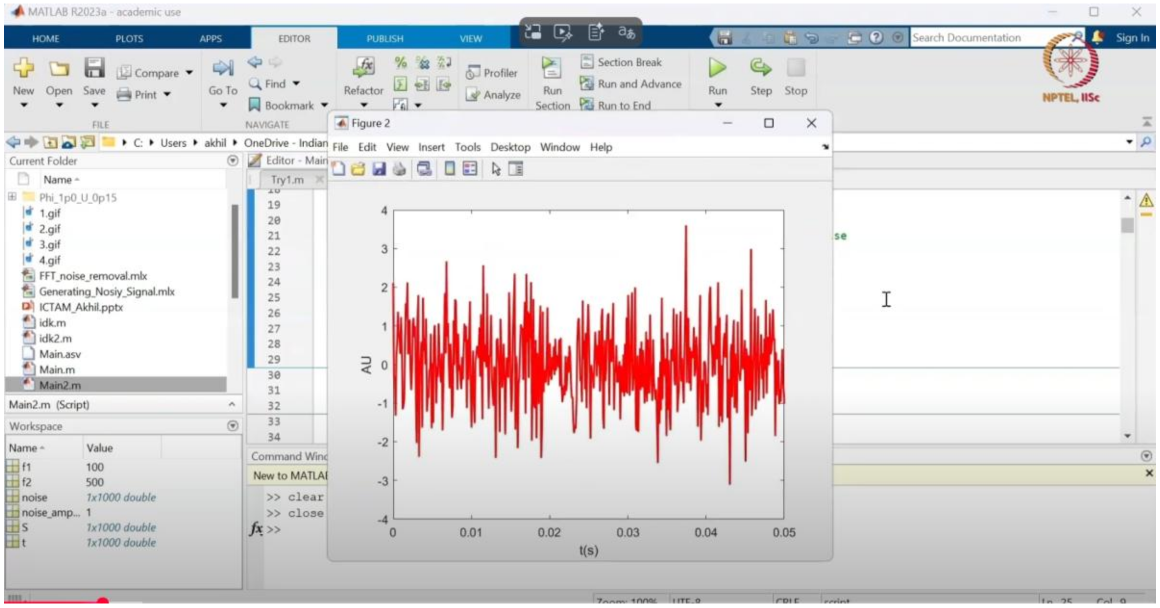
But the data that we get from the experiments won't be clean like the signal that we see here. They'll be associated with a lot of noise. In most cases in experiments, the noise is Gaussian in nature. What that means is that if we plot the histogram of the noise alone, it will end up having a Gaussian profile that is centered around zero. And to replicate this noise here, what we're going to do is create Gaussian noise.



```
hold on;
xlabel('t(s)')
ylabel('AU')
xlim([0, 0.05]);

%% Noise
rng(12345);                         % Set a random seed for reproducibility
noise_amplitude = 0.25;                % Amplitude of noise
noise = noise_amplitude * randn(size(t)); % Generate reproducible random noise

figure(2)
plot(t,noise,"LineWidth",1.5,"Color",[1 0 0 1],'LineStyle','-');
hold on;
xlabel('t(s)')
ylabel('AU')
xlim([0, 0.05]);
```

```
New to MATLAB? See resources for Getting Started.
>> clear all
>> close all
```

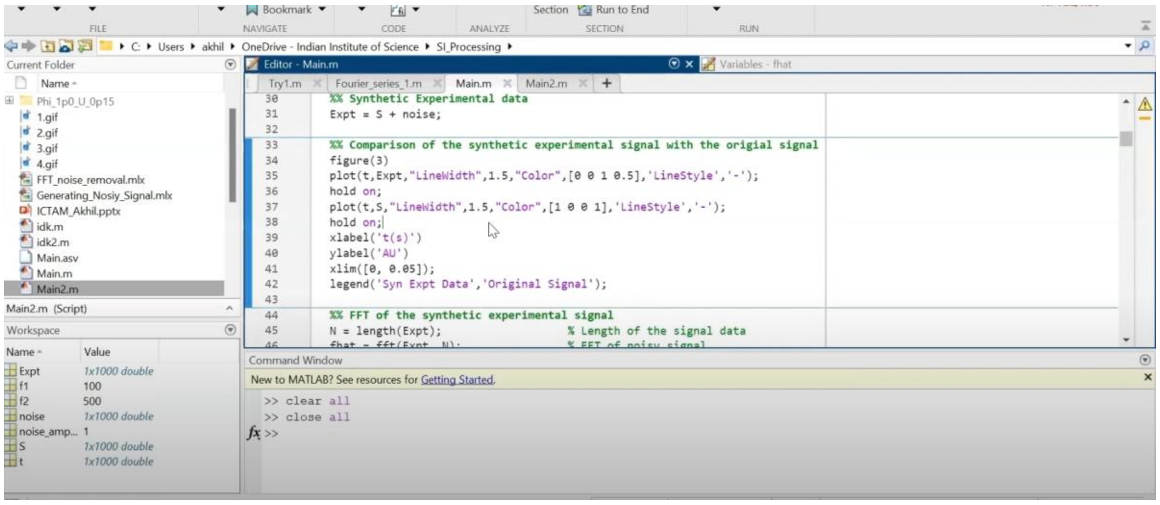
And that is created by this RAND function. And again, the spread of the Gaussian noise is determined by this noise amplitude. Here I am going to take a noise amplitude of 1 so that the noise becomes comparable to the signal itself. So let us generate the noise first. And now what we are going to do is that we are going to add this noise to the signal.



Name	Value
f1	100
f2	500
noise	1x1000 double
noise_amp...	1
S	1x1000 double
t	1x1000 double

```
>> clear
>> close
>>
```

And let us look at what this is like. Yeah, now the plot in blue actually shows a signal plus the noise added. And this is how an experimental signal typically looks. It will be associated with many spurious oscillations. And so this is a representative of an experimental signal that we have and that we are going to proceed with for the rest of the analysis.

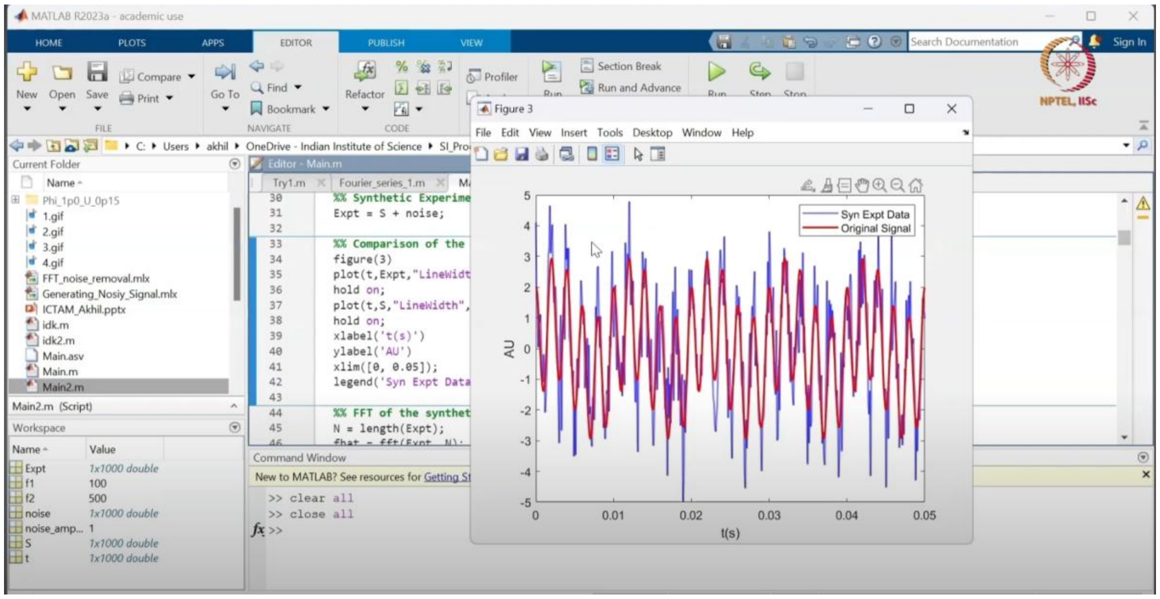


```
%% Synthetic Experimental data
Expt = S + noise;

%% Comparison of the synthetic experimental signal with the origial signal
figure(3)
plot(t,Expt,"LineWidth",1.5,"Color",[0 0 1 0.5],'LineStyle','-');
hold on;
plot(t,S,"LineWidth",1.5,"Color",[1 0 0 1],'LineStyle','-');
hold on;
xlabel('t(s)')
ylabel('AU')
xlim([0, 0.05]);
legend('Syn Expt Data','Original Signal');

%% FFT of the synthetic experimental signal
N = length(Expt);                    % Length of the signal data
```

Name	Value
Expt	1x1000 double
f1	100
f2	500
noise	1x1000 double
noise_amp...	1
S	1x1000 double
t	1x1000 double

New to MATLAB? See resources for Getting Started.

```
>> clear all
>> close all
>>
```

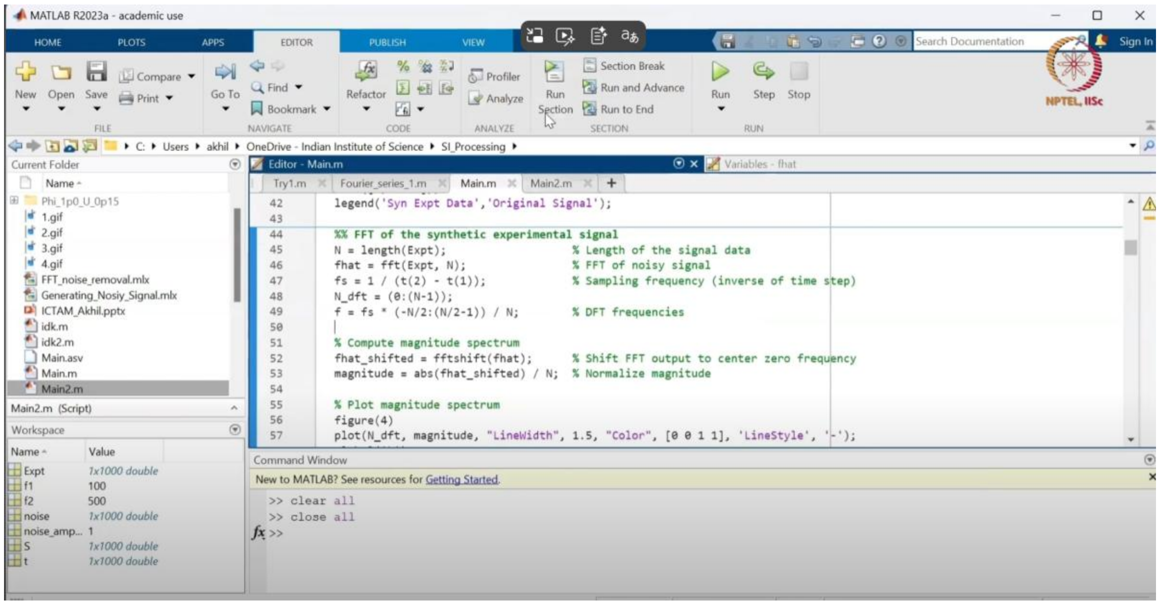


```
%% Synthetic Experime
Expt = S + noise;

%% Comparison of the
figure(3)
plot(t,Expt,"LineWidt
hold on;
plot(t,S,"LineWidth",
hold on;
xlabel('t(s)')
ylabel('AU')
xlim([0, 0.05]);
legend('Syn Expt Data

%% FFT of the synthet
N = length(Expt);
```

Name	Value
Expt	1x1000 double
f1	100
f2	500
noise	1x1000 double
noise_amp...	1
S	1x1000 double
t	1x1000 double

```
>> clear all
>> close all
>>
```

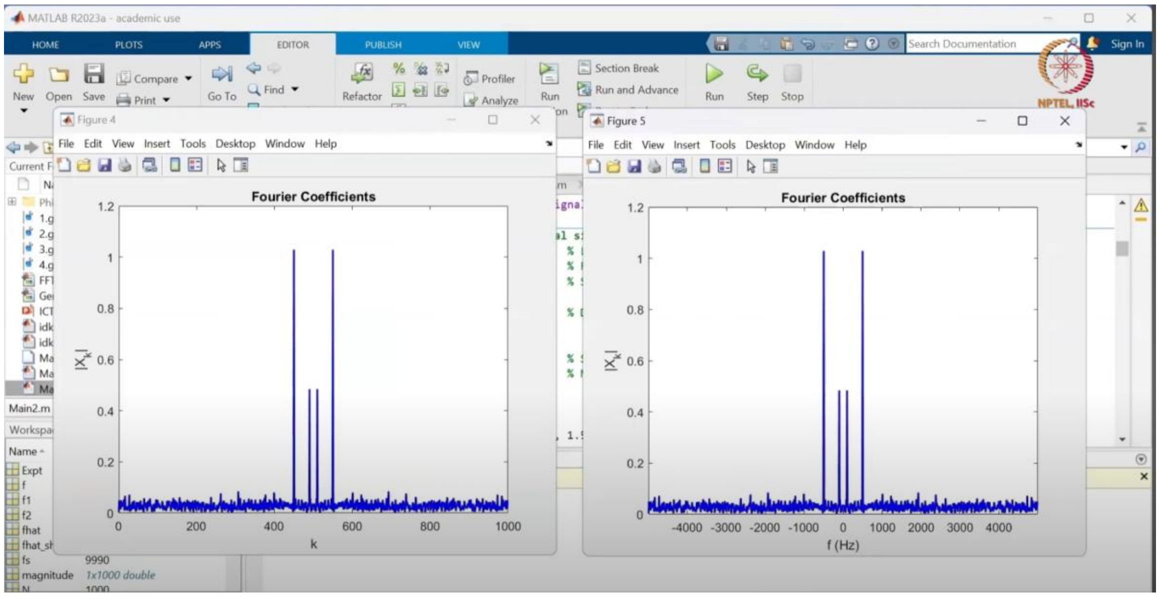
We are going to call this the synthetic experimental data for now. We will first start with doing an FFT over the synthetic experimental data. The reason we are going to do FFT is so that we can identify what the dominant frequencies associated with our synthetic experimental data are. Let us run this part of the code. So now we have obtained the variations of the Fourier coefficients.



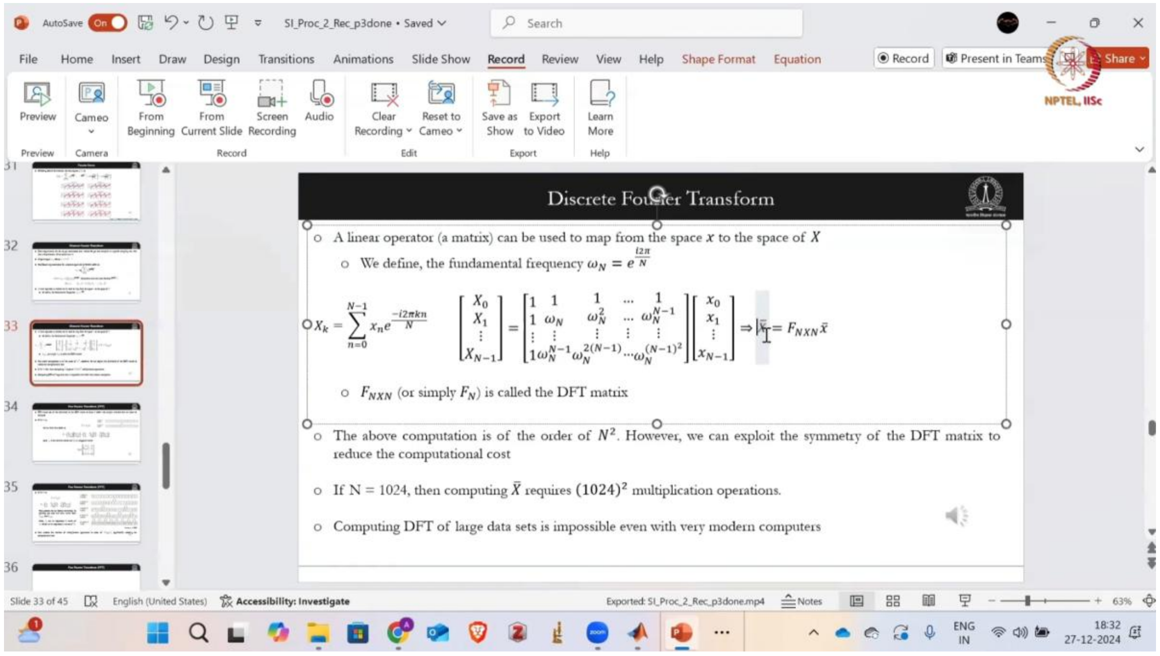
```
legend('Syn Expt Data','Original Signal');

%% FFT of the synthetic experimental signal
N = length(Expt);                   % Length of the signal data
fhat = fft(Expt, N);                % FFT of noisy signal
fs = 1 / (t(2) - t(1));             % Sampling frequency (inverse of time step)
N_dft = (0:(N-1));
f = fs * (-N/2:(N/2-1)) / N;        % DFT frequencies

% Compute magnitude spectrum
fhat_shifted = fftshift(fhat);      % Shift FFT output to center zero frequency
magnitude = abs(fhat_shifted) / N;  % Normalize magnitude

% Plot magnitude spectrum
figure(4)
plot(N_dft, magnitude, "LineWidth", 1.5, "Color", [0 0 1 1], 'LineStyle', '-');
```

Name	Value
Expt	1x1000 double
f1	100
f2	500
noise	1x1000 double
noise_amp...	1
S	1x1000 double
t	1x1000 double

```
>> clear all
>> close all
>>
```



So what we have estimated here is, let me just pull up the PPT. So, in the previous lecture, we saw that we need to estimate these coefficients X_K, which basically show the projection of my data vector onto the basis functions. So the FFT function in MATLAB essentially does that. It takes the experimental data, the data vector, and n, which is basically the total number of points in the data vector, as input. And then it estimates the capital X_K values.



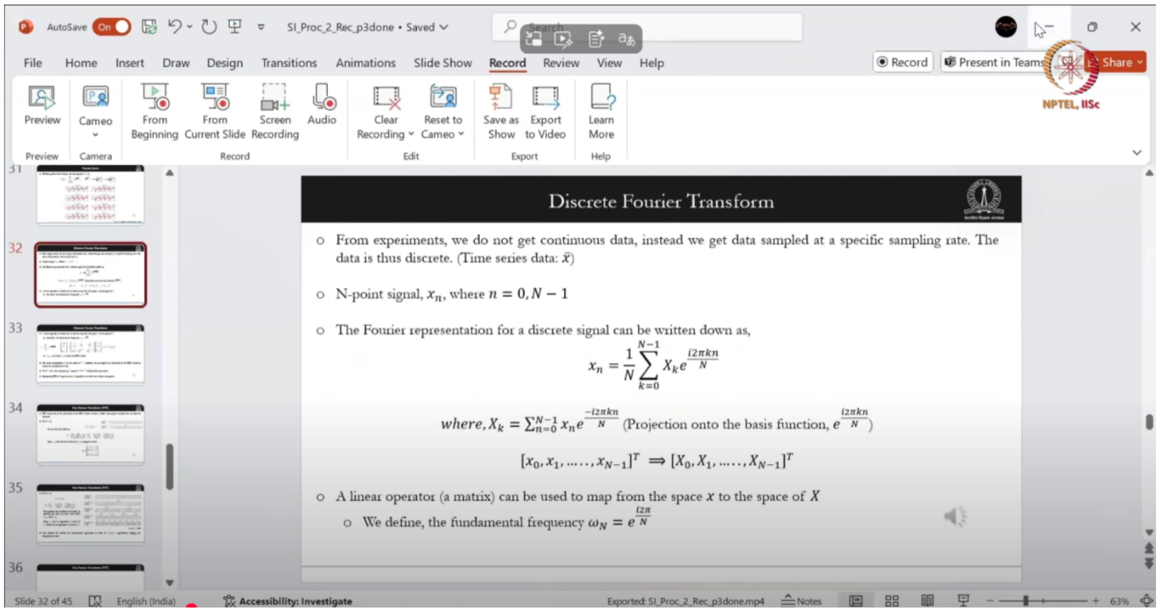
Discrete Fourier Transform

- A linear operator (a matrix) can be used to map from the space x to the space of X
  - We define, the fundamental frequency $\omega_N = e^{\frac{i2\pi}{N}}$

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-i2\pi kn}{N}} \qquad \begin{bmatrix} X_0 \\ X_1 \\ \vdots \\ X_{N-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_N & \omega_N^2 & \cdots & \omega_N^{N-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega_N^{N-1} & \omega_N^{2(N-1)} & \cdots & \omega_N^{(N-1)^2} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{bmatrix} \Rightarrow \bar{X} = F_{NXN}\bar{x}$$

  - F_{NXN} (or simply F_N) is called the DFT matrix
- The above computation is of the order of N^2. However, we can exploit the symmetry of the DFT matrix to reduce the computational cost
- If N = 1024, then computing $\bar{X}$ requires $(1024)^2$ multiplication operations.
- Computing DFT of large data sets is impossible even with very modern computers

Basically, it estimates the Fourier coefficients that correspond to different basis functions. And this is a plot of how these coefficients vary. And the value of k here again varies from 0 to n minus 1. Let us look at the value of n; n here is 1000. So, basically, the value of k goes from 0 to 999.



Discrete Fourier Transform

- From experiments, we do not get continuous data, instead we get data sampled at a specific sampling rate. The data is thus discrete. (Time series data: $\bar{x}$)
- N-point signal, x_n, where $n = 0, N-1$
- The Fourier representation for a discrete signal can be written down as,

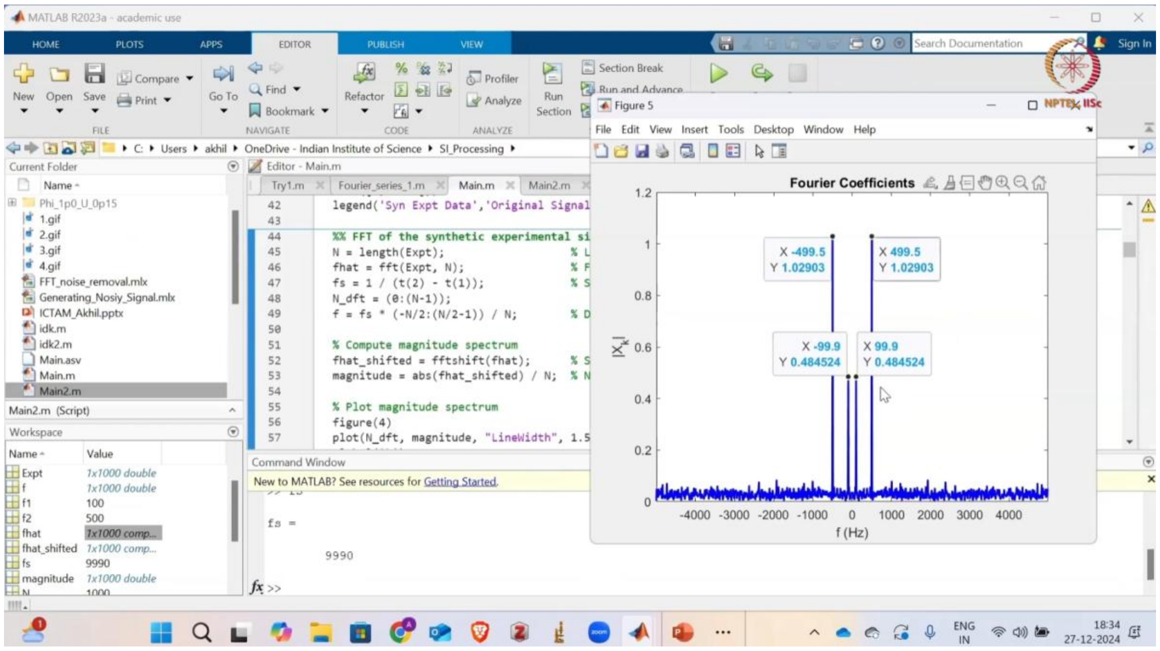
$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k e^{\frac{i2\pi kn}{N}}$$

$$where, X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-i2\pi kn}{N}} \text{ (Projection onto the basis function, } e^{\frac{i2\pi kn}{N}}\text{)}$$

$$[x_0, x_1, \ldots\ldots, x_{N-1}]^T \Rightarrow [X_0, X_1, \ldots\ldots, X_{N-1}]^T$$

- A linear operator (a matrix) can be used to map from the space x to the space of X
  - We define, the fundamental frequency $\omega_N = e^{\frac{i2\pi}{N}}$

But to actually interpret it in a physical sense, we need to convert this k in terms of frequencies, the frequencies of the waves associated with these. So we can actually mathematically show that this can be mapped to minus Fs over 2 to plus fs over 2. Fs is basically nothing but the sampling frequency. Here, the value of fs is 9990. So that basically means that this spectrum runs from minus 4,995 to plus 4,995.



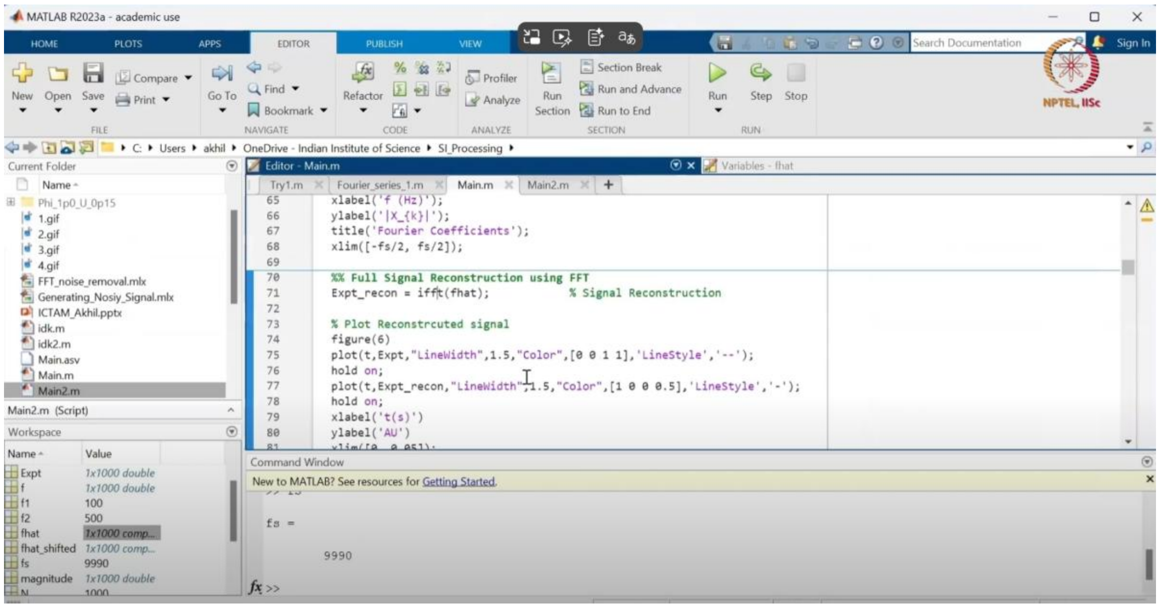
```
legend('Syn Expt Data','Original Signal

%% FFT of the synthetic experimental si
N = length(Expt);                      % L
fhat = fft(Expt, N);                   % F
fs = 1 / (t(2) - t(1));                % S
N_dft = (0:(N-1));
f = fs * (-N/2:(N/2-1)) / N;           % D

% Compute magnitude spectrum
fhat_shifted = fftshift(fhat);         % S
magnitude = abs(fhat_shifted) / N;     % N

% Plot magnitude spectrum
figure(4)
plot(N_dft, magnitude, "LineWidth", 1.5
```

```
fs =

        9990
```

Name	Value
Expt	1x1000 double
f	1x1000 double
f1	100
f2	500
fhat	1x1000 comp...
fhat_shifted	1x1000 comp...
fs	9990
magnitude	1x1000 double
N	1000

And if we just look at the spectrum, we can see that there are actually four dominant peaks. So two are associated with frequencies of 100 Hz and -100 Hz, and the two had frequencies of 500 Hz and -500 Hz. Now, if we just go into the original data from which we created our experimental signal, we will see that there are two main frequencies: 500 and 100. So again, if we look into these peaks and add up the amplitudes associated with 100 Hertz and minus 100 Hertz, they come close to one, which is basically the frequency in the original signal, which is the amplitude of the frequency in the original signal that had a frequency of 100 Hertz. And again, if we add up the amplitude that is associated with the 500 Hertz spectrum and the minus 500 Hertz frequency, we end up with a value close to 2, which is again the amplitude of the wave that was associated in the original signal with the frequency of 500 Hertz.

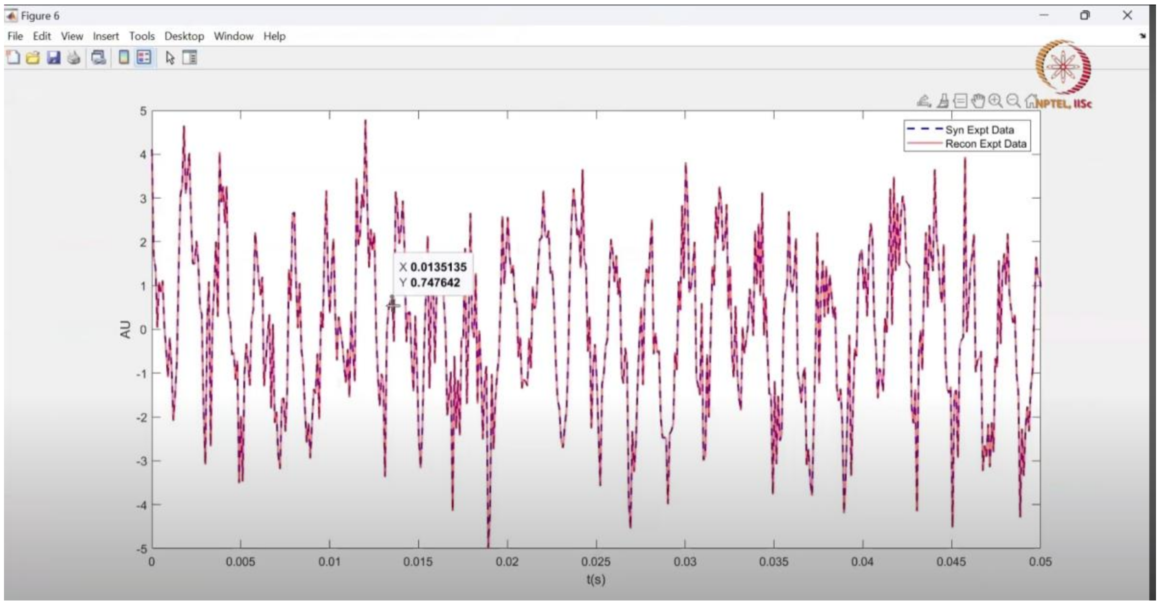
So what has happened here is that the magnitude of xk has been divided between the positive and negative spectra. We will discuss this further. later on, but first let us reconstruct the signal. Since we have already estimated how these Fourier coefficients vary, to reconstruct the signal all we need to do is multiply these Fourier coefficients with the basis functions and then sum them all up. So, that is what this function, IFFT, does, and let us look at how the reconstructed signal compares against the synthetic data.



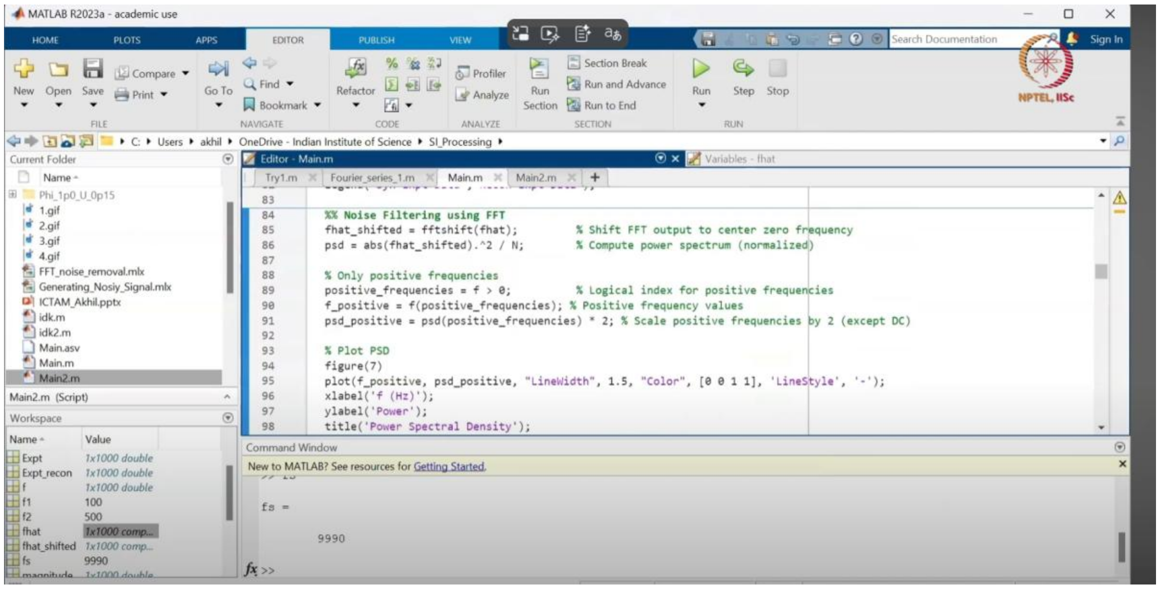
```
xlabel('f (Hz)');
ylabel('|X_{k}|');
title('Fourier Coefficients');
xlim([-fs/2, fs/2]);

%% Full Signal Reconstruction using FFT
Expt_recon = ifft(fhat);            % Signal Reconstruction

% Plot Reconstrcuted signal
figure(6)
plot(t,Expt,"LineWidth",1.5,"Color",[0 0 1 1],'LineStyle','--');
hold on;
plot(t,Expt_recon,"LineWidth",1.5,"Color",[1 0 0 0.5],'LineStyle','-');
hold on;
xlabel('t(s)')
ylabel('AU')
```

```
fs =

        9990

>>
```

Name	Value
Expt	1x1000 double
f	1x1000 double
f1	100
f2	500
fhat	1x1000 comp...
fhat_shifted	1x1000 comp...
fs	9990
magnitude	1x1000 double
N	1000



So we can see that the reconstructed signal accurately captures the synthetic experimental data that we created. And so now we can use this reconstructed signal. We can perform some filtering on this so that we can extract only the signal and remove the noise. So, that is going to be the idea. Now, to do that, we are going to first plot the power spectral density.

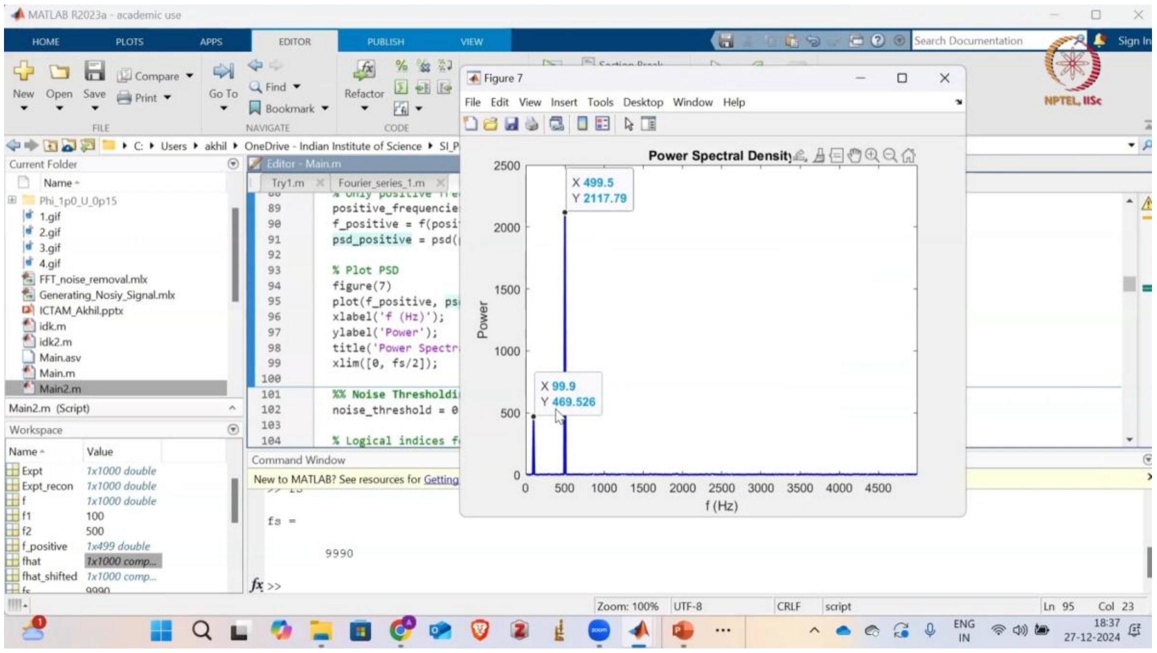


```
%% Noise Filtering using FFT
fhat_shifted = fftshift(fhat);         % Shift FFT output to center zero frequency
psd = abs(fhat_shifted).^2 / N;        % Compute power spectrum (normalized)

% Only positive frequencies
positive_frequencies = f > 0;          % Logical index for positive frequencies
f_positive = f(positive_frequencies); % Positive frequency values
psd_positive = psd(positive_frequencies) * 2; % Scale positive frequencies by 2 (except DC)

% Plot PSD
figure(7)
plot(f_positive, psd_positive, "LineWidth", 1.5, "Color", [0 0 1 1], 'LineStyle', '-');
xlabel('f (Hz)');
ylabel('Power');
title('Power Spectral Density');
```

```
fs =

        9990
```

So, what we had earlier was basically the magnitude of xk, how this magnitude of xk varies for different values of k, or how it varies over the frequency space. Now, instead of looking in terms of the X_K, let us look in terms of the power that is associated with each value of the frequency. So, to get the value of the power associated with each wave, we are going to multiply X_K by its conjugate. And this can again be mathematically shown that the power associated with a particular frequency is basically nothing but xk into the conjugate of X_K. So once we do that, we basically get this power spectral density.

And since I already told you that the energy is getting distributed here around the positive and negative spectrum, to make more sense out of the data, what we can do is transfer all the energy from the negative spectrum into the positive spectrum by basically just multiplying by a factor of two. And that is what we are doing here. Let us now plot this power spectral density. Now here we see that there are only two peaks: one that is associated with a frequency of 100 hertz and the other that is associated with a frequency of 500 hertz. These two peaks correspond to the data basically inside our original signal.

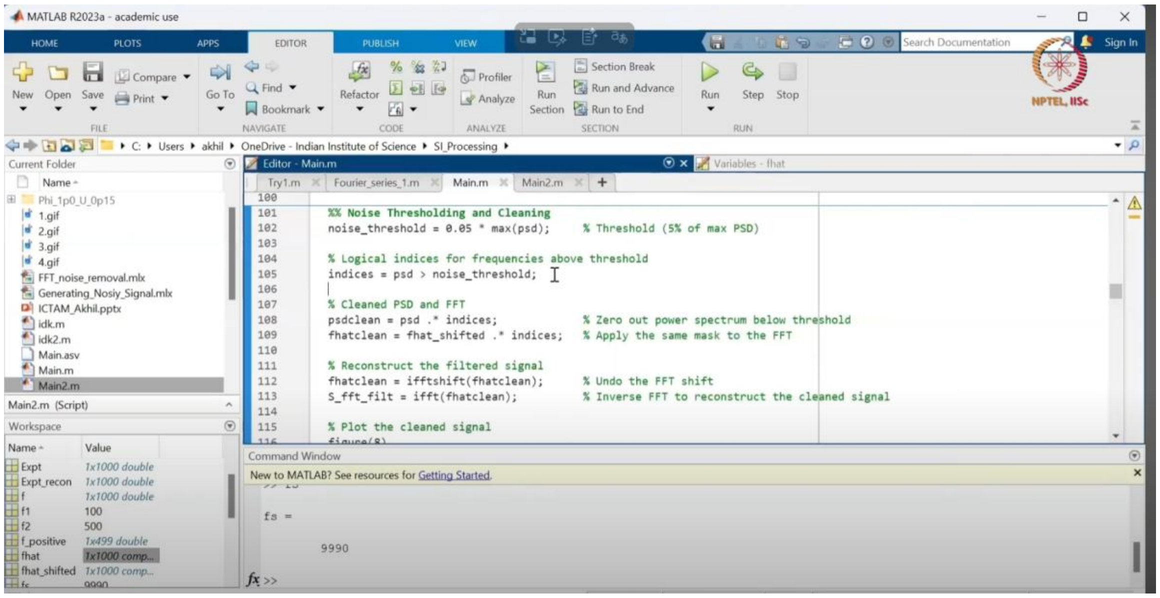


```
positive_frequencie
f_positive = f(posi
psd_positive = psd(

% Plot PSD
figure(7)
plot(f_positive, ps
xlabel('f (Hz)');
ylabel('Power');
title('Power Spectr
xlim([0, fs/2]);

%% Noise Thresholdi
noise_threshold = 0

% Logical indices f
```

```
fs =

         9990
```

So this is basically the basis for extracting the data from the noise here. So we can see that the energy associated with the data signals, which have these 100 and 500 Hertz, is very high. Whereas the energy that is associated with the noise is very, very low. The noise is not even comparable to the actual signals. So this will be the basis for separating noise from the data.

So what we are going to do is decide on a threshold. We are going to say that if the power associated with a particular frequency is greater than 5% of the maximum PSD value, and here the maximum PSD value occurs at 500 Hertz, let's look at the maximum value here. So it's around 2117. So if it is at least 5% of this value, then that signal will be used to reconstruct the filtered signal. If it is not, then we are going to discard that value.



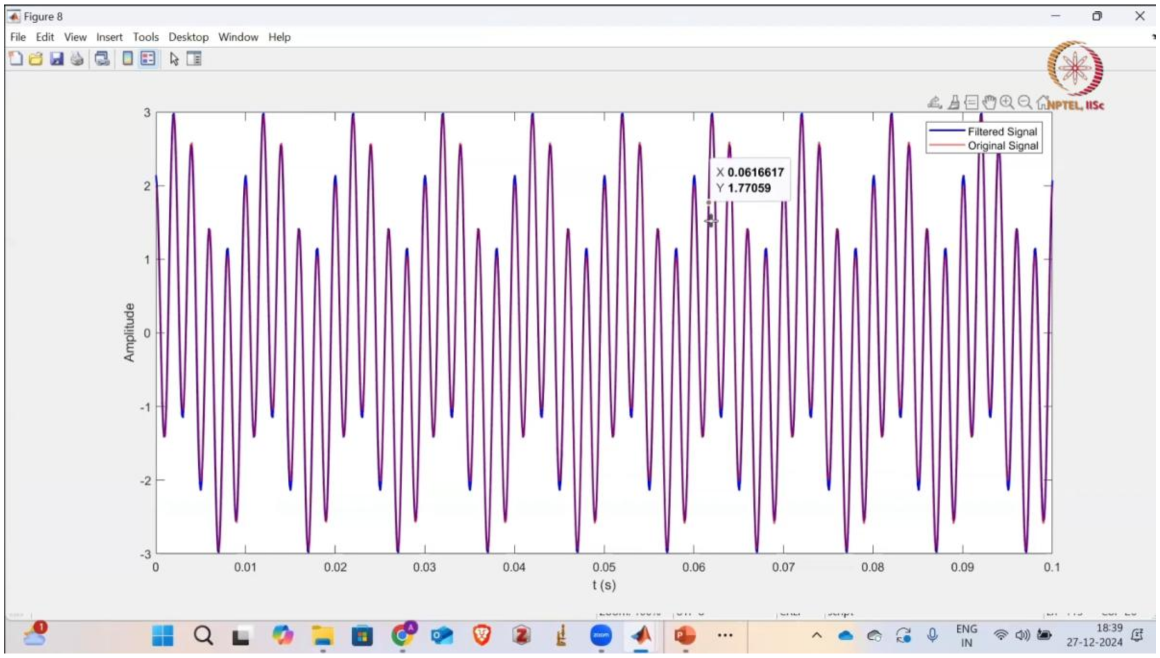
MATLAB R2023a - academic use
HOME
PLOTS
APPS
EDITOR
PUBLISH
VIEW
C: ▸ Users ▸ akhil ▸ OneDrive - Indian Institute of Science ▸ SI_Processing
Editor - Main.m
Try1.m
Fourier_series_1.m
Main.m
Main2.m
NPTEL, IISc

```
%% Noise Thresholding and Cleaning
noise_threshold = 0.05 * max(psd);     % Threshold (5% of max PSD)

% Logical indices for frequencies above threshold
indices = psd > noise_threshold;

% Cleaned PSD and FFT
psdclean = psd .* indices;             % Zero out power spectrum below threshold
fhatclean = fhat_shifted .* indices;   % Apply the same mask to the FFT

% Reconstruct the filtered signal
fhatclean = ifftshift(fhatclean);      % Undo the FFT shift
S_fft_filt = ifft(fhatclean);          % Inverse FFT to reconstruct the cleaned signal

% Plot the cleaned signal
```

Name	Value
Expt	1x1000 double
Expt_recon	1x1000 double
f	1x1000 double
f1	100
f2	500
f_positive	1x499 double
fhat	1x1000 comp...
fhat_shifted	1x1000 comp...

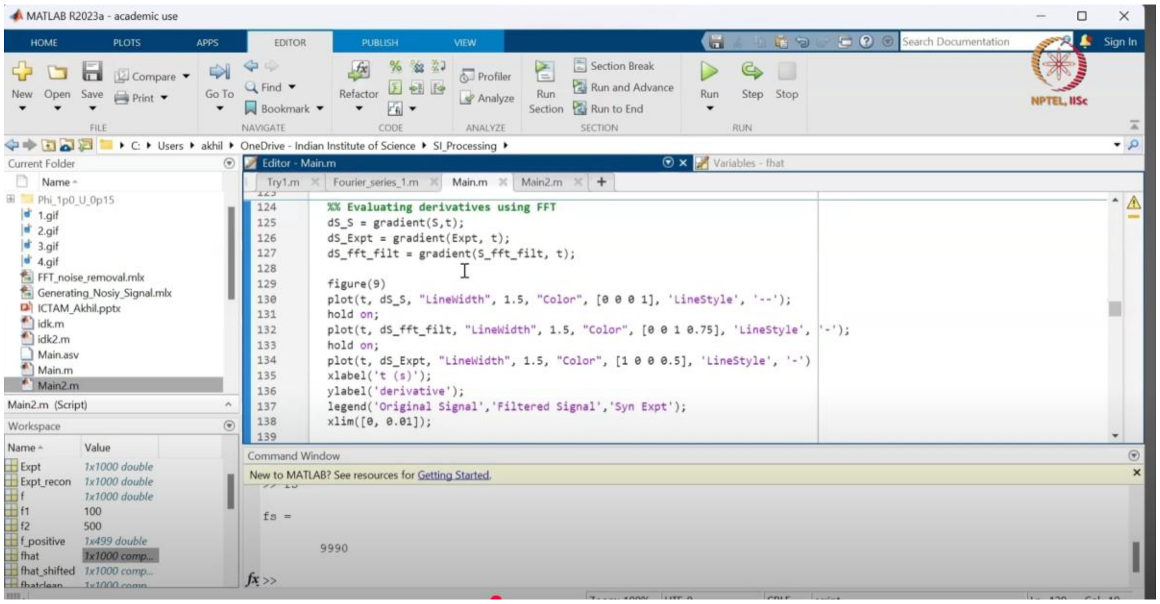
```
fs =

        9990
```

And we are going to reconstruct the signal using FFT now, based only on the dominant modes that have the highest amount of power. So once we do this thresholding, we can now reconstruct the signal. And so this is how the reconstructed signal looks. The plot is basically in the blue. So we can see that this curve compares quite well with the original, the ground truth signal from which we started.



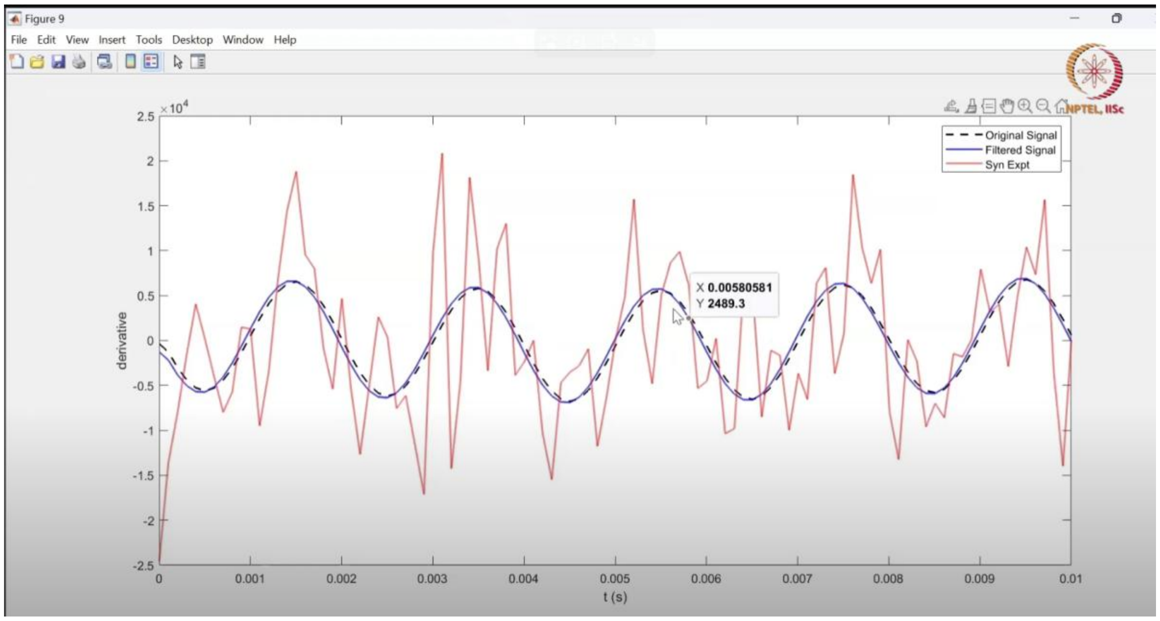
So FFT, using FFT, we have basically filtered out noise from the signal pretty effectively. And this becomes especially handy when we are estimating the derivatives. In most of our applications, estimating the derivatives of the data signals becomes very important. For example, if we are tracking, say, the position of a flame inside the channel, the flame is continuously moving, and we want to track the velocity of the flame, we are going to get the data series of the position of the flame, and we need to estimate the derivative so that we can estimate the velocity associated with it. You can always find parameters of interest where we need to estimate the derivatives of the time series signals.

Especially the estimation of derivatives becomes highly erroneous when we have a lot of noise in the data. Let us compare the gradient that we estimate from this filtered signal that we obtained after FFT filtering. And what if we estimate the gradients directly from the synthetic experimental data? So let us run this section of the code. So, we can see that the plot here in this orange basically shows the gradient estimated directly from the synthetic experimental data. We can see that it is highly erroneous compared to the actual ground truth.



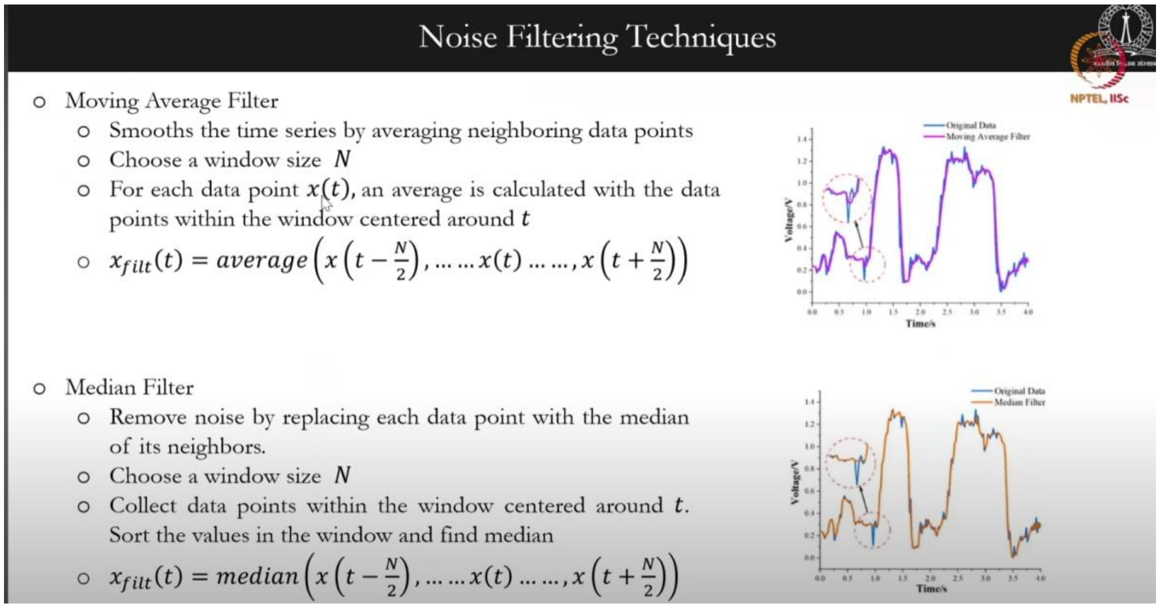
```
%% Evaluating derivatives using FFT
dS_S = gradient(S,t);
dS_Expt = gradient(Expt, t);
dS_fft_filt = gradient(S_fft_filt, t);

figure(9)
plot(t, dS_S, "LineWidth", 1.5, "Color", [0 0 0 1], 'LineStyle', '--');
hold on;
plot(t, dS_fft_filt, "LineWidth", 1.5, "Color", [0 0 1 0.75], 'LineStyle', '-');
hold on;
plot(t, dS_Expt, "LineWidth", 1.5, "Color", [1 0 0 0.5], 'LineStyle', '-')
xlabel('t (s)');
ylabel('derivative');
legend('Original Signal','Filtered Signal','Syn Expt');
xlim([0, 0.01]);
```

```
fs =

        9990
```



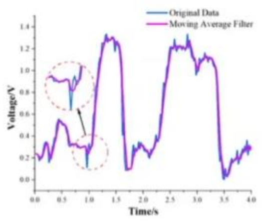
The ground truth is shown by this dashed line here, the dashed line in black. And we can see that the gradient estimated from the filtered signal actually compares quite well with the ground truth data. So FFT has very effectively filtered the noise out from the signal. We can also use other techniques to filter out data from the noise, and we are going to look into three important techniques that are used. One is the moving mean filter, the other is the median filter, and the third one is the Sawitzki-Goulet filter.

Let us take a brief look into that theory. So the way the moving mean filter works is that we first need to choose a window size parameter. We need to input a parameter called n into this filter. And basically, it creates a window that encloses n points and is centered around the point where we need the filtered value. And then it basically estimates the average of all the data points that are enclosed within that endpoint window.

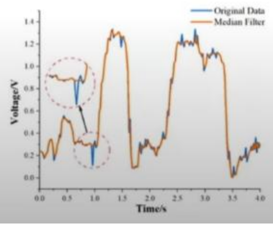


Noise Filtering Techniques

- Moving Average Filter
  - Smooths the time series by averaging neighboring data points
  - Choose a window size N
  - For each data point $x(t)$, an average is calculated with the data points within the window centered around t
  - $x_{filt}(t) = average\left(x\left(t-\frac{N}{2}\right), \dots\dots x(t) \dots\dots, x\left(t+\frac{N}{2}\right)\right)$

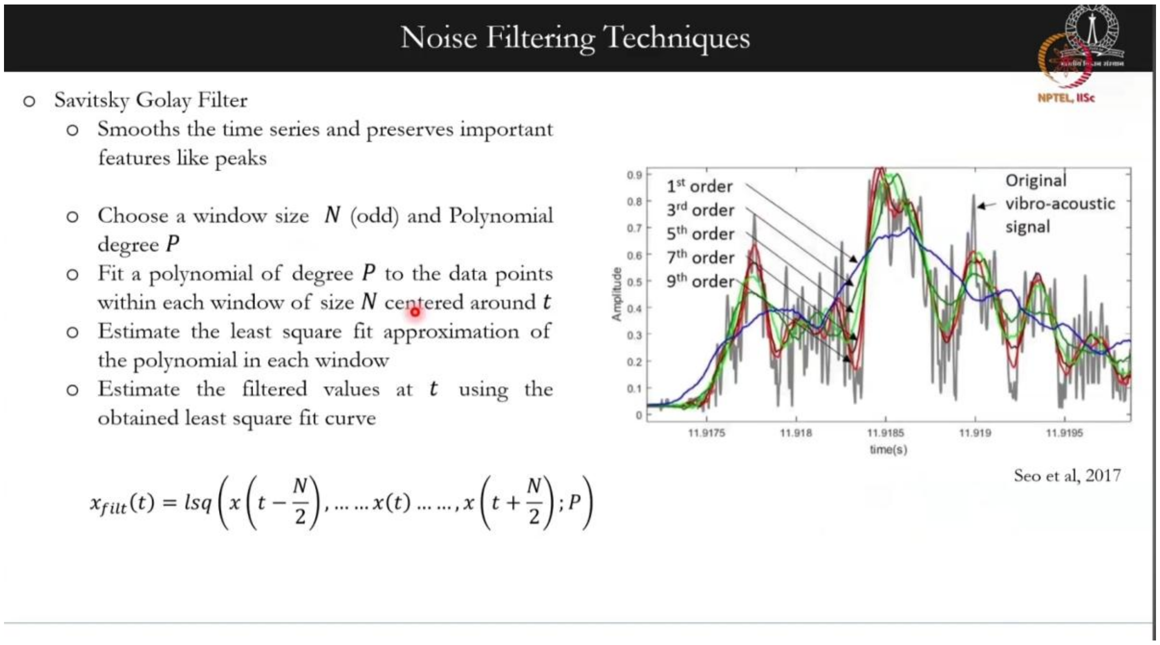


- Median Filter
  - Remove noise by replacing each data point with the median of its neighbors.
  - Choose a window size N
  - Collect data points within the window centered around t. Sort the values in the window and find median
  - $x_{filt}(t) = median\left(x\left(t-\frac{N}{2}\right), \dots\dots x(t) \dots\dots, x\left(t+\frac{N}{2}\right)\right)$



And once it estimates that average, the filtered value at that point is basically nothing but that computed average. And it performs this operation over the entire data series, sliding this window of size throughout the entire data series. and basically generates a whole new time series of filtered data. So, that is what the moving average filter does. And the median filter is very similar, except that here instead of computing the average of all the data inside that moving window, it estimates the median of those endpoints that are enclosed within that endpoint window.

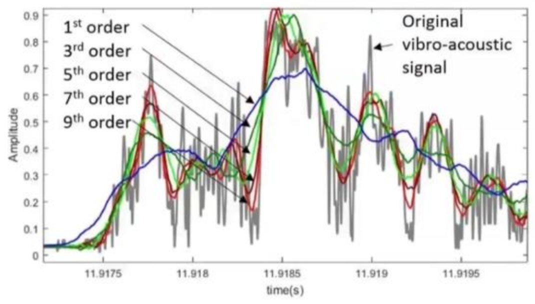
And once it estimates the median, the filtered value is basically nothing but the estimated median. And again, it performs this operation over the entire data series at all points and essentially generates a whole new series of filtered data points. So as we can see here that these filters are very effective in removing these spurious oscillations. So these oscillations get smeared out when we use this moving average filter or median filter, and that's how it tackles noise. The other commonly used filter is the Savitzky-Golay filter.



Noise Filtering Techniques

- Savitsky Golay Filter
  - Smooths the time series and preserves important features like peaks
  - Choose a window size N (odd) and Polynomial degree P
  - Fit a polynomial of degree P to the data points within each window of size N centered around t
  - Estimate the least square fit approximation of the polynomial in each window
  - Estimate the filtered values at t using the obtained least square fit curve

$$x_{filt}(t) = lsq\left(x\left(t-\frac{N}{2}\right), \dots \dots x(t) \dots \dots, x\left(t+\frac{N}{2}\right); P\right)$$

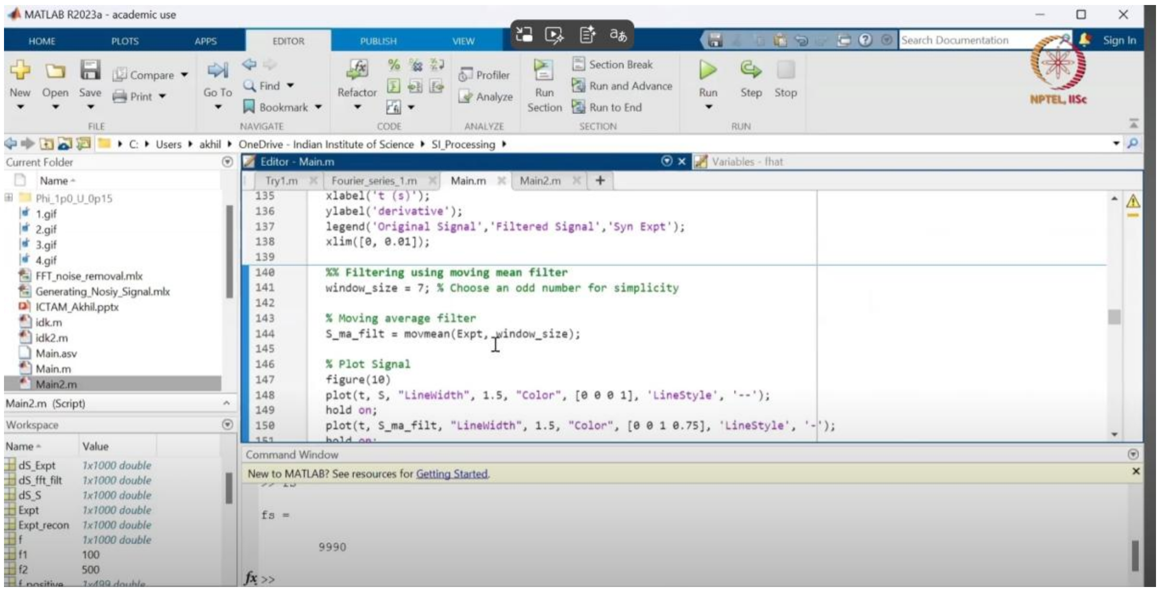


Seo et al, 2017

So here, along with the parameter n that indicates the number of points enclosed in a moving window, we also need to specify another parameter, which is basically the degree of the polynomial. So here, what it does is that it basically fits a polynomial of degree P to the points that are inside that endpoint window. Using the least squares fit approximation. And once it has that polynomial approximation, it uses it to estimate the filtered data at that point. And again, it performs this operation on all the data points by sliding that moving window.

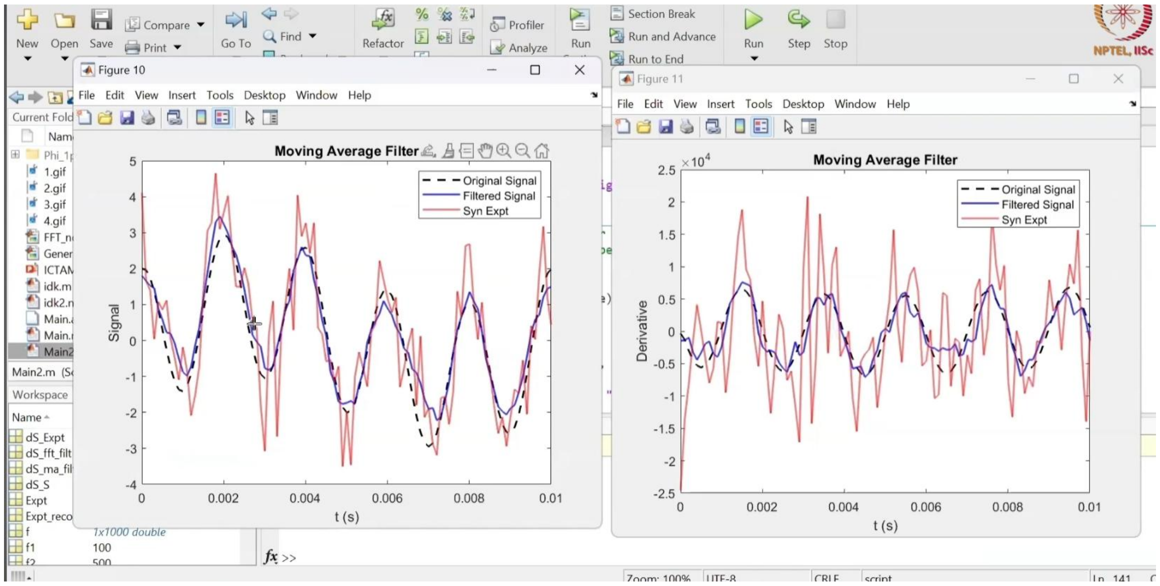
And in each of these moving windows, it estimates the least squares fit approximation of polynomial degree P. Estimates that approximation and then uses that approximation to estimate the filtered value. Again, it generates a whole new time series of filtered values. So, that is how these three filters work. Let us now compare how effective these filters are in tackling the noise that we generated.

So going back to the code, here, this is a moving average filter, and I have chosen a window size of seven. Again, this is arbitrary, as you can choose a different moving window size and get slightly different results. So let us just run this portion of the code. So we can see that although the signal is fairly captured, it is not doing as good a job as what FFT filtering actually did. FFT filtering was able to almost match the original signal with a very good level of accuracy.

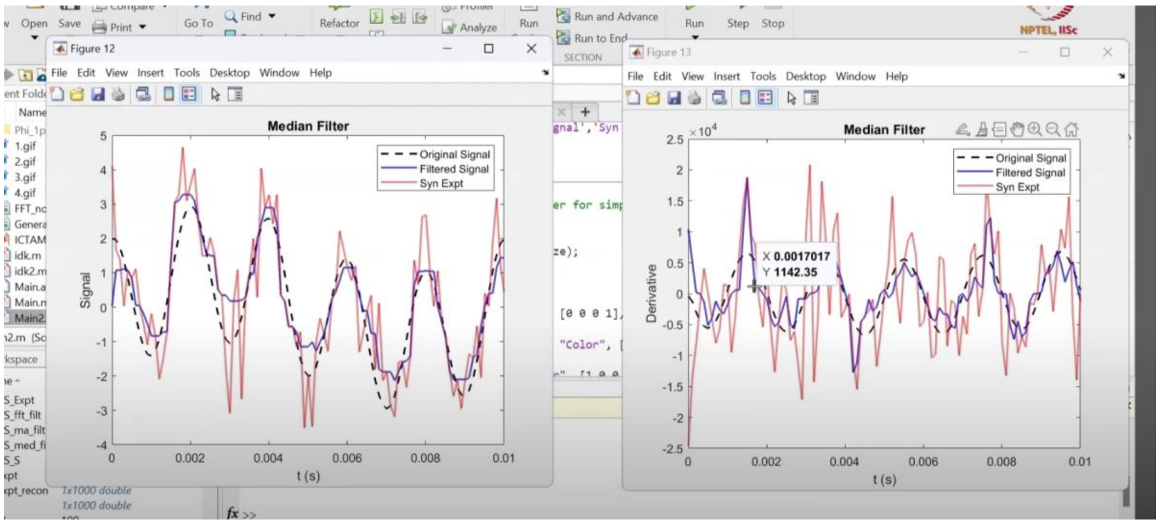


```
xlabel('t (s)');
ylabel('derivative');
legend('Original Signal','Filtered Signal','Syn Expt');
xlim([0, 0.01]);

%% Filtering using moving mean filter
window_size = 7; % Choose an odd number for simplicity

% Moving average filter
S_ma_filt = movmean(Expt, window_size);

% Plot Signal
figure(10)
plot(t, S, "LineWidth", 1.5, "Color", [0 0 0 1], 'LineStyle', '--');
hold on;
plot(t, S_ma_filt, "LineWidth", 1.5, "Color", [0 0 1 0.75], 'LineStyle', '-');
```

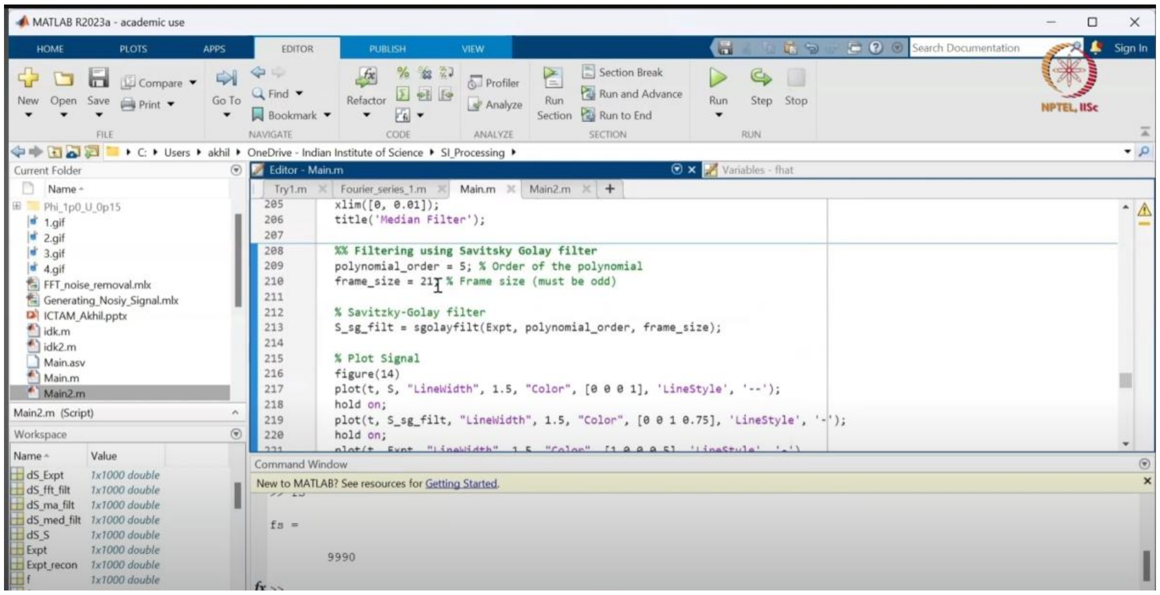
```
fs =

        9990
```



But the moving average filter does not do that good of a job. Let's run it for the median filter now. Even for the median filter, I have chosen the same size. The window size is still seven.

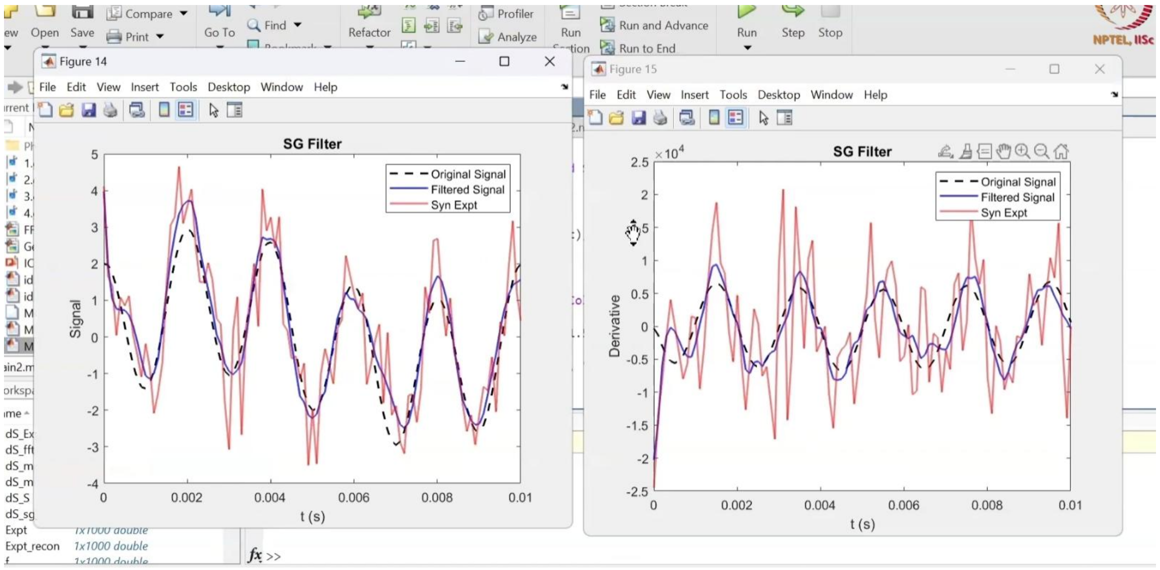


So we observe almost similar results. In fact, in some places, we see that the spurious oscillations are not even removed very well by the median filter. So the median filter underperforms even more than the moving average filter. Although the estimated derivatives are nearly the same order, the FFT filter did a much better job. Now let us go into the Savitzky-Golay filter. Here, the frame size, basically the window size, is taken as 21, and the polynomial order is chosen as 5.



```
xlim([0, 0.01]);
title('Median Filter');

%% Filtering using Savitsky Golay filter
polynomial_order = 5; % Order of the polynomial
frame_size = 21; % Frame size (must be odd)

% Savitzky-Golay filter
S_sg_filt = sgolayfilt(Expt, polynomial_order, frame_size);

% Plot Signal
figure(14)
plot(t, S, "LineWidth", 1.5, "Color", [0 0 0 1], 'LineStyle', '--');
hold on;
plot(t, S_sg_filt, "LineWidth", 1.5, "Color", [0 0 1 0.75], 'LineStyle', '-');
hold on;
```

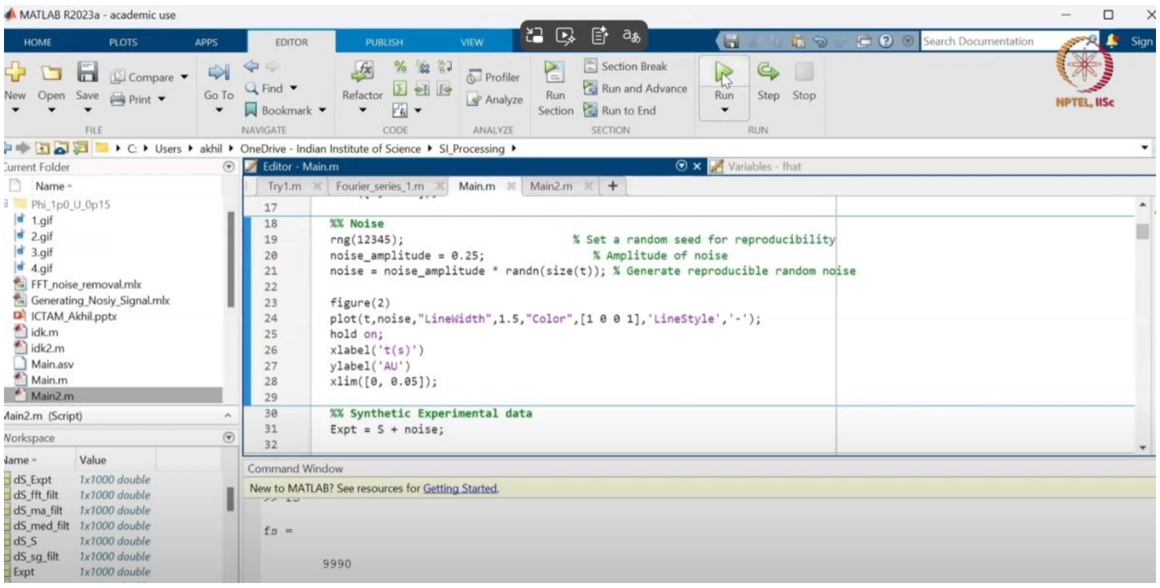
```
fs =

        9990
```



Again, you can change these parameters to obtain a slightly different fit. That can obviously be done. But let us just compare how this looks. We can see that this does a slightly better job than the moving mean filter and the moving average filters. One thing I forgot to mention was that these plots basically represent the gradient, and this is basically the signal.

The filtering using FFT was not more effective than the SC filter. But what we can do right now is reduce the noise amplitude. We'll just reduce the noise amplitude and see how good these filters are. Earlier, I had a noise amplitude of 1, which was comparable to the amplitude of the signal as well.

So let us reduce that noise now. So I'll make this say 0.25 and let us run the code now. So, we have all the plots. So, for the AC filter, we can see that it has improved significantly. So, as I reduce the noise amplitude, the filtering has become a lot more effective.

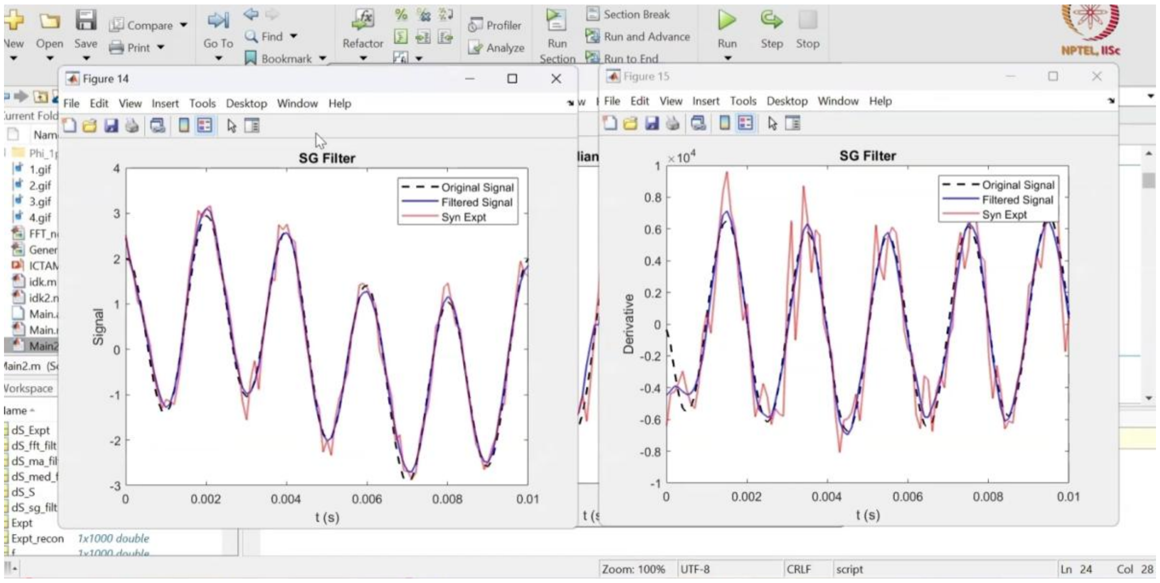

MATLAB R2023a - academic use
HOME PLOTS APPS EDITOR PUBLISH VIEW
Editor - Main.m
Try1.m Fourier_series_1.m Main.m Main2.m
C: ▸ Users ▸ akhil ▸ OneDrive - Indian Institute of Science ▸ SI_Processing ▸
NPTEL, IISc

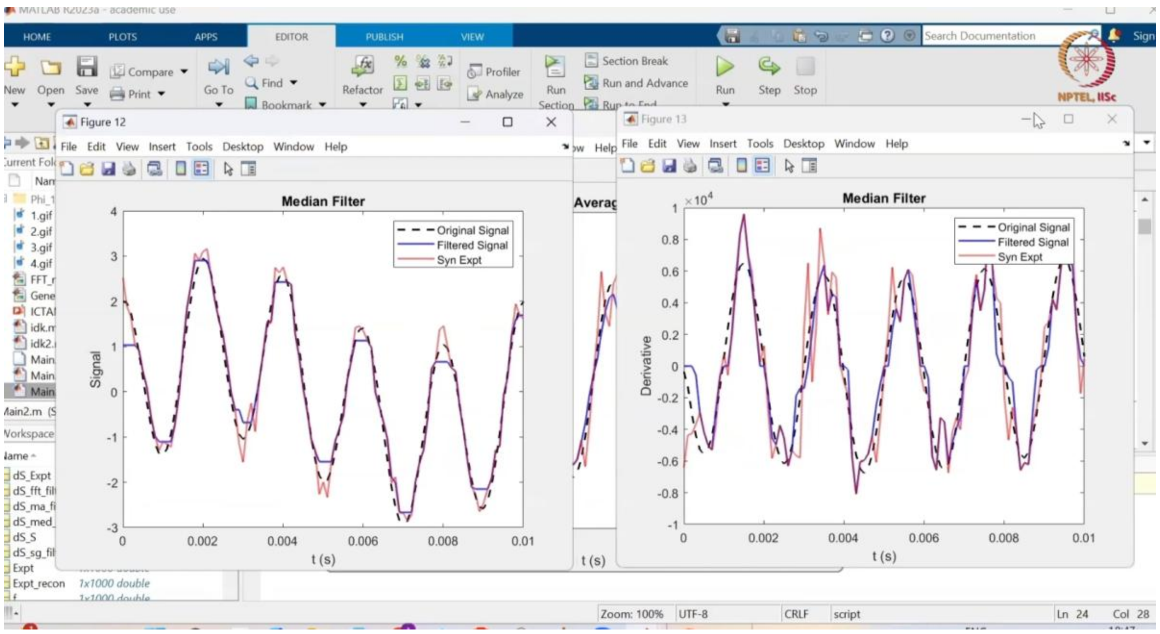
```
%% Noise
rng(12345);                      % Set a random seed for reproducibility
noise_amplitude = 0.25;              % Amplitude of noise
noise = noise_amplitude * randn(size(t)); % Generate reproducible random noise

figure(2)
plot(t,noise,"LineWidth",1.5,"Color",[1 0 0 1],'LineStyle','-');
hold on;
xlabel('t(s)')
ylabel('AU')
xlim([0, 0.05]);

%% Synthetic Experimental data
Expt = S + noise;
```

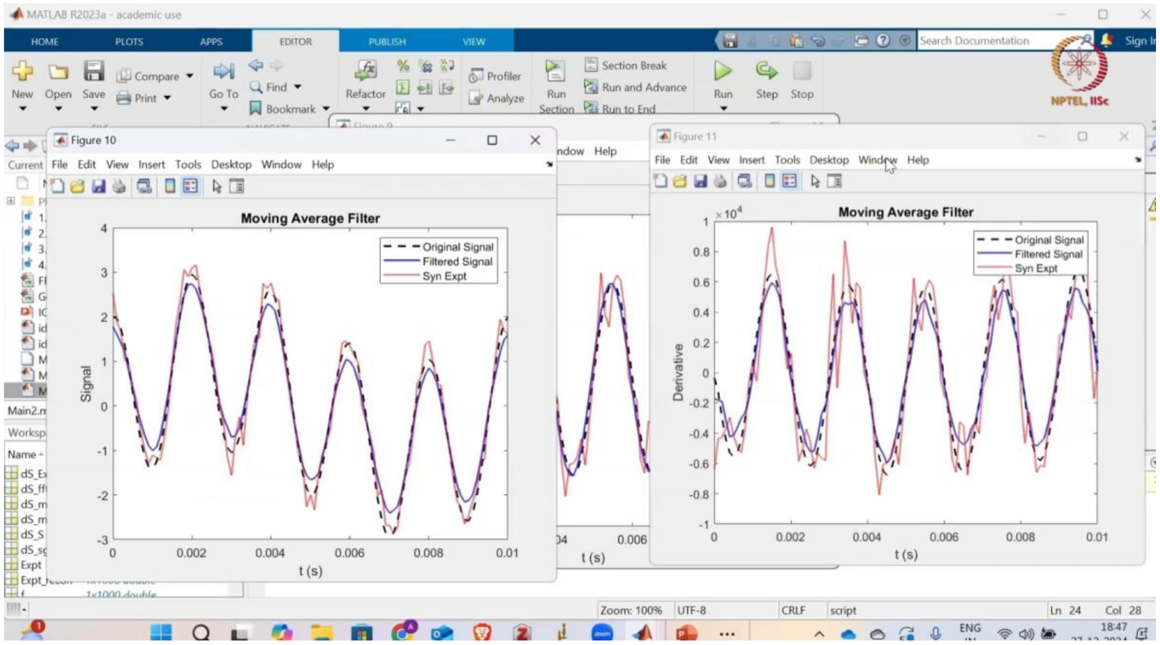
```
fs =

        9990
```

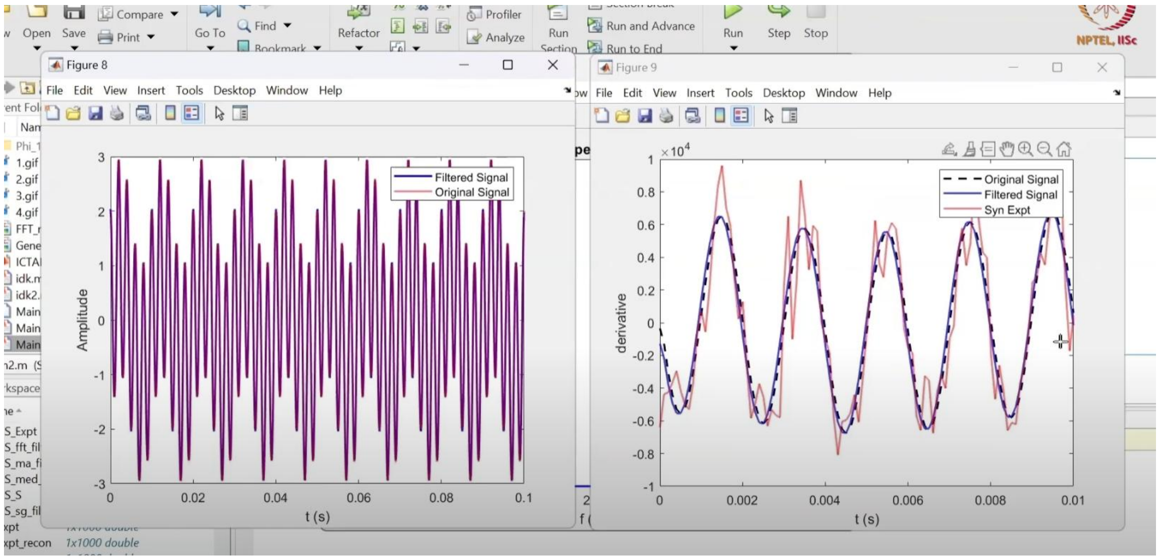
Name	Value
dS_Expt	1x1000 double
dS_fft_filt	1x1000 double
dS_ma_filt	1x1000 double
dS_med_filt	1x1000 double
dS_S	1x1000 double
dS_sg_filt	1x1000 double
Expt	1x1000 double

The original signal compares fairly to the filtered signal. The same median filter; again, the quality has slightly improved. But again, there are still spurious oscillations that it has not very effectively removed. Almost the same thing can be said about the moving average filter. It's better than what we were getting when the noise amplitude was very high.

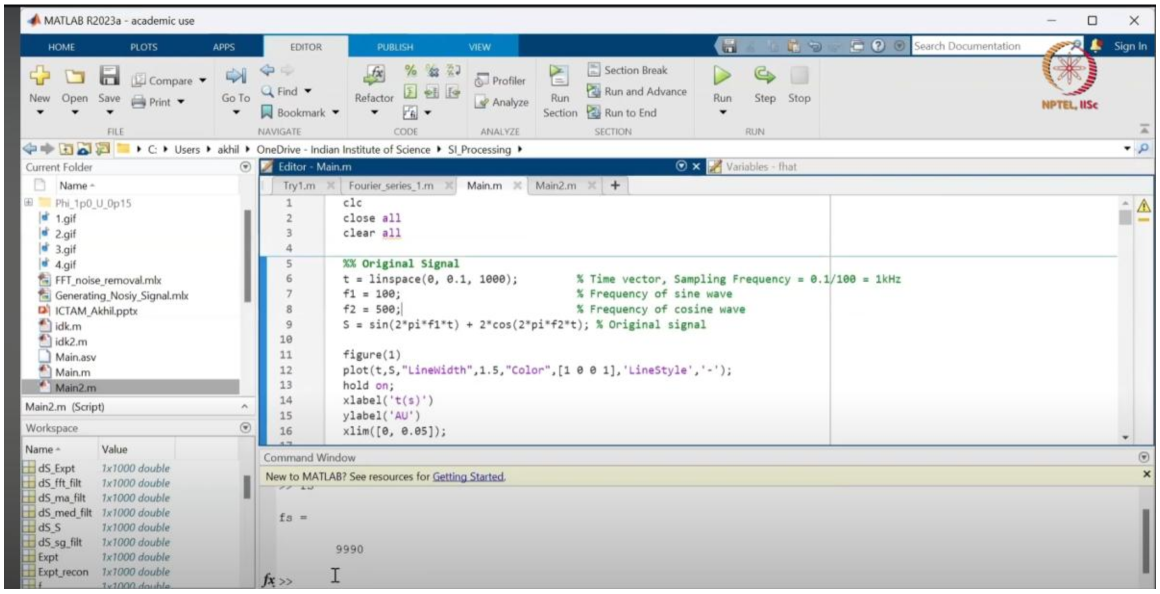



MATLAB R2023a - academic use
HOME PLOTS APPS EDITOR PUBLISH VIEW
Figure 12
Median Filter
Original Signal
Filtered Signal
Syn Expt
Signal
t (s)
Figure 13
Median Filter
Derivative
$\times 10^4$
NPTEL, IISc


MATLAB R2023a - academic use
HOME PLOTS APPS EDITOR PUBLISH VIEW
Figure 10
Moving Average Filter
Original Signal
Filtered Signal
Syn Expt
Signal
t (s)
Figure 11
Moving Average Filter
Derivative
$\times 10^4$
NPTEL, IISc



Now let us compare again with the FFT. So filtering using FFT, we can see that there is almost a very nice overlap between the original signal and the filtered signal. So this was just a demonstration to show how effective the FFT is in actually removing the noise. And this technique is quite often used to separate data from noise in experiments. One other thing that I wanted to show was the effect of changing the sampling rate.



```
clc
close all
clear all

%% Original Signal
t = linspace(0, 0.1, 1000);          % Time vector, Sampling Frequency = 0.1/100 = 1kHz
f1 = 100;                            % Frequency of sine wave
f2 = 500;                            % Frequency of cosine wave
S = sin(2*pi*f1*t) + 2*cos(2*pi*f2*t); % Original signal

figure(1)
plot(t,S,"LineWidth",1.5,"Color",[1 0 0 1],'LineStyle','-');
hold on;
xlabel('t(s)')
ylabel('AU')
xlim([0, 0.05]);
```

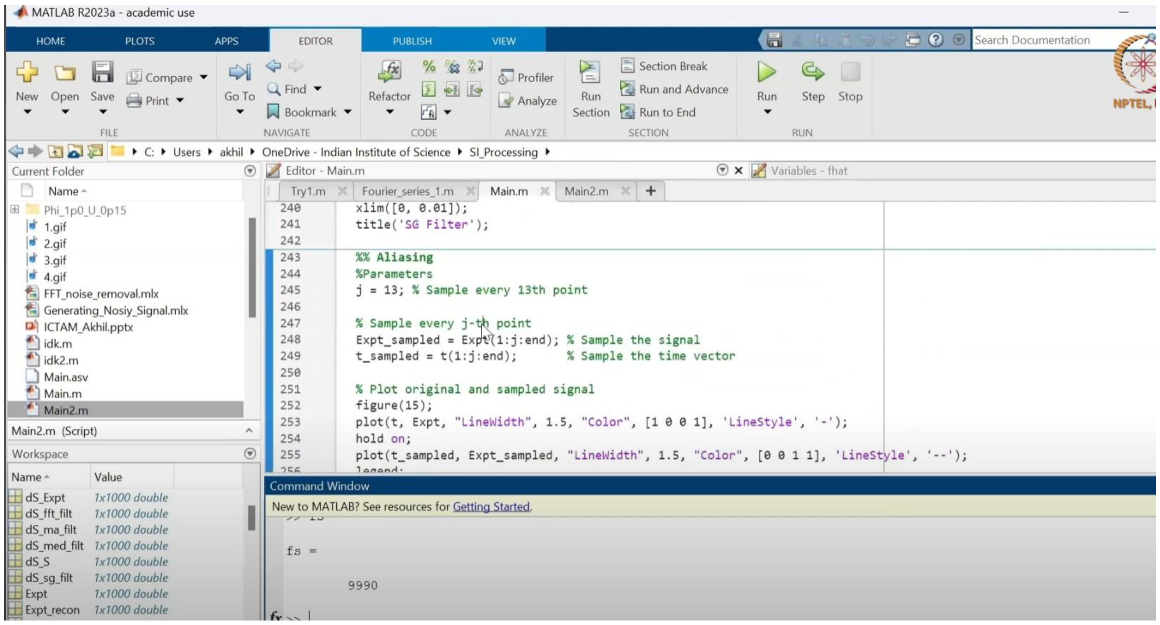
Workspace

Name	Value
dS_Expt	1x1000 double
dS_fft_filt	1x1000 double
dS_ma_filt	1x1000 double
dS_med_filt	1x1000 double
dS_S	1x1000 double
dS_sg_filt	1x1000 double
Expt	1x1000 double
Expt_recon	1x1000 double

Command Window

New to MATLAB? See resources for Getting Started.

```
fs =

        9990

fx >>
```



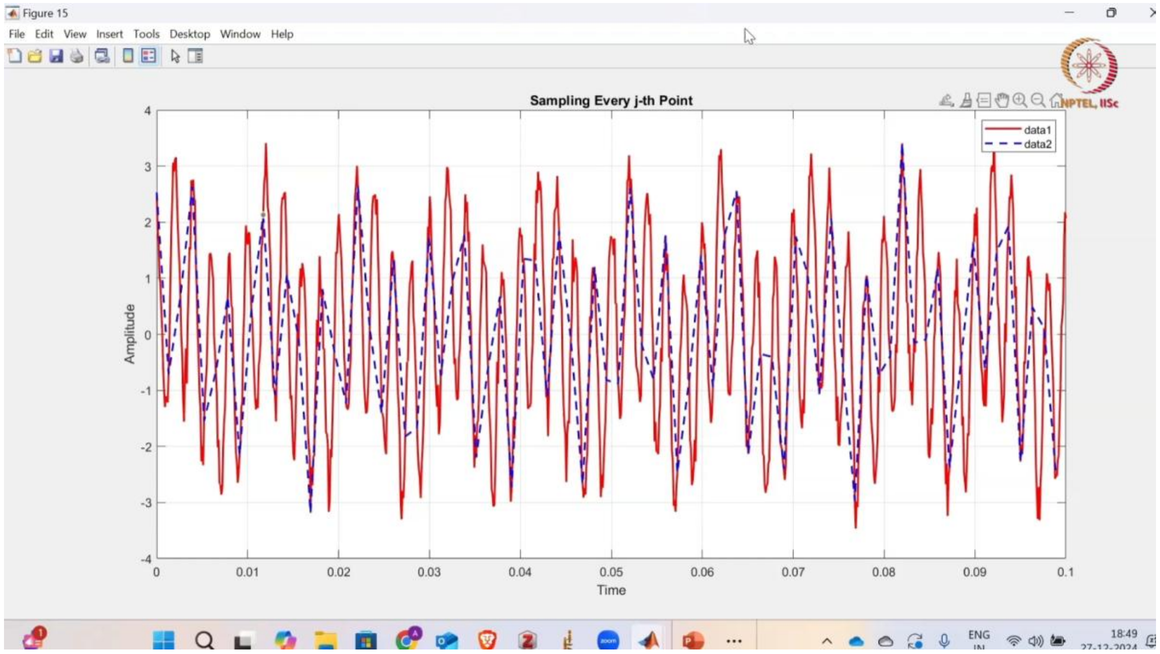
```
xlim([0, 0.01]);
title('SG Filter');

%% Aliasing
%Parameters
j = 13; % Sample every 13th point

% Sample every j-th point
Expt_sampled = Expt(1:j:end); % Sample the signal
t_sampled = t(1:j:end);       % Sample the time vector

% Plot original and sampled signal
figure(15);
plot(t, Expt, "LineWidth", 1.5, "Color", [1 0 0 1], 'LineStyle', '-');
hold on;
plot(t_sampled, Expt_sampled, "LineWidth", 1.5, "Color", [0 0 1 1], 'LineStyle', '--');
legend;
```

```
fs =

        9990
```

Name	Value
dS_Expt	1x1000 double
dS_fft_filt	1x1000 double
dS_ma_filt	1x1000 double
dS_med_filt	1x1000 double
dS_S	1x1000 double
dS_sg_filt	1x1000 double
Expt	1x1000 double
Expt_recon	1x1000 double

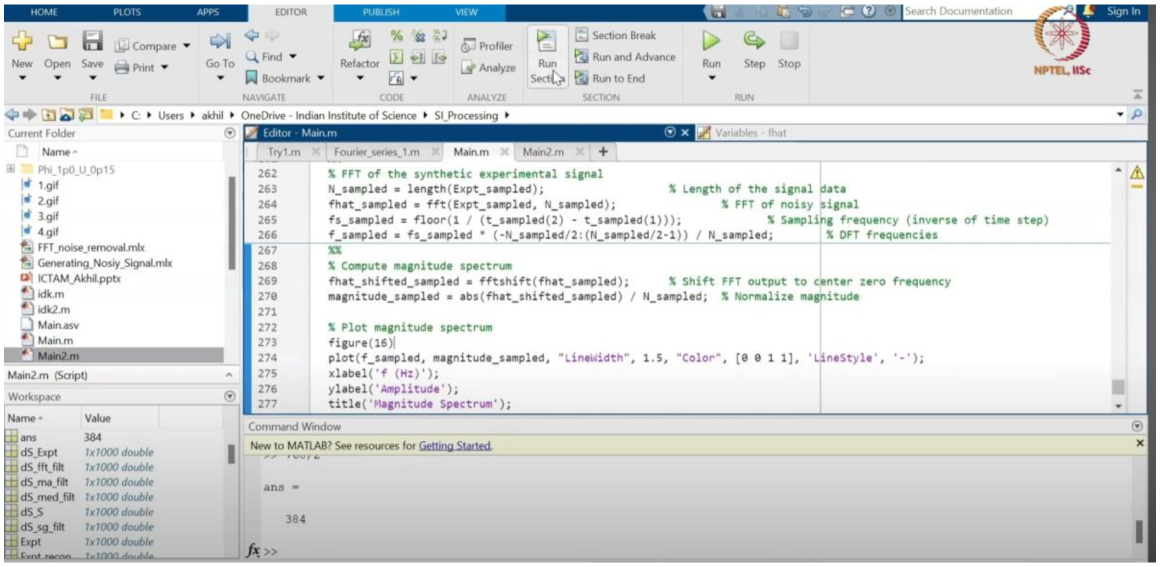
So, let us look into that now. So, basically, in the previous section, the sampling rate, specifically the value of Fs, was around 9,990. And our frequencies of interest, the frequencies that we were interested in capturing, were 100 Hz and 500 Hz. So, as per the Nyquist criterion, Fs by 2 is much greater than these values. So, we would not have any problems capturing them. But what if we change the sampling rate? What if we sample it at a slightly different rate? Now what we are going to do is sample every 13th point.

Basically, what it does is reduce the sampling frequency by a factor of 13. So let us run this portion of the code. So now this is the sample data. The plot that you're seeing is in blue dotted lines.



So that is the sample data. When we sample it at... 1 by the 13th of the initial sampling rate. So, let us look at what the sampling rate is right now. So, let us look at the new sampling rates. So, this is 768, and we can see that now the Fs sampled by 2, which is basically 768 by 2, is 384, and 384 is basically less than 500, where 500 is one of the frequencies that we are interested in capturing. So now, as per the Nyquist criteria, we will not be able to capture this 500 hertz.

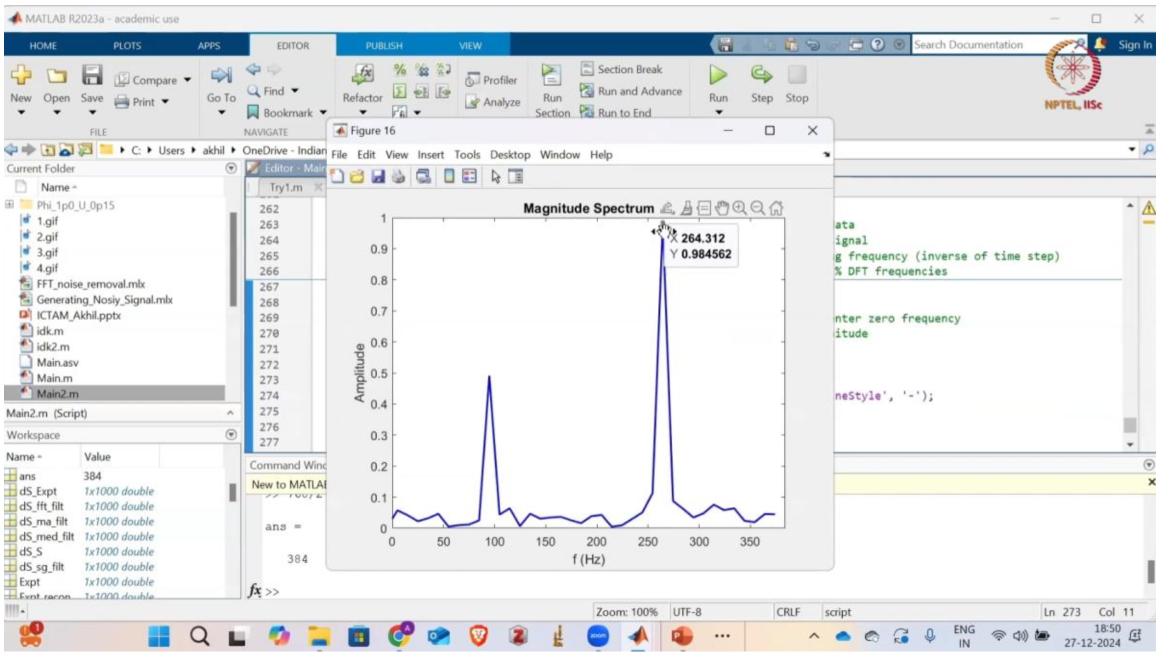
So let us run the FFT on this and see what value we actually get. So, let us run this portion and we see here that the 100 hertz is still kind of captured, whereas the signal that was initially at 500 hertz has now been aliased to 264 hertz, and this is the aliasing that we were mentioning when we discussed the Shannon-Nyquist criteria. Sampling frequencies less than twice the frequency of our interest, the energy associated with that phenomenon will be mapped onto another frequency. So, the energy that was associated with 500 Hz here got mapped to 264 Hz, and that is what happened here. So this is something that we need to keep in mind when we do experiments. If we know that we need to capture a particular frequency of oscillation, we should always make sure that our sampling rate is at least two times that value.



```
% FFT of the synthetic experimental signal
N_sampled = length(Expt_sampled);                    % Length of the signal data
fhat_sampled = fft(Expt_sampled, N_sampled);                 % FFT of noisy signal
fs_sampled = floor(1 / (t_sampled(2) - t_sampled(1)));              % Sampling frequency (inverse of time step)
f_sampled = fs_sampled * (-N_sampled/2:(N_sampled/2-1)) / N_sampled;        % DFT frequencies
%%
% Compute magnitude spectrum
fhat_shifted_sampled = fftshift(fhat_sampled);      % Shift FFT output to center zero frequency
magnitude_sampled = abs(fhat_shifted_sampled) / N_sampled;  % Normalize magnitude

% Plot magnitude spectrum
figure(16)
plot(f_sampled, magnitude_sampled, "LineWidth", 1.5, "Color", [0 0 1 1], 'LineStyle', '-');
xlabel('f (Hz)');
ylabel('Amplitude');
title('Magnitude Spectrum');
```

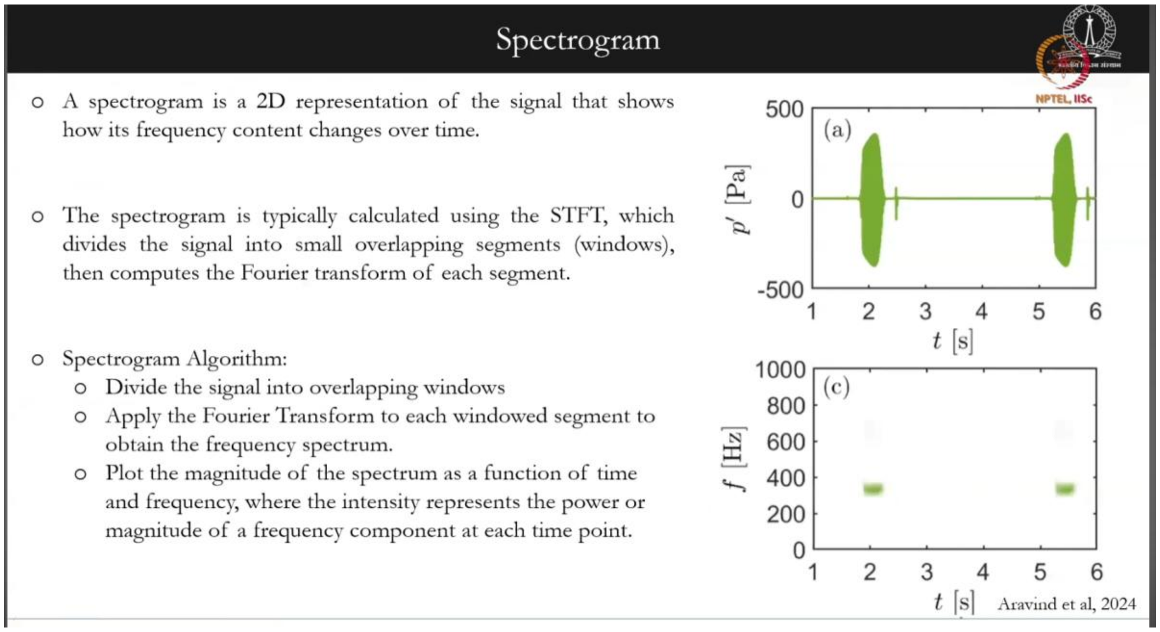
New to MATLAB? See resources for Getting Started.

```
ans =

   384
```



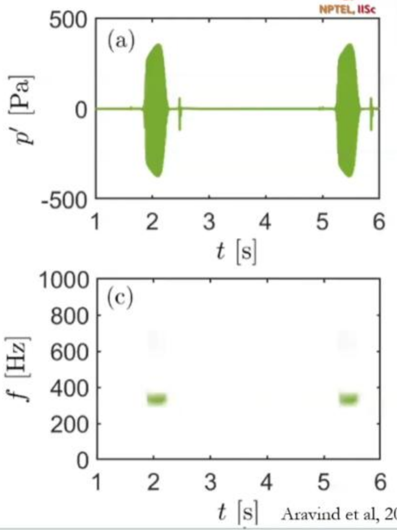
This is a very important thing to note. Another thing that I want to discuss is the spectrograms. So basically, in an FFT curve, if you look at an FFT curve like this, we know that there are dominant frequencies in my signal. However, I have no information about when this frequency occurs in my time series data. Now let us look into the data series here. So here I have a data series that actually shows pressure variations in a particular chamber.

And it shows that there are some pressure fluctuations at a particular frequency over this small period of time, maybe from around 1.7 seconds to around 2.2 seconds. And after that, there are not many oscillations in the pressure. And again, these oscillations occur during the time period from, say, around 5.



Spectrogram

- A spectrogram is a 2D representation of the signal that shows how its frequency content changes over time.
- The spectrogram is typically calculated using the STFT, which divides the signal into small overlapping segments (windows), then computes the Fourier transform of each segment.
- Spectrogram Algorithm:
  - Divide the signal into overlapping windows
  - Apply the Fourier Transform to each windowed segment to obtain the frequency spectrum.
  - Plot the magnitude of the spectrum as a function of time and frequency, where the intensity represents the power or magnitude of a frequency component at each time point.



Aravind et al, 2024

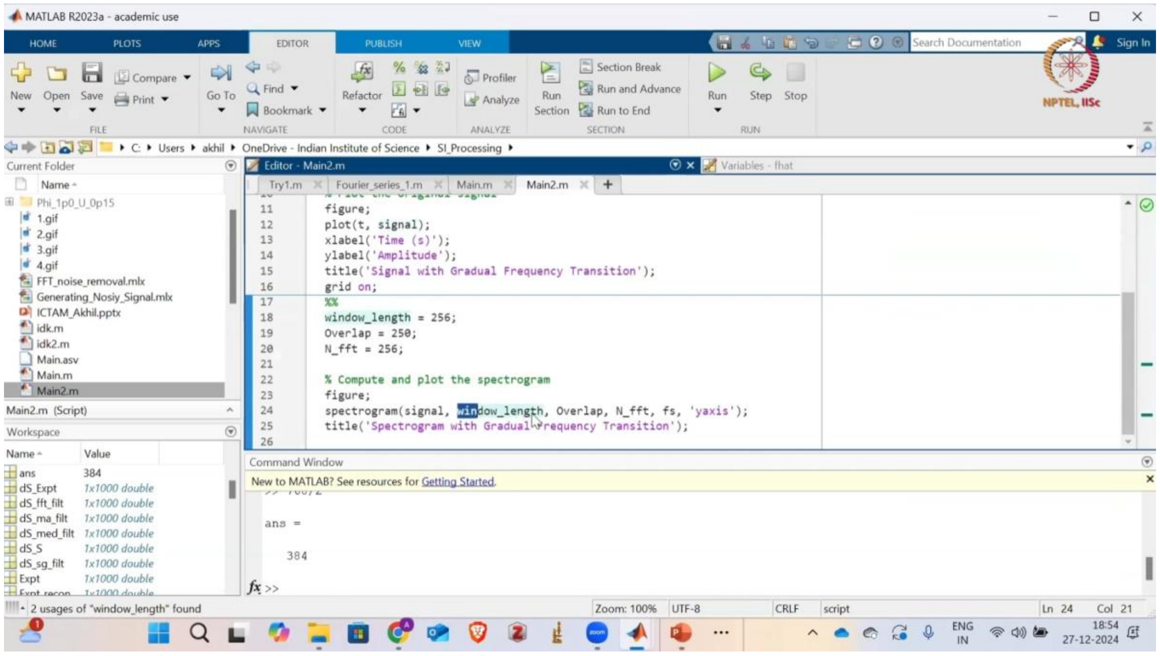
3 to, say, 5.7. So, basically, frequencies are occurring periodically here. If we do an FFT of this, it will still show us what the dominant peak is. It will still tell us what the frequency associated with these oscillations is, but it will not give us information on when these oscillations are actually happening in my time series. And that can be very important information here. And that is where the spectrogram comes in. So, what this spectrogram does is that it basically creates a moving window again.

So, it creates a window that encloses, say, a certain number of points. It performs FFT on that, then it slides over a few points and performs FFT again on that; similarly, it completely slides over the entire time series and performs FFT in each window. And then it displays it as a 2D contour, basically. And this is what we finally end up with. So here the x-axis is again the time axis, and the y-axis is the frequency axis. Here, the color and the intensity of the color basically show what the magnitude of that X_k vector is.

So, here we can see that in the time period from around 1.7 to around 2.2 seconds, because of these frequencies, the dominant frequency was around 350 hertz, and its magnitude was very high. Again, the same frequency occurred after a particular time period that corresponds to these fluctuations. And again, it represents a magnitude in terms of color. So that is basically what the spectrogram does.

It's basically nothing but a moving-window FFT. And again, the implementation of this in MATLAB is pretty easy. So here it directly uses a function called a spectrogram. And to this function, the inputs that we need to give is the data signal, whatever we have. The other input that we need to provide is the window length, which is essentially the length of the window that I was talking about where the FFT is computed in each window.

So that is the window length. And then there is the overlap. Overlap basically shows what the overlap is between two consecutive windows. So if I have a window length of 256 and an overlap of 250, that basically means that between two consecutive windows, the window just moves by six data points. That's it. And then NFFT shows the number of points over which the FFT is performed inside each window.

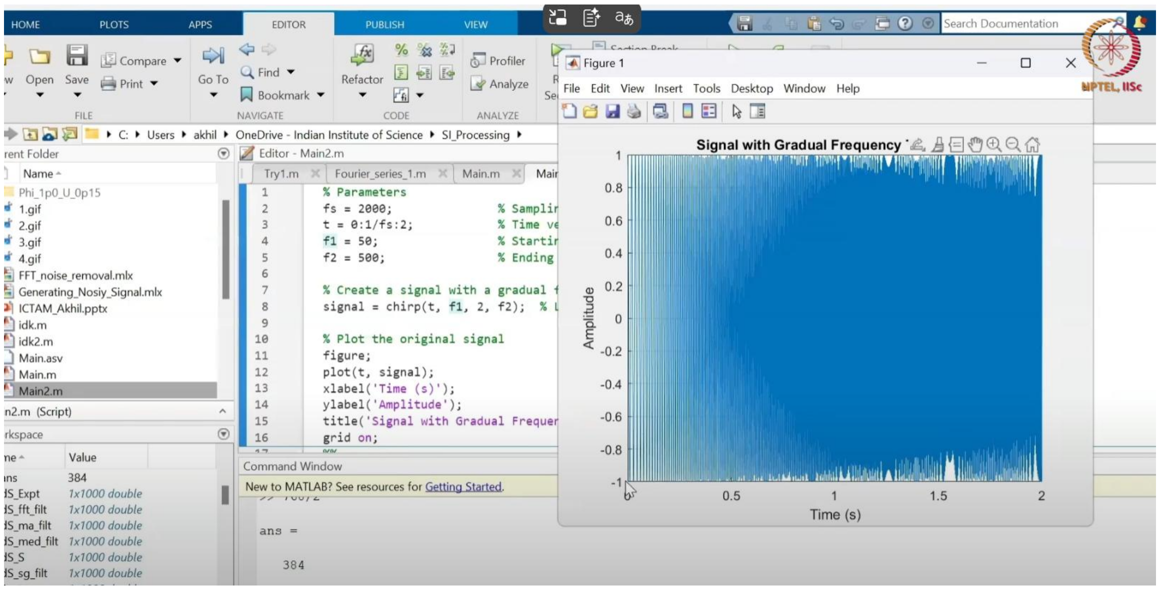


```
figure;
plot(t, signal);
xlabel('Time (s)');
ylabel('Amplitude');
title('Signal with Gradual Frequency Transition');
grid on;
%%
window_length = 256;
Overlap = 250;
N_fft = 256;

% Compute and plot the spectrogram
figure;
spectrogram(signal, window_length, Overlap, N_fft, fs, 'yaxis');
title('Spectrogram with Gradual Frequency Transition');
```

```
ans =

   384
```

And then Fs is basically nothing but the sampling rate. Here, to illustrate this example, I have taken a very simple signal. It's called a chirp signal. So let us first look at what this signal is.

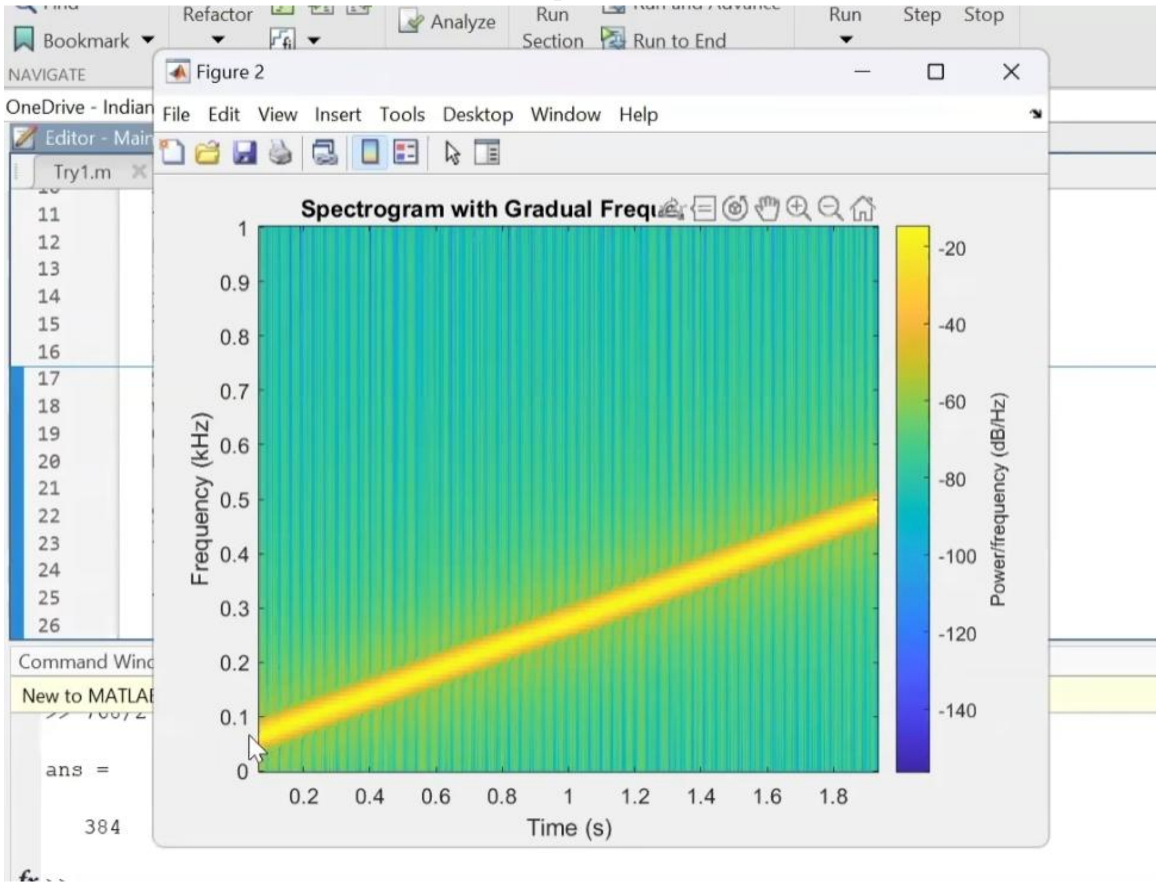
So, let us run the section. I will just close the other plots and then run the section. So, here we can see that the frequency actually varies from around 50 hertz to about 500 hertz in approximately 2 seconds. So, basically, the amplitude is nearly constant, but the frequency varies from 50 to 500 as we go from 0 to 2 seconds. And now we are going to, our aim is to actually capture this variation. We need a plot that shows that the frequency actually varies from the dominant frequency.



```
% Parameters
fs = 2000;            % Samplir
t = 0:1/fs:2;         % Time ve
f1 = 50;              % Startir
f2 = 500;             % Ending

% Create a signal with a gradual f
signal = chirp(t, f1, 2, f2);  % L

% Plot the original signal
figure;
plot(t, signal);
xlabel('Time (s)');
ylabel('Amplitude');
title('Signal with Gradual Frequer
grid on;
```

```
New to MATLAB? See resources for Getting Started.

ans =

   384
```

It actually evolves from 50 hertz to 500 hertz. And that is exactly what this spectrogram plot shows. Again, it shows that here the unit isn't for kilohertz. So here the frequency varies from around 50 hertz at approximately zero seconds to around 500 hertz as I move to about two seconds. So, the important thing is that in FFT, we get no information about how the dominant frequencies evolve over time.



```
ans =

   384
```

Whereas the spectrogram actually gives that information. It actually shows how the dominant frequencies are evolving over time. So, this completes the demo video.