Digital Communication using GNU Radio

Prof. Kumar Appaiah

Department of Electrical Engineering

Indian Institute of Technology Bombay

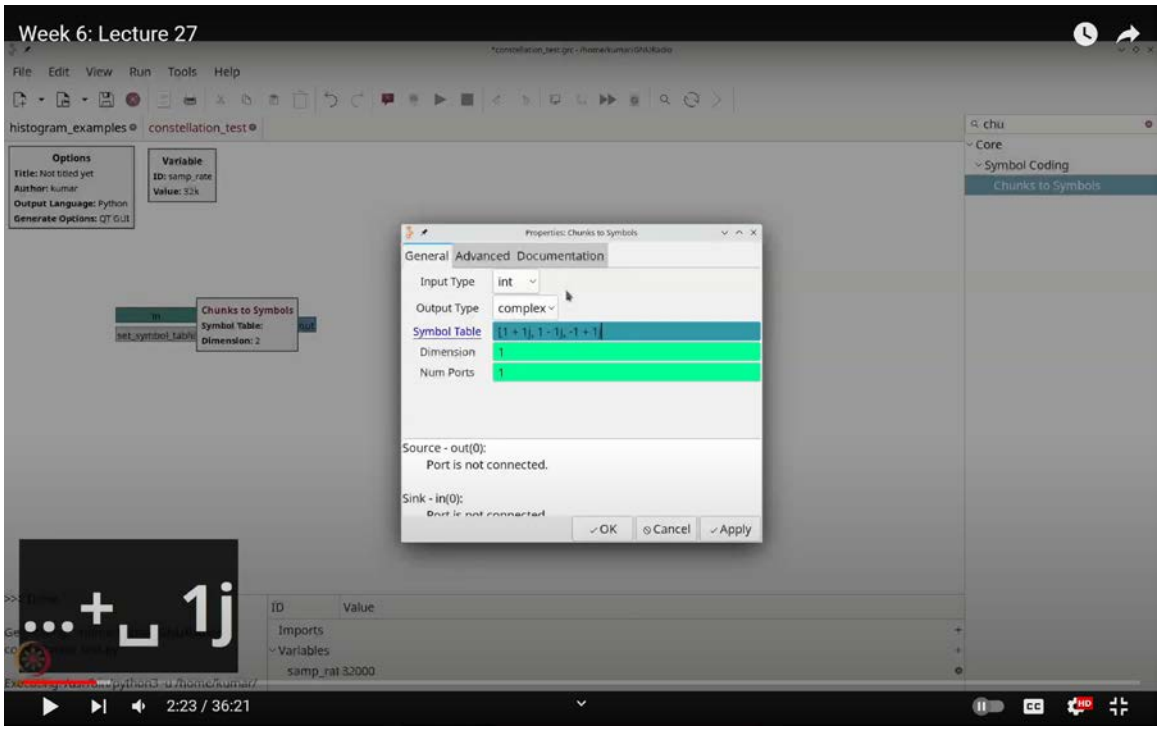
Week-06

Lecture-27

Visualising Symbol Error Rate in GNU Radio

Welcome to this lecture on Digital Communication using GNU Radio. I'm Kumar Appiah. In this session, we will build on the concepts from our previous lectures and explore how to utilize GNU Radio's constellation objects for decoding received values. Specifically, we'll address how to determine the minimum distance point in the presence of additive Gaussian noise and convert that to symbols. Additionally, we will examine the symbol error rate under noisy conditions. Integrating these concepts will enable us to create robust simulations that validate both theoretical and practical knowledge.

(Refer Slide Time: 02:23)



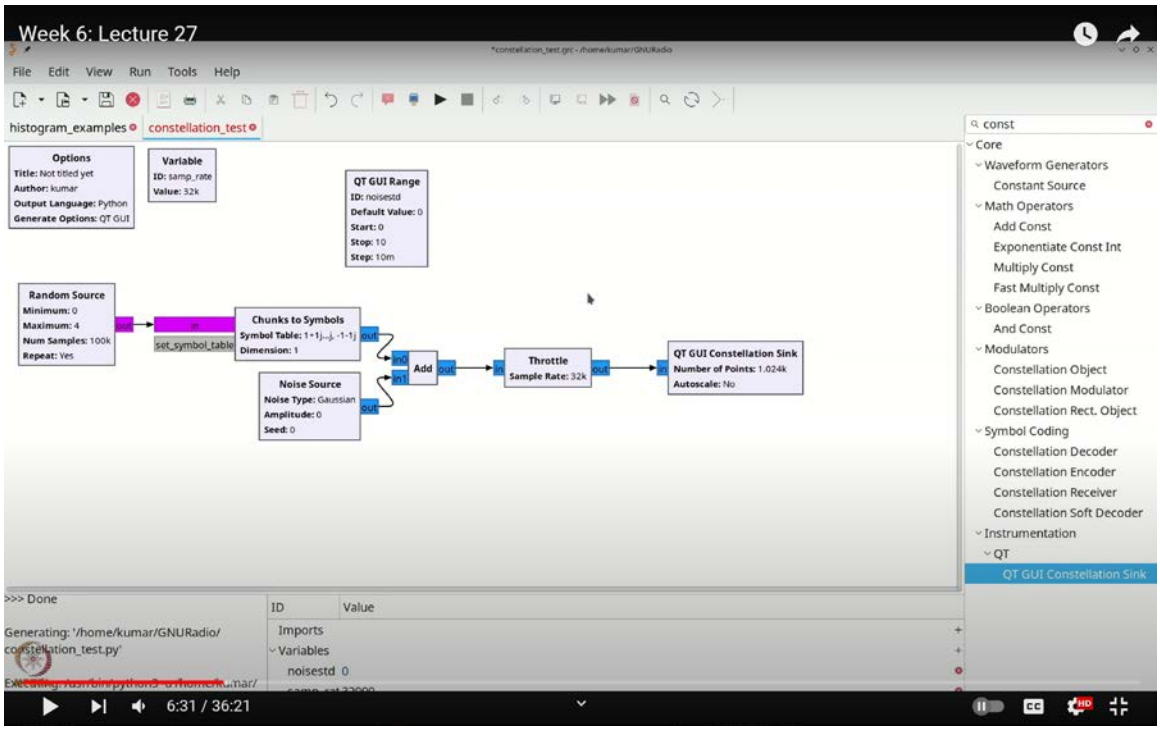
Let’s start by constructing and generating constellations from symbols in GNU Radio.

First, we’ll examine the Chunks to Symbols block. To find it, press Ctrl+F (or Command+F) and type "chunk". Drag the Chunks to Symbols block into your workspace. This block functions as a table lookup. You need to provide a symbol table, and the Chunks to Symbols block will map integer inputs (such as 0, 1, 2, etc.) to the corresponding symbol from the symbol table.

For this demonstration, we will use a commonly employed constellation, such as QPSK or QAM-4. We will define this constellation by specifying its elements in a list. Enter the following symbols into the list: **`[1 + 1j, 1 - 1j, -1 + 1j, -1 - 1j]`**. Each integer input to the block will correspond to one of these symbols. For instance, an input of 0 will yield the symbol **`1 + 1j`**.

With this setup, we can effectively map chunks of data to their respective symbols and proceed with our analysis of symbol error rates in the presence of noise.

(Refer Slide Time: 06:31)



If the input value is 1, the corresponding output will be the symbol associated with 1. If the input is 2, the output will be the symbol associated with 2, and similarly for 3. For convenience in future use, I will change the data type to a byte. This modification is practical since a byte can represent the values 0, 1, 2, or 3 just like the previous setup.

To generate these symbols randomly, we need to introduce a Random Source. Press Ctrl+F (or Command+F) and type "random" to find the random source block. Place it at the beginning of your flow graph and configure it to output a byte, which will be one of 0, 1, 2, or 3.

As noted in the previous lecture, a common issue with the random source is that it outputs the same sequence of random values repeatedly. To address this, we will increase the size of the random source to 100,000. This adjustment ensures that the sequence of random values is sufficiently varied and not limited by repetitive patterns.

Once you've set this up, connect the random source to the input of the Chunks to Symbols block. Before visualizing the constellation, let's add the capability to include noise. Press Ctrl+F (or Command+F) and type "noise" to find the noise source block. Add this block to your flow graph and configure it to allow control over the noise's standard deviation or variance. Change the amplitude parameter to "noise std" and click OK.

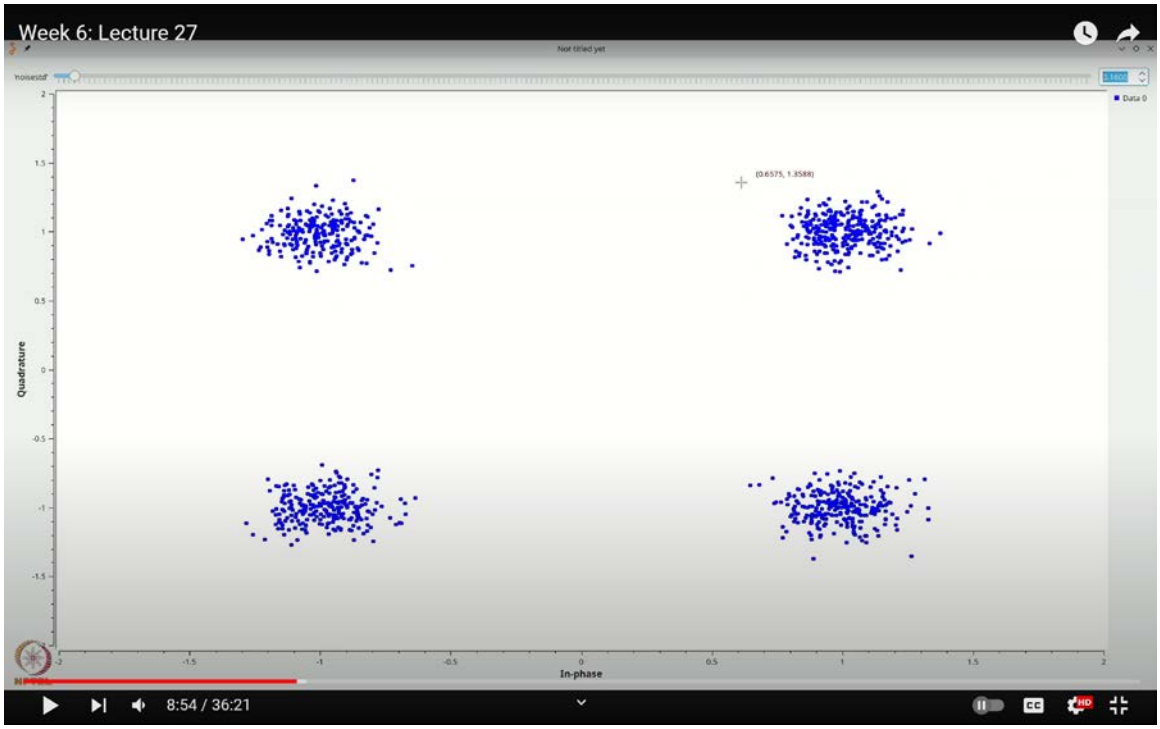
To control the noise level dynamically, add a QTGUI Range block. Press Ctrl+F (or Command+F) and type "range" to locate it. Place this range block in your workspace, double-click to open its properties, and set its ID to "noise std." This will turn it into a control knob for adjusting the amount of noise. Set the default value to 0, indicating no noise, with the standard deviation ranging up to 10 and the step size set to 0.01.

Now, you have a way to adjust the noise level. Since the default value of "noise std" is 0, the amplitude will show 0, which is expected. Next, we need to combine the symbols with the noise. Press Ctrl+F (or Command+F) and type "add" to find the Add block. Place it in the flow graph and connect the output of the Chunks to Symbols block and the Noise Source block to this Add block.

We will now connect the Chunks to Symbols block with the Noise Source block. To manage the flow and prevent overloading, we need to add a Throttle block. Press Ctrl+F (or Command+F) and type "throttle" to locate it. Add this throttle block to your flow graph, and then connect it to a Constellation Sink.

Visualizing this using a Time Sink would only display a series of 1s and -1s, as the signals represent values like 1 + 1j, 1 - 1j, -1 + 1j, and -1 - 1j. In this setup, the real part of the signal is either +1 or -1, and the imaginary part is also either +1 or -1. Therefore, to get a more meaningful visualization, we should use a Constellation Sink. Press Ctrl+F (or Command+F) and type "constellation" to find the QTGUI Constellation Sink. Add it to the flow graph and connect it appropriately.

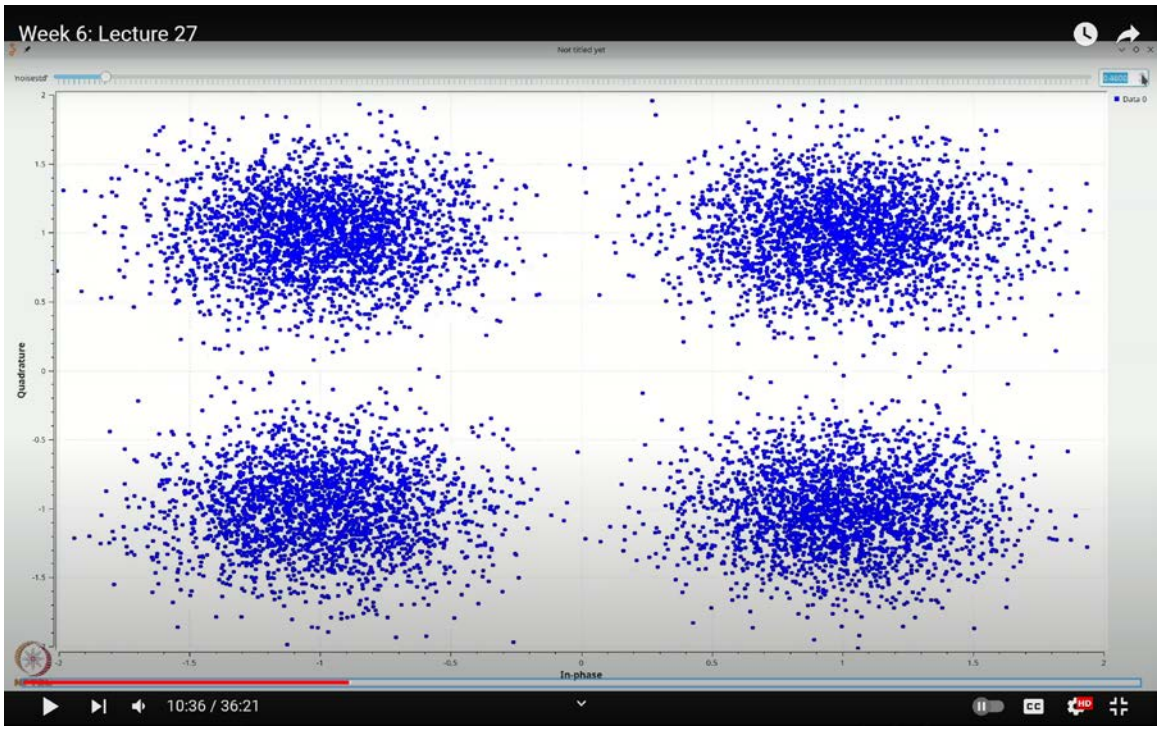
(Refer Slide Time: 08:54)



Now, we are ready to visualize the constellation generated by this flow graph. So, what does a constellation mean in this context? Let’s explore it. You should see four distinct points, which are located at 1 + 1j, 1 - 1j, -1 - 1j, and -1 + 1j. Essentially, this visualization acts like a 2D histogram, but without the counting of occurrences. The Constellation Sink

is showing you all the values it receives from the input. Your random source generates values 0, 1, 2, and 3 randomly, each selecting one of these constellation points.

(Refer Slide Time: 10:36)



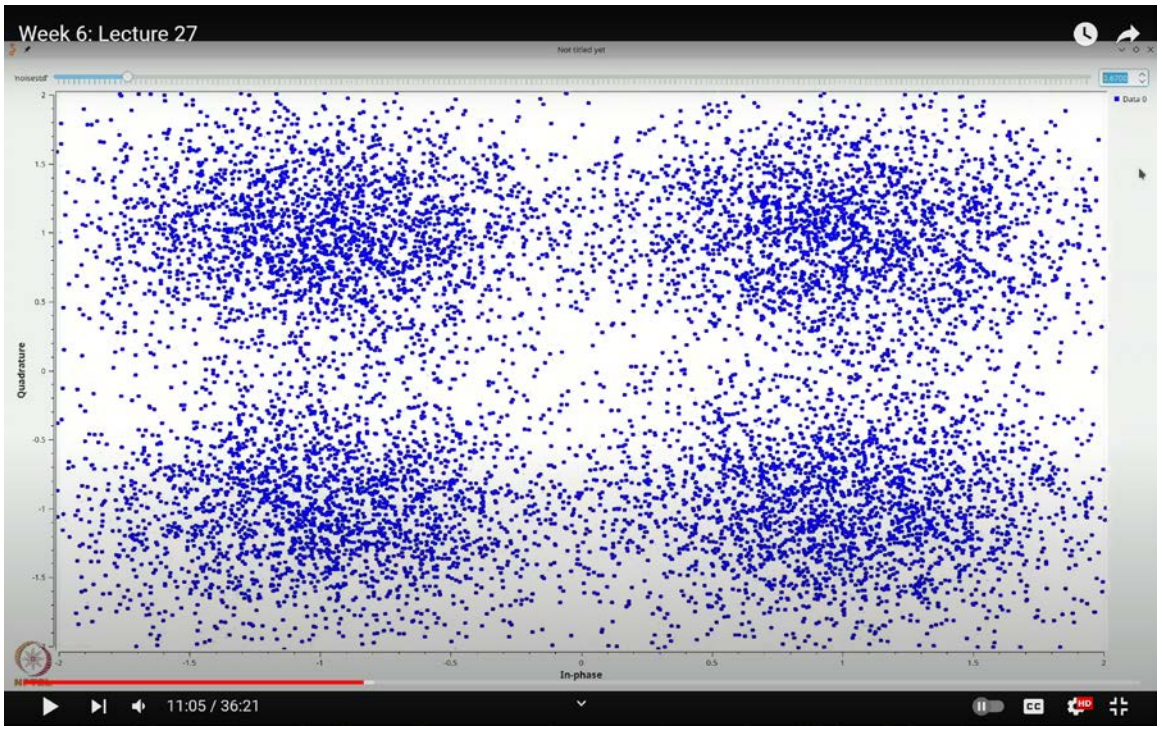
Let’s now add a bit of noise to see how it affects the constellation. If we set the noise standard deviation to 0.02, you will notice a slight movement in the constellation points. What’s happening here? You are adding Gaussian noise to each of the points, such as 1 + 1j. To understand this, imagine that without noise, you would see a single point, like an impulse, precisely at 1 + 1j. If we were using a 2D histogram, each point would appear as a stick of uniform height. However, the introduction of noise slightly shifts these points, creating a visual dispersion around their original locations.

Now, instead of a single stick, what you see is a small 2D Gaussian distribution centered around your 1 + 1j point, which spreads out slightly due to the added noise.

Why is it only slight? This is because the noise standard deviation is set to 0.02, representing a very small amount of noise, or in communication terms, a very high Signal-to-Noise Ratio (SNR) scenario. As we increase the amount of noise, the spread of the points

becomes more pronounced. For instance, if you add noise to 1 + 1j, the resulting Gaussian noise begins to scatter the points over a broader area.

(Refer Slide Time: 11:05)



To enhance visualization, let's make a slight adjustment. Double-click the QTGUI Constellation Sink, and you'll see that the number of points is set to 1024. We can increase this number to 1024 to add more density to our constellation. Additionally, you can set the grid and choose not to auto-scale the axes, as the x and y axes are correctly aligned.

Execute the flow graph again. With increasing noise, for example, to 0.1, you'll notice a denser constellation. This is indeed a Gaussian distribution: points are concentrated around the center, but as you move further from the center, they become sparser. In a high SNR scenario, if you want to compute it, a point like 1 + 1j corresponds to an amplitude of $\sqrt{2}$, with a noise standard deviation (σ) of 0.1. You can calculate the SNR ratio and convert it to decibels, which might be around 14 dB in linear scale.

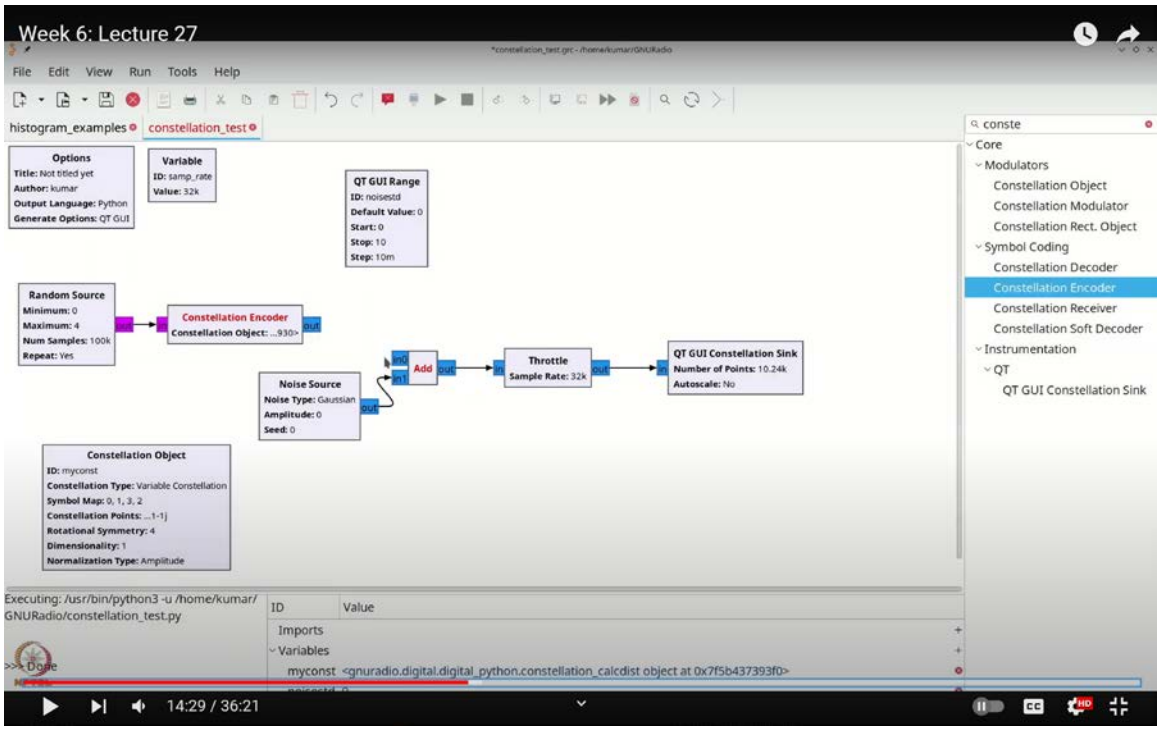
However, as you continue to increase the noise, the situation becomes more problematic. With higher noise levels, the likelihood that your 1 + 1j point gets distorted by the additive

noise and is mistakenly detected as, say, -1 + 1j, 1 - 1j, or even -1 - 1j increases significantly. If the noise becomes excessively large, the situation can become chaotic, leading to a substantial number of detection errors.

It's very clear that as we increase the noise power or decrease the signal-to-noise ratio (SNR), the number of symbol errors we encounter will rise. However, it’s crucial to have a method to quantify or measure these errors. We’ll address how to do this shortly.

To accurately determine the symbol error rate or identify whether a symbol error has occurred, we need to implement minimum distance decoding. As discussed in previous lectures, minimum distance decoding is optimal for equiprobable symbols under Gaussian noise conditions.

(Refer Slide Time: 14:29)



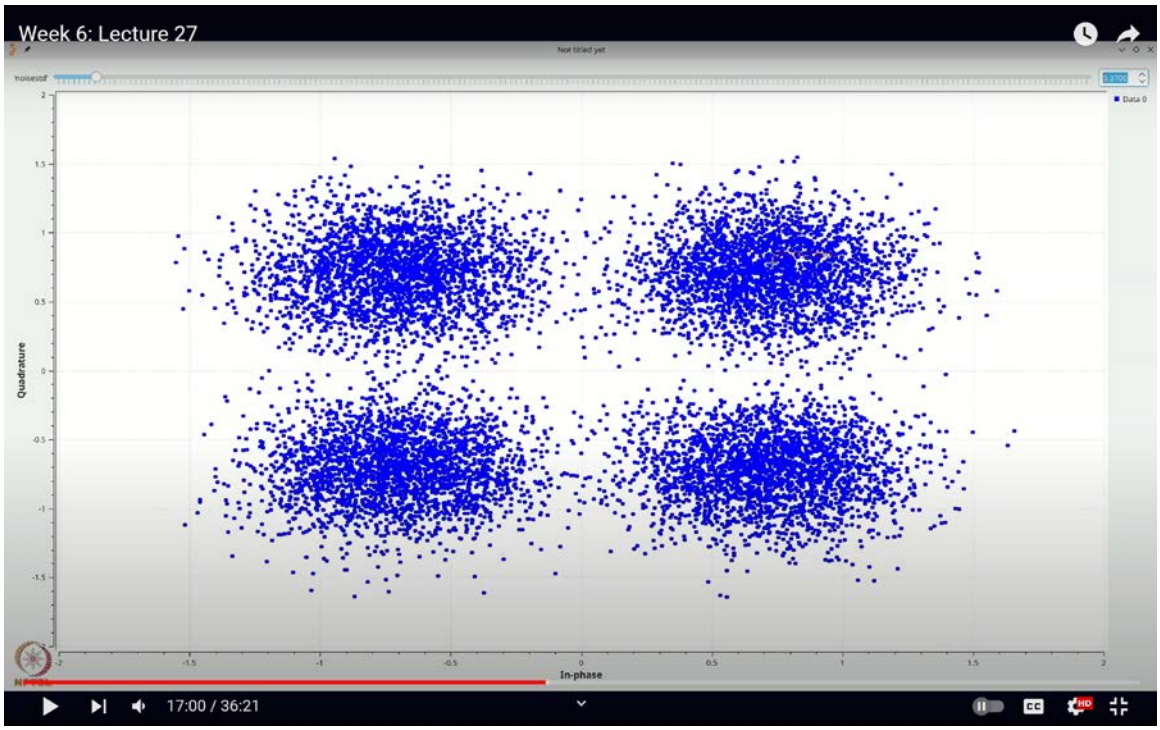
The Chunks to Symbols block specifies the symbols, but to decode these symbols at the receiver and compare them with the transmitted symbols to check for errors, we need a minimum distance decoder. This can be achieved through computations in GNU Radio. Since this process can be cumbersome and is a common task, GNU Radio offers a more

sophisticated and user-friendly built-in constellation approach. This method simplifies minimum distance decoding and other related operations.

Let’s now explore this approach. We will look at three blocks, but first, let's start with the initial two. Begin by removing the Chunks to Symbols block. Then, press Ctrl+F (or Command+F) to find and add the Constellation Object block to the flow graph.

The Constellation Object block is unique, much like the Variable and Range blocks, it doesn’t have any inputs or outputs. Instead, it serves as a specification that generates a variable of type constellation. Double-click the Constellation Object block to view its settings. The default type is Variable Constellation, and it includes options like BPSK, QPSK, DPQPSK, 8PSK, and 16QAM. For simplicity, we'll start with the Variable Constellation.

(Refer Slide Time: 17:00)

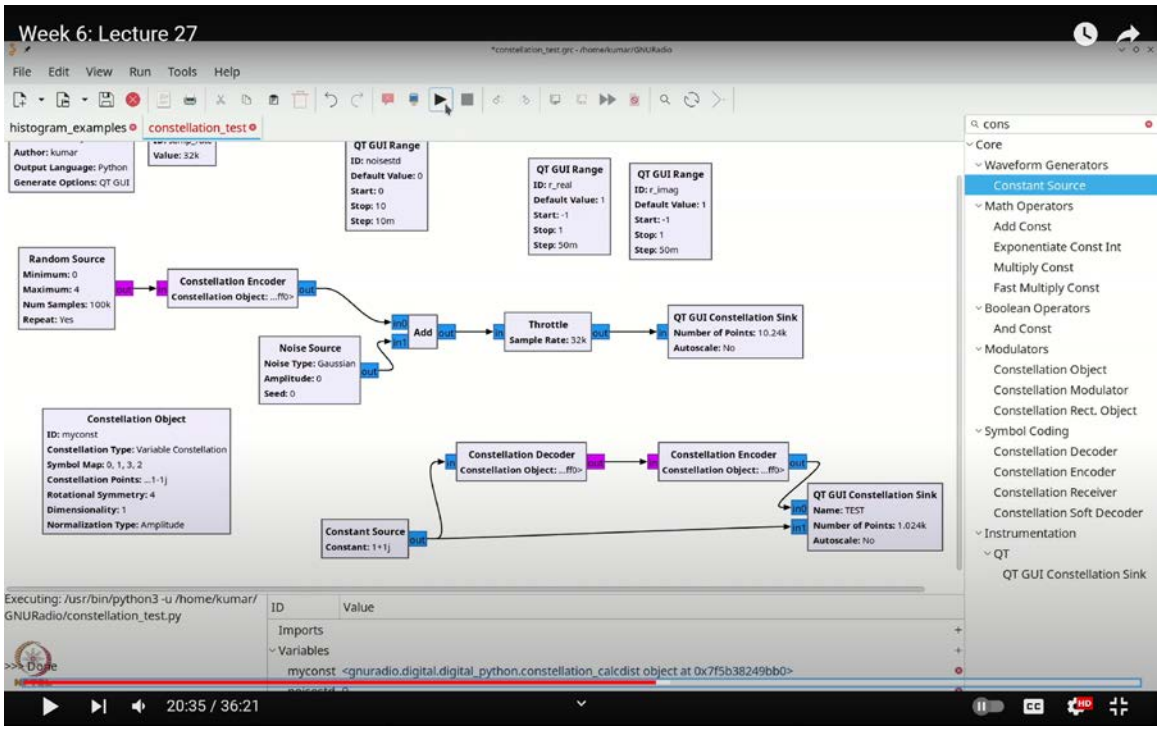


The symbol map is set to 0, 1, 3, 2. This specific mapping has a particular reason, as these values correspond to bits, which we will discuss later. For now, you can disregard the details. In essence, if the random source generates a 0, it will output the first symbol; if it

generates a 1, it will output the second symbol; if it generates a 2, it will output the third symbol; and if it generates a 3, it will output the fourth symbol.

At this point, we've covered some essential components, but there are additional aspects that we won't delve into right now. What you need to know is that this variable will generate a constellation. Let's proceed by confirming this setup with "OK."

(Refer Slide Time: 20:35)



Next, we'll add the Constellation Encoder block. Press Ctrl+F (or Command+F) and type **`SACONSTE`** to locate it. Drag the Constellation Encoder block onto the flow graph and connect it accordingly.

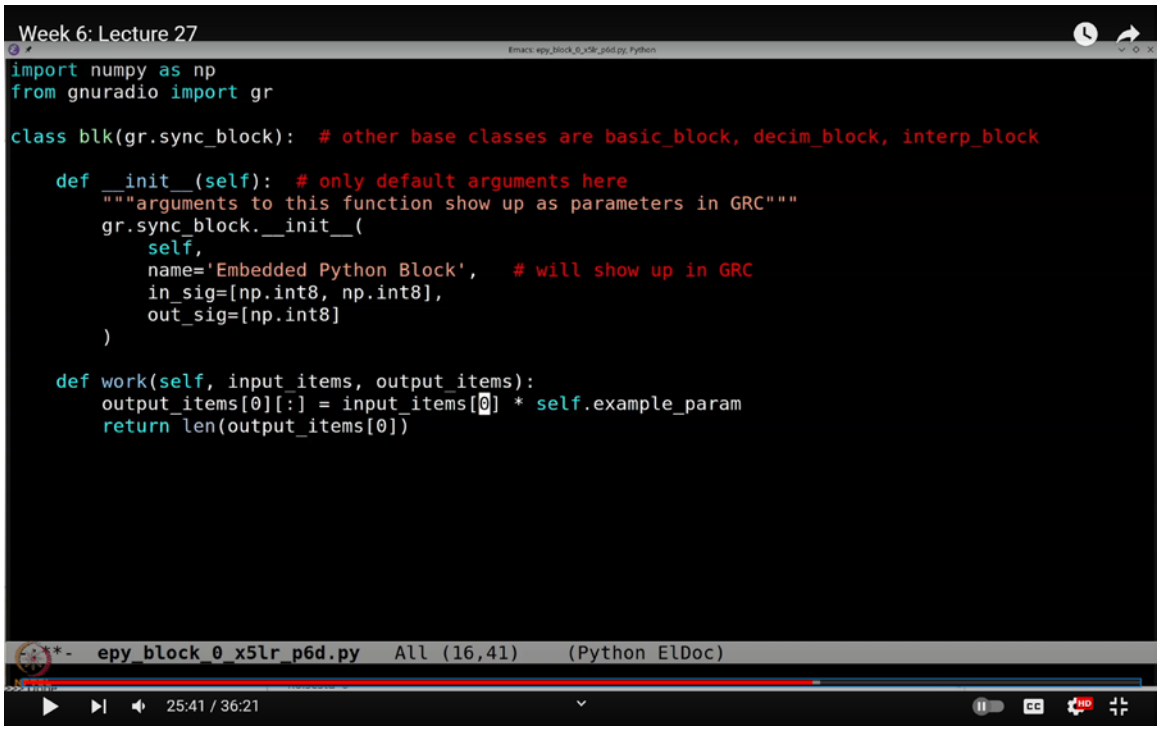
Double-click the Constellation Encoder block to open its settings. You'll see that it's currently set to use the "variable constellation 0" object. For better clarity and convenience, let's rename this to **`MYCONST`**. So, update the name to **`MYCONST`** and apply the changes.

Now, let’s connect this block in the flow graph and execute it. Before we proceed, observe that the constellation has shifted slightly to 0.7 + 0.7j. This change from 1 + 1j to 0.7 + 0.7j occurs because the Constellation Encoder block normalizes the constellation. Specifically, the value of 0.7 + 0.7j represents $1/\sqrt{2} + 1/\sqrt{2}j$.

Let’s explore why this normalization happens. Double-click the Constellation Object block and look for the Normalization Type setting. When set to “amplitude,” the constellation values are scaled so that the squared values, divided by the number of points, equal 1.

If you change the normalization type to “none,” you’ll see the values return to 1 + 1j. Conversely, setting it to “power” will also result in $1/\sqrt{2} + 1/\sqrt{2}j$. We’ll discuss the exact implications of these settings in detail later, but for now, we’ll keep it set to “amplitude.”

(Refer Slide Time: 25:41)



When you execute the flow graph, you'll notice that adding noise introduces a 2D Gaussian distribution around your constellation points. As the variance or standard deviation of this

2D Gaussian increases to a very large value, some of these points will start to drift into other decision regions. Remember, each decision region corresponds to a specific quadrant in this constellation setup. For instance, the first quadrant represents the decision region for 1 + 1j, the second quadrant for 1 - 1j, and so on. Any point that crosses these boundaries will lead to potential issues with symbol detection.

As mentioned earlier, the minimum distance decoder is optimal for this scenario. Its role is to determine the closest constellation point to the received data point. This is particularly effective under Gaussian noise with equiprobable symbols. Fortunately, GNU Radio provides an implementation of this minimum distance decoder.

To use it, we need the Constellation Decoder block. Press Ctrl+F (or Cmd+F) and type **`const`** to find the decoder block. Drag it onto the flow graph. This block requires you to specify which constellation it should use, so double-click on the decoder and set it to use **`MYCONST`**.

Before relying on this decoder for actual decoding, let’s verify that it performs minimum distance decoding as expected. We can test this by adjusting the position of a test point and checking if the decoder accurately identifies the closest constellation point. To facilitate this, we’ll create two ranges and construct a received value to evaluate the decoder’s performance.

Here’s how we can set it up: Add a Constellation Encoder block after the constellation decoder. Copy and paste it using Ctrl+C or Cmd+C, and Ctrl+V or Cmd+V. Then, add a Constellation Sink block to visualize the results. Label this setup as "test" to distinguish it from other configurations. Finally, construct a received symbol to test how well the constellation decoder performs.

To set this up, I need two real values that I will combine into a complex number. First, I'll use Ctrl-F to find the Range block and add it to the flow graph. Double-click on the block and name it **`R_real`** for the real part of the received value. Set its range to start at -1 and end at 1, with a step size of 0.05 and a default value of 1.

Next, I'll copy this block using Ctrl-C and paste it with Ctrl-V to create another range block. Rename this new block **`R_image`** for the imaginary part of the received value.

To view the results, I'll add two inputs. Press Ctrl-F and double-click to add a Constant Source. In this block, set the constant value to **`R_real + 1j * R_image`**. Although this block is called "constant," it will actually vary with the ranges we set, and you will see its effect in the constellation sink.

I'll also connect this constant source to the decoder. The decoder will process this constant source and reconstruct the original symbol, which you'll see as a red point on the constellation diagram. The decoded symbol will be displayed in blue.

Now, let's execute the flow graph and examine the results. Initially, we have a symbol of 1 + 1j, which gets decoded to 0.7 + 0.7j due to normalization. Let's add a grid for better visualization. As you move the received symbol, the minimum distance decoder should correctly identify the closest constellation point.

For example, if you move the received point to the left of 0, the minimum distance decoder should correctly decode it to $-1/\sqrt{2} + j/\sqrt{2}$. Moving it downward should decode it to $-1/\sqrt{2} - j/\sqrt{2}$. If you adjust the point to the right, set it to 0.3 to observe its effect.

The constellation decoder is functioning correctly, performing minimum distance decoding as expected. As you move the test point around, the decoder accurately identifies the closest constellation point. For example, when testing with 1 - 1j and 1 + 1j, it provides the correct results. Similarly, it accurately decodes 0.7 - 0.8, demonstrating its effectiveness in identifying the minimum distance point. Essentially, the constellation decoder does the work for you.

Now, let's simplify our setup by removing the test components and keeping only the constellation decoder. Our next task is to determine the decoded point from the noisy output and compare it with the original input from the random source.

To achieve this, we will first obtain the decoded output and then compare it with the original source using a custom Python block. If you recall from a previous lecture, we

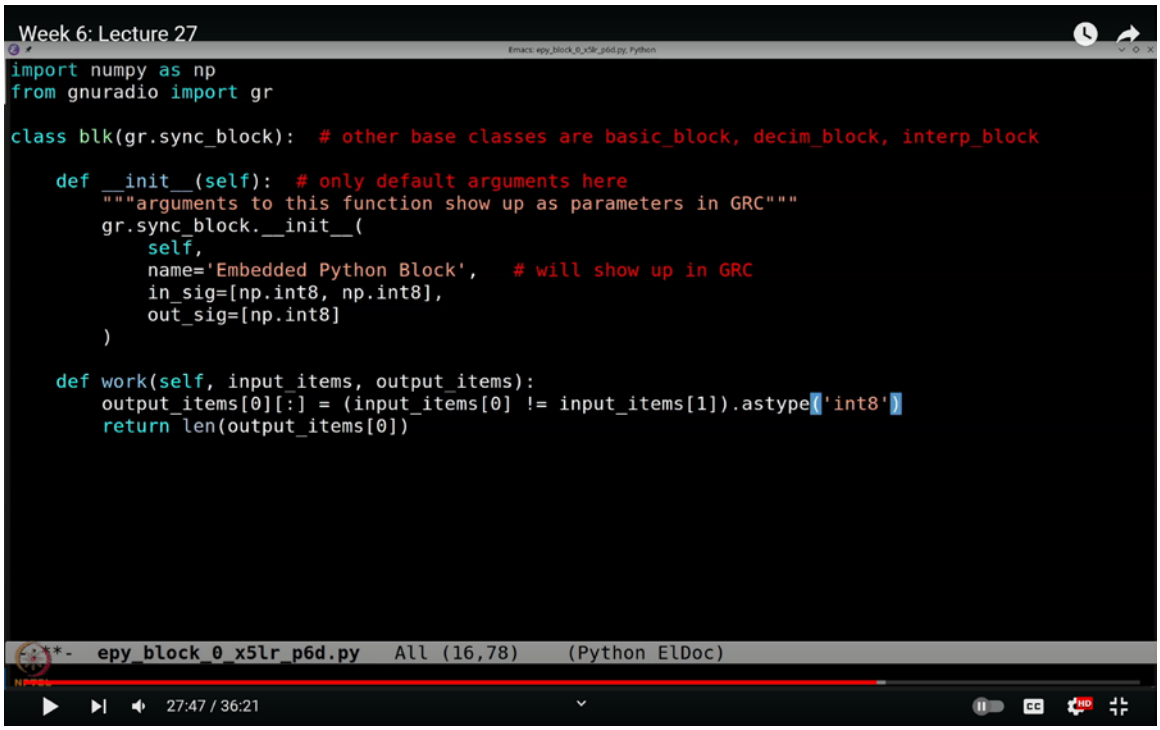
discussed creating custom Python blocks, which is the approach we will use here to identify errors and count them with a histogram.

Start by connecting the noisy output to the constellation decoder. Next, we will extract the output from the random source and build a comparison system using an embedded Python block.

To do this, press Ctrl-F (or Cmd-F) and search for "embedded Python block." Add this block to your flow graph, double-click it, and choose "Open in Editor." You can select any editor suitable for Python editing.

Once in the editor, you will see a template for the block. Since we are comparing two integers, but want to view the results as float64 in a histogram, we need to adjust the block accordingly. Remove unnecessary comments, documentation, and parameters. We will handle two inputs, which are bytes (int8), and the output can be either int8 or float, depending on your preference.

(Refer Slide Time: 27:47)



Let's use **`int8`** for our output since we're only dealing with zeros and ones. Ideally, we should stick with **`int8`** to keep things simple. We don't need any additional attributes for this.

Our task is to compare the first element of our input array with the second element, check where they differ, and then output these differences so they can be visualized in a histogram. To do this, we will use a Python snippet to handle the comparison.

First, we'll import the **`numpy`** library:

```
import numpy as np
```

Let's create two arrays to illustrate:

```
x = np.array([1, 2, 3])
```

```
y = np.array([1, 3, 3])
```

We want to identify where these arrays differ. By using:

```
x != y
```

`numpy` will return an array like **`[False, True, False]`**. This indicates which elements differ: **`True`** at positions where **`x`** and **`y`** are not equal, and **`False`** where they are the same.

To convert these boolean values to integers (where **`True`** becomes **`1`** and **`False`** becomes **`0`**), use:

```
(x != y).astype(np.int8)
```

This converts the boolean array into an array of **`int8`** values, marking differences as **`1`** (error) and similarities as **`0`** (no error).

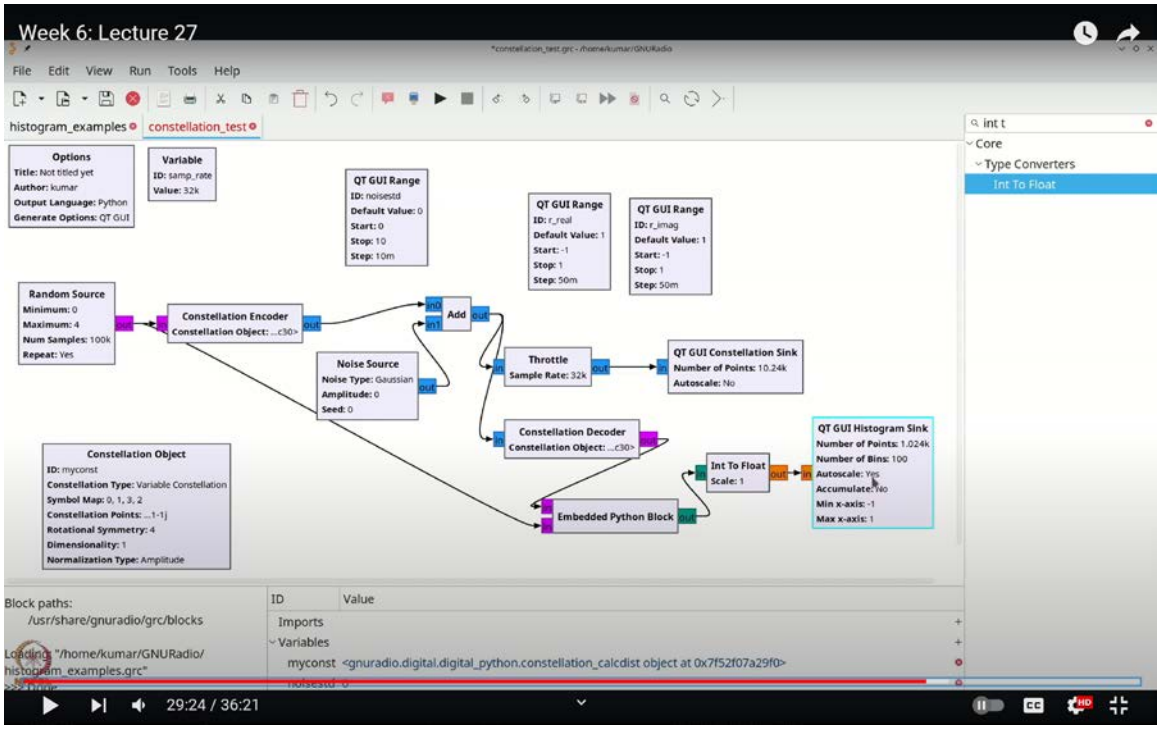
In our embedded Python block, we'll implement:

```
output_array = (input_items[0] != input_items[1]).astype(np.int8)
```

This line will create an output array where **`1`** represents an error (difference) and **`0`** represents no error.

Save and exit the editor. GNU Radio should now correctly identify that you need two byte inputs and one byte output. Connect these inputs and output accordingly.

(Refer Slide Time: 29:24)



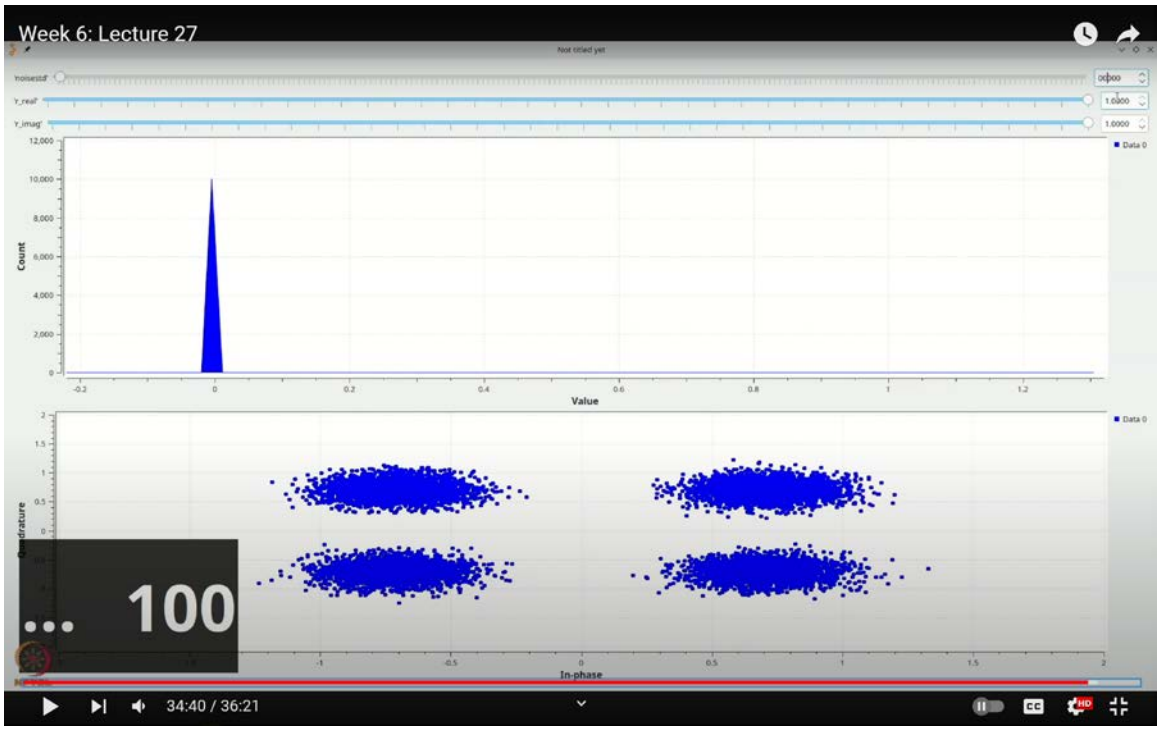
Next, add a histogram to analyze how many points differ. Press Ctrl-F (or Cmd-F) and search for "histogram." You will need to configure the histogram to accept the appropriate data type for processing. Make sure to adjust the settings to match the data type you are working with.

Let's adjust the settings for better accuracy. First, press Ctrl-F and look for "byte to float." If "byte to float" is inconvenient, we'll switch to integer processing for simplicity. To do this, double-click the block and select "Open in Editor." Change the data type from **`int8`** to **`int32`** to handle integer values properly.

Now, to facilitate easier conversion, we'll look for "int to float." Instead of converting directly to float here, we'll handle it in the next step to keep things organized. The values we are dealing with are **`0`**s and **`1`**s. Adjust the histogram settings by double-clicking it and setting more points on the x-axis, for instance, from **`-0.2`** to **`1.2`** on the y-axis.

Execute the flow graph. Initially, you should see 10,240 zeros, indicating no errors. Start increasing the noise level. At a noise standard deviation of **`0.2`**, there are still no errors, demonstrating a stable setup.

(Refer Slide Time: 34:40)



As you increase the noise standard deviation to around **`1`**, you will observe that some numbers fall into error regions. This fraction of errors indicates the number of values detected incorrectly out of the total 10,240. You will notice that the constellation performance degrades as the noise increases.

For a noise standard deviation of **`0.25`**, there are still relatively few errors. Increase it to **`0.35`**, and you'll see a slight increase in errors, suggesting that symbol errors are becoming more frequent.

When the noise standard deviation approaches **`0.5`**, the signal-to-noise ratio (SNR) is lower, resulting in a stable number of errors. The constellation shows that most points are still correctly identified, but errors are more frequent.

As you continue increasing the noise, the likelihood of errors increases significantly. With further increases, the number of errors grows substantially. This indicates that the current SNR is insufficient for reliable communication with a low symbol error rate, making it clear that the system struggles to maintain performance under high noise conditions.

When the noise level becomes so high that it overwhelms the signals, you end up with a situation where only about 3,200 to 3,300 symbols are correctly decoded, while approximately 7,000 are incorrect. This disparity occurs because you're transmitting one of four possible symbols. With such high noise levels, if you were to make random guesses, you'd be correct about one-fourth of the time.

In other words, if you were guessing numbers between 0 and 3, even if you guessed blindly and always chose 0, you would still be correct one-fourth of the time. This results in a correct-to-incorrect ratio of 1 to 3. The histogram clearly shows that out of 10,000 symbols, around 7,000 are wrong, while 3,000 are correct.

As you continue to increase the noise, this ratio remains close to 1 to 3. For instance, if the noise variance reaches 100 squared, equivalent to an extremely low signal-to-noise ratio, you'll observe that only about 2,500 symbols are correct, and around 7,500 are incorrect, giving a ratio of approximately 1 to 3. This means that only about 25% of the symbols are correctly decoded. This result aligns with the principle that, with four equally probable symbols, random guessing would yield correct results approximately 25% of the time.

With this approach, you now have a straightforward method to evaluate the performance of a communication system by simply observing the symbol error rates. From this, you can infer how well the system performs at various signal-to-noise ratios (SNRs). Generally, when the SNR is high, say around 100 or so, you can expect excellent performance. However, as the SNR decreases, you will start to encounter more errors. In cases where the error rate is significantly high, such as when the errors are very frequent, you will need to

make adjustments. This might involve reducing the transmission rate, increasing the power, or adjusting the data rate, among other trade-offs. We will explore these adjustments and their implications in future lectures.

In this lecture, we have demonstrated how combining a constellation encoder, a constellation decoder, and noise can help you visualize and measure the impact of noise on various constellations. By adjusting different parameters and settings in GNU Radio, you can simulate and assess the performance of different constellations under various conditions.

Additionally, if you are conducting hardware experiments, integrating them with GNU Radio provides a convenient way to benchmark your system's performance. In the upcoming lectures, we will build on this foundation to develop more complex simulations that closely resemble real-world scenarios. Thank you.