Digital Communication using GNU Radio

Prof. Kumar Appaiah

Department of Electrical Engineering

Indian Institute of Technology Bombay

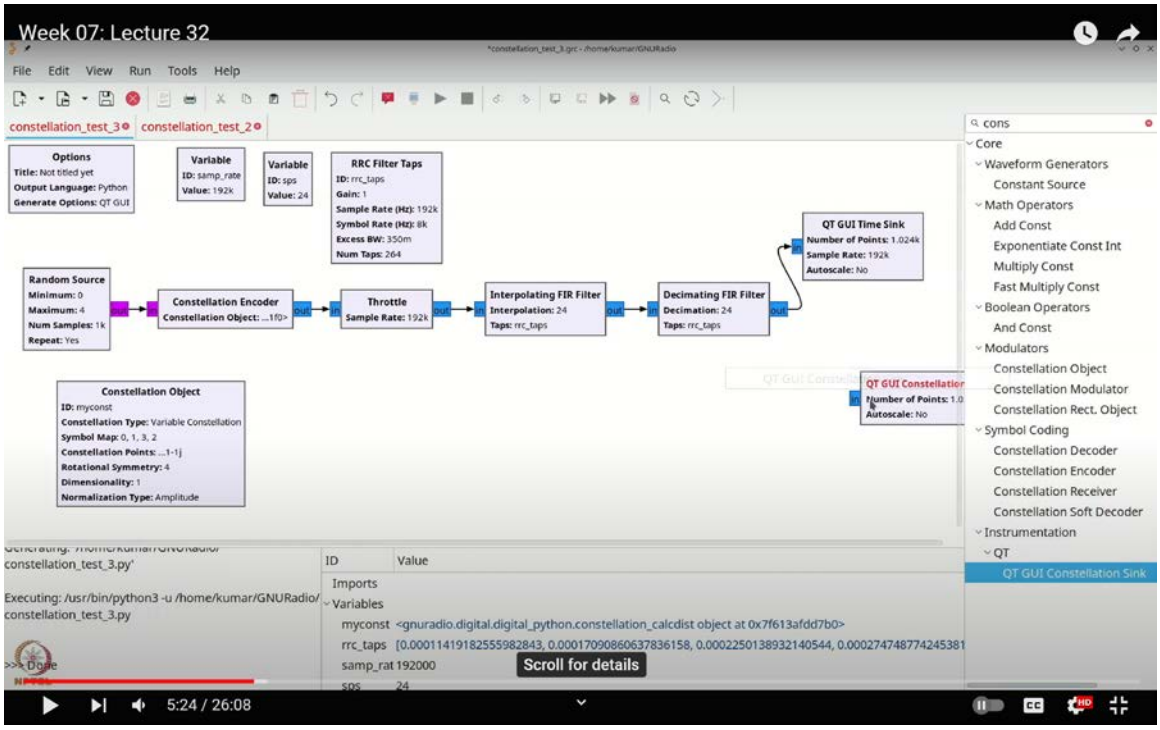
Week-07

Lecture-32

End-to-End Digital Communication System Simulation in GNU Radio

Welcome to this lecture on Digital Communication using GNU Radio. My name is Kumar Appiah, and I am with the Department of Electrical Engineering at IIT Bombay. In this lecture, we will integrate all the components we've covered in previous sessions to create a complete system simulation. This involves generating data, converting it into symbols, upconverting to the passband, receiving it with noise, downconverting to the baseband, and finally, visualizing and verifying that we can accurately recover our symbols both with and without noise. This process will give us a comprehensive understanding of a practical communication system.

(Refer Slide Time: 05:24)



Let's start, as usual, by introducing a random source. Press Ctrl+F or Command+F and search for "random." Select the random source, and configure it to output bytes. Since we will be using QPSK, set the parameter to 4.

Next, add a constellation encoder. Press Ctrl+F or Command+F, search for "encoder," and select the constellation encoder. Connect this to the random source. Now, add a throttle. Press Ctrl+F or Command+F, search for "throttle," and connect it accordingly.

Before proceeding, we need to define the constellation. Search for "constellation object" using Ctrl+F or Command+F, and select it. Double-click to open it, and name it "my$_c$onst," which is perfectly fine. Leave the default variable constellation setting as it is; this will be a gray-coded QPSK constellation.

Now, we need to ensure that we use the constellation we have created. Our next step is to apply a pulse shaping filter. For this, we'll use a standard root raised cosine (RRC) filter. Press Ctrl+F or Command+F, search for "RRC," and select the RRC filter taps. Double-click to configure it. I made one adjustment: I changed the sampling rate to 1920.00 for convenience.

We will also use a "samples per symbol" variable. Press Ctrl+F or Command+F, search for "variable," and place it in your setup. Name it "SPS" (samples per symbol). I want the symbol rate to be 8000, so set this to **`Samp rate / 8000`**. The double division here ensures integer division, automatically computing the samples per symbol as 4. This means the symbol rate is essentially **`Samp rate / SPS`**, which will correctly yield 8000. This configuration will automatically compute the RRC filter taps for us.

Next, we'll pulse shape our constellation-encoded symbols. To do this, we'll add an interpolating FIR filter. Press Ctrl+F or Command+F, search for "interpolating FIR filter," and connect it. Set the interpolation factor to SPS because we want this filter to produce 24 samples for each input symbol. Use "RRC_taps" as the filter taps.

To verify that this filter works correctly in conjunction with the decimating FIR filter, let's add a decimating FIR filter. Press Ctrl+F or Command+F, search for "decim," and select

the decimating FIR filter. Connect it, set the decimation to SPS, and use the same "RRC_taps."

Finally, to check that everything is working as expected, add a constellation sync and a time sync. Press Ctrl+F or Command+F, search for "time sync," and add it. Then, add a constellation sync as well. This setup will allow us to visualize and verify that the constellation looks as expected.

When we run this, it appears quite small and very tiny. The reason for this is that the filter has a scaling issue. To correct this, double-click on the filter and set the gain to about 5, which should address the problem. This adjustment will bring the result closer to 0.707.

However, this is not exactly 0.0707 because this value should actually be the square root of 24. To get this right, you can compute the square root of 24 manually, or you can use the numpy module. To do this, press Ctrl+F or Command+F, type "import," and import numpy as np. Then, use **`np.sqrt(SPS)`** to calculate the square root of the samples per symbol (SPS). This will handle the scaling correctly, and you should see a constellation value very close to 0.0707.

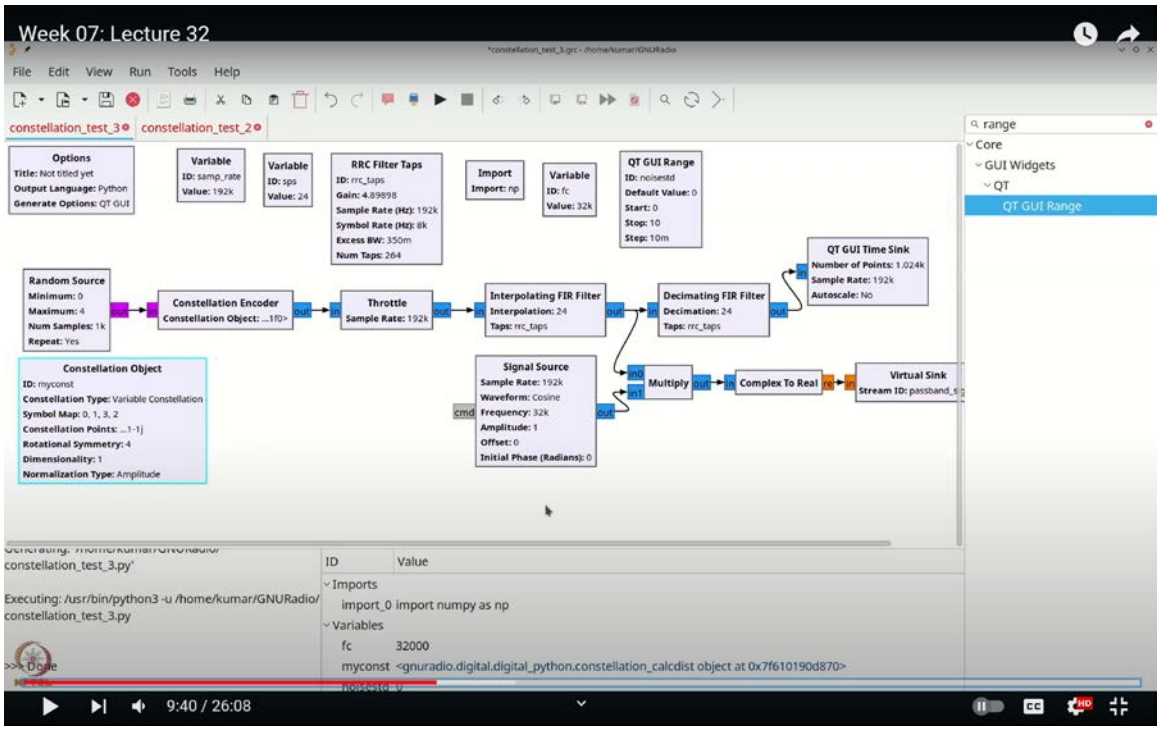
We now have a clean stream of data displayed, showing the baseband complex data in terms of its real and imaginary parts after decimation. This setup provides a good template for verifying that the received data matches what is expected. For now, we can temporarily remove the constellation sink, as we do not need it.

Our next step is to convert this signal to passband. To do this, we will add an **`$f_c$`** variable. Press Ctrl+F or Command+F, search for "variable," and create an **`$f_c$`** variable. Set **`$f_c$`** to 32000, which should be a safe choice given that our baseband signal is around 8000; this should not cause any issues. You can adjust this value if necessary.

Next, we need to modulate the output of the interpolating FIR filter using a complex exponential signal. Ideally, this would involve computing **`$e^{(j2\pi f_c t)}$`**, which can be expressed using **`$\sqrt{2}\cos(2\pi f_c t)$`** and **`$\sqrt{2}\sin(2\pi f_c t)$`**. However, to simplify, we can use a single complex signal source. Press Ctrl+F or Command+F, search for "signal source,"

and select it. Set it to cosine with a frequency of `$\mathbf{f_c}$`. The amplitude can be adjusted later, and we may need to apply some scaling as well.

(Refer Slide Time: 09:40)



We will now multiply these two signals. To do this, press Ctrl+F or Command+F and search for "multiply" to find the multiplication block. Use this block to multiply the two signals and then extract the real part. Specifically, you want the real part of $x \cdot e^{j2\pi f_c t}$.

For this purpose, let's use the "Complex to Real" block. Press Ctrl+F or Command+F, and search for "complex to real" to find this block, which is more appropriate than using "Complex to Float" in this context. By using the "Complex to Real" block, we will obtain our passband signal.

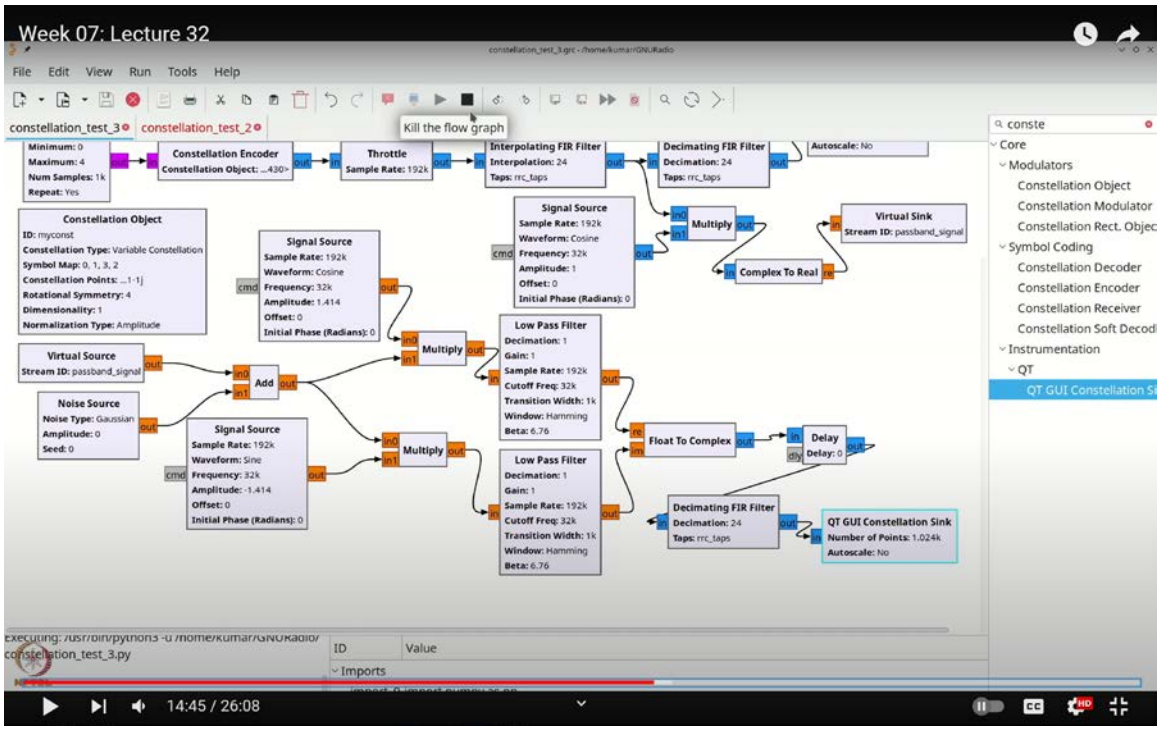
Next, I will route this passband signal to a virtual sink to facilitate further processing. Press Ctrl+F or Command+F, search for "virtual sink," and use it to label the signal as `**passband_signal**`. With this setup, we now have our passband signal ready for the receiver.

At the receiver end, we need to introduce noise to the signal and then attempt to recover the original signal. To add noise, let's start by including a noise source. We will add a range for our noise standard deviation (`**noise_std**`). Although the exact scaling is not crucial at this stage, you can adjust it later if needed. We will apply some Gaussian noise.

For the next step, place a "Virtual Source" by pressing Ctrl+F or Command+F, searching for "virtual source," and setting the stream ID to `**passband_signal**`. Arrange the virtual source by moving it down to create some space. Ensure the virtual source is properly connected to the signal flow.

Now, to add noise, press Ctrl+F or Command+F and search for "noise." We will use a real noise source, which mimics real-world noise conditions, set with the `**noise_std**`. To integrate the noise, grab an "Add" block by pressing Ctrl+F or Command+F, searching for "add," and ensure it is configured as a float add. Connect this block accordingly to the signal flow.

(Refer Slide Time: 14:45)



Now that we have our virtual source with the passband signal and added noise, we can proceed with the receiver processing. The receiver's tasks involve multiplying by $\sqrt{2}\cos(2\pi f_c t)\ and\ -\sqrt{2}\sin(2\pi f_c t)$, followed by filtering the outputs with a low-pass filter to recover the baseband signal.

Let's implement this step-by-step. First, add a signal source by pressing Ctrl+F or Command+F and typing "signal." Place this block in your flowgraph, and configure it with a frequency $\mathbf{f_c}$, set it to cosine, and use an amplitude of $\sqrt{2}$. For convenience, approximate $\sqrt{2}$ as 1.414, and set the data type to float.

Next, duplicate this signal source by copying it (Ctrl+C) and pasting it (Ctrl+V). Adjust the new source to output $-\sqrt{2}\sin(2\pi f_c t)$.

Now, we will multiply these signals with the passband signal. To do this, press Ctrl+F or Command+F, search for "multiply," and select a real multiplier block. Connect the passband signal to both multipliers, and then connect each multiplier to the corresponding signal sources. Duplicate the multiplier setup if needed.

After the multiplication, we need to filter the results to remove the 2 $\mathbf{f_c}$ component. Add two low-pass filters by pressing Ctrl+F or Command+F, searching for "low pass filter," and selecting a float low-pass filter. Set the decimation to 1, and configure the cutoff frequency. In this case, you can set the cutoff frequency to $\mathbf{f_c}$ or a reasonable value such as 1000 Hz.

Copy and paste the low-pass filter to create a pair, ensuring both filters have the same settings. This filtering process will remove the 2 $\mathbf{f_c}$ components from the signals.

Finally, combine the filtered signals to reconstruct the baseband signal. Before completing the setup, ensure to perform any necessary decimation steps to match the sample rate requirements.

Let's start by combining the filtered signals and then applying decimation. To do this, we'll use a "float to complex" block, so press Ctrl+F or Command+F, and search for "float to

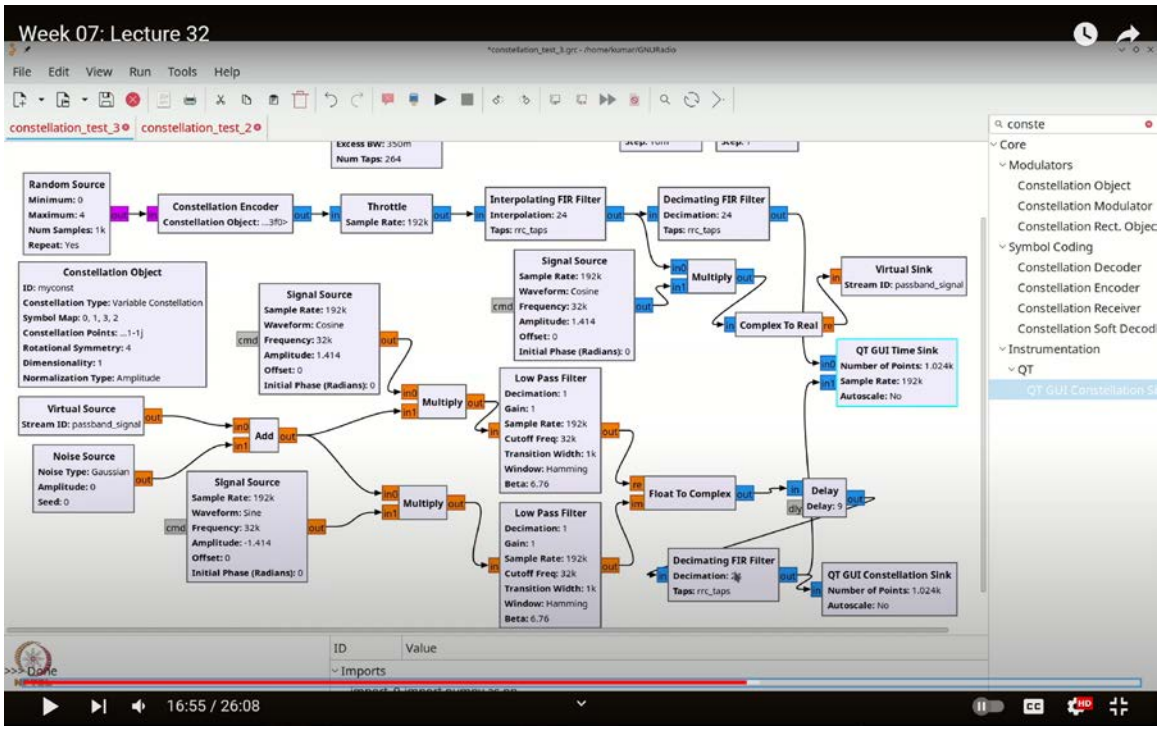
complex." This will allow us to proceed with matched filtering since we now have our baseband signal ready.

Next, we will use a decimating filter, which we can copy from earlier. Press Ctrl+C to copy and Ctrl+V to paste the decimating filter. However, we'll need to add a delay element to ensure that the sampling is properly aligned. To add a delay, press Ctrl+F or Command+F, search for "delay," and select a delay block.

We will configure this delay element with a range from 1 to the samples per symbol (SPS), which should be suitable for our needs. Name this delay element accordingly.

Now, we need to connect the delay element to our decimating filter. The delay is essential because we introduce a causal element with the low-pass filter, and we need to account for its length and group delay to accurately recover our signal.

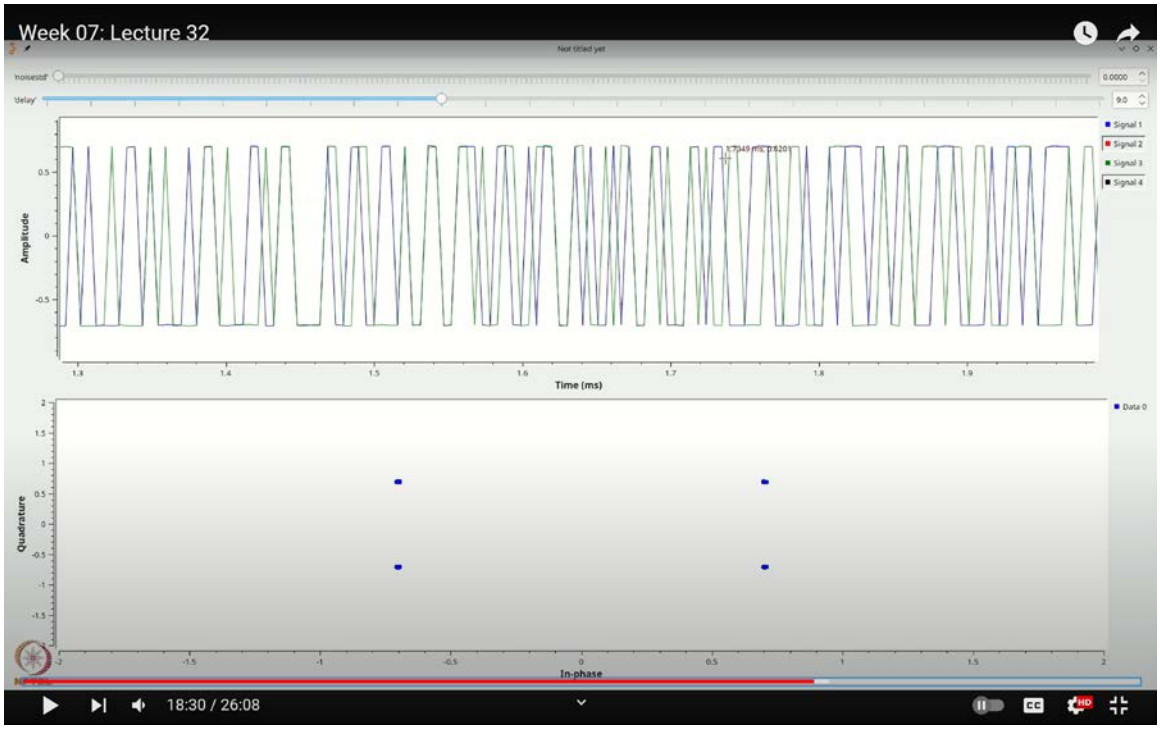
(Refer Slide Time: 16:55)



Finally, add a constellation sync block to visualize the recovered signal. Press Ctrl+F or Command+F, search for "constellation sync," and add this block to the flowgraph.

If you execute the flowgraph and encounter issues, ensure that the delay element is correctly set. Make sure to configure the delay setting appropriately so that it reflects the delay we introduced.

(Refer Slide Time: 18:30)

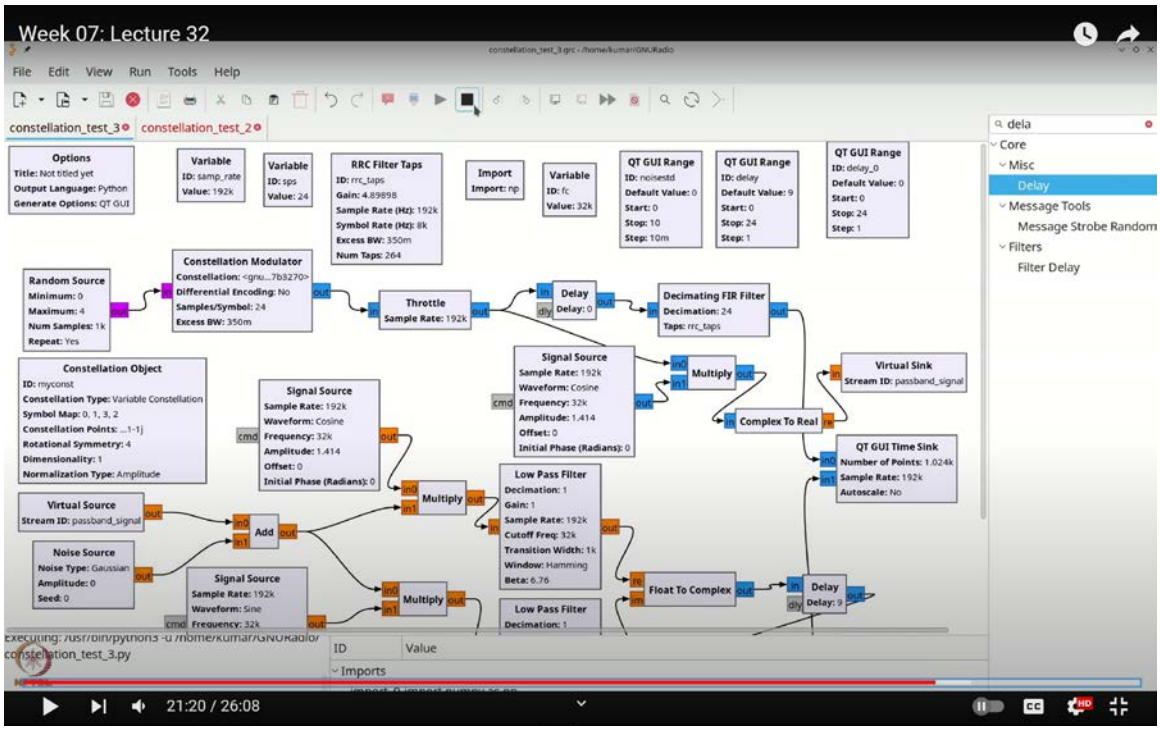


Now, when you execute the flowgraph and adjust the delay, you'll notice that the constellation starts to align, with a result of 0.5, 0.5. This indicates a scaling issue. Specifically, we needed the original $e^{j2\pi f_c t}$ to be scaled to approximately 1.414. By setting the delay to 9, we correct this issue. For convenience, let's set the default delay range to 9 so we don't have to constantly adjust it.

As you increase the noise, you'll observe that the constellation behaves similarly to how it did in earlier scenarios. Now, for the final step, although we should ideally be able to recover the data as well, I won't delve into adding a bit error rate or symbol error rate counter at this point. This is because practical delays need to be aligned before performing data computation. In real-world systems, delays can be more significant than our set delay

of 9 due to various system issues. Ultimately, recovering the signal and managing transmission start times will need to be addressed, which we will explore later.

(Refer Slide Time: 21:20)

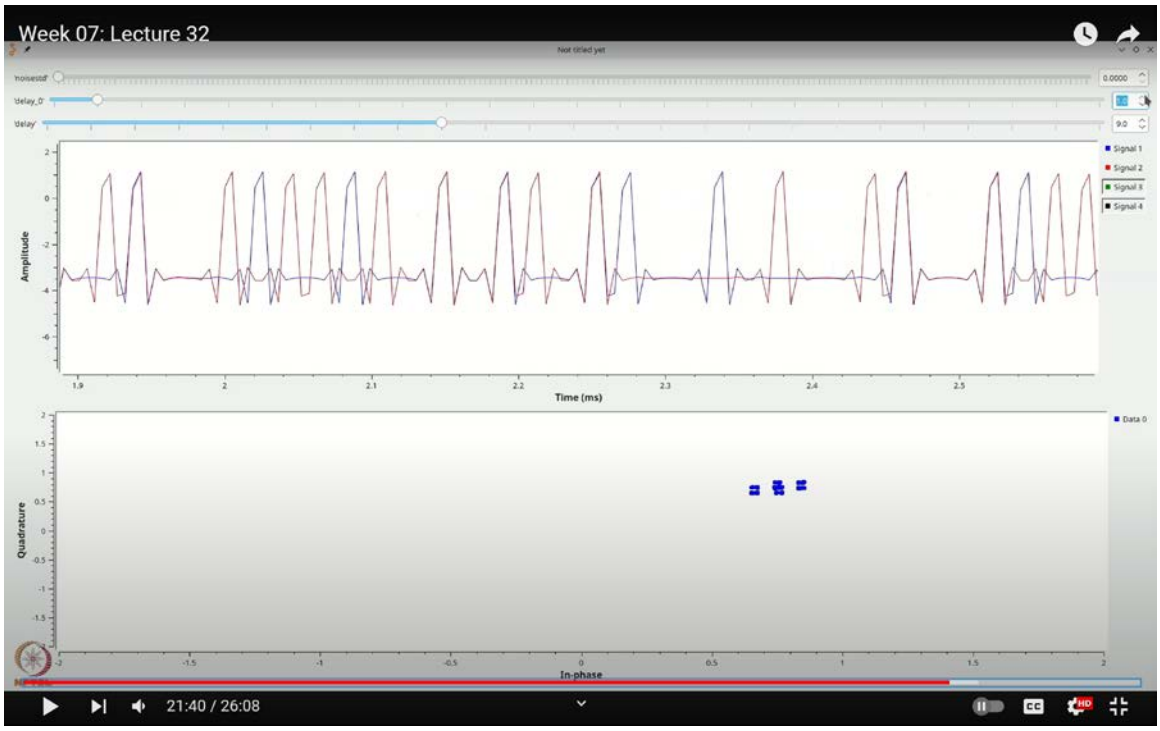


For now, we can get a preliminary idea of whether the two sequences match. One approach is to use the time sync block to compare the outputs. Connect the output from the decimating filter to the same time sync block. This allows us to examine if there is any discernible pattern.

Let's pause and zoom in for a closer look. In the time sync view, signal 1 and signal 3 represent the real part, while signal 2 and signal 4 represent the imaginary part. Temporarily hide signals 2 and 4 to simplify the view.

Now, observe the remaining signals. Although it’s not easy to discern, the blue signal represents the original, and the green signal should lag behind. Any patterns you see should appear first in the blue signal and then in the green signal. This will help confirm if the sequences are indeed aligned.

(Refer Slide Time: 21:40)



When analyzing the signals, any pattern observed in the blue signal should eventually appear in the green signal, albeit with a delay. Let's examine a specific location to identify if we can see a pattern. For instance, if we hide the green signal and focus on the blue one, we can look for distinct features such as four sharp peaks surrounded by two less pronounced peaks.

In the blue signal, if we observe four sharp peaks followed by two less sharp peaks, these should also appear in the green signal but delayed. As you can see, the four sharp peaks and two less sharp peaks are evident in the blue signal and are similarly reflected in the green signal, confirming a noticeable delay.

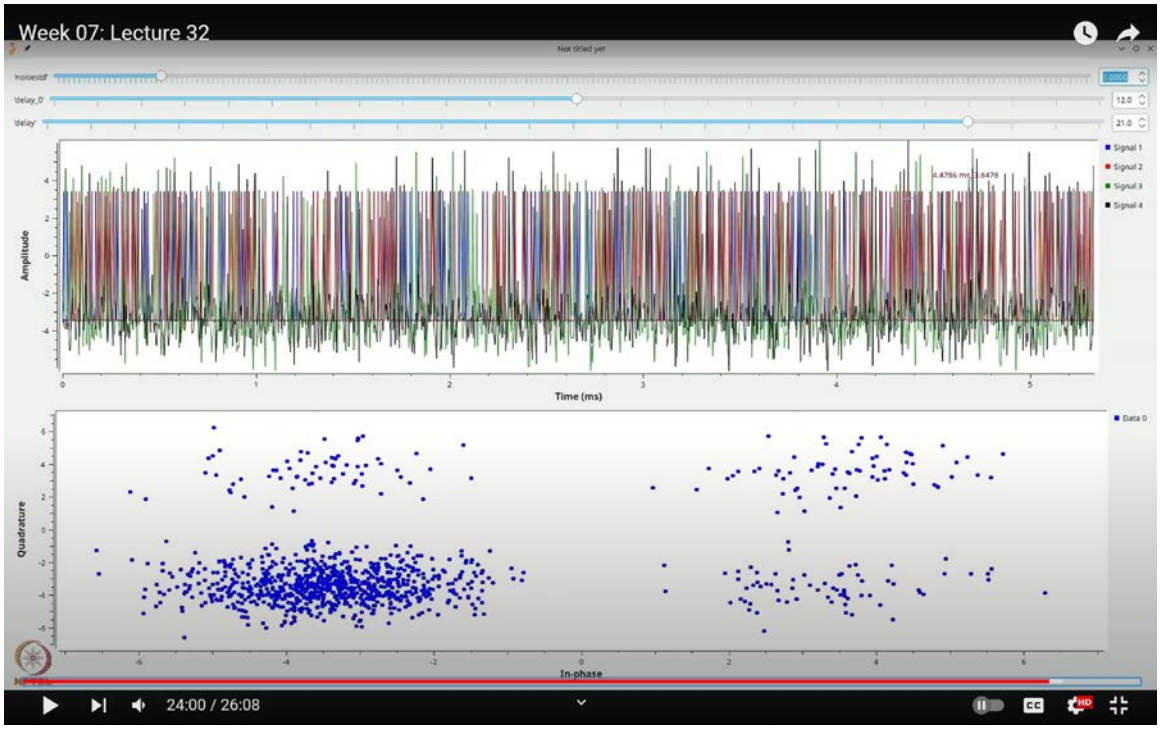
To quantify this, consider the first sharp peak in the blue signal occurs at 1.63 microseconds, while in the green signal it appears at 1.68 microseconds, indicating a practical delay. Let's now check the imaginary part of the signals. If we hide the other signals and focus on the red (early) and black (delayed), we should see the large valley

between the two flat peaks in the red signal appearing about half a millisecond later in the black signal.

Indeed, this large valley is present in both signals, validating that the delay is consistent. Aligning your transmissions should allow for accurate data recovery without major issues.

For simplicity, since using a constellation encoder and interpolating FIR filter is a common practice, you might consider combining these functions into a single block to streamline the process.

(Refer Slide Time: 24:00)



For instance, you could simplify your setup by removing both the interpolating FIR filter and the constellation encoder. Instead, use the constellation modulator, which has some convenient features. It includes an integrated RRC generator, and by specifying the constellation as **`MYCONST`**, setting differential encoding to **`NO`**, and configuring samples per symbol to **`SPS`**, you can automate the modulation process. Just make sure to connect it appropriately in your flow graph.

Upon executing the flow graph, you'll notice that the constellation signals may still exhibit some issues due to delay-related problems. Unlike the sharp peaks seen previously, these signals might not be as distinct. This suggests that the internal filter used by the constellation modulator might be introducing some delay.

To address this, add an additional delay. You can do this by copying and pasting a delay block, renaming it to **`delay_0`**, and setting its value to 0. Insert this delay between the relevant components in your flow graph. After adding this delay, execute the flow graph again. You should observe that the appearance of the signals improves, with the peaks becoming clearer. However, if you continue to increase the delay beyond a certain point, the signal quality may deteriorate.

Next, adjust the delay on the receiver side. With the receiver delay set to 0, gradually increase it from 0 to 9. If you originally had a delay of 12 on the sender side, and you add 9 on the receiver side, you should reach a total delay of approximately 21, which aligns with the expected value. You might notice a slight artifact in the signal, likely due to the delay being between 12 and 13. This minor artifact is typical, but the overall result is acceptable.

Keep in mind that using the constellation modulator can present challenges due to its internal filter, which might not match the RRC filter you are using. Although the constellation demodulator could potentially handle these discrepancies, for the moment, sticking with this approach should suffice.

This flow graph might seem complex, but it provides a comprehensive view of how these systems can be constructed. Practically speaking, it covers essential aspects: data generation, modulation, upconversion to passband, downconversion to baseband (including removal of spurious frequencies), matched filtering, and constellation recovery. You've also encountered some issues, such as the impact of noise.

Let's first set the delay values to 12 and 21. High levels of noise can introduce significant problems. For instance, excessive noise can cause considerable interference and distortions, which must be addressed. Additionally, there are other practical challenges we

haven't explored yet, such as incorrect carrier frequencies ($\mathbf{f_c}$) at the receiver or phase offsets at the source. These issues can complicate data recovery and will be addressed in future lectures.

In this lecture, we have assembled various components of practical modulation and demodulation. Specifically, we covered how to convert random data into symbols, shape these symbols into a waveform, upconvert using a carrier frequency, add noise, downconvert, apply matched filtering, and finally recover the symbols and bits. This forms the basic framework of a communication system.

In subsequent lectures, we will use this foundational template to explore more practical aspects, including frequency offsets, phase offsets, channel impairments, and other real-world challenges. We will relax our assumptions about the receiver and investigate how receiver impairments affect system performance. Thank you for your attention.