Digital Communication using GNU Radio

Prof. Kumar Appaiah

Department of Electrical Engineering

Indian Institute of Technology Bombay

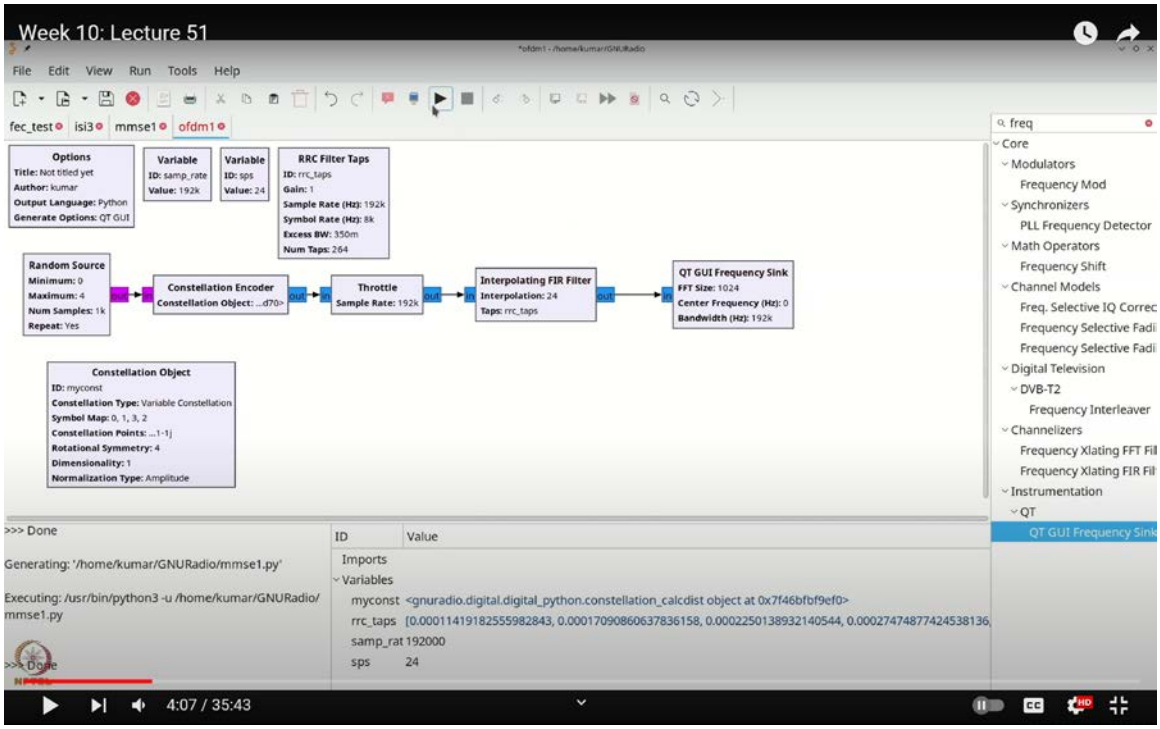
Week-10

Lecture-51

Equalisation using OFDM in GNU Radio

In the previous lecture, we thoroughly explored the various aspects of Orthogonal Frequency Division Multiplexing (OFDM). To recap, the core idea behind OFDM is to distribute your data across multiple parallel narrowband frequency channels. This approach allows you to effectively parallelize the transmission, transforming it into several channels that are relatively easy to equalize.

(Refer Slide Time: 04:07)



In this lecture, we will delve into using GNU Radio to implement an FFT-based OFDM system. We will demonstrate how parallelizing the channels in the frequency domain helps

mitigate the effects of inter-symbol interference (ISI) to a significant extent. This reduction is achieved because the transmission is split into parallel channels. However, it's important to note that while ISI is reduced in this manner, it does reappear in the frequency domain as different channel gains.

We will explore this concept by applying various PSK and QAM modulation schemes, examining how the use of OFDM impacts the receiver design. Our approach will involve first constructing a traditional system that does not utilize OFDM, and then gradually integrating the OFDM components to observe the differences.

Let's begin with our first task, which involves creating a random data source. Once we've added this random source, we'll configure it for QPSK modulation. Specifically, we will set the maximum value to 4 and use the byte data type for simplicity. Next, we'll introduce a constellation object along with an encoder. You can easily find these by using the search function, just press Control+F (or Command+F on a Mac) and type in "constellation" to locate the encoder.

We'll connect these elements together, defining the constellation as **`myconst`**, and ensure that our encoder is set to use this **`myconst`** configuration. At this stage, we'll also need to add a throttle block, which you can similarly find by searching for "throttle."

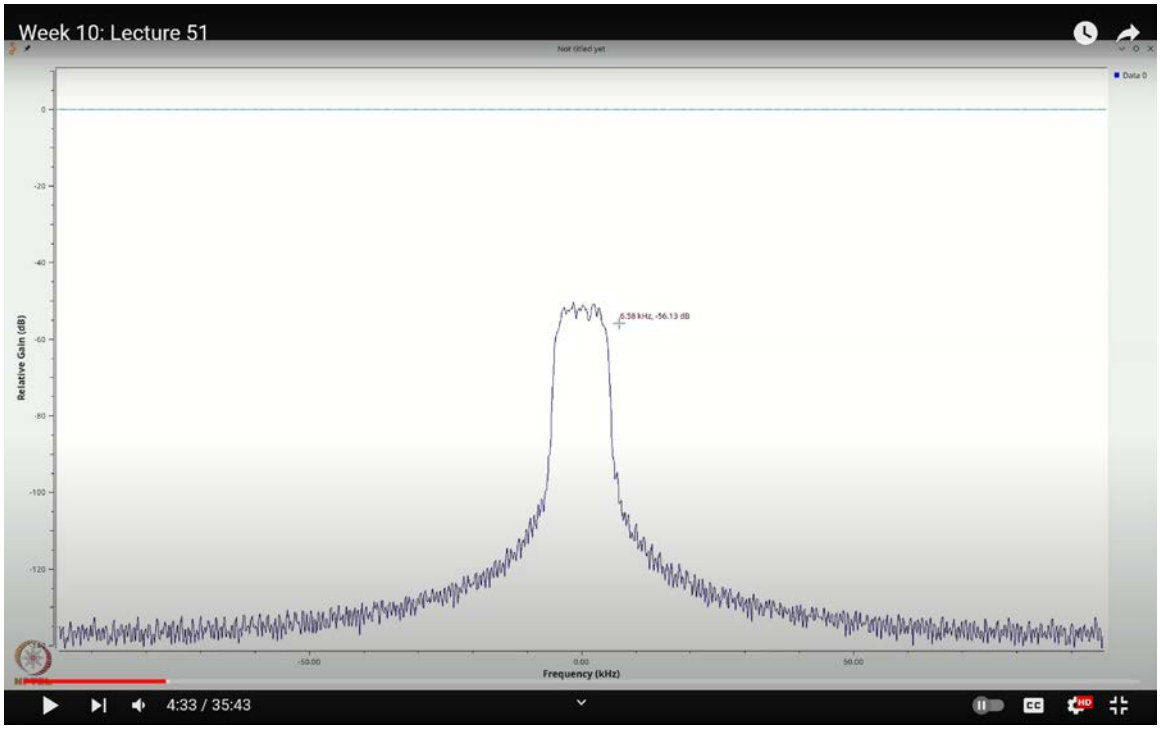
Moving forward, we'll incorporate a Root Raised Cosine (RRC) filter to handle pulse shaping. We'll set the sampling rate to 192,000, while limiting our symbol rate to 8,000. To manage this, we'll create another variable called **`SPS`** (samples per symbol). You can find the variable block by searching, and we'll configure **`SPS`** to be the result of the sampling rate divided by the symbol rate, ensuring we obtain an integer value of 8,000.

With the **`SPS`** variable in place, we'll next define our RRC filter taps. Again, search for "RRC" to find the filter taps block, which we will label as **`RRC_taps`**. Given our symbol rate of 8,000, this will define our RRC filter. Finally, we'll perform an interpolation using a FIR (Finite Impulse Response) filter to adjust the sample rate.

So, the next step is to incorporate an interpolating FIR filter into our design. To do this, we'll search for it by using Control+F (or Command+F on Mac), and then locate the

interpolating FIR filter. The interpolation factor will be set to **`SPS`**, while the filter taps will correspond to the **`RRC taps`** that we defined earlier. This follows the traditional approach we've been using to generate a properly sampled signal.

(Refer Slide Time: 04:33)



Now, we can visualize this signal using a QT GUI frequency sink. Again, use Control+F (or Command+F) to search for "FREQ" and grab the QT GUI frequency sink. Once connected, you'll notice that the frequency spectrum of the signal is spread across a bandwidth roughly between -8 kHz and +8 kHz. After performing a bit of averaging, you will see that the active bandwidth is actually within the range of -5 kHz to +5 kHz. This slight reduction is due to the 1.3 factor introduced by the root raised cosine filter, which is completely expected and makes sense.

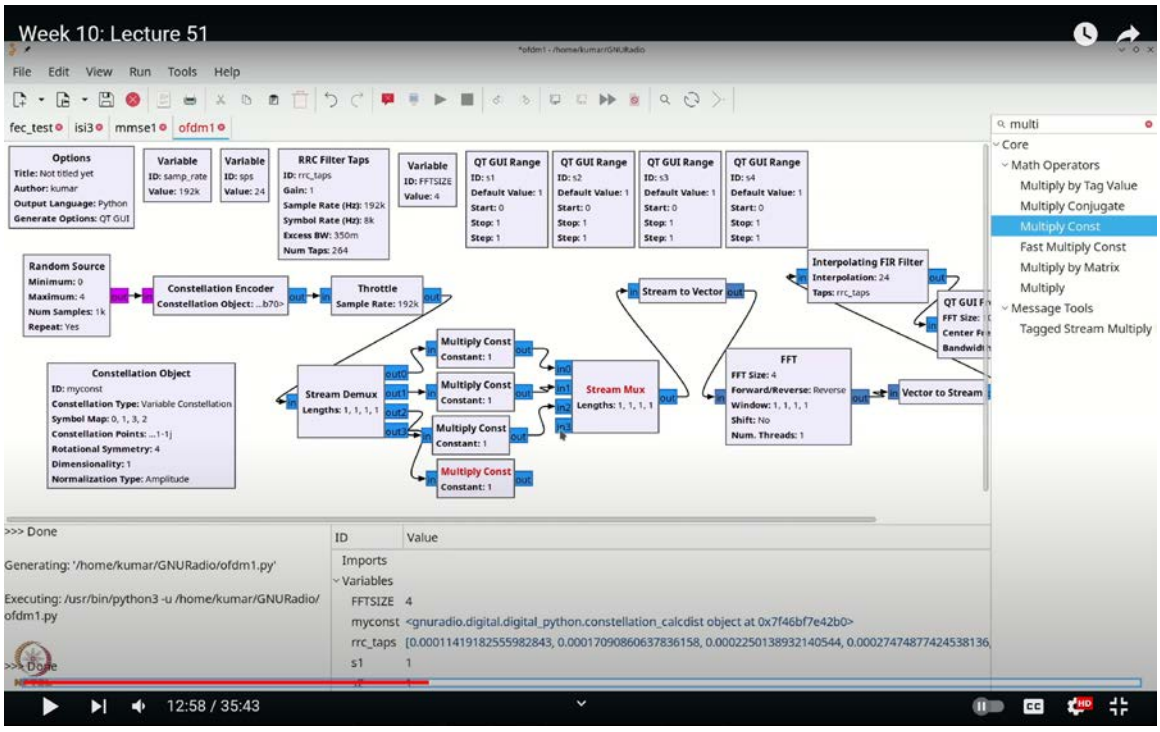
Moving forward, we will transition from traditional modulation techniques to OFDM modulation. To do this, we will take the output of the encoder (or from the throttle) and perform an Inverse Discrete Fourier Transform (IDFT). As discussed in the lecture, the

simplest way to implement OFDM is to take your modulated symbols, apply an IDFT to them, and then transmit the result.

To set this up in GNU Radio, we'll begin by creating a variable that defines the FFT size for the IDFT. Use Control+F (or Command+F) to search for "variable" and define a new variable called `**FFTSIZE**`. Setting this up as a variable will allow you to easily adjust the FFT size later. For now, we can set it to, say, 4.

Next, we'll disconnect the existing signal flow so we can insert the necessary OFDM elements. Specifically, we will need to perform a Discrete Fourier Transform (DFT) and then its inverse (IDFT). The FFT block will be used for this purpose. Use Control+F (or Command+F) to search for "FFT", and you'll find the FFT block under the Fourier analysis section.

(Refer Slide Time: 12:58)



However, the FFT block in GNU Radio works with vector inputs rather than stream inputs, because the FFT algorithm operates on blocks of data instead of individual samples. So, before we can connect our signal to the FFT block, we need to convert the continuous

stream into vectors of a fixed size. This can be done by inserting a stream-to-vector conversion block.

To do this, once again, use Control+F (or Command+F) and search for "stream to vector". After inserting the block, double-click on it to configure the settings. We will set the number of items to match our FFT size, which in this case is 4. Make sure that both the number of items and the vector length are configured correctly, so that the input is divided into chunks of size 4.

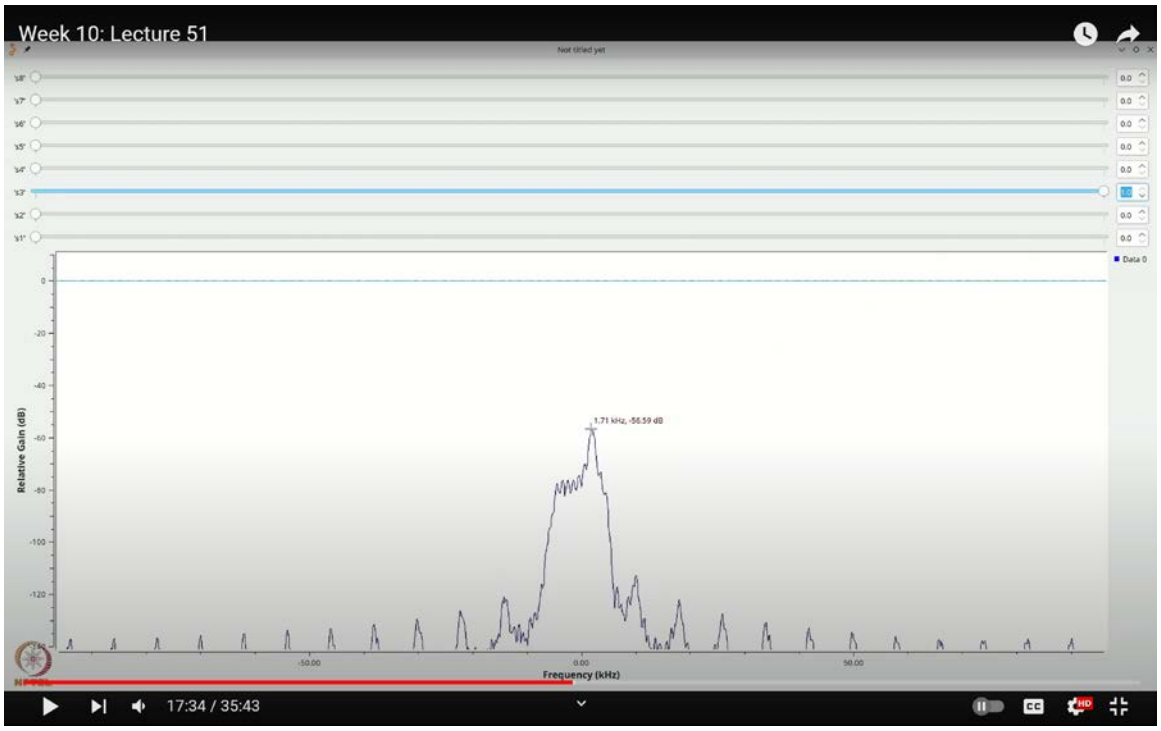
With this setup, the output from the stream-to-vector block can now be fed into the FFT block. However, don't forget that after performing the FFT, you will need to convert the output back into a stream to continue processing and viewing it in the frequency sink. This ensures smooth signal flow throughout the entire OFDM processing chain.

We are now one step closer to creating a fully functional OFDM system in GNU Radio.

The first step is to configure the FFT block. We'll start by setting the FFT size to **`FFTSIZE`**. Since we're aiming to perform an inverse DFT (IDFT), we need to adjust the FFT block by setting it to operate in reverse mode. By default, the FFT block applies a Blackman-Harris window, which is typically useful for spectral analysis as it smoothens the spectral display. However, in our case, we want to visualize the spectrum directly without any modifications. So, we'll modify the window function to be all ones, specifically by defining it as **`square bracket 1 times FFT size`**. This ensures that the window function is effectively neutral.

Next, we need to deal with the FFT shift feature, which rotates the FFT output for easier viewing. However, since we don't require any rotation here, we'll set the shift to 'none'. With these adjustments, the connection should be set up correctly. After performing the inverse DFT, the next step is to convert the output back into a continuous stream. For this, we use the "vector to stream" block. After inserting the block, double-click it and set the number of items to the FFT size. Now, make sure that the previous "stream to vector" block is also updated to use **`FFTSIZE`** instead of a fixed number like 4, allowing for more flexibility when changing the FFT size in the future.

(Refer Slide Time: 17:34)



Once the flow graph is executed, you should observe a similar spectrum to the one before, this makes sense because we're essentially sending similar data through the system. However, it would be more insightful to visualize how the spectrum or symbols behave when we enable or disable specific frequency subcarriers. To accomplish this, we can explore various methods, but one straightforward approach is to use demultiplexing and multiplexing.

Here’s the plan: we will break the stream down into smaller sub-streams, manipulate specific subcarriers, and then reassemble the stream. We'll begin by creating four QT GUI range sliders, which will act as on/off switches for each subcarrier. Start by using Control+F (or Command+F) to search for the "range" block. Once inserted, name the first one **`S1`** and set the default value to 1. The range will be configured with a start value of 0, a stop value of 1, and a step size of 1, effectively turning the subcarriers on and off.

Next, we'll duplicate this range block three times (using Ctrl+C and Ctrl+V), labeling the new blocks as **`S2`**, **`S3`**, and **`S4`**, respectively. These will control the activation of the four parallel streams (or subcarriers).

After setting up the range blocks, we need to split the main stream into parallel sub-streams using a stream demultiplexer (stream demux). This block takes the continuous input stream and splits it into multiple parallel streams. Search for the "stream demux" block via Control+F (or Command+F), and insert it into the flow graph. Double-click the block and configure the number of outputs to match the FFT size, in this case, 4. The lengths of the outputs should be set to **`[1 times FFT size]`**, ensuring that each of the four outputs corresponds to one of the subcarriers.

Now, we'll connect the output of the modulation stage to the demux block, splitting the data into four parallel streams. Once we have the demuxed streams, the next step is to reassemble them back into a single stream. We accomplish this by using a stream multiplexer (stream mux) block. Search for "stream mux" and insert it into the flow graph. This block will perform the inverse operation of the demux, recombining the individual streams back into one continuous data stream.

Lastly, once the structure is in place, you can tidy up the flow graph for better readability and ensure everything is properly connected.

Let's tidy this up later. Now, for the stream multiplexer (streammux), I'll double-click and set the number of inputs to **`FFTSIZE`**, and for simplicity, we'll set the lengths as **`[1 times FFT size]`**. Once the streammux is configured, we'll connect its output appropriately. The next step is to toggle these individual streams on and off using the ranges we set up earlier.

To do this, we'll create four constant sources that act as multipliers for each stream. Use Ctrl+F (or Cmd+F) and search for the "multiply const" block. I'll insert the first one and name it **`S1`**. Then, I'll duplicate it three times by using Ctrl+C and Ctrl+V, and name them **`S2`**, **`S3`**, and **`S4`**. Once these multipliers are placed, we connect them to the corresponding streams.

Now, if we execute this flow graph, you'll initially see the entire spectrum. But let’s switch off all subcarriers by setting all ranges to zero. As expected, no spectrum will be visible. Now, let’s turn on only **`S1`**. You should observe that only one portion of the spectrum near zero is being used, occupying approximately one-fourth of the total spectrum, which makes sense since we’ve divided the spectrum into four subcarriers.

Next, let’s switch on **`S2`**. You'll notice a similar spectral shape, but this time it will appear around the 2 kHz mark, again, this is logical as the spectrum is being split into four parts. When only **`S3`** is enabled, the spectrum appears around the 4 kHz mark on the positive side and also symmetrically near -4 kHz. This behavior is consistent with the sampling theorem and what you've learned in DSP. Finally, when **`S4`** is on, the center appears around -2 kHz.

In essence, by enabling and disabling these streams, we’ve split the signal into four parallel subcarriers, and you can clearly see how each subcarrier occupies its respective portion of the frequency spectrum.

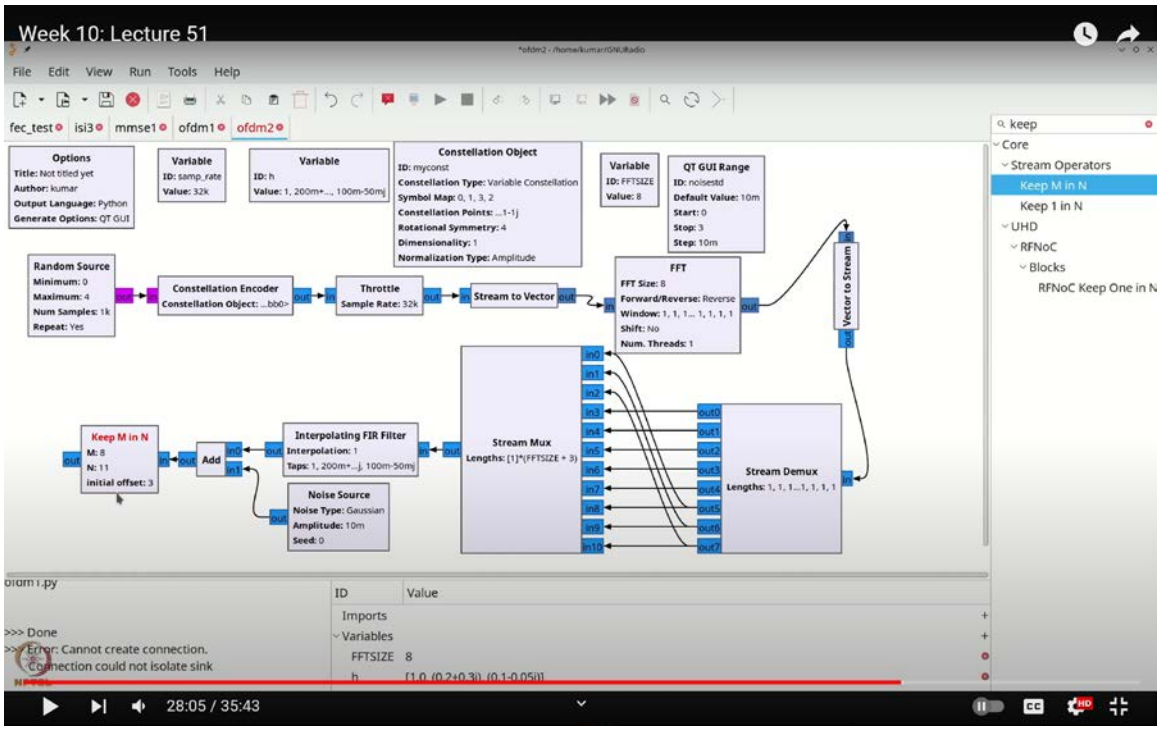
Now, let's explore what happens when we increase the number of subcarriers. A simple way to do this is by increasing the **`FFTSIZE`** from 4 to 8. However, this will require us to add more ranges and multipliers to handle the additional subcarriers. Let’s go ahead and do that now. I'll select the current range and multiplier blocks, copy them using Ctrl+C, and paste them with Ctrl+V. These new blocks will be named **`S5`**, **`S6`**, **`S7`**, and **`S8`**, respectively.

I'll also duplicate the corresponding multiply constants, connecting **`out4`** to **`S5`**, **`out5`** to **`S6`**, **`out6`** to **`S7`**, and **`out7`** to **`S8`**. With everything connected and in place, we can execute the flow graph again.

This time, let’s only enable **`S1`** and keep all others off. You’ll notice the spectrum is even narrower than before, centered around 0 kHz. Now, turn on **`S2`**, and you’ll see a portion of the spectrum centered around 1 kHz. With eight subcarriers, we’re dividing the 8 kHz spectrum into eight equal parts. Similarly, enabling **`S3`** places the spectrum around 2 kHz, and **`S4`** around 3 kHz. When **`S5`** is activated, the spectrum splits between 4 kHz and -4

kHz, a familiar artifact of the sampling theorem and the way the discrete-time Fourier transform (DTFT) is computed, repeating between -π and π.

(Refer Slide Time: 28:05)



This effectively demonstrates that we've parallelized the channels, distributing the signal across multiple subcarriers. The spectrum clearly shows how the data is placed in parallel, validating that our system works as expected.

Now that we've achieved this, it's clear that using OFDM allows your data to appear in the frequency domain without altering the spectral characteristics. It uses the same spectrum but distributes the data across different subcarriers, effectively parallelizing the channel. Our next objective is to demonstrate the ease of equalization in OFDM, specifically by ensuring that, even in the presence of a channel, we can equalize the signal using one-tap equalization.

To keep things straightforward, let's simulate a scenario where we assume one symbol per sample rate, which means our simulation will have one symbol for every sample. We'll start by adding a random source and a constellation encoder. So, as usual, use Ctrl+F (or

Cmd+F) to search for "random source." We'll configure the random source to output 4 bytes and then proceed to add the constellation encoder. For this example, let's call the encoder **`myconst`**, though we haven't created it yet. We'll define **`myconst`** as our constellation object, and with that, our constellation is ready.

Next, we'll add a throttle block to regulate the flow. However, since we are introducing a channel in the simulation, it's important to remember that we need to add a cyclic prefix to account for the channel. Let's assume our channel has at most 3 taps. We'll create a channel model by defining a variable named **`h`**. Let's say the channel model is represented by the following complex taps: **`[1, 0.2 + 0.3j, 0.1 - 0.05j]`**. This will represent our baseband channel.

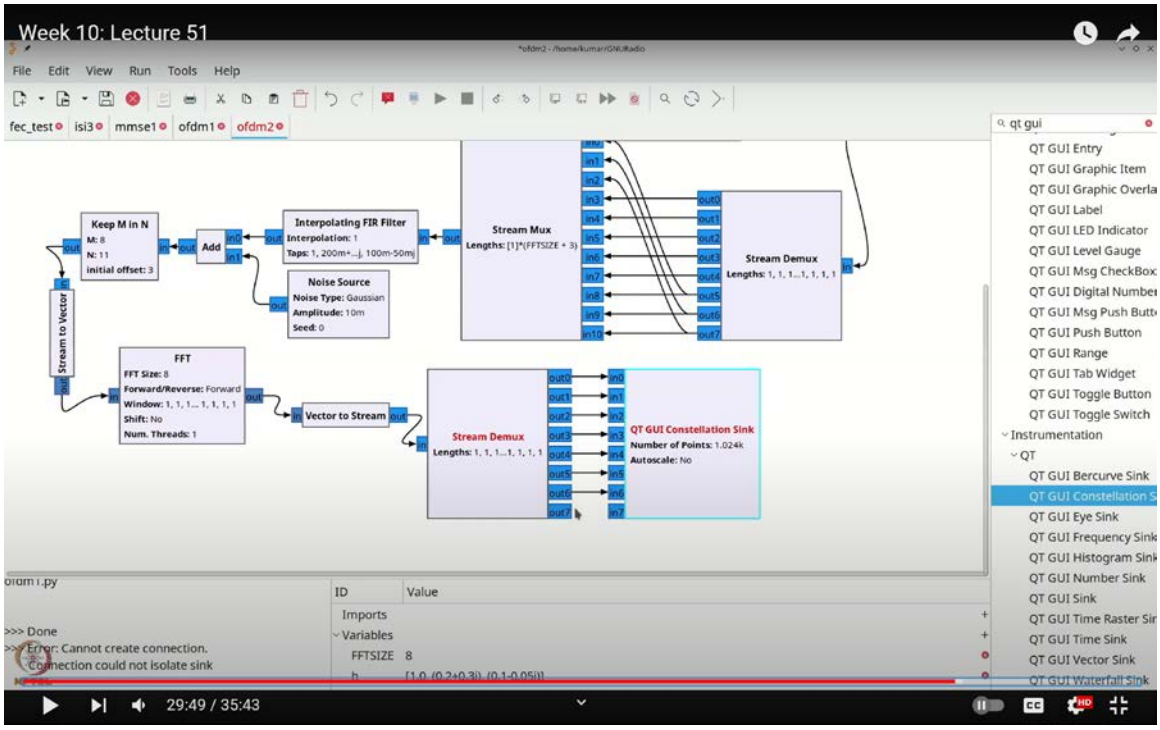
Now, we need to perform a convolution between the transmitted data and this channel. Before doing that, we'll incorporate OFDM, making sure to add a cyclic prefix of length at least 2 to accommodate the channel. To begin, we'll use Ctrl+F (or Cmd+F) to search for and add the necessary blocks. We'll start by bringing in a variable block for the FFT size. Let's name this variable **`FFTSIZE`** and set its initial value to 8.

Next, we'll use a "stream to vector" block to convert the incoming data stream into vectors of size **`FFTSIZE`**. Search for this block using Ctrl+F (or Cmd+F). We'll configure the number of items in this block to match the **`FFTSIZE`** variable. Following that, we'll grab an FFT block. Again, we'll set the FFT size according to our **`FFTSIZE`** variable and configure it for inverse FFT (IDFT). We'll also set the FFT window function to **`[1 times FFT size]`**, an array of ones, and disable shifting.

After the inverse FFT, we'll convert the vectors back into a stream by using a "vector to stream" block. Once this is done, we'll need to demultiplex the stream in order to add a cyclic prefix. Search for the "stream demux" block, configure its length to **`[1 times FFT size]`**, and set the number of outputs to **`FFTSIZE`**. This will effectively split the stream into multiple parallel outputs.

Now, to handle the cyclic prefix, we'll use a stream multiplexer (streammux). This will allow us to insert the cyclic prefix into the data stream. We'll add a streammux block and configure it to accommodate the additional data for the cyclic prefix.

(Refer Slide Time: 29:49)



Once all the components are set, we'll rotate the blocks for better visibility using the arrow keys. This will help organize the flow graph neatly. Now, with the cyclic prefix added and everything properly aligned, we'll be able to effectively simulate the OFDM system with channel convolution and observe how the one-tap equalization performs under these conditions.

Now, I'll grab a stream mux and double-click to adjust its parameters. Let's set the cyclic prefix length to 3. This means the length of the stream will be **`[1 * FFT size + 3]`**, so the total length becomes **`FFT size + 3`**. This creates a larger stream mux that accounts for the cyclic prefix. I'll use the right and left arrow keys to rotate the block as needed for the connections.

I’ll now connect the "back parts" of the data stream directly, since the cyclic prefix is positioned at the front. After connecting these back parts, I’ll connect **`out 7`** to **`IN2`**, **`out 6`** to **`IN1`**, and **`out 5`** to **`IN0`**. This step introduces the cyclic prefix by prefixing the actual symbols with the last 3 symbols, ensuring that the convolution won't distort the data.

Next, we bring in the channel model by searching for the "interpolating FIR filter" using Ctrl+F (or Cmd+F). The filter block also needs to be rotated, use the left arrow twice to do that. The filter taps will be set to **`H`**, which performs the convolution. Now that we have our output from the convolution, we’ll introduce noise into the system.

Search for a noise source using Ctrl+F (or Cmd+F) and rotate it using the left arrow key. After that, grab an "add" block and rotate it as well. The noise source’s amplitude will be set by a variable called **`noise_std`**. To adjust this, we need a range block, use Ctrl+F (or Cmd+F) to find it. This range will allow us to vary the standard deviation of the noise, starting from 0.01, with a stop value of 3, and increments of 0.01. The default value for **`noise_std`** should be 0.01, and the range will start at 0.

Now we have a noise source incorporated, and our output will clearly be affected by both the channel and noise. To make the flow graph clearer, I’ll move the constellation object over to free up some space. With this adjustment, the flow graph looks more organized.

At the receiver side, the first task is to remove the cyclic prefix and then perform equalization. The equalization process will be simple, just a single-tap equalizer. Now, there are two ways to remove the cyclic prefix. One method involves using a stream demux to separate the **`FFT size + 3`** elements and send the first 3 to a null source. But, we will use a simpler approach.

We’ll use a block called **`keep m in n`** (Ctrl+F or Cmd+F). This block is quite handy, it lets you keep only **`m`** elements out of **`n`**, starting from a defined initial offset. In our case, the first 3 elements are always going to be the cyclic prefix, so we set the initial offset to 3 and keep only **`FFT size`** elements out of **`FFT size + 3`**. By doing this, we effectively remove the cyclic prefix from the data stream.

The remaining task is to take the FFT of the signal to see what we get back. I'll search for the "stream to vector" block, rotate it using the arrow keys, and set the number of items to **`FFT size`**. After that, I'll grab the FFT block and configure it. The FFT block needs to be in forward mode, so I'll copy the existing block, set it to forward, and connect it to the stream.

Now, I'll take the result and convert it back to a stream using the "vector to stream" block. After rotating the block appropriately, I'll prepare to visualize the output using a QT GUI constellation sink. Search for the constellation sink (Ctrl+F or Cmd+F) and set the number of inputs to 8, as we are working with 8 parallel streams.

Finally, I'll demultiplex the stream so that we can view each of the 8 constellations individually on the GUI. By copying the necessary blocks and arranging them, we can view each stream's constellation separately on the QT GUI.

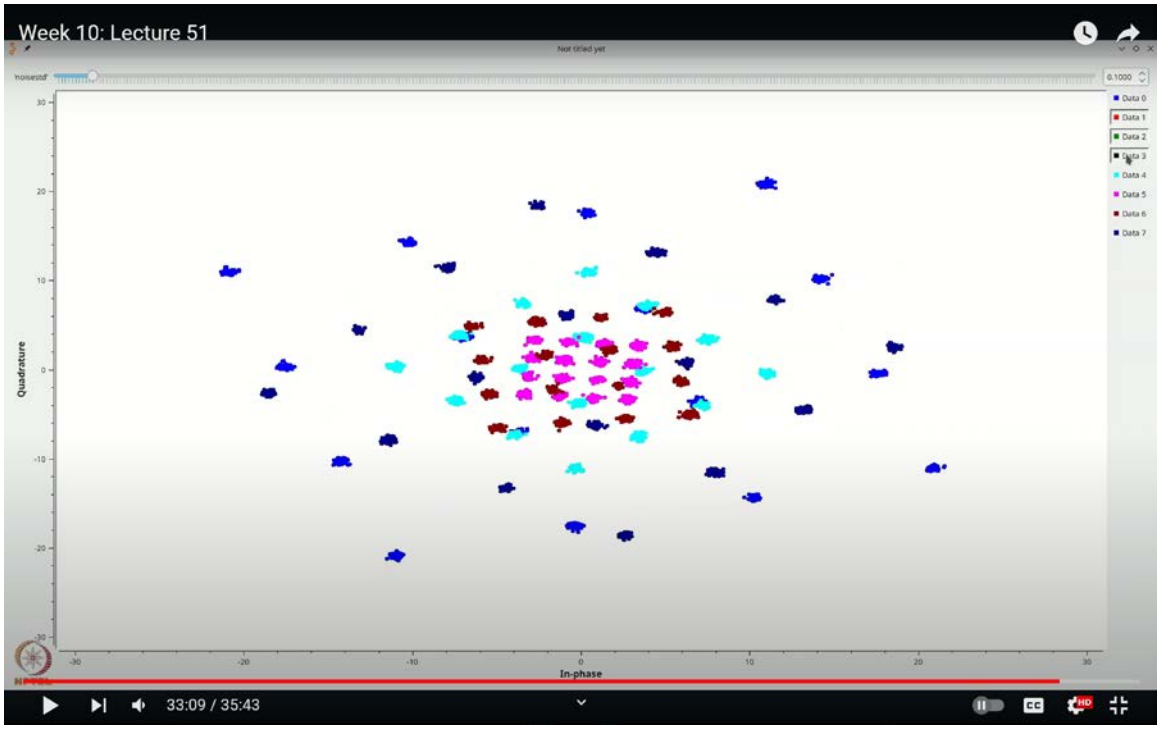
Now that everything is connected, we are ready to run the flow graph. When we execute it, a pattern appears, but one minor issue is that all the signals seem to be the same color, which can make it difficult to differentiate between them. To fix this, you can double-click on the respective elements and change their colors in the configuration settings. For example, we can assign blue to the first stream, red to the second, green to the third, black to the fourth, cyan to the fifth, magenta to the sixth, dark red to the seventh, and dark blue to the eighth. This provides better visual separation of the streams.

Once we execute the flow graph again, we encounter a scaling issue due to the FFT, which automatically scales the output. We haven't accounted for this scaling yet. So, let's examine just one of the outputs by modifying the display settings. Interestingly, we observe a QPSK-like constellation, but it's rotated. As you might know, a rotation in the QPSK constellation can be easily corrected with a simple multiplication to de-rotate it.

Let's move on to the second stream. It also shows a rotated QPSK-like constellation. The third stream displays a similar pattern, as does the fourth. This indicates that the inter-symbol interference introduced by the channel is not significantly detrimental because all that's required to recover the signal is a simple scaling adjustment. However, you may

notice that the amplitudes of some constellations vary, some are higher, some lower. This variation is a result of the frequency-selective nature of the channel, which affects different parts of the signal differently.

(Refer Slide Time: 33:09)



Different subcarriers are affected by the channel with varying degrees of impact. Interestingly, the amplitudes of certain subcarriers tend to be close to one another, such as subcarrier 0 and data1, subcarriers 2 and 3, and subcarriers 3 and 4. However, subcarrier 1 and subcarrier 4 may exhibit noticeable differences, as well as subcarrier 1 and subcarrier 6. This behavior is due to the gradual variation of the channel's frequency response, which manifests in the frequency domain.

Now, let's experiment by changing the channel. For instance, let's modify the taps to **`0.7 + 0.7j`** and **`-0.1`** or **`-0.4`**. With this altered channel, you'll observe new characteristics across the subcarriers. Let's examine them one by one: subcarrier 1, subcarrier 2, and so on. You'll notice that subcarriers 7 and 8 appear weaker.

If you’re skeptical about whether this system is functioning properly, we can further demonstrate it by switching to a QAM 16 constellation. We’ll adjust the constellation to QAM 16 and modify the random source to output values between 0 and 16. When viewed, you'll notice numerous dots on the display. Let’s focus on just one of the streams. You can clearly see a rotated QAM 16 constellation. The same rotation appears in the other streams as well, though some show weaker constellations due to the channel’s effect. Nevertheless, they all exhibit the characteristic QAM 16 pattern, simply rotated.

This illustrates the power of OFDM. The technique of parallelizing the channel allows for very simple equalization, just a single-tap equalizer suffices. It’s almost as if the transmission is pre-equalized, without any detailed knowledge of the channel. If you’re interested in diving deeper into OFDM using GNU Radio, there are more sophisticated implementations that handle the allocation of data into OFDM frames, transmission, reception, and more. Those are worth exploring on your own.

In this lecture, we built a basic OFDM simulation in GNU Radio, specifically a baseband OFDM system. We demonstrated that even in the presence of a channel, OFDM can mitigate the channel’s effects by transforming a wideband signal into multiple narrowband sub-channels using the DFT. We saw that regardless of the constellation used, each subcarrier undergoes only a multiplication by a complex number, which makes equalization straightforward. This is why OFDM is effective in converting a frequency-selective wideband channel into several nearly frequency-flat narrowband channels. It is an extremely useful technique, which explains its widespread adoption in modern wireless and wired communication standards, including Wi-Fi, LTE, and many others. Thank you.