

Digital Communication using GNU Radio

Prof. Kumar Appiah

Department of Electrical Engineering

Indian Institute of Technology Bombay

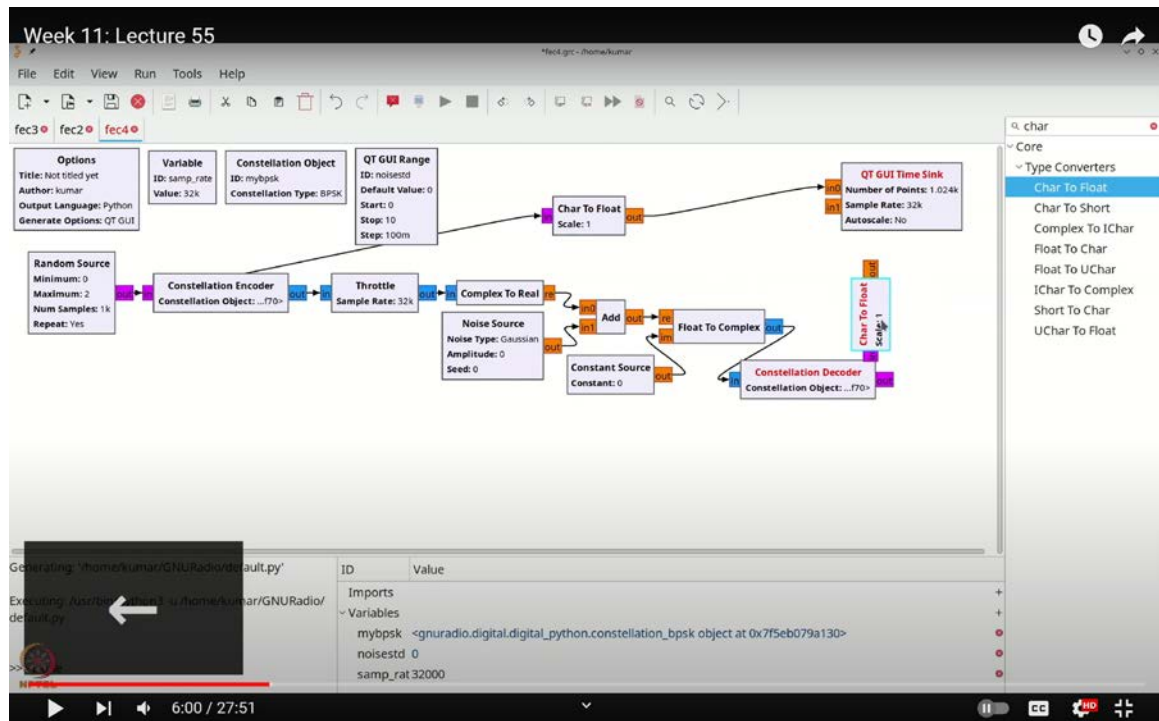
Week-11

Lecture-55

Repetition Codes in GNU Radio

Welcome to this lecture on Digital Communication using GNU Radio. My name is Kumar Appiah, and I am from the Department of Electrical Engineering at IIT Bombay. In our previous lectures, we introduced the concept of linear block codes and explored repetition codes in detail, including their practical applications. Today, we will use GNU Radio to build both a repetition code encoder and decoder.

(Refer Slide Time: 06:00)



Encoding with repetition codes is straightforward in GNU Radio. We will utilize the Repeat block to duplicate the bits as needed. For decoding, we will employ majority logic

decoding. This means we will count the number of 1s in the received bits: if the count exceeds $\frac{n-1}{2}$, we will interpret the bit as a 1; otherwise, we will interpret it as a 0. To achieve this, we will create a custom Python block that counts the 1s. For instance, in a 3-bit repetition code, if we observe 2 or more 1s, we will decode it as a 1, and if there are 0 or 1 ones, we will decode it as a 0.

We will also demonstrate the robustness of the repetition code in the presence of noise by using a histogram to analyze the effect of noise. Our setup will simulate a binary symmetric channel using Binary Phase Shift Keying (BPSK) with additive white Gaussian noise (AWGN).

Let's get started. First, we will add a random source to generate our data. Use Control-F or Command-F to search for "random" and place the random source block in the workspace. As usual, we will convert the data to the byte data type, setting the maximum value to 2.

Next, we need to add constellation-related blocks. Use Control-F or Command-F to search for "constellation." Add the constellation object and the constellation encoder blocks. Set the constellation object to BPSK and name it ``myBPSK``. Connect the constellation encoder to this object, so it uses the ``myBPSK`` configuration.

We are now ready to add a throttle block. Again, use Control-F or Command-F to search for "throttle" and place the throttle block in the workspace. With the throttle in place, we can now add a noise source to introduce AWGN into our system.

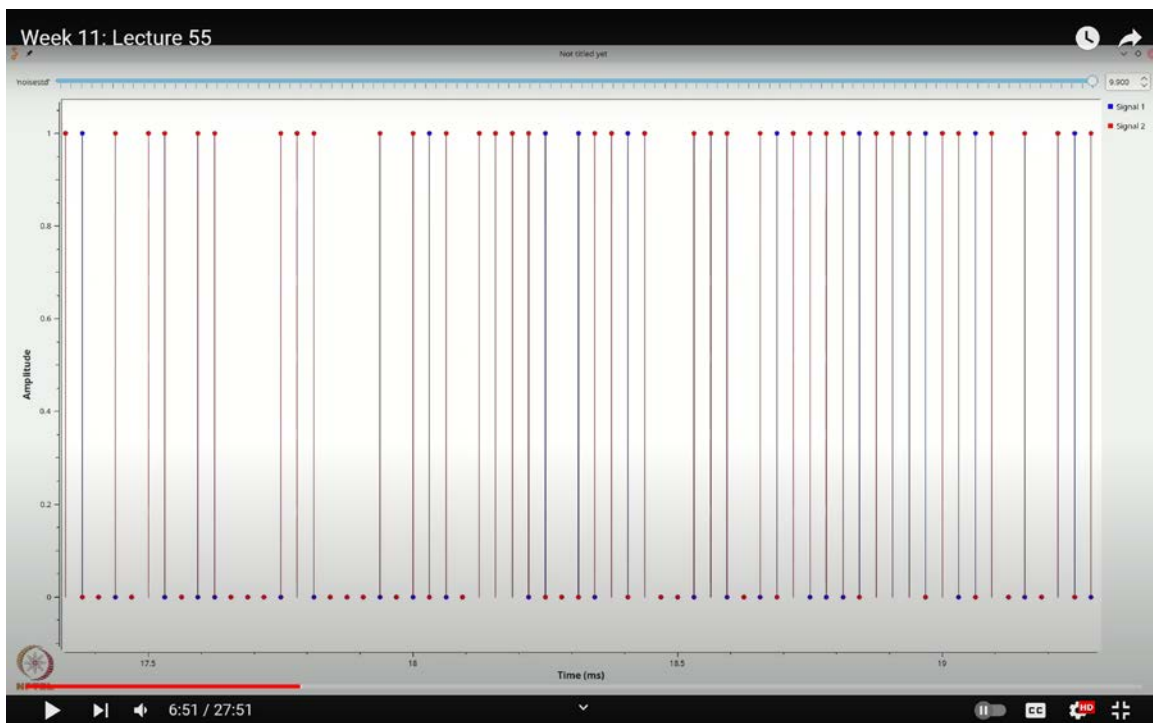
Before proceeding, let's make the setup more concrete. For Binary Phase Shift Keying (BPSK), we can use a real channel setup. To begin, use Control-F or Command-F to search for "complex to real" and add the Complex to Real block. Next, add a noise source by searching for "noise" and placing the Noise Source block in the workspace. Set the noise source to float type and adjust the amplitude to match the noise standard deviation (noise STD) so that you can control the amplitude. Search for "range" and add a Range block to manage the noise STD. Double-click the Range block, name it "noise STD," set the default value to 0, set the maximum value to 10, and step size to 0.1.

Now we can proceed with a few additional steps. To decode the constellation, first, add a Float to Complex block. For this, the imaginary part should be set to 0, so include a Constant Source block, set it to float type, and assign a value of 0. Connect this Constant Source to the imaginary input of the Float to Complex block.

Next, add a Constellation Decoder block. Search for "constellation decoder" and add it to the workspace. Once you have the Constellation Decoder, configure it to use the `myBPSK` object you set up earlier. This will allow you to access the decoded bits.

To compare the original data with the decoded data, use two methods. First, add a Time Sink block by searching for "time sink." Rotate the Time Sink using the left key, double-click it to set the type to float, and configure it to have two inputs. Since the data is in bytes, you need to convert it. Add a Char to Float block by searching for "char" and place it in the workspace. Connect the original data source to the first input of the Time Sink and connect the decoded data source to the second input.

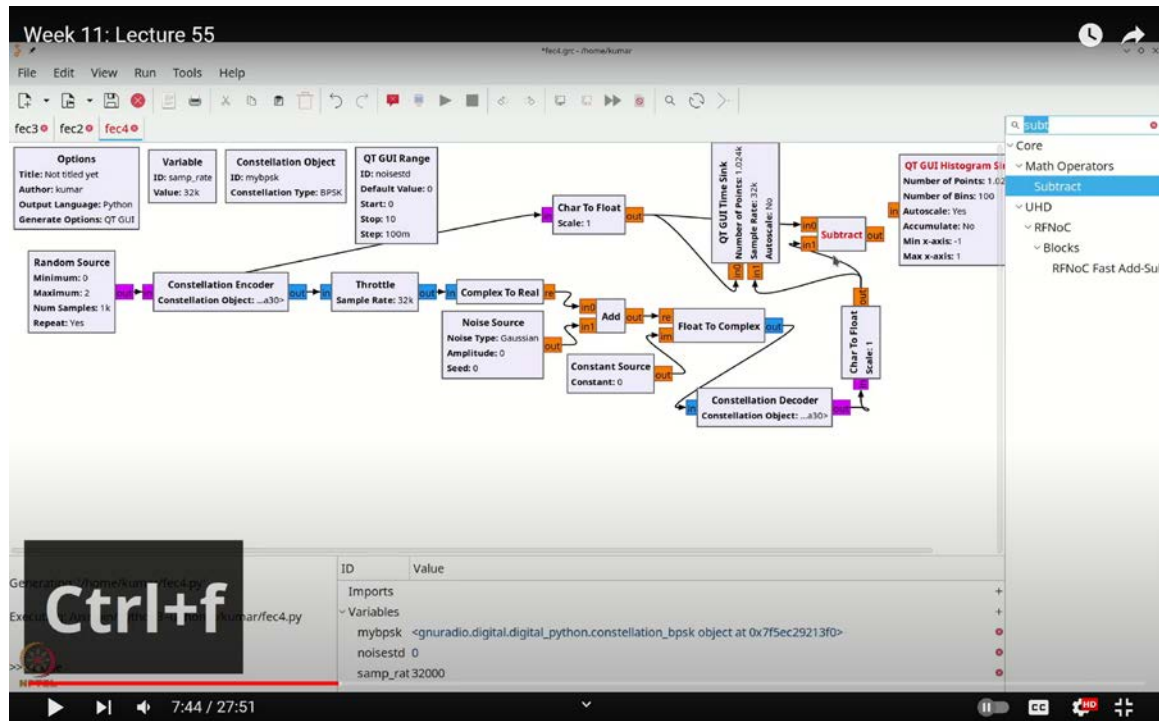
(Refer Slide Time: 06:51)



Now you can visualize the data. If there are no errors, the red and blue lines should overlap.

As you increase the noise STD, you will begin to see discrepancies where the red and blue lines diverge, indicating errors. For instance, where the blue line represents the decoded data and the red line represents the original data, any mismatch signals an error.

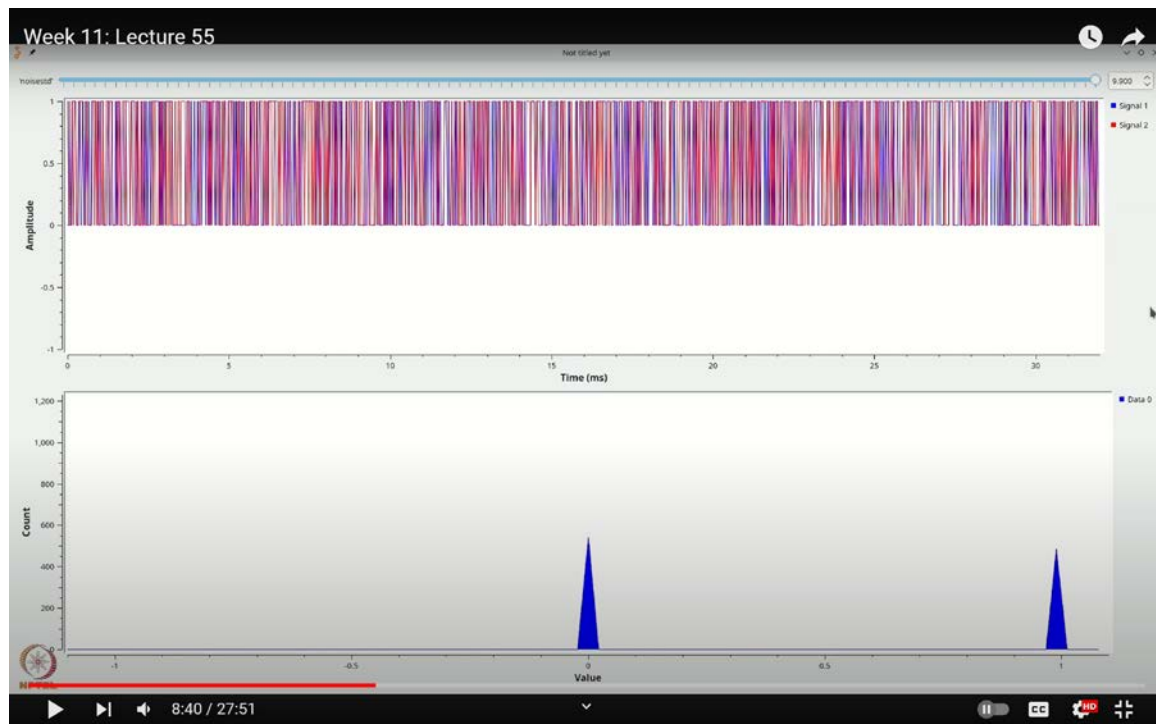
(Refer Slide Time: 07:44)



To further analyze the errors, use a Histogram Sink block. Search for "histogram sink," add it to the workspace, and connect it to a subtraction operation between the original and decoded data. Subtracting these will yield values of 1 (for 1 - 0), 0 (for 1 - 1 or 0 - 0), or -1 (for 0 - 1). This histogram will help you visualize the distribution of errors.

We can subtract these two values and then take their absolute value to determine if there is an error. If the result is 0, it indicates no error; if the result is 1, it indicates an error. To accomplish this, search for "subtract" using Control-F or Command-F, add the Subtract block, and set it to float type. Subtract the original stream from the decoded stream, then add an Absolute Value block by searching for "abs," set this block to float type as well, and connect it to the output of the Subtract block.

(Refer Slide Time: 08:40)



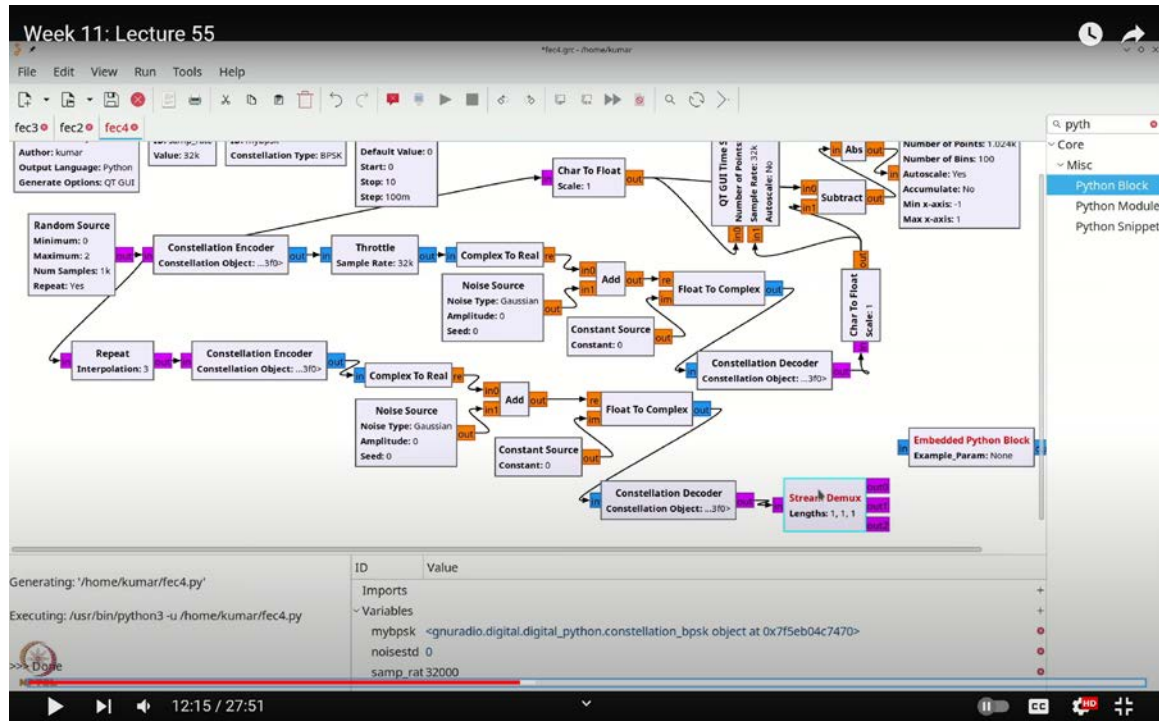
Next, connect this output to a Histogram Sink block to visualize the results. When you execute this flow graph, the histogram will display in blue, indicating that 1000 out of 1000 bits were decoded correctly. As you increase the noise standard deviation (noise STD), the histogram will show errors with a value of 1. For instance, with increased noise, you might observe around 400 errors and approximately 600 correct decodings. If you reduce the bit error probability (P), you might see around 300 errors and about 600-700 correct decodings.

This setup allows you to adjust the error probability and observe the impact on decoding accuracy. With a lower error probability, you will achieve better decoding accuracy, while a higher probability results in more errors. At an extreme error probability, the performance resembles a coin toss, with a bit error rate of 50%.

Having established a basic binary symmetric channel, our next step is to incorporate a code to enhance performance. We will use a repetition code for this purpose. The simplest way to implement a repetition code is to repeat the transmitted bits and decode them using

majority logic at the receiver.

(Refer Slide Time: 12:15)



To implement this, search for "repeat" using Control-F or Command-F and add the Repeat block. Double-click the Repeat block and set the interpolation factor to 3 and the type to byte. Connect this to the random source. You could also repeat the encoded output, but for simplicity, let's add another Constellation Encoder. Copy and paste the Constellation Encoder block and connect it appropriately.

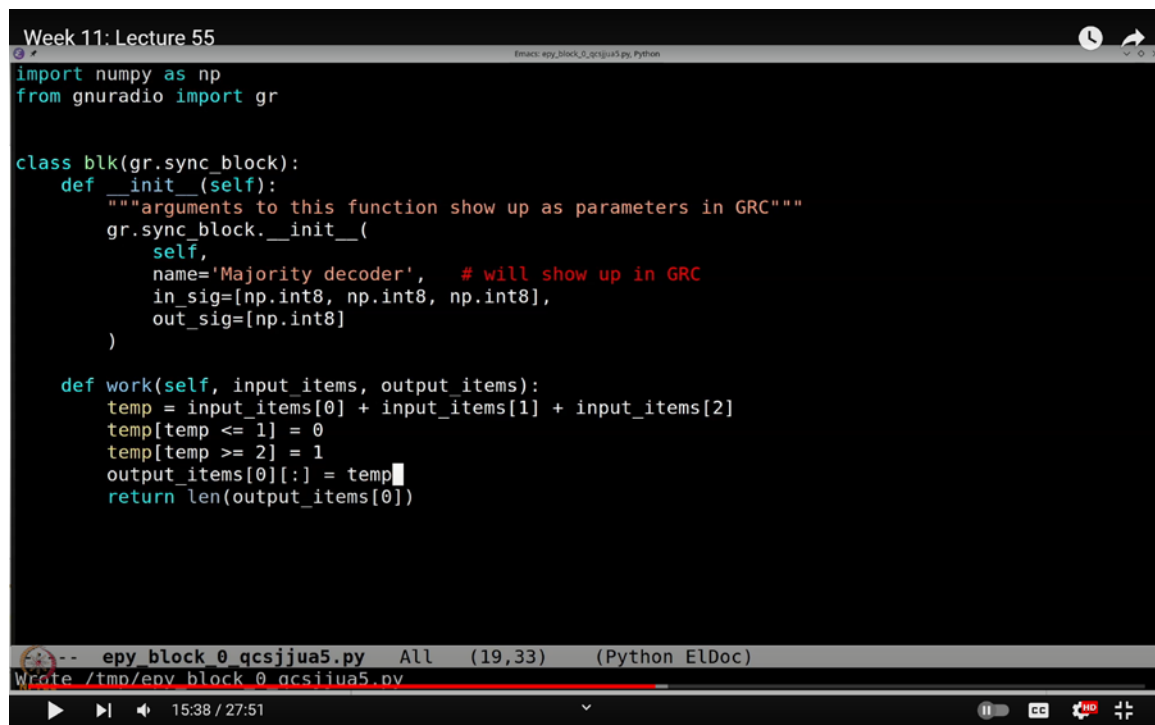
Since the interpolation rate has changed to 3, we need to adjust the noise source accordingly. Copy and paste a fresh Noise Source block and set it to use the same noise standard deviation (noise STD). Finally, add another Add block and connect it to the noise sources. This setup ensures that the repetition code is effectively applied and allows us to observe improvements in performance.

Next, I'll need to convert the data back from float to complex. To do this, I'll copy all three of these objects using Control-C and Control-V and move them over to the appropriate location. I'll convert this to real, and then use the constellation decoder. The decoded output

can be viewed using the QtGUI Time Sink or the Histogram Sink.

However, before viewing, I need to perform majority logic decoding to reduce from 3 bits to 1 bit. To achieve this, I'll split the output into three separate streams. For this, I'll use the Stream Demux block. Search for "streamDemux" using Control-F or Command-F, add the Stream Demux block, and connect it to the output. Remember to adjust the type settings, double-click the block and set it to byte. You can also use the compact notation ``111`` or, alternatively, ``1*1*3`` to indicate the split into three parts.

(Refer Slide Time: 15:38)

A screenshot of a video player window titled "Week 11: Lecture 55". The video content shows a Python code editor with the following code:

```
import numpy as np
from gnuradio import gr

class blk(gr.sync_block):
    def __init__(self):
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Majority decoder', # will show up in GRC
            in_sig=[np.int8, np.int8, np.int8],
            out_sig=[np.int8]
        )

    def work(self, input_items, output_items):
        temp = input_items[0] + input_items[1] + input_items[2]
        temp[temp <= 1] = 0
        temp[temp >= 2] = 1
        output_items[0][:] = temp
        return len(output_items[0])
```

The video player interface includes a progress bar at the bottom showing the current time as 15:38 out of 27:51. The status bar at the very bottom indicates the file path and editor: "epy block 0 qcsjjua5.py All (19,33) (Python ElDoc)" and "Wrote /tmp/epy_block_0 qcsjjua5.py".

Now, to perform majority decoding, there are several approaches. The essential idea is that if there are 0 or 1 ones among the three bits, it is concluded that a 0 was sent. Conversely, if there are 2 or 3 ones, it is concluded that a 1 was sent. This process was discussed in class, but to implement majority logic decoding, we will use a Python block. Search for "Python" using Control-F or Command-F, and add the Python block.

Double-click the Python block to open the editor. Choose any convenient editor and edit the block to implement majority decoding logic. We need three byte inputs and one byte

output for this process.

Start by removing any unnecessary comments and parameters from the code. We'll name this function `majority_decoder`. We will use `np.int8` for the inputs, specifying it three times since we need three inputs. The output will also be a single `np.int8`.

The `np.int8` type represents an 8-bit integer where each bit is either 0 or 1, but only the least significant bit matters here. The core task is to count the number of 1s and 0s in these three parallel inputs and determine the sent value based on the majority.

Now, let's adjust the work function. We'll implement the majority decision logic carefully. Our approach will be to sum up the inputs. Each input entry is either 0 or 1. By adding the three inputs, the possible results are 0, 1, 2, or 3.

Let's use a temporary variable, `temp`, to represent the sum of the inputs: `temp = input_items[0] + input_items[1] + input_items[2]`. The `temp` array will consist of these sums, with each entry being either 0, 1, 2, or 3. To map these sums to output values, we will set the output to 1 if `temp` is 2 or 3, and to 0 if `temp` is 0 or 1.

We'll first handle the case where `temp` is 0 or 1. So, wherever `temp` is less than or equal to 1, we set the output to 0. Wherever `temp` is 2 or more, we set it to 1. Finally, we map these results to the output items, completing the majority decoding process.

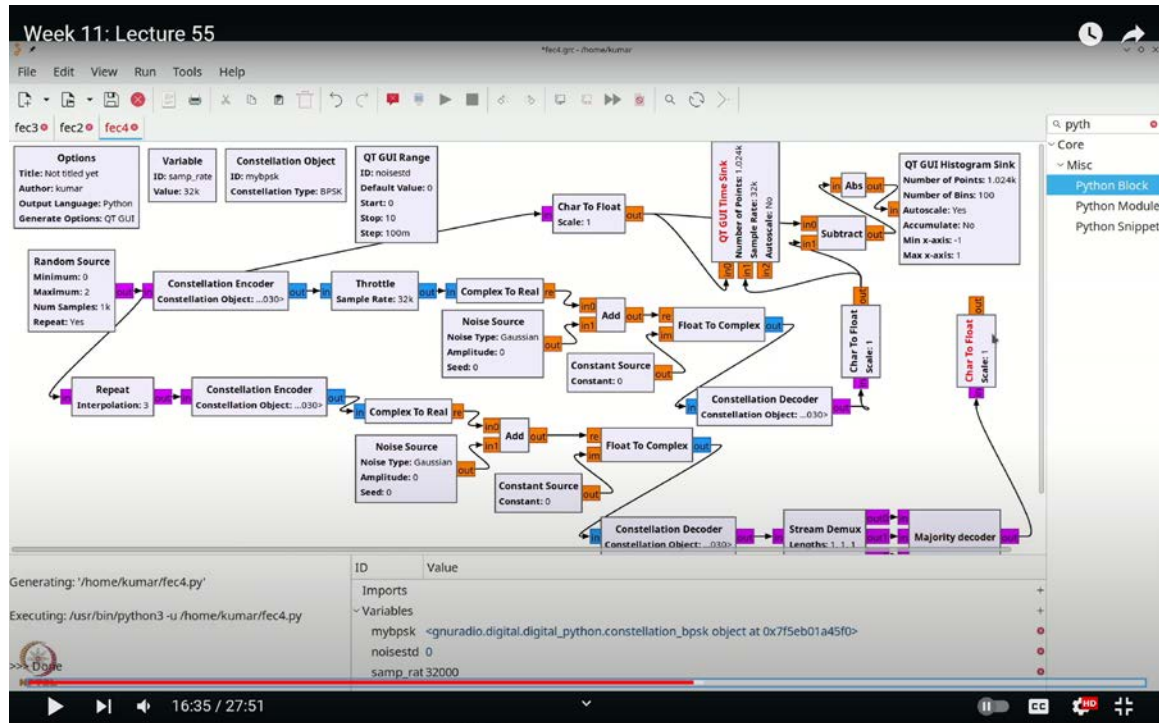
Why does this method work? Each `input_items` array, `input_items[0]`, `input_items[1]`, and `input_items[2]`, contains the same length arrays with the 3-bit repeated code that has been transmitted through the binary symmetric channel. Our goal is to deduce the sent bit based on the most probable scenario. The most probable case is that either a single error occurred or no error occurred.

If a single error occurred, for instance, sending `000` might result in outputs like `000`, `001`, `010`, or `100`. Therefore, if we observe a single `1` among the three, we can confidently conclude that `000` was sent. Conversely, if there are 2 or more `1`s, we conclude that `111` was sent, and map it back to the correct bit.

Now, save this function and exit. We have successfully implemented the majority decoder.

Next, we can connect this decoder and compare its output.

(Refer Slide Time: 16:35)



First, let's perform a time-based comparison. Add three inputs to the time sink. Set the number of inputs to 3, and copy the character-to-float block (Control-C, Control-V). Connect these blocks as needed. When you execute the flow graph, you should see that the green line matches the blue line, indicating no errors.

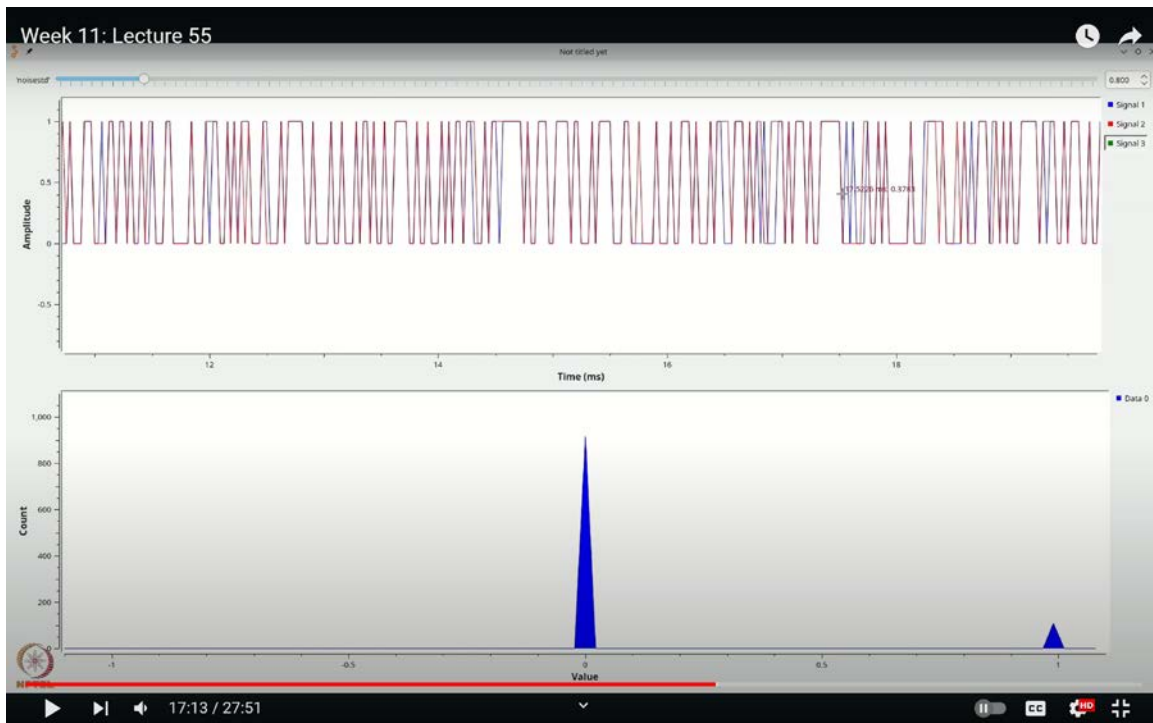
If you increase the noise, the green line will track the blue line more closely. By adjusting the error level, you'll notice that the green and blue lines have fewer mismatches compared to the red line. With higher noise, the red line will show more errors.

To better understand the error distribution, use a histogram to visualize the data. Let's modify the histogram accordingly.

Let's proceed by double-clicking to adjust the settings. First, we'll disable auto-scaling and add two inputs. For the second input, you need to perform the same operations: subtraction and absolute value, similar to what was done for the first input.

To do this, copy the existing subtraction block (Ctrl+C, Ctrl+V). Connect this new subtraction block to the corresponding output, and link the original signal to this block. Then, copy and add an absolute value block (Ctrl+C, Ctrl+V). Connect this block to the output as well. Adjust the layout by moving these blocks slightly to the right for better visibility.

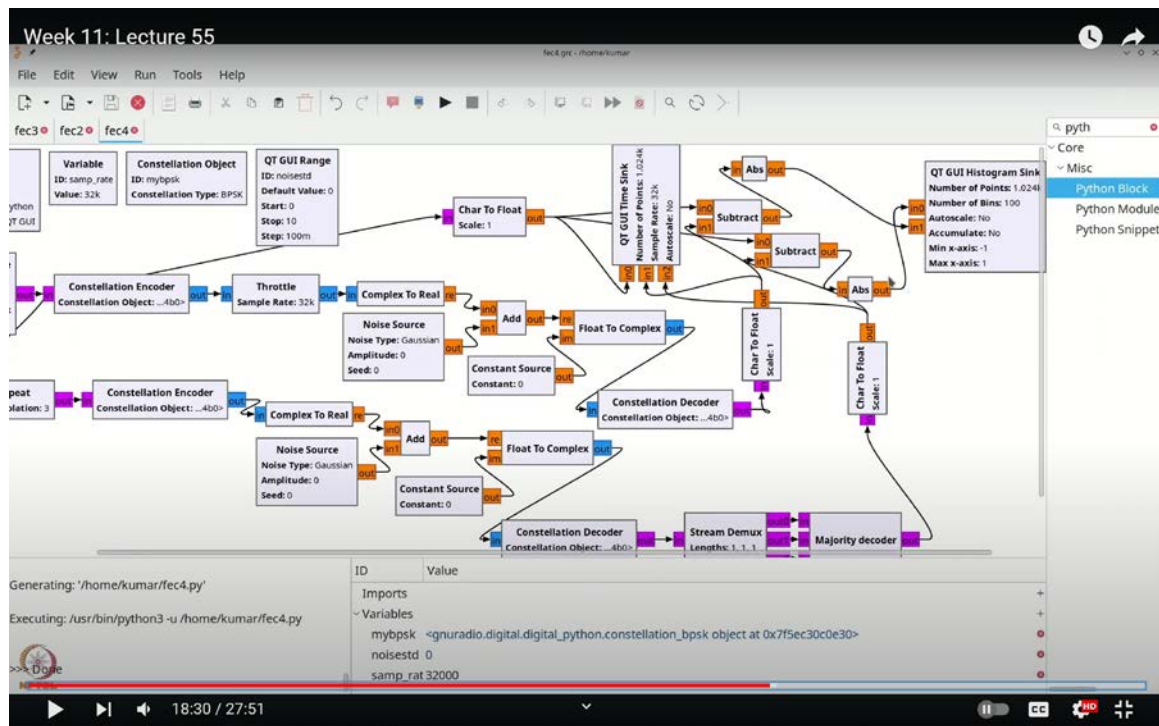
(Refer Slide Time: 17:13)



Executing this flow graph will show overlapping red and blue signals. If needed, swap the connections for clarity. Increasing the noise will reveal that the blue signal, which represents the case without coding, exhibits a higher number of errors compared to the red signal with repetition coding. The repetition coding provides robust error correction, demonstrating fewer errors even as noise increases.

As noise levels rise, both signal peaks will decrease, but the red signal will remain below the blue signal due to the effective error correction of repetition coding. Increasing the level of repetition coding improves performance. For instance, if you set the interpolation to 5 and adjust the majority decoder to handle 5 errors, make the necessary modifications.

(Refer Slide Time: 18:30)

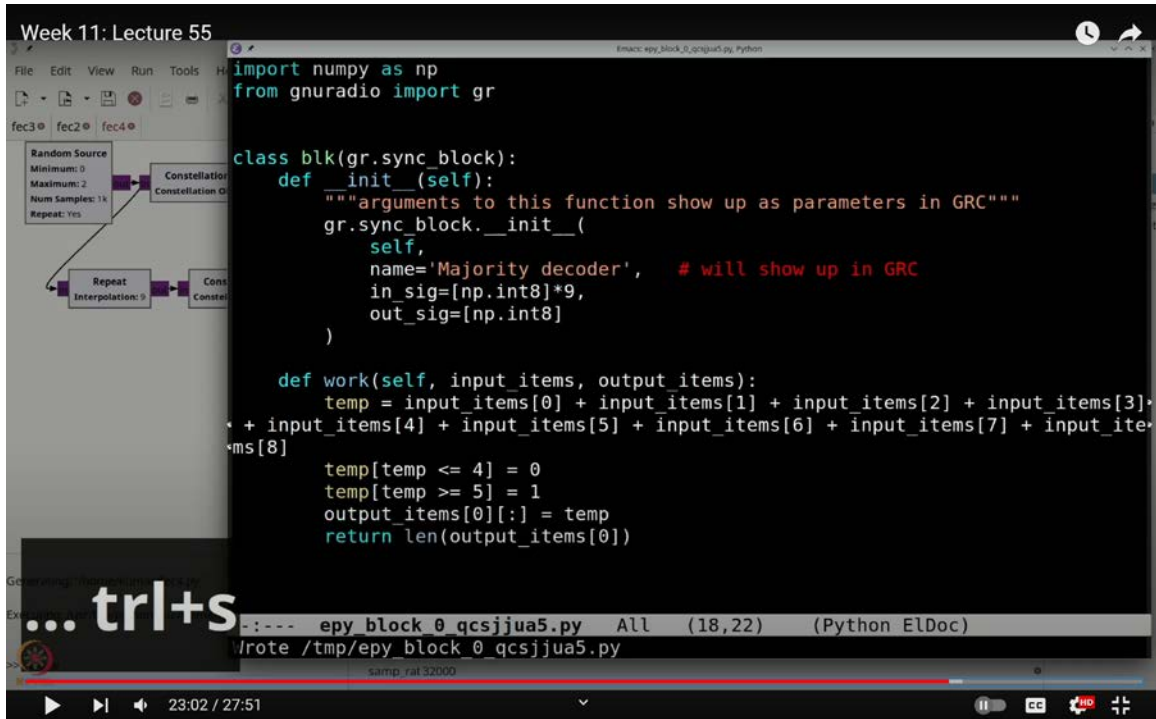


(Refer Slide Time: 19:08)



Add two inputs to the majority decoder block. Adjust the majority logic so that 0, 1, or 2 errors result in a 0, and 3, 4, or 5 errors result in a 1. Ensure that the syntax is correct, there should be a comma in the code. Fix any syntax errors to proceed.

(Refer Slide Time: 23:02)



The screenshot shows a video player interface. On the left, a GRC flow graph is visible with blocks like 'Random Source', 'Constellation', 'Repeat Interpolation: 9', and 'Constellation'. On the right, a Python code editor shows the following code:

```
import numpy as np
from gnuradio import gr

class blk(gr.sync_block):
    def __init__(self):
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Majority decoder', # will show up in GRC
            in_sig=[np.int8]*9,
            out_sig=[np.int8]
        )

    def work(self, input_items, output_items):
        temp = input_items[0] + input_items[1] + input_items[2] + input_items[3] +
        + input_items[4] + input_items[5] + input_items[6] + input_items[7] + input_items[8]
        temp[temp <= 4] = 0
        temp[temp >= 5] = 1
        output_items[0][:] = temp
        return len(output_items[0])
```

The video player controls at the bottom show the time is 23:02 / 27:51.

Connect these adjustments and run the flow graph again. You will observe that with a higher number of errors, the red signal (with repetition coding) performs significantly better. Comparing with the same noise standard deviation, say 1.5, the red signal shows a notable improvement, demonstrating the enhanced error-correcting capability of the repetition coding method.

Keep in mind that the 5-bit repetition code can correct up to two errors. Even with a significant amount of noise, the error correction capability of the 5-bit repetition code remains highly effective. Let's take this a step further by increasing the repetition code to 9.

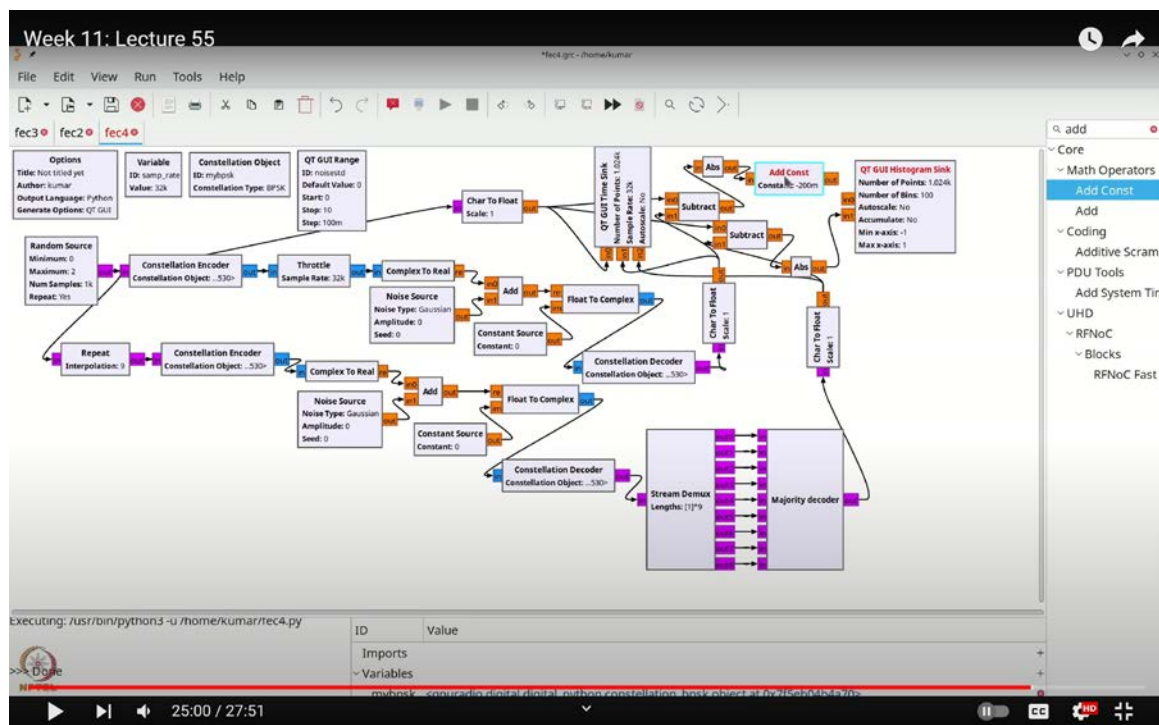
Here's what we need to do: set this value to 9, adjust the majority decoder accordingly, and ensure that all related blocks are also updated to handle 9 repetitions. For better accuracy,

you can use a wildcard notation like `*9` to specify 9 inputs and similarly adjust the majority logic to account for these changes.

In the case of a 9-bit repetition code, you will use the following logic:

- If the number of ones is 0, 1, 2, 3, or 4, the output should be 0.
- If there are 5 or more ones, the output should be 1.

(Refer Slide Time: 25:00)



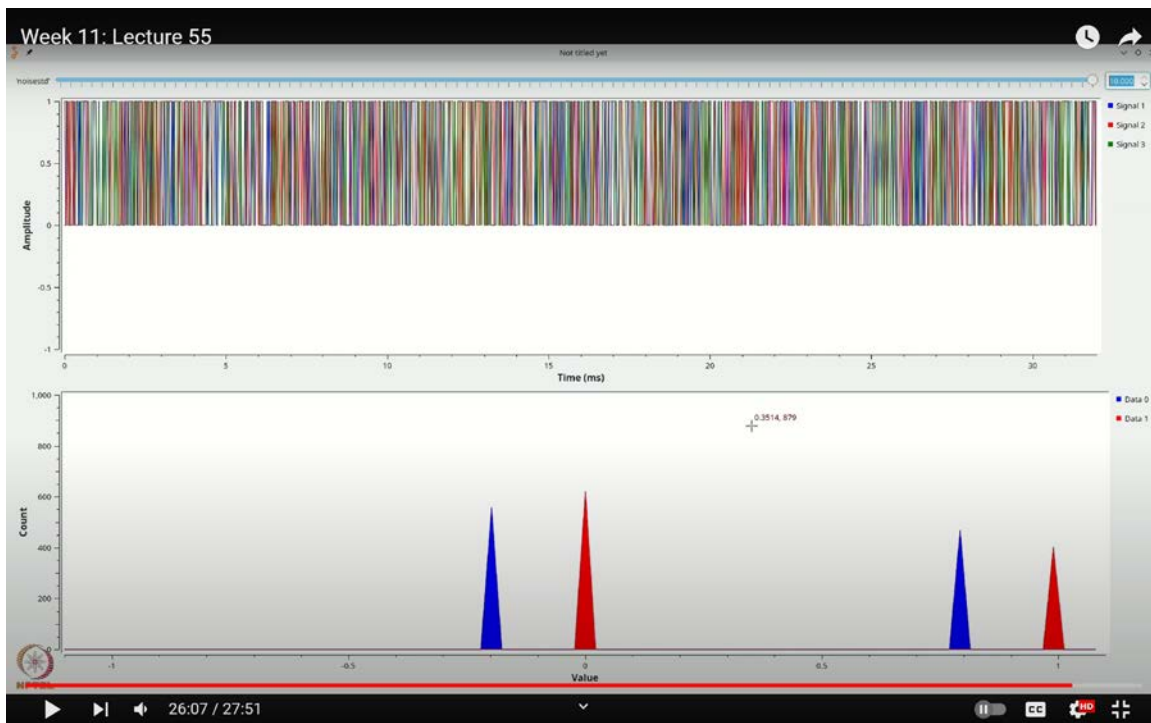
Let's evaluate the performance with these changes. Connect the necessary blocks, and run the flow graph with a noise standard deviation of 1.5. You'll observe that the error rate with the 9-bit repetition code is significantly lower, with only about 5% of the bits showing errors while the rest are correctly decoded. Even as you increase the noise, the red signal (with repetition coding) demonstrates remarkable robustness, showcasing the enhanced error-correcting performance of the 9-bit repetition code.

However, it's important to note that while repetition coding provides excellent error correction, it comes at the cost of reducing the effective data rate to one-ninth. This trade-

off in efficiency means that while repetition codes are highly reliable, they may not always be practical in scenarios where efficiency is crucial. Therefore, in our next lecture, we will explore more efficient coding techniques, such as Hamming codes.

One final tip: when using the histogram sink, overlapping blue and red bars can make it difficult to differentiate their heights. To resolve this, you can offset one of the data sets slightly. For instance, you can add a constant to one of the signals. To do this, use the "Add Constant" block. Set the constant to a value like -0.2. By adding this constant before sending the data to the histogram, you can clearly see the histogram bins for each data set separately.

(Refer Slide Time: 26:07)



When you run the flow graph, you'll see that the histograms are displayed separately, making it easier to compare the heights of the data sets. This visualization becomes even more convenient when adding noise, as it allows you to compare the heights directly on the same graph. This approach significantly simplifies the visualization process.

One important observation is related to extreme noise conditions. For example, setting the

noise standard deviation to 10, equivalent to a noise variance of 100, which implies a signal-to-noise ratio (SNR) of approximately 1/100, results in poor performance for both coded and uncoded systems. Although the repetition code still shows better performance by correcting some errors, this reinforces our hypothesis: as the error probability P approaches 0.5 (meaning the error vector E_v approaches zero), error performance deteriorates significantly if the noise level is too high. This confirms that even with error correction coding, performance can be severely impacted if the noise is overwhelming.

In the context of BPSK, where the noise variance N_0 is very high, distinguishing between +1 and -1 becomes challenging. During this lecture, we constructed a repetition code-based error correction system in GNU Radio. We observed that while repetition codes can still encounter errors, they are significantly more robust compared to uncoded systems. The repetition code provided substantial error correction even under high noise conditions, as evidenced by the histograms comparing correctly and incorrectly decoded bits.

However, a major drawback of the repetition code is its inefficiency in terms of data rate. For a 9-bit repetition code, the data rate is reduced to one-ninth of the original, resulting in considerable overhead and performance loss. In our next lecture, we will explore Hamming codes, which effectively correct single-bit errors while maintaining a much higher data rate than repetition codes. Thank you.