Digital Communication using GNU Radio

Prof. Kumar Appaiah

Department of Electrical Engineering

Indian Institute of Technology Bombay

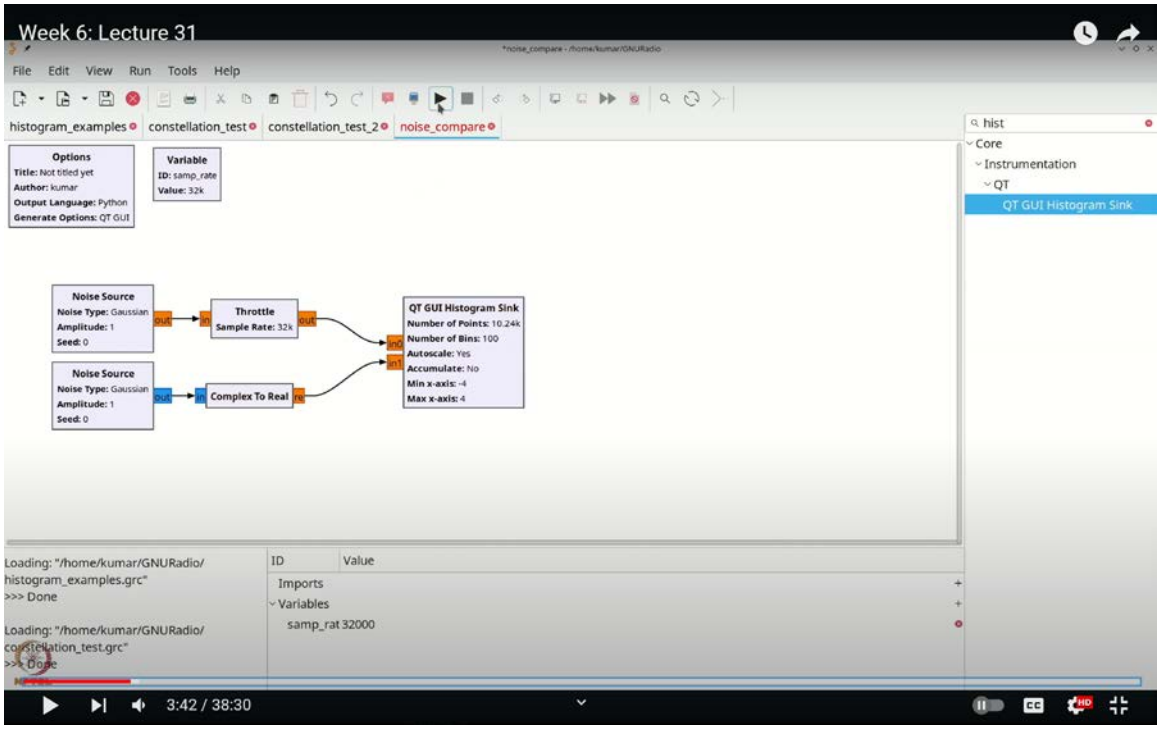
Week-06

Lecture-31

Computing Bit Error Rates in GNU Radio

In our previous GNU Radio exercises, we've primarily focused on symbol error rates. However, it's important to recognize that there is a significant difference between symbol error rates and bit error rates. In today's lecture, we will delve into the methodology for calculating bit error rates (BER) for various constellations. We aim to verify that these bit error rates align with the theoretical expectations discussed earlier in class.

(Refer Slide Time: 03:42)



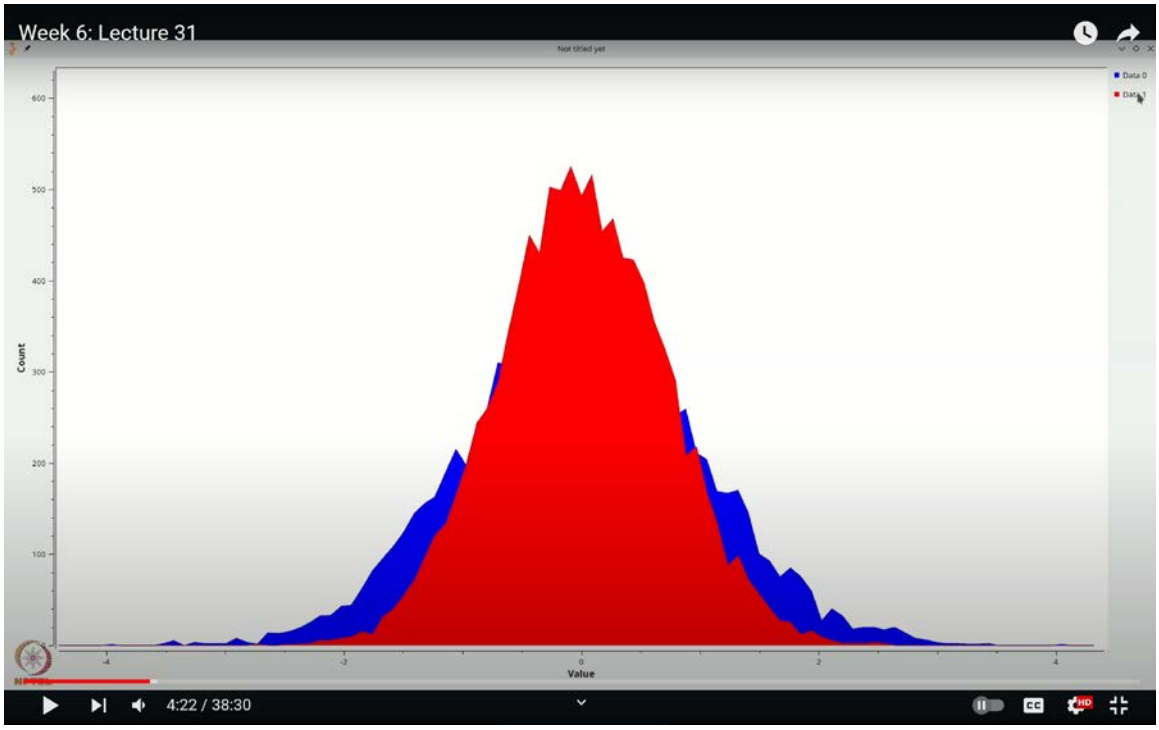
To achieve this, we will generate random symbols and observe the resulting bit errors associated with these symbol errors. Additionally, we will confirm that the calculated bit

error rates are consistent with our theoretical evaluations. Before diving into the specifics of bit error rates, it's crucial to examine the Gaussian noise source available in GNU Radio.

Let's start by locating the Gaussian noise source in GNU Radio by using the search function (Ctrl+F or Cmd+F) and typing "noise" to bring up the noise source block. For reasons that will become evident soon, we'll also need to grab a second noise source.

I’m going to copy and paste this noise source by pressing Ctrl+C (or Cmd+C on Mac) and then Ctrl+V (or Cmd+V) to duplicate it. Now, let’s configure the first noise source to output a float type. Both noise sources should have an amplitude of 1, but with the first set to float and the second to complex.

(Refer Slide Time: 04:22)



Next, we’ll want to visualize the distribution of these two noise sources, ideally on the same histogram. To do this, we’ll extract the real part of the complex noise source. We can accomplish this by using the "Complex to Float" block, which we can find by searching with Ctrl+F or Cmd+F. However, for simplicity, let's use the "Complex to Real" block, which directly provides the real part.

Once we have the "Complex to Real" block, we'll add a throttle block to control the flow of data. Again, search for "Throttle" using Ctrl+F or Cmd+F, and place it in the flow graph, configuring it to output a float type.

Let's connect everything together now. Next, we'll grab a Qt GUI histogram sink to visualize our data. You can do this by searching for "hist" using Ctrl+F or Cmd+F. We want to compare the histograms of two signals, so double-click on the histogram sink to configure it.

As usual, let's increase the number of data points to improve the clarity of our visualization. Setting the number of bins to 100 should work well. For the x-axis, we'll set the range from -4 to 4 since we know that most of the values of a Gaussian distribution lie between -3 and 3. Additionally, we'll configure the histogram to display two inputs by setting the number of inputs to 2. Now, connect the signals to the histogram, and we're ready to visualize the flow graph.

Upon running the flow graph, you will observe two different data sequences. The first, labeled as Data 0, represents the real part of the noise source, while the second, labeled as Data 1 and shown in red, represents the real part of the complex noise source. As you can see, the complex noise source (red) is narrower and taller compared to the real noise source, which appears fatter and shorter. If you hide one of the sources, you'll notice that the difference in shapes is even more pronounced.

Let's make our analysis a bit more precise. We can add one more data point and increase the sampling rate to 192,000 to refine our visualization. With these adjustments, the distinction becomes clearer: the red histogram is noticeably taller, while the blue one is shorter and wider.

Now, what does this difference imply? Recall the expression for a Gaussian distribution:

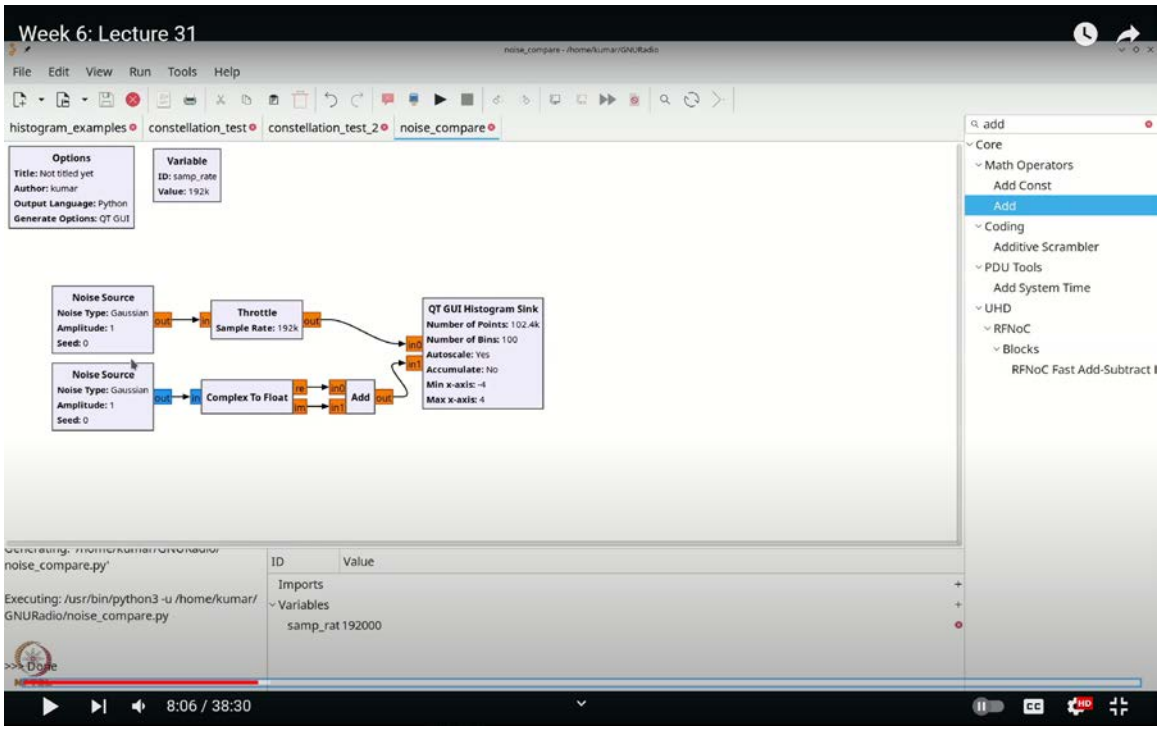
$$\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}}$$

For a zero-mean Gaussian, when the standard deviation (σ) is smaller, the distribution is narrower and taller. Conversely, when σ is larger, the distribution becomes wider and shorter. This observation confirms that the real part of the complex noise source has a lower variance.

Why is this? The complex Gaussian noise source generates noise with both real and imaginary components, each contributing half of the total variance. Thus, the red part in our histogram has a variance of 0.5, and the imaginary part (if visualized) would also have a variance of 0.5.

If you're skeptical, we can prove this by leveraging the fact that the Gaussian noise source essentially consists of two independent Gaussian distributions. By extracting the independent Gaussian components from the real and imaginary parts and summing them, we should obtain similar variances.

(Refer Slide Time: 08:06)



To demonstrate this, let’s remove the previous setup and introduce a "Complex to Float" block to separate the real and imaginary parts of the complex noise source. These two

components are independent of each other. Now, search for an adder block (Ctrl+F or Cmd+F, type "ADD") and set it to float. Connect the real and imaginary parts to the adder and visualize the output.

You'll observe that the two histograms overlap, indicating that both parts have zero mean and the same variance. This confirms our hypothesis: the real and imaginary parts of the complex Gaussian noise source each have a variance of 0.5. The overall variance depends on the amplitude. For instance, if the amplitude is set to 4, the total variance would be 16, and each part would have a variance of 8. If the amplitude is $\sqrt{2}$, the total variance is 2, with each part contributing a variance of 1.

Understanding this behavior is crucial when working with real-valued constellations, such as BPSK, PAM-4, or PAM-16. It's essential to use real noise sources, as the complex noise won't contribute to the calculations, which could lead to slight errors in your bit error rate (BER) or symbol error rate (SER) computations.

To compute bit errors, we need to evaluate XORs. Before proceeding, let's take a brief detour to understand how XORs are implemented in Python so we can build a robust Python block for our calculations.

Start by importing NumPy. Then, create a small array of integers that represent some of the message values in your MPAM system. For example, in PAM-4, our messages could be one of 0, 1, 2, or 3. Let's assume our transmitted messages are 1, 0, 1, 2, 1, and 3, which correspond to the bit sequence 01 (since 1 can be represented in binary as 01), 00, 01, 10, 01, and 11.

Now, let's introduce two bit errors into the sequence. Suppose the first "1" flips to "0", changing the sequence to "00". Additionally, assume both bits of the "3" flip, also resulting in "00". These changes will allow us to observe the impact of bit errors on the sequence.

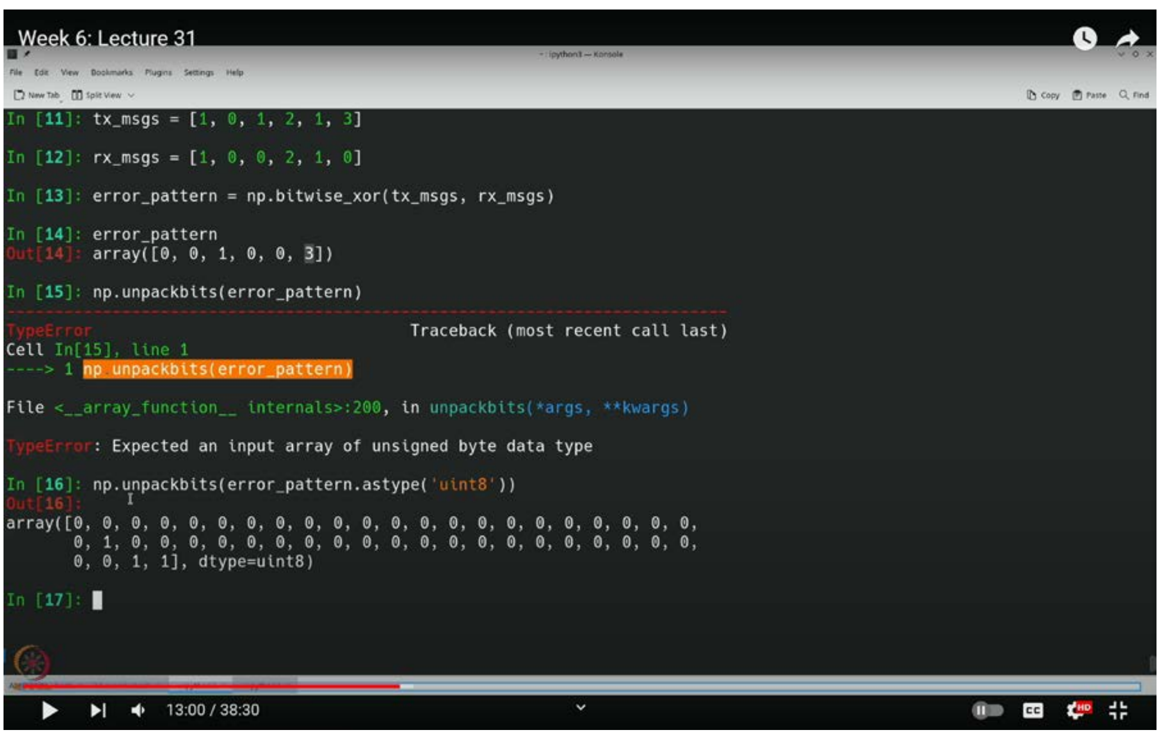
Let's examine the results. As you can see, the bit sequence 0, 1 was altered to 0, 0, while the bit sequence 1, 1 was also changed to 0, 0. This indicates that within our 12-bit sequence, we have introduced a total of 3 bit errors.

To determine the exact number of bit errors, we will perform a bitwise XOR operation on these sequences. NumPy provides a function for this bitwise XOR operation. By inspecting the error patterns, we can see that there are no bit errors in some places. For clarity, let’s clean up the visualization:

- In the first case, there is no bit error detected.
- In the second case, no bit error is observed.
- In the third case, there is exactly 1 bit error, where 0, 1 changed to 0, 0.
- In the fourth case, no bit errors are present.
- In the fifth case, there are 2 bit errors, which is evident because the sequence 1, 1 changed to 0, 0.

So, we end up with a total of 3 bit errors, represented by the pattern 1, 1. To calculate the total number of bit errors, note that it's not simply the sum of individual errors, such as 1 + 3 = 4. Instead, this total includes 1 bit error from one sequence and 2 bit errors from another.

(Refer Slide Time: 13:00)

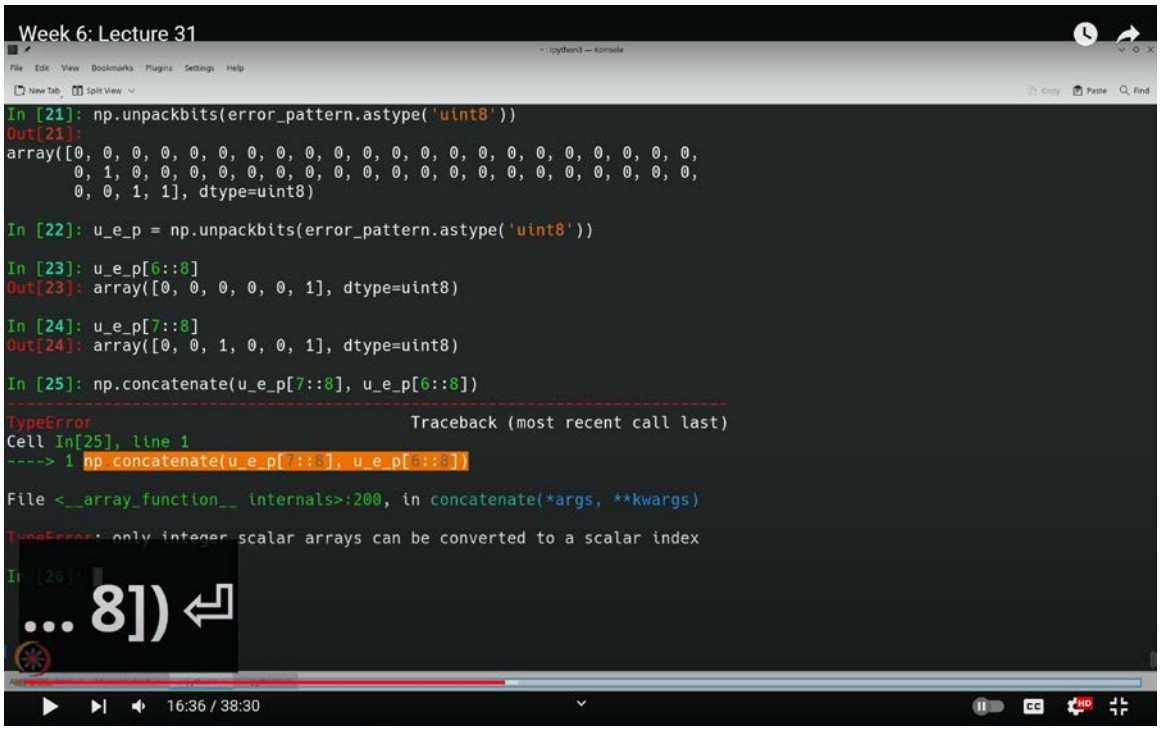


To accurately count the bit errors, we use NumPy's **`unpackbits`** function. This function requires unsigned byte data types, so we will first convert our array to this type. When you run the code, each original 1, 2, 3, 4, 5, and 6-bit sequence will be converted into 8-bit patterns, resulting in a bit array of length 48.

These 48 bits represent the bitwise representations of the original bytes. For instance, the byte 01 and 03 will be converted to their bitwise forms. By summing these bits, you will count the total number of bit changes. This approach provides a reliable method to compute the number of bit errors that we will use in GNU Radio as well.

Now, we want to output only the bit errors corresponding to the symbols that generated those errors. In other words, if you look at a specific error pattern, it originated from 6 PAM4 or QPSK symbols, which correspond to 12 bits in total. Our goal is to output exactly 12 zeros or ones to indicate whether each of those 12 bits is in error.

(Refer Slide Time: 16:36)



To achieve this, we need to convert our data into 12-bit patterns. Although our data consists of 48 elements, we only need the last 2 bits from each of the 8 elements. To obtain this, we

will extract the 7th and 8th bytes from each of these elements and then concatenate them together. While the order of these precise bit error locations may not be correct, that's acceptable for our purpose since we're interested in histogramming the errors, and the order isn't significant.

Let's call this process "UEP unpacked error patterns." We'll extract the 7th bit and every 8th bit after that using the slicing notation **`6::8`**. This notation gives us the required error pattern. For example, if a bit sequence got flipped, the resulting pattern will reflect this change.

Specifically, using **`7::8`** provides every 8th bit starting from the 7th bit. This is well-documented in Python's slicing notation. We will use the **`np.concatenate`** function to combine these bits into a single array. This concatenated array should have 12 elements, corresponding to the 12 bits we are analyzing.

We will rewrite this code in our Python block to compute bit errors.

Next, let's rebuild our PAM or QAM-based communication system, but this time, we will focus on bit errors rather than symbol errors. We'll start by creating a QAM-4 bit error rate calculator.

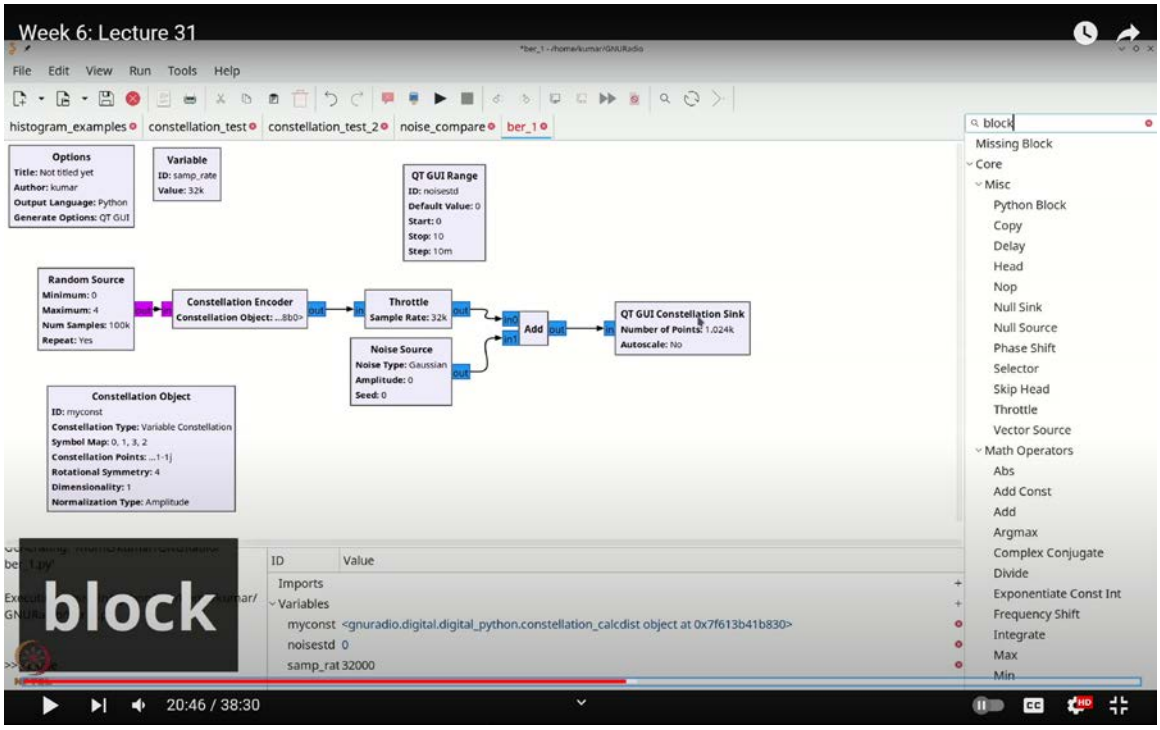
First, we need to set up a random source. Search for "random source" using Ctrl+F or Cmd+F, and configure it to output values from 0 through 4, converting these to bytes.

Next, we'll add a constellation encoder by searching for "ENCOD" and placing the encoder in the flow graph. This encoder requires a constellation object, which we'll name **`MY_CONST`**. To create this constellation object, search for "CONST" and select the appropriate constellation object. By default, the constellation object is set to a QPSK-like constellation, which will suffice for our purposes.

Let's proceed with the setup. We'll name our constellation object **`my_const`**. Next, we need to add a throttle to control the flow of data. Search for "throttle" using Ctrl+F or Cmd+F and add it to the flow graph.

We will also add noise to our system. First, add a "range" block by searching for "range." Name this block **`noise_STD`**. Configure it to start at 0, end at 10, and use a step size of 0.01. Then, add a "noise source" by searching for "noise source" and set its amplitude to **`noise_STD`**.

(Refer Slide Time: 20:46)



Now, connect these components together. To visualize the results, add a "constellation sink" by searching for "constellation sink." Increase the number of samples in the sink to allow for averaging, which will help us observe the constellations. As you increase the noise level, you'll notice that the constellations become wider and more diffuse due to the noise.

Our next objective is to compute the bit error rates. For this, note that each QPSK symbol corresponds to two bits. Therefore, we need a block that outputs twice the number of samples as it receives.

To address this, add a Python block by searching for "block" and selecting the Python block. Double-click it to open the editor. Unlike previous cases where a symbol error rate

might involve a simple yes/no (0 or 1) per symbol, here we deal with QPSK symbols. Each QPSK symbol can produce two possible bit error patterns, so we cannot use a simple sink block.

Instead, we will use an "interp" block, which ensures proper timing and converts each input symbol or symbol pair into two output symbols, which are zeros and ones. This will allow us to histogram the bit errors. Remove unnecessary parameters and replace the sink block with the interp block. Name it **`QPSKPERCOUNTER`**. Set the inputs and outputs to **`int8`** to handle pairs of 0s and 1s.

Let's refine our approach to generalize the bit error rate (BER) computation. We don't need to use the example parameter; instead, we can use the decimation rate as a parameter to make the system adaptable to various quadrature amplitude modulation (QAM) formats. We'll name this parameter **`decim_rate`** and set it to 2, initially tailored for QPSK. However, this approach will be flexible enough to extend to other modulation schemes. We'll refer to this as **`general_br_counter`**, making it applicable to formats like PAM4 and other symbol sets.

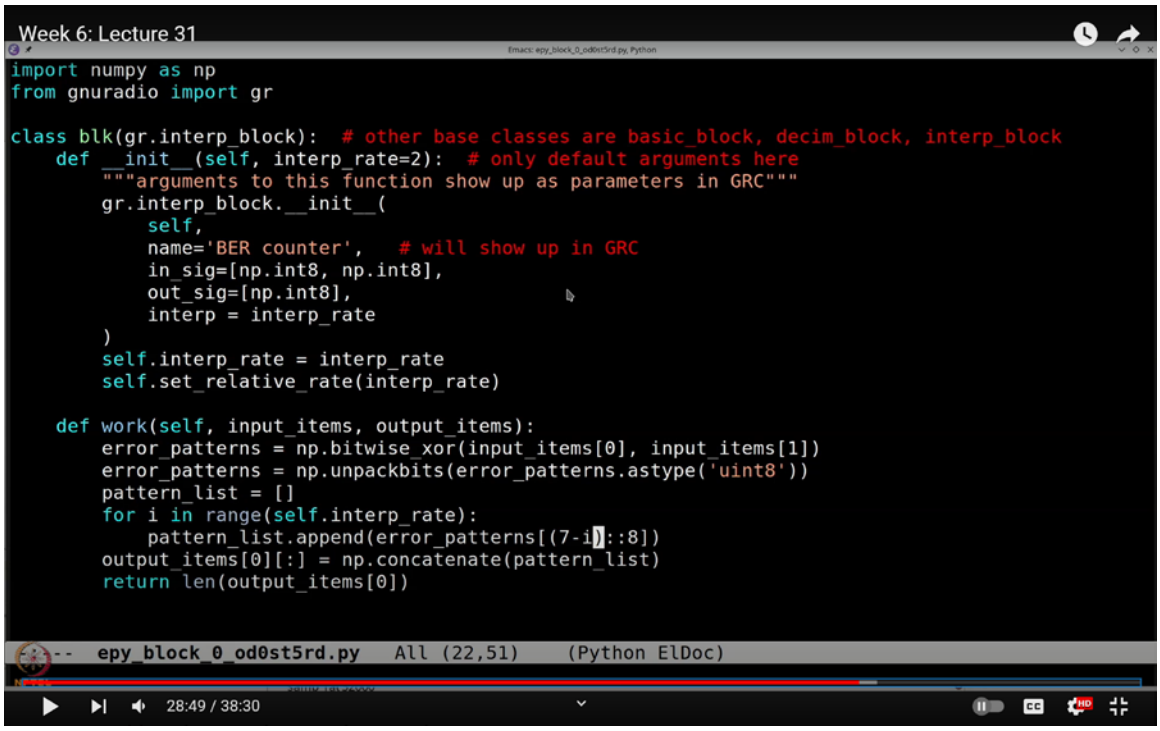
Next, we should adjust the parameter from **`decim_rate`** to **`interp_rate`**, and update our code accordingly. We'll set **`interp`** to **`interp_rate`** and assign **`self.interp_rate`** to **`interp_rate`**. We will also configure **`self.set_relative_rate`** to **`interp_rate`**. This setting helps GNU Radio determine how many samples will be output based on the interpolation rate.

Now, let's outline the sequence of commands for computing XOR, performing unpacking, and obtaining the correct sequences. The goal is to process the error patterns and convert them into unpacked form. For instance, if the error pattern is 3, it represents 2 errors (binary 11). We need to convert these errors to **`uint8`** format and extract the 7th and 8th bits, concatenate them, and include them in the output sequence.

Here's what the code does: It calculates the bitwise XOR between input items 1 and 2, which marks the number of bit errors. The **`br_counter`** will handle various formats such

as QPSK and QAM-16. For QAM-16, which involves 16 patterns corresponding to 4 bits per symbol, the interpolation rate is 4.

(Refer Slide Time: 28:49)



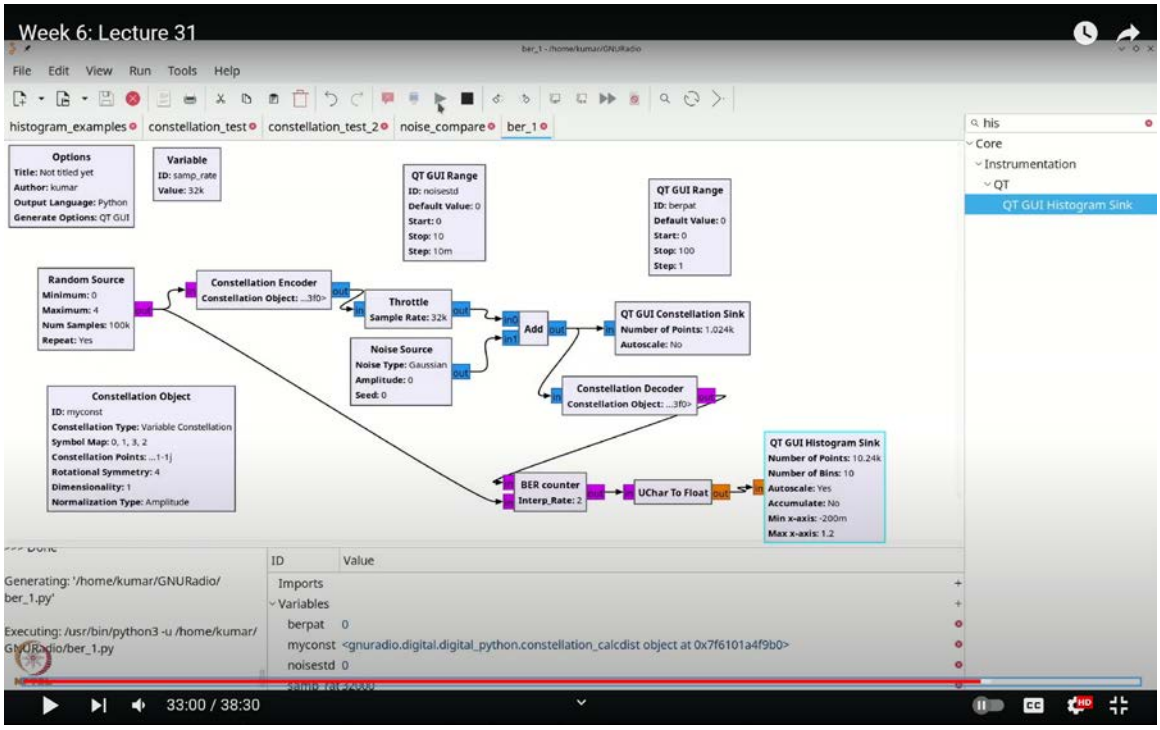
Depending on the interpolation rate (corresponding to the constellation size, e.g., QAM-16), we compute errors starting from the 8th bit (least significant bit) and concatenate these errors. Although it's possible to loop through all 8 bits, processing each bit individually is more efficient. For instance, if the interpolation rate is 2, the sequence will be processed as 7 minus 0 for the 8th bit, 7 minus 1 for the 7th bit, and so on. By concatenating these error patterns, we effectively calculate the bit errors.

This method provides a robust way to calculate bit errors efficiently and is adaptable to various modulation schemes.

Let's evaluate the bit error rate (BER) counter we've implemented in GNU Radio. Although there are potentially more efficient methods for this task, our goal now is to verify if our custom BER counter operates correctly.

First, save and exit the current setup. We should now see a new block named **`ber-counter`** with an interpolation rate of 2. To test this block, we will input fixed values and check if it performs as expected.

(Refer Slide Time: 33:00)



We will start by adding a couple of constant sources. Use **`Ctrl+F`** (or **`Cmd+F`** on Mac) and search for "constant source." Set one constant source to output a byte with a value of 0. Add another constant source, but this time, set it as a variable constant and name it **`ber-pat`**. Connect these sources to the **`ber-counter`** block.

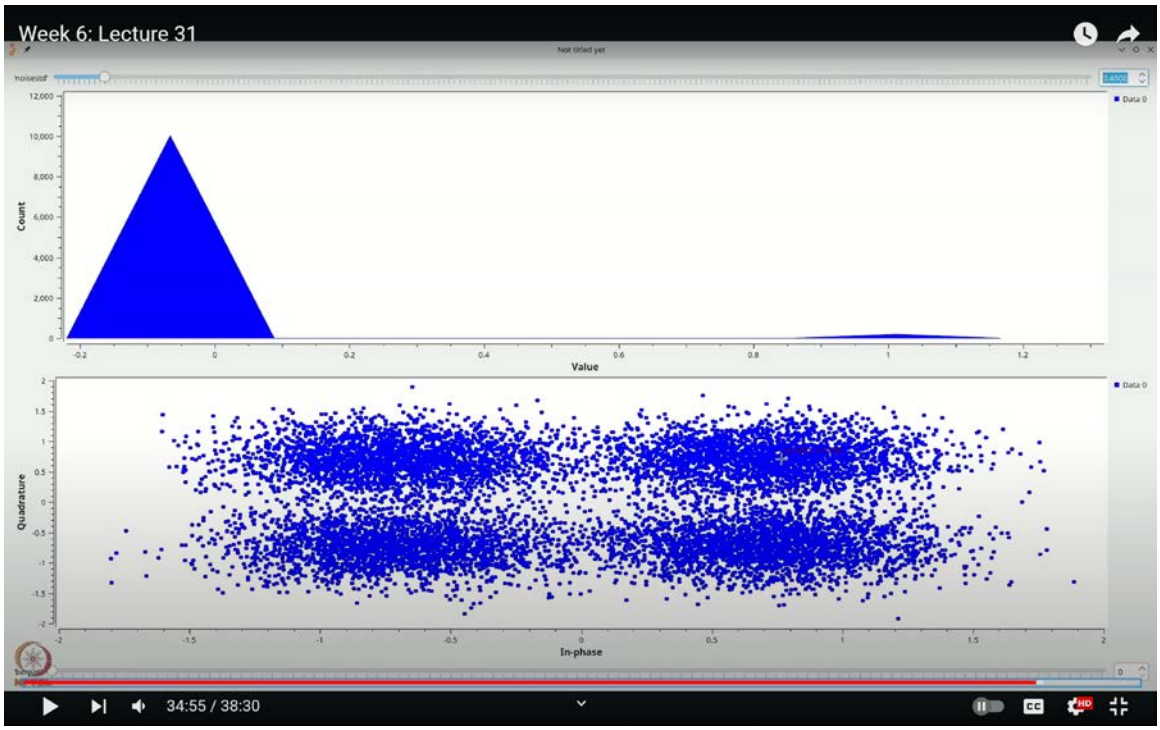
Next, create a range block by using **`Ctrl+F`** (or **`Cmd+F`**) and search for "range." Configure this block to be named **`ber-pat`** with a default value of 0, starting at 0 and stopping at 100. This integer can be converted to a byte, so no further adjustment is needed here.

To visualize the results, add a time sink by searching for "time sink" and place it in the flow graph. Set the time sink to float. Additionally, include a "UChar to Float" block to

convert unsigned characters to float values. Connect this block accordingly to ensure proper visualization.

Run the flow graph. You will observe that XORing 0 with 0 yields 0, while XORing 0 with 1 produces 1. Similarly, XORing 1 with 0 results in 1, and XORing 1 with 1 results in 0, indicating a flipped bit.

(Refer Slide Time: 34:55)



In summary, if you have 0 and 0, there are no errors. XORing 0 and 1 (or 1 and 0) results in errors approximately half the time, as reflected in the visual pattern. When XORing 1 and 1, you encounter 2 errors. This confirms that our BER counter is functioning correctly, as it accurately identifies bit errors in the tested scenarios.

Let's remove the constant sources from our setup and keep only the **`UChar to Float`** block. We'll now add a histogram block to analyze the data.

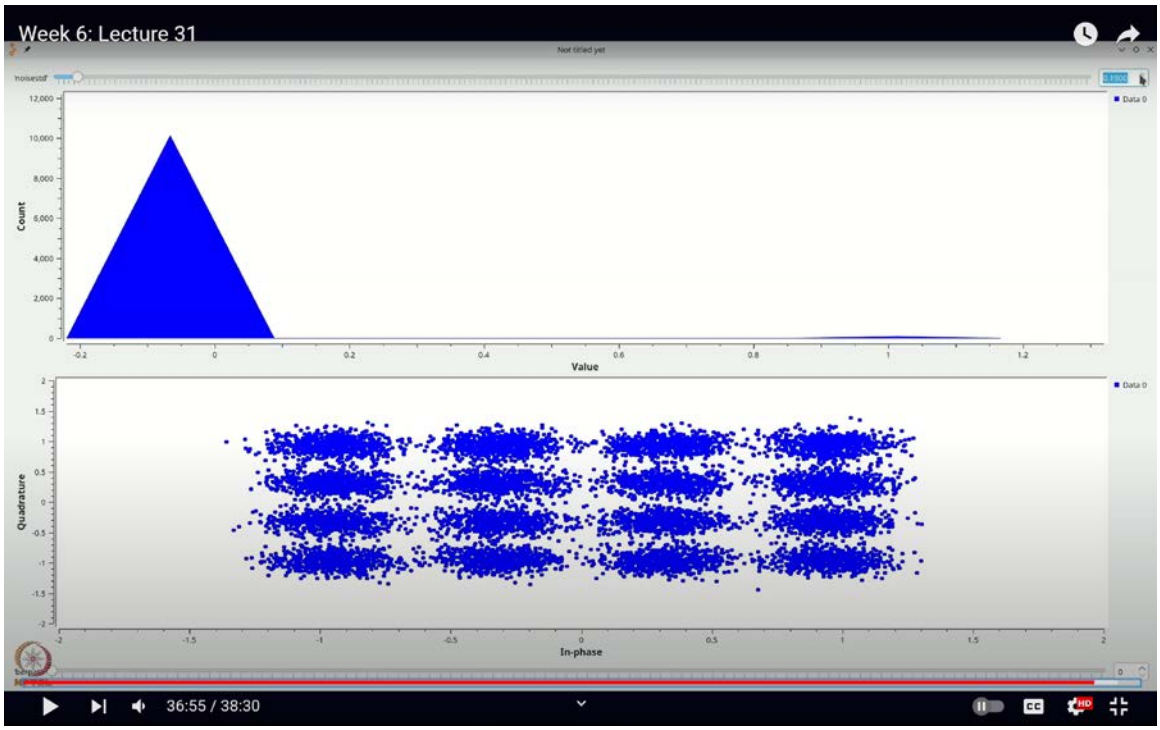
The next step is to recover the constellation by incorporating a decoder and then measuring the bit errors. To do this, add a constellation decoder by using **`Ctrl+F`** (or **`Cmd+F`**) to

search for "decoder." Select the constellation decoder and configure it to use **`mycons`** as the constellation. Connect this decoder to the appropriate parts of the flow graph, linking it to the original source.

Now, add a histogram block by using **`Ctrl+F`** (or **`Cmd+F`**) to search for "histogram." Place the histogram sink in the flow graph and connect it accordingly. Set the number of bins to 10, as we are primarily interested in counting the values 0 and 1. Configure the histogram range from -0.2 to 1.2, focusing on counting zeros and ones, where zeros represent no errors and ones represent errors.

Execute the flow graph. Initially, you should see a histogram peak at 0, indicating no errors. As you increase the noise, you will observe a rise in the error count. If the noise is increased significantly, the histogram will show both values (0 and 1) at approximately the same height, suggesting a bit error rate of 50%. This is interesting because a 50% error rate implies that the signal is indistinguishable from random noise—essentially a coin toss.

(Refer Slide Time: 36:55)



To refine the analysis, increase the noise slightly and observe the behavior of the QPSK constellation. The constellation points will spread out, forming a larger blob as the noise level increases. To get a clearer picture, you can adjust the constellation to include more points. As you continue to increase the noise, observe how the constellation grows, especially when the noise variance or standard deviation approaches values like 0.2 or 0.3.

As you start to increase the noise level, you'll observe some symbols crossing over, which leads to bit errors. This is consistent with what we've discussed in the lectures: increasing noise results in more bit errors.

In this setup, we're using an inbuilt constellation where E_b (bit energy) is $\frac{E_s}{2}$, since E_s (symbol energy) is set to 1. Therefore, E_b is half of E_s . Also, N_0 (noise power spectral density) is set to 1 because we've chosen the noise to have unit variance. By performing the necessary calculations, you can easily determine the bit error rate.

As expected, increasing the noise level causes more bit errors, while reducing the noise decreases the bit errors. The value of 1 in your results represents the number of bit errors. By calculating the ratio of bit errors to the total number of symbols transmitted, you can estimate the bit error rate of the system. This should align well with the bit error rate formula used in class. For instance, setting the noise level close to 0.4 or 0.5 will show an increase in bit errors, allowing you to verify the consistency of the bit error rate formula.

For a final check, let's change the constellation to QPSK and observe if it impacts the results. You should find that the effects are similar: as the noise increases, so do the number of errors.

Before we conclude, here's a crucial point to consider: if you switch to a different modulation scheme like 16-QAM, our bit error rate counter will still function correctly. However, you'll need to adjust the parameters accordingly. Specifically, update the random source to range from 0 to 16 and configure the BER counter to handle values from 0 to 4. This adjustment is necessary because 16-QAM involves 4 bits per symbol.

Running the simulation now reveals a normalized constellation. Even a slight increase in noise will start to introduce bit errors. For the same noise level that we used in the QAM-4 case, you'll notice that the impact of noise is significantly higher in the QAM-16 case. This demonstrates that QAM-16 is indeed much less resistant to bit errors compared to QAM-4, which aligns with our expectations.

You can perform a similar analysis for PAM-8, but remember to adjust the parameters accordingly: set the modulation order to 8 and update the bit error rate counter to handle values from 0 to 3.

In this lecture, we've learned how to build a simple bit error rate (BER) testing mechanism in GNU Radio. With a few modifications, we've managed to effectively characterize bit errors and visualize the bit error rate by comparing the fraction of correctly decoded symbols to incorrectly decoded ones using a histogram sink. In the next lecture, we will integrate these concepts to simulate a practical communication system. Thank you.