Digital Communication using GNU Radio

Prof. Kumar Appaiah

Department of Electrical Engineering

Indian Institute of Technology Bombay

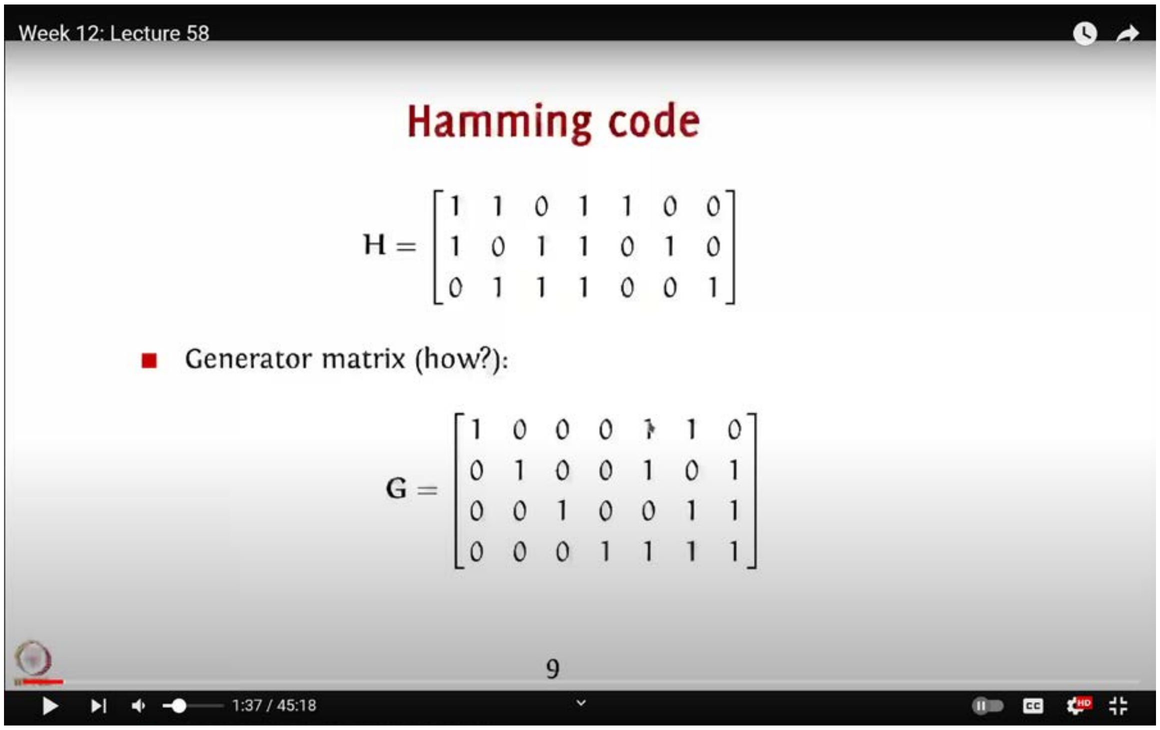
Week-12

Lecture-58

(7,4) Hamming Code in GNU Radio

Welcome to this lecture on Digital Communication using GNU Radio. My name is Kumar Appiah, and I am a member of the Department of Electrical Engineering at the Indian Institute of Technology, Bombay. In this session, we will continue our exploration of error correction codes, with a specific focus on binary Hamming codes. We will implement these codes using GNU Radio and embedded Python blocks, which provide us with a powerful means of customizing the coding process.

(Refer Slide Time: 01:37)



Hamming code

$$H = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

- Generator matrix (how?):

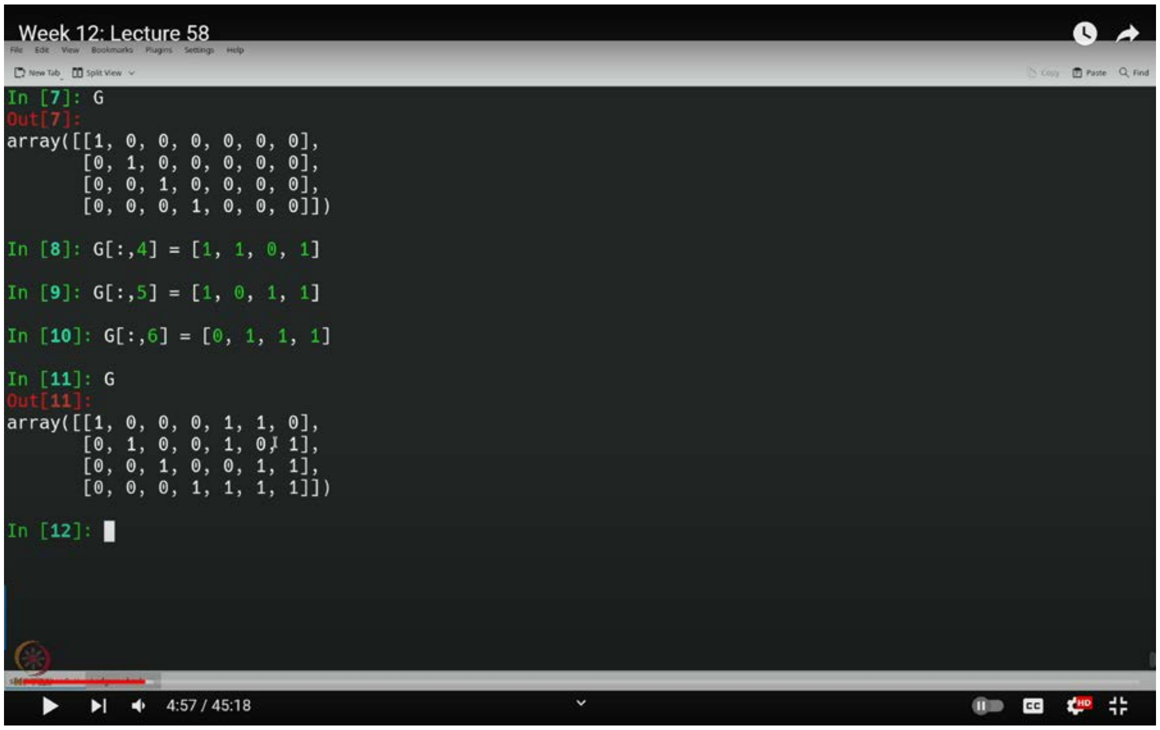
$$\mathbf{G} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

9

Our primary focus will be on the (7,4) Hamming code, which encodes 4 information bits into 7 coded bits. This code is particularly valuable because it can correct any single-bit error without any loss of fidelity. We will verify the functionality of the Hamming code and evaluate its performance under various scenarios.

In the previous lecture, we defined the Hamming code using a parity check matrix. We began with a matrix containing the rows 1101, 1011, and 0111, followed by the identity matrix. We then used this to create the generator matrix, which included the identity matrix and the transpose of the initial matrix, resulting in columns of 1101, 1011, and 0111. Today, we'll see how to implement this generator matrix in Python, which we will then translate into a GNU Radio Python block.

(Refer Slide Time: 04:57)



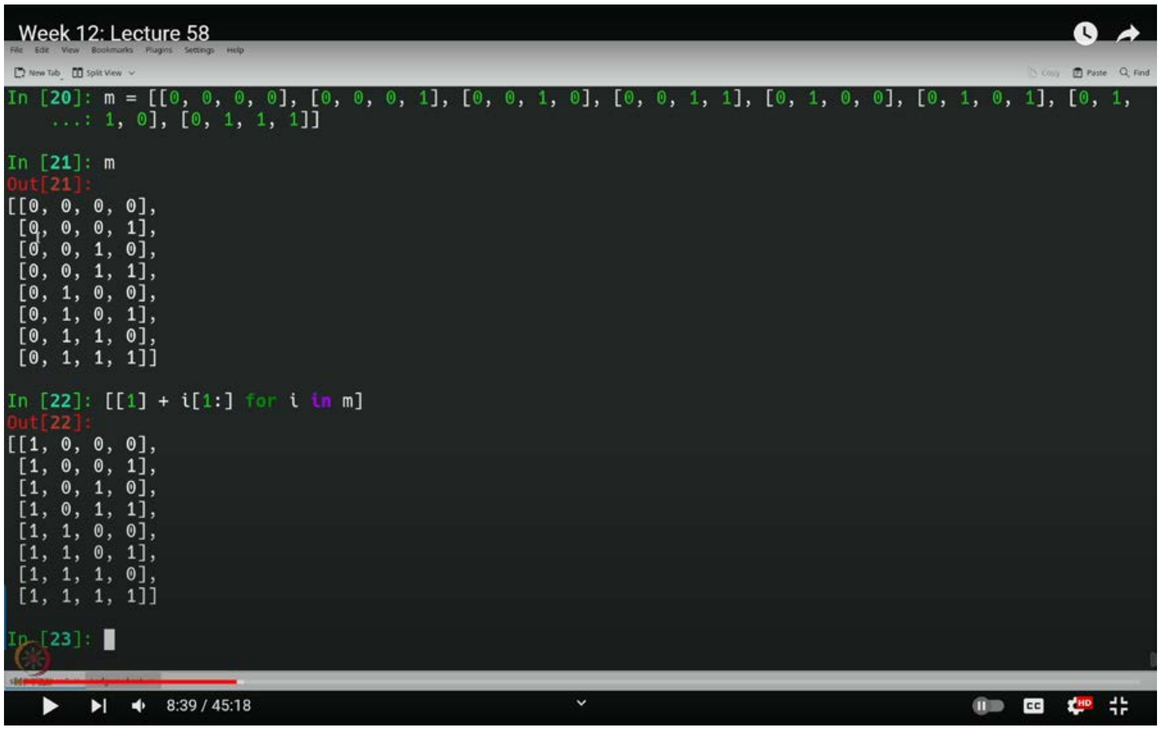
```
In [7]: G
Out[7]:
array([[1, 0, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0]])

In [8]: G[:,4] = [1, 1, 0, 1]

In [9]: G[:,5] = [1, 0, 1, 1]

In [10]: G[:,6] = [0, 1, 1, 1]

In [11]: G
Out[11]:
array([[1, 0, 0, 0, 1, 1, 0],
       [0, 1, 0, 0, 1, 0, 1],
       [0, 0, 1, 0, 0, 1, 1],
       [0, 0, 0, 1, 1, 1, 1]])

In [12]:
```

To start, we'll use a Python prompt to create the generator matrix and verify all possible code words for the (7,4) Hamming code, which has 16 distinct 7-bit code words. We'll begin by writing the generator matrix.

First, we need to import the **`numpy`** library. Type **`import numpy as np`** to bring in this essential tool. Next, we'll construct the generator matrix G.

We'll create a zero array to start, ensuring that the data type is set to integers by specifying **`dtype=int`**. We will define this array as a 4 by 7 matrix. Given that the first 4 by 4 submatrix is an identity matrix, we can access this portion of the matrix using **`[:, :4]`**. This operation selects the first submatrix, allowing us to build the generator matrix accordingly.

(Refer Slide Time: 08:39)



```
In [20]: m = [[0, 0, 0, 0], [0, 0, 0, 1], [0, 0, 1, 0], [0, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 1], [0, 1,
    ...: 1, 0], [0, 1, 1, 1]]

In [21]: m
Out[21]:
[[0, 0, 0, 0],
 [0, 0, 0, 1],
 [0, 0, 1, 0],
 [0, 0, 1, 1],
 [0, 1, 0, 0],
 [0, 1, 0, 1],
 [0, 1, 1, 0],
 [0, 1, 1, 1]]

In [22]: [[1] + i[1:] for i in m]
Out[22]:
[[1, 0, 0, 0],
 [1, 0, 0, 1],
 [1, 0, 1, 0],
 [1, 0, 1, 1],
 [1, 1, 0, 0],
 [1, 1, 0, 1],
 [1, 1, 1, 0],
 [1, 1, 1, 1]]

In [23]:
```

To verify our generator matrix G, we first set it to I_4, which gives us the identity matrix. Now, we need to properly specify the 5th, 6th, and 7th columns of G (corresponding to columns 4, 5, and 6, respectively, because Python uses zero-based indexing). Let’s proceed to fill in these entries. We will update the matrix as follows:

G[:, 4] = ... # Set the 5th column

G[:, 5] = ... # Set the 6th column

G[:, 6] = ... # Set the 7th column

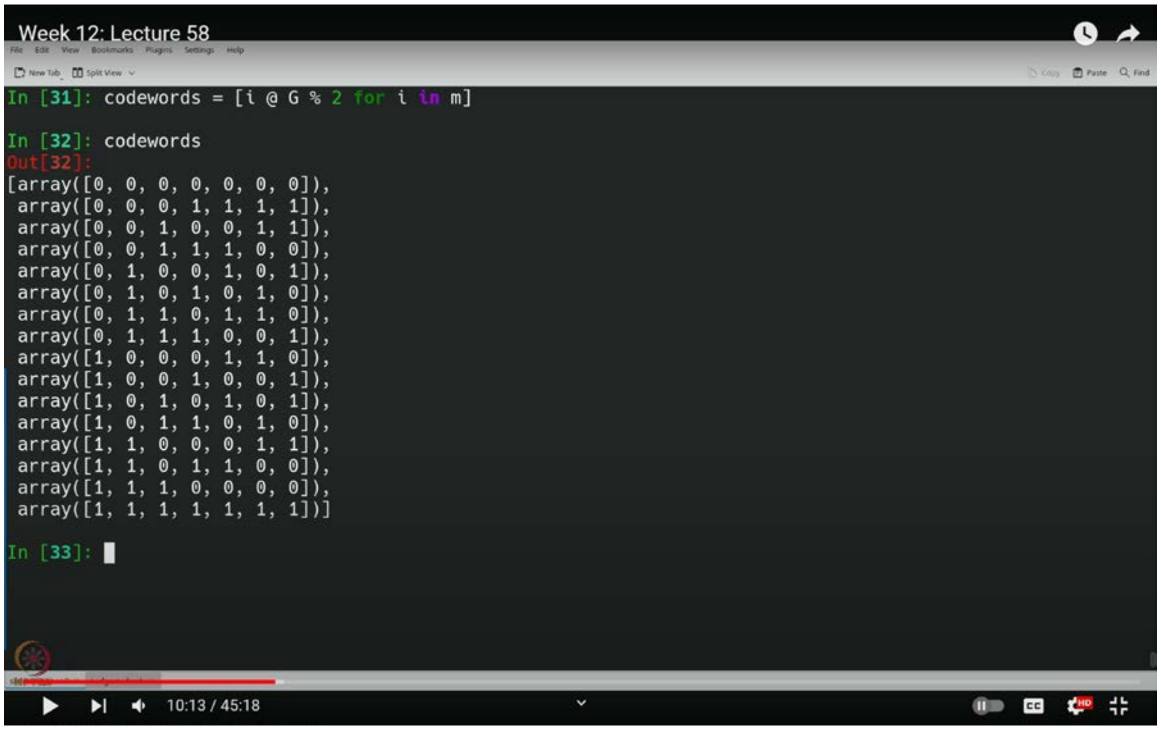
Upon inspection, G should now match the generator matrix we intended.

Next, let’s construct the parity check matrix H. Since we have already created the generator matrix, deriving the parity check matrix is straightforward. The parity check matrix H consists of the identity matrix on the right and the transpose of matrix A on the left. Thus, we can take the non-identity part of G and use it to populate H.

We begin by initializing H with zeros. For the (7,4) Hamming code, the parity check matrix H is of size 3 x 7. We need to place the identity matrix in the last three columns. This is done using the following code:

H[:, -3:] = np.i(3) # Place the 3x3 identity matrix in the last three columns

(Refer Slide Time: 10:13)



Week 12: Lecture 58

```
In [31]: codewords = [i @ G % 2 for i in m]

In [32]: codewords
Out[32]:
[array([0, 0, 0, 0, 0, 0, 0]),
 array([0, 0, 0, 1, 1, 1, 1]),
 array([0, 0, 1, 0, 0, 1, 1]),
 array([0, 0, 1, 1, 1, 0, 0]),
 array([0, 1, 0, 0, 1, 0, 1]),
 array([0, 1, 0, 1, 0, 1, 0]),
 array([0, 1, 1, 0, 1, 1, 0]),
 array([0, 1, 1, 1, 0, 0, 1]),
 array([1, 0, 0, 0, 1, 1, 0]),
 array([1, 0, 0, 1, 0, 0, 1]),
 array([1, 0, 1, 0, 1, 0, 1]),
 array([1, 0, 1, 1, 0, 1, 0]),
 array([1, 1, 0, 0, 0, 1, 1]),
 array([1, 1, 0, 1, 1, 0, 0]),
 array([1, 1, 1, 0, 0, 0, 0]),
 array([1, 1, 1, 1, 1, 1, 1])]

In [33]:
```

Now, to complete H, we take the second part of G (the part that excludes the identity matrix), transpose it, and insert it into the first part of H:

H[:, :4] = G[:, 4:].T

This gives us the parity check matrix H we need. Recall from the lecture that if G has the form $[I\ A]$, then H should be $[A^T\ I]$ for a binary matrix. To verify that G and H are indeed correct, we can perform matrix multiplication modulo 2:

result = (H @ G.T) % 2

The result should be a matrix of all zeros, confirming that H and G are correctly paired.

To find all 16 codewords, we can manually encode all possible 4-bit vectors. For example, let’s start with the following 4-bit messages:

0000

0001

0010

0011

0100

0101

0110

0111

These are 8 of the codewords. To generate the remaining codewords, prepend a 1 to each of the above messages:

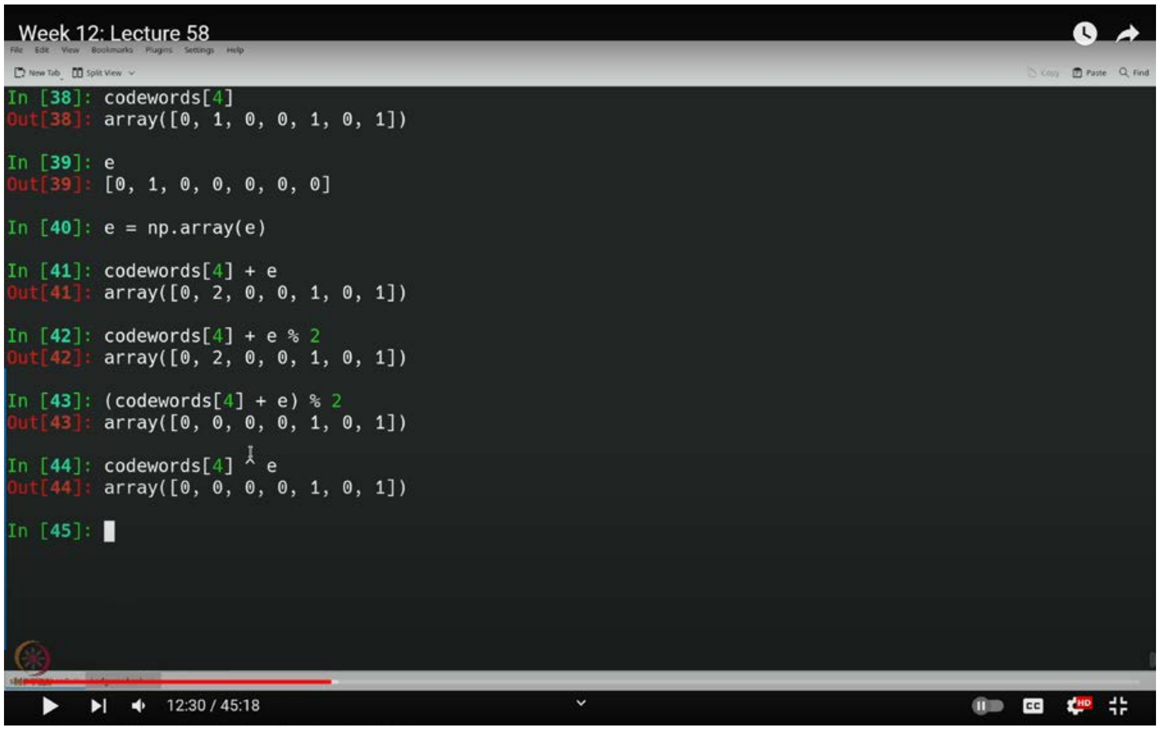
codewords = [np.concatenate(([1], m)) for m in messages]

Here, **`messages`** contains all 4-bit vectors, and this list comprehension prepends a 1 to each vector. By adding these new vectors to the original list, we obtain all 16 binary codewords for the (7,4) Hamming code.

In fact, there are more efficient methods to achieve this, but I am sticking to this approach for simplicity. To generate the codewords, I will use a list comprehension. Essentially, I will multiply each message vector m by the transpose of G. Here’s how you can do it:

codewords = [i @ G.T % 2 for i in m]

(Refer Slide Time: 12:30)



```
In [38]: codewords[4]
Out[38]: array([0, 1, 0, 0, 1, 0, 1])

In [39]: e
Out[39]: [0, 1, 0, 0, 0, 0, 0]

In [40]: e = np.array(e)

In [41]: codewords[4] + e
Out[41]: array([0, 2, 0, 0, 1, 0, 1])

In [42]: codewords[4] + e % 2
Out[42]: array([0, 2, 0, 0, 1, 0, 1])

In [43]: (codewords[4] + e) % 2
Out[43]: array([0, 0, 0, 0, 1, 0, 1])

In [44]: codewords[4] ^ e
Out[44]: array([0, 0, 0, 0, 1, 0, 1])

In [45]:
```

Here, **`i @ G.T`** represents matrix multiplication of each vector i in m with G transposed, and **`% 2`** ensures we perform modulo 2 operations. This generates all 16 codewords. If you notice any discrepancies, it's likely because the modulo 2 operation was not applied, so ensure you use **`% 2`** to correct this.

To verify that these are indeed valid Hamming codewords, we can use the parity check matrix H. By checking:

[H @ i % 2 for i in codewords]

we should confirm that all results are zero, indicating that these are valid codewords.

Next, observe that any two codewords differ in at least 3 bits. For example, if you compare the first codeword with the second, they differ in 4 bits. This distance property ensures that the code is robust against errors.

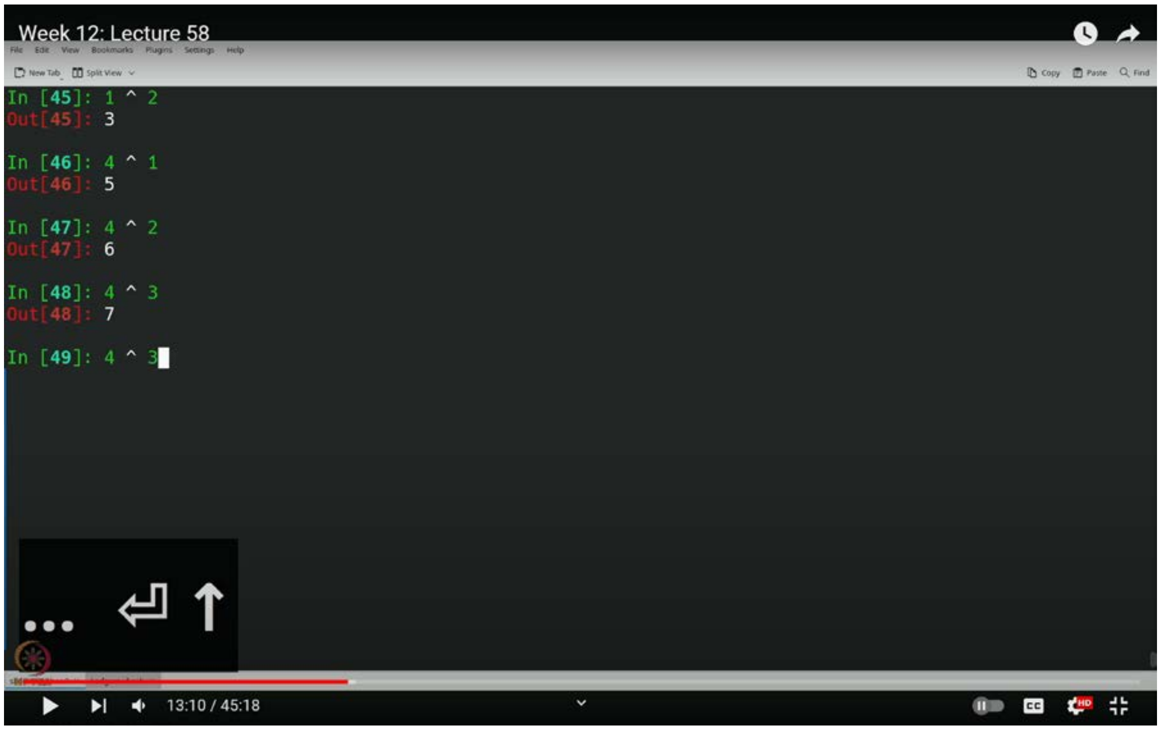
Now, if you need to implement this in GNU Radio, you will be performing XOR operations. One method is to add the vectors and take modulo 2, but a more straightforward approach is to use the XOR operator directly.

For instance, let’s consider an error vector **`010000`**. If we calculate:

syndrome = H @ e % 2

we obtain the error syndrome. To correct errors and find the correct codeword, you would XOR this error vector with the codeword. Instead of manually performing XOR and modulo operations, you can use Python’s built-in XOR operator (`^`), which simplifies the process.

(Refer Slide Time: 13:10)



```
In [45]: 1 ^ 2
Out[45]: 3

In [46]: 4 ^ 1
Out[46]: 5

In [47]: 4 ^ 2
Out[47]: 6

In [48]: 4 ^ 3
Out[48]: 7

In [49]: 4 ^ 3
```

For example:

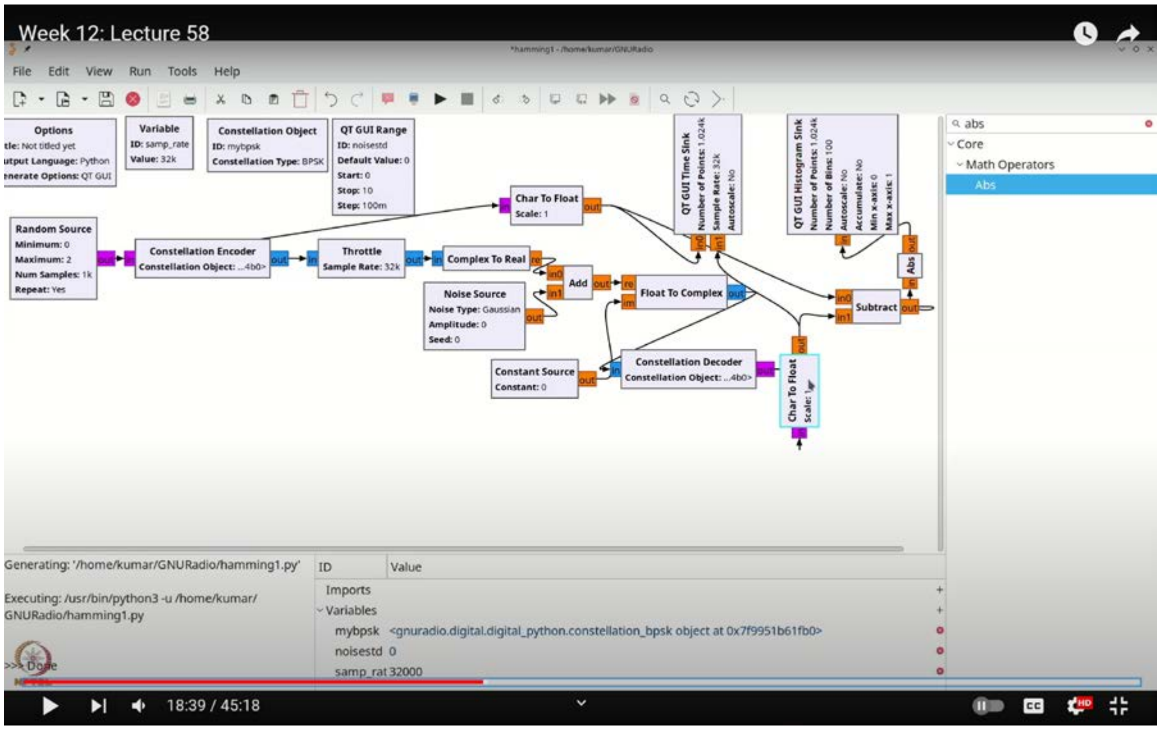
- **`1 ^ 2`** results in **`3`** because in binary, **`1`** is **`0001`** and **`2`** is **`0010`**, so their XOR gives **`0011`**.
- **`4 ^ 1`** results in **`5`** because **`4`** is **`0100`** and **`1`** is **`0001`**, so their XOR gives

`0101`.

- **`4 ^ 2`** results in **`6`** and **`4 ^ 3`** results in **`7`**, while **`4 ^ 4`** results in **`0`**.

We will use the XOR operator for our GNU Radio implementation. The approach will involve constructing a binary symmetric channel using a Gaussian channel model and applying the Hamming code in parallel. This will demonstrate improved error performance compared to a standard channel without error correction.

(Refer Slide Time: 18:39)



Options
Title: Not titled yet
Output Language: Python
Generate Options: QT GUI

Variable
ID: samp_rate
Value: 32k

Constellation Object
ID: mybpsk
Constellation Type: BPSK

QT GUI Range
ID: noisestd
Default Value: 0
Start: 0
Stop: 10
Step: 100m

Random Source
Minimum: 0
Maximum: 2
Num Samples: 1k
Repeat: Yes

Constellation Encoder
Constellation Object: ...4b0>

Throttle
Sample Rate: 32k

Char To Float
Scale: 1

Noise Source
Noise Type: Gaussian
Amplitude: 0
Seed: 0

Constant Source
Constant: 0

Constellation Decoder
Constellation Object: ...4b0>

QT GUI Time Sink
Number of Points: 1.024k
Sample Rate: 32k
Autoscale: No

QT GUI Histogram Sink
Number of Points: 1.024k
Number of Bins: 100
Autoscale: No
Accumulate: No
Min x-axis: 0
Max x-axis: 1

```
Generating: '/home/kumar/GNURadio/hamming1.py'

Executing: /usr/bin/python3 -u /home/kumar/
GNURadio/hamming1.py

>>> Done
```

ID	Value
Imports	
Variables	
mybpsk	<gnuradio.digital.digital_python.constellation_bpsk object at 0x7f9951b61fb0>
noisestd	0
samp_rat	32000

Let's put together the components for our system. We'll start with a random source. Since we are using Binary Phase Shift Keying (BPSK), a binary source is sufficient. We'll need to configure a few essential elements: a BPSK constellation object, a constellation encoder, and a constellation decoder. We'll set up these components as follows:

1. BPSK Constellation Object: We'll name this **`myBPSK`** and configure it for BPSK modulation.

2. Constellation Encoder: We'll use the **`myBPSK`** constellation object for encoding.

3. Constellation Decoder: We'll also use the **`myBPSK`** constellation object for decoding.

Next, we'll connect these components. To manage the flow of data, we'll add a throttle block to control the rate.

After that, we will add the following blocks:

- Complex to Real Converter: To convert complex numbers to real numbers, we'll use the "complex to real" block.
- Noise Source: We'll include a noise source to simulate real-world conditions. This will be a real noise source with adjustable amplitude and noise standard deviation. We’ll set the range for noise from 0 to 10, with a step of 0.1.
- Adder: We'll use an adder block to combine the noisy signal with the original signal.

To prepare the signal for decoding:

- Float to Complex Converter: We'll convert the float values back to complex numbers. The real part will be taken from the signal, and the imaginary part will be set to 0 using a constant source.

We’ll visualize the results with:

- Time Sink: Add a time sink to observe the signals. The time sink will have two inputs, and we’ll configure it to display float values. Additionally, we will include a "char to float" block to convert and view the bytes as float values.

To visualize errors:

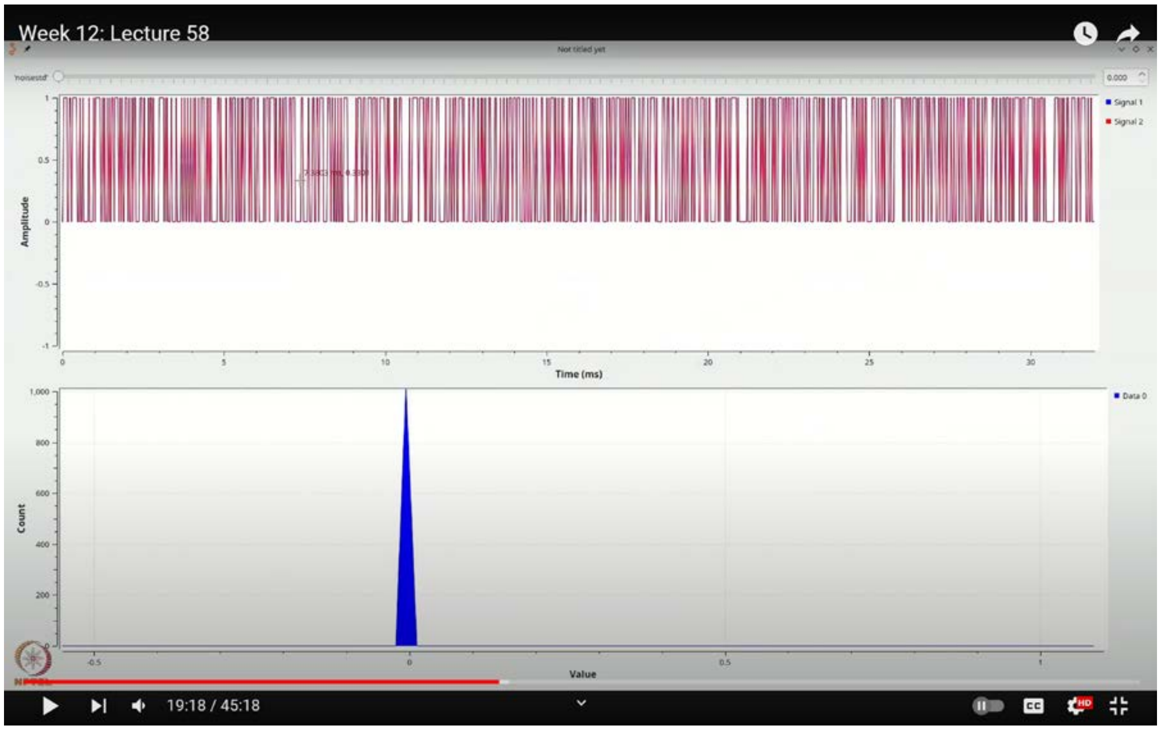
1. Histogram Sink: Add a histogram sink to observe the distribution of errors. We’ll use the subtract block to find the difference between the expected and actual values, and the absolute value block to get the magnitude of errors.

2. Layout Adjustments: Rotate and arrange the blocks for clarity. This will help in visualizing the error patterns effectively.

Finally, run the flow graph. Initially, with no noise, you should see no errors. As you increase the noise level, the blue and red signals will start to diverge, indicating errors. Adjust the histogram range and scale to better visualize the errors. Adding an absolute value block will help in displaying the magnitude of the errors.

By adjusting these settings, you’ll be able to clearly see and analyze the error patterns in your BPSK system.

(Refer Slide Time: 19:18)



As you can see, errors are appearing in the results. To address this, we'll need to adjust the settings. Ensure that the accumulation option is turned offset it to "no." Adding a grid to the visualization will help us better understand the error distribution. Now, we can effectively visualize errors by comparing the number of detected errors with the actual number of correct bits.

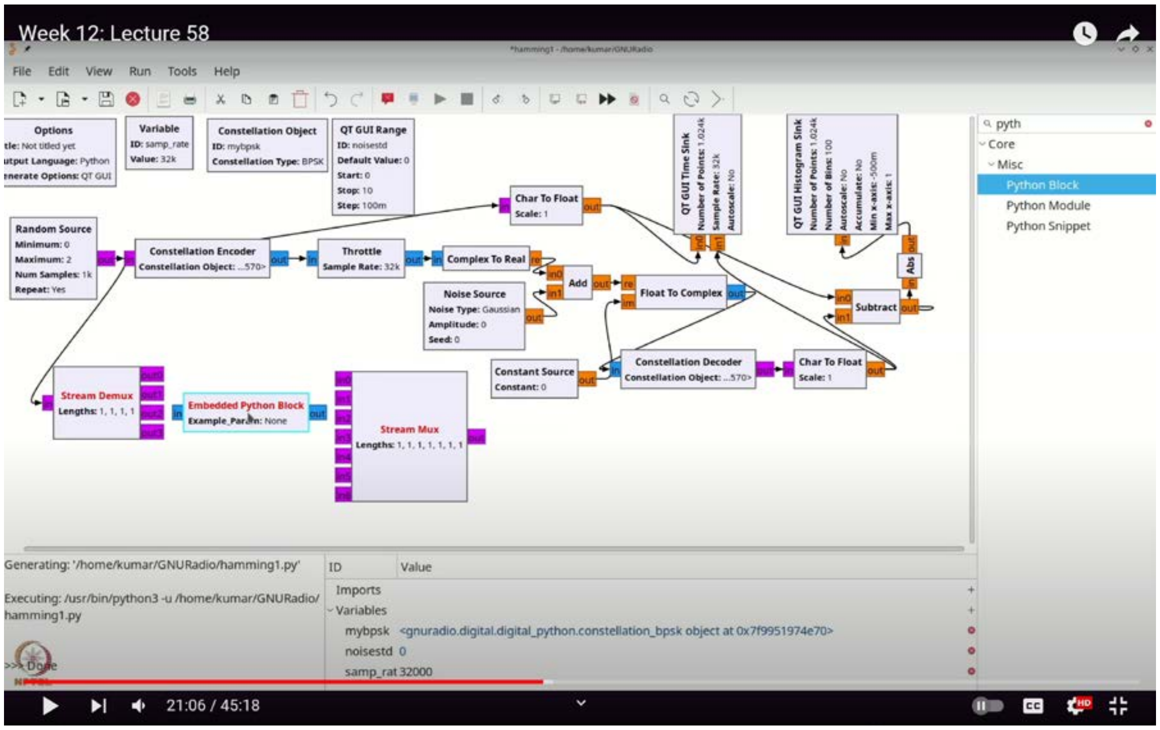
We have our base setup complete. Next, we need to incorporate a Hamming code into our flow graph. To do this, we'll modify the existing setup to accommodate Hamming encoding. We will continue using the random source but will replicate several components to manage the Hamming code processing.

Here the plan is:

1. Random Source: We will use this as before but now need to handle multiple bits for Hamming encoding.

2. Hamming Encoding: We'll use a Python block to perform the Hamming encoding. This block will take in 4 bits at a time and output 7 bits.

(Refer Slide Time: 21:06)



```
Generating: '/home/kumar/GNURadio/hamming1.py'

Executing: /usr/bin/python3 -u /home/kumar/GNURadio/
hamming1.py

>>> Done
```

ID	Value
Imports	
Variables	
mybpsk	<gnuradio.digital.digital_python.constellation_bpsk object at 0x7f9951974e70>
noisestd	0
samp_rat	32000

To achieve this:

- Demultiplexer: First, we need to demultiplex the data stream to handle 4-bit chunks. Press **`Ctrl+F`** or **`Command+F`** and search for "stream demux." Add a stream demux block and configure it to split the stream into 4-bit segments. Set the data type to byte.
- Multiplexer: We’ll also need a stream mux block to combine the encoded 7-bit segments back into a single stream. Similarly, press **`Ctrl+F`** or **`Command+F`**, search for "stream mux," and add this block. Configure it to handle 7-bit segments

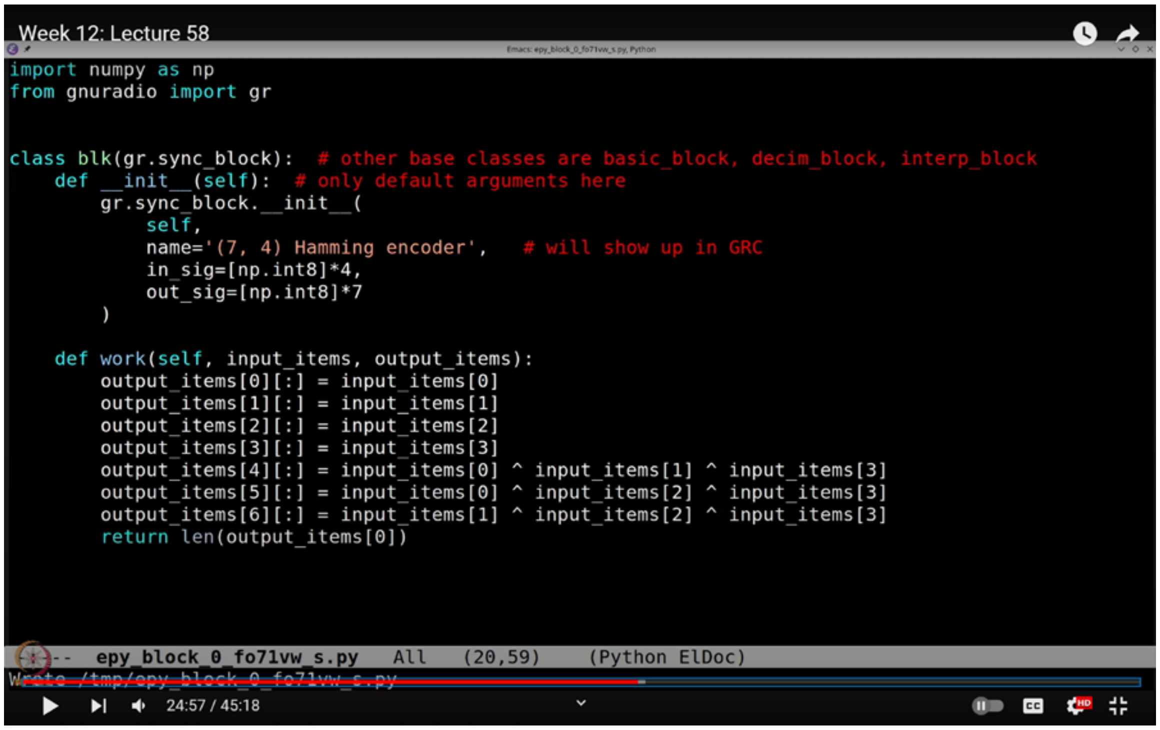
with byte data type.

- Hamming Encoder: Implement the Hamming encoder as a Python block. Press **`Ctrl+F`** or **`Command+F`**, search for "Python," and add the Python block. Double-click to open the editor and choose your preferred Python editor for coding. Remove any unnecessary content to focus on your Hamming encoding implementation. You don't need the example parameter provided.

With these adjustments, you'll be able to integrate the Hamming code into your flow graph effectively and analyze the performance.

We don't need this section. We'll name our block "7 for Hamming Encoder." This block will accept 4 **`int8`** inputs and produce 7 **`int8`** outputs, and we won't be using any example parameters. We will set the input items as 0, 1, 2, and 3, and the output items as 0 through 6. Our goal is to perform XOR operations based on these inputs.

(Refer Slide Time: 24:57)



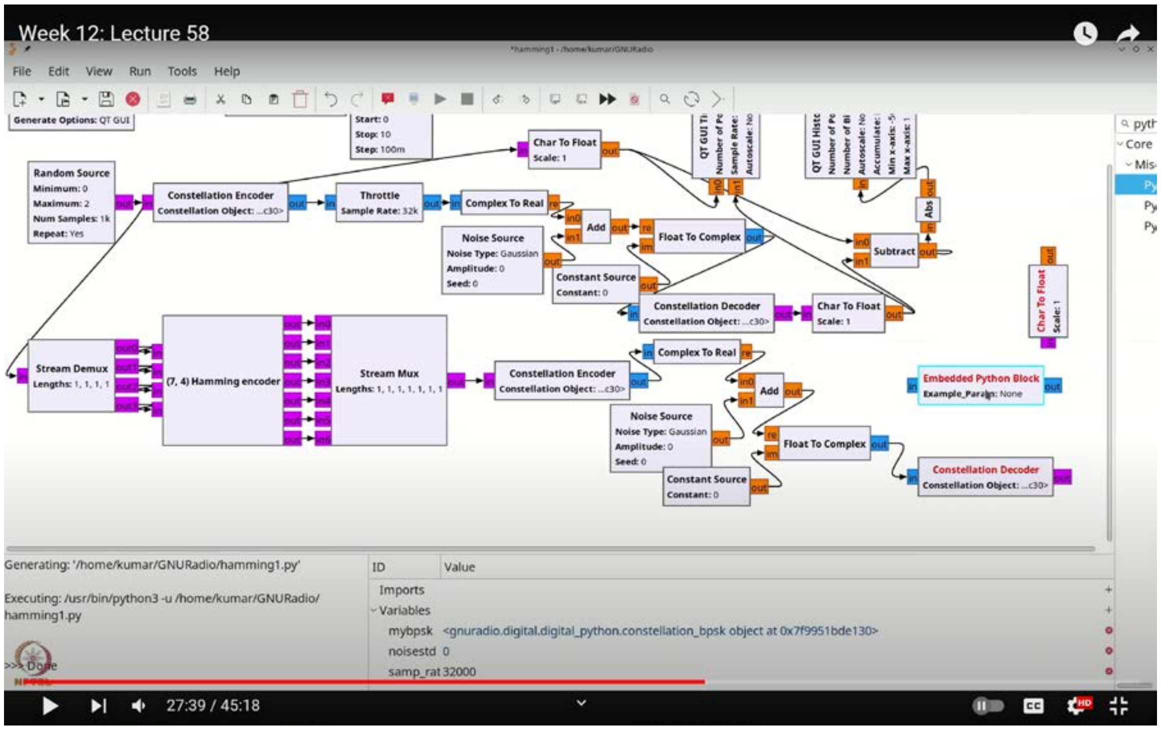
```
import numpy as np
from gnuradio import gr


class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    def __init__(self):  # only default arguments here
        gr.sync_block.__init__(
            self,
            name='(7, 4) Hamming encoder',   # will show up in GRC
            in_sig=[np.int8]*4,
            out_sig=[np.int8]*7
        )

    def work(self, input_items, output_items):
        output_items[0][:] = input_items[0]
        output_items[1][:] = input_items[1]
        output_items[2][:] = input_items[2]
        output_items[3][:] = input_items[3]
        output_items[4][:] = input_items[0] ^ input_items[1] ^ input_items[3]
        output_items[5][:] = input_items[0] ^ input_items[2] ^ input_items[3]
        output_items[6][:] = input_items[1] ^ input_items[2] ^ input_items[3]
        return len(output_items[0])

--  epy_block_0_fo71vw_s.py   All   (20,59)    (Python ElDoc)
Wrote /tmp/epy_block_0_fo71vw_s.py
```

To understand how to perform these XOR operations, let's look at the generator matrix. It is clear that the first 4 output bits are directly the same as the input bits because they appear unchanged. The fifth bit is calculated by XORing the first, second, and fourth bits from the input. Similarly, the sixth bit is computed by XORing the first, third, and fourth bits, and the seventh bit results from XORing the second, third, and fourth bits.

Now, let’s implement this using the XOR operator in the GNU Radio Python block. For the first 4 bits, we simply map them directly from the inputs. For the remaining bits, we will use the XOR operations:

- Output item 4 (the fifth bit) is computed as: **`input[0] XOR input[2] XOR input[3]`**
- Output item 5 (the sixth bit) is computed as: **`input[0] XOR input[1] XOR input[3]`**
- Output item 6 (the seventh bit) is computed as: **`input[1] XOR input[2] XOR input[3]`**

(Refer Slide Time: 27:39)



```
Generating: '/home/kumar/GNURadio/hamming1.py'

Executing: /usr/bin/python3 -u /home/kumar/GNURadio/
hamming1.py

>>> Done
```

ID	Value
Imports	
Variables	
mybpsk	<gnuradio.digital.digital_python.constellation_bpsk object at 0x7f9951bde130>
noisestd	0
samp_rat	32000

You can verify that this implementation matches the XOR operations specified by the generator matrix. Ensure that these operations correspond correctly to the expected patterns, like 1101, 1011, and 0111.

With this, we can save and exit the Python block editor. Our Hamming encoder is now ready. While we could have implemented this as a matrix operation, this approach provides a clearer instructional example.

Next, we'll need to connect the stream and multiplex it back for transmission. To do this, we’ll copy and paste the components required for modulation and noise addition:

1. Copy the Encoder: **`Ctrl+C`**, **`Ctrl+V`** to duplicate the Hamming encoder.

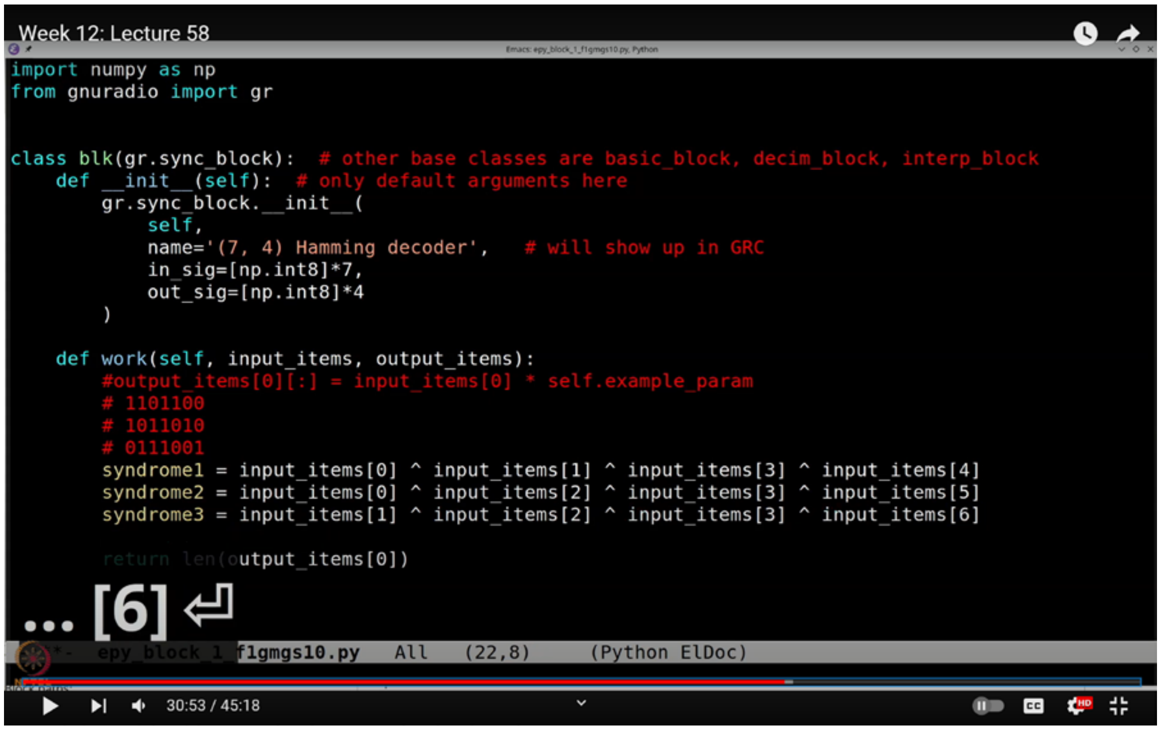
2. Add Complex to Real: **`Ctrl+C`**, **`Ctrl+V`** to duplicate the Complex to Real block.

3. Add Noise: **`Ctrl+C`**, **`Ctrl+V`** to duplicate the Noise block.

4. Add Adder: **`Ctrl+C`**, **`Ctrl+V`** to duplicate the Adder block.

5. Add Float to Complex: **`Ctrl+C`**, **`Ctrl+V`** to duplicate the Float to Complex block, along with the Constant Source.

(Refer Slide Time: 30:53)



```
import numpy as np
from gnuradio import gr


class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    def __init__(self):  # only default arguments here
        gr.sync_block.__init__(
            self,
            name='(7, 4) Hamming decoder',   # will show up in GRC
            in_sig=[np.int8]*7,
            out_sig=[np.int8]*4
        )

    def work(self, input_items, output_items):
        #output_items[0][:] = input_items[0] * self.example_param
        # 1101100
        # 1011010
        # 0111001
        syndrome1 = input_items[0] ^ input_items[1] ^ input_items[3] ^ input_items[4]
        syndrome2 = input_items[0] ^ input_items[2] ^ input_items[3] ^ input_items[5]
        syndrome3 = input_items[1] ^ input_items[2] ^ input_items[3] ^ input_items[6]

        return len(output_items[0])
```

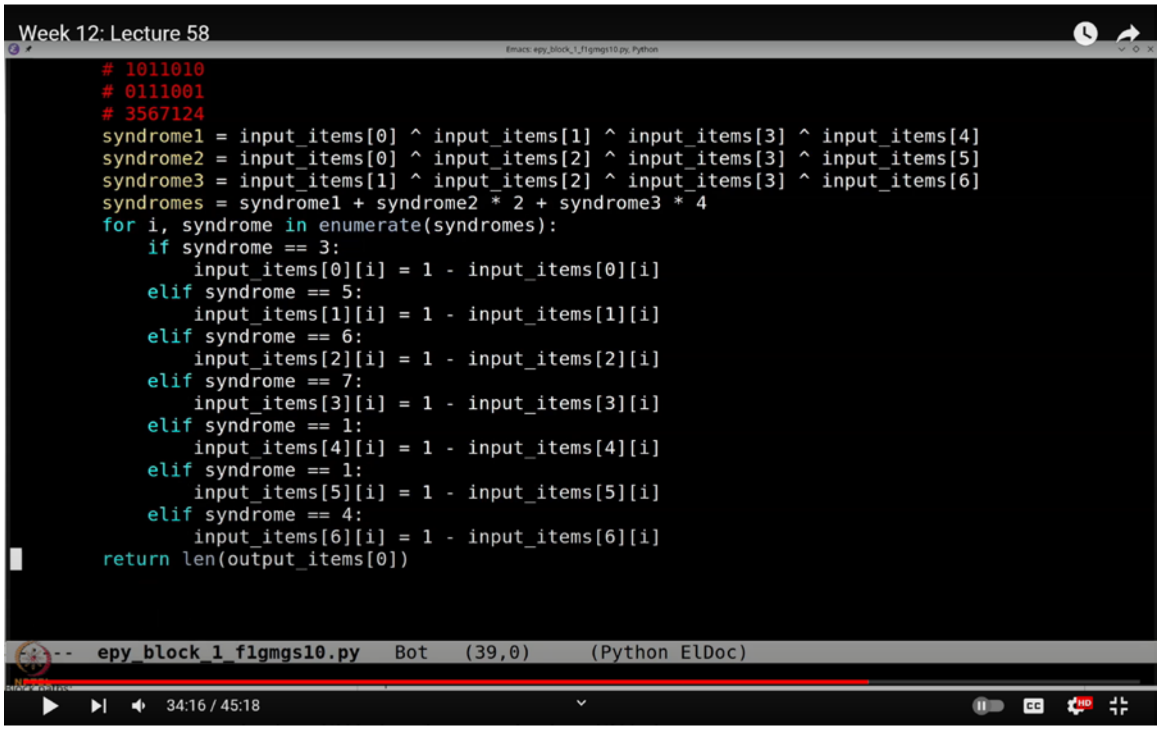
```
-:**-  epy_block_1_f1gmgs10.py   All   (22,8)      (Python ElDoc)
```

Connect these blocks accordingly and arrange them neatly to improve the layout. Position the stream demux, Hamming encoder, and stream mux blocks effectively to streamline the flow.

Let's organize our setup for decoding. First, we'll position the noise source, complex-to-real conversion, and constant source. After that, we need to add the constellation decoder and adjust the layout to create some space for this component. We'll rotate this decoder to the left for better alignment.

Before we proceed with converting characters to floats, we need to implement Hamming decoding. For this, we'll use a syndrome-based decoder utilizing the parity check matrix we discussed earlier. By examining the syndrome and identifying its corresponding index in the parity check matrix, we can pinpoint the location of errors and perform the necessary corrections.

(Refer Slide Time: 34:16)



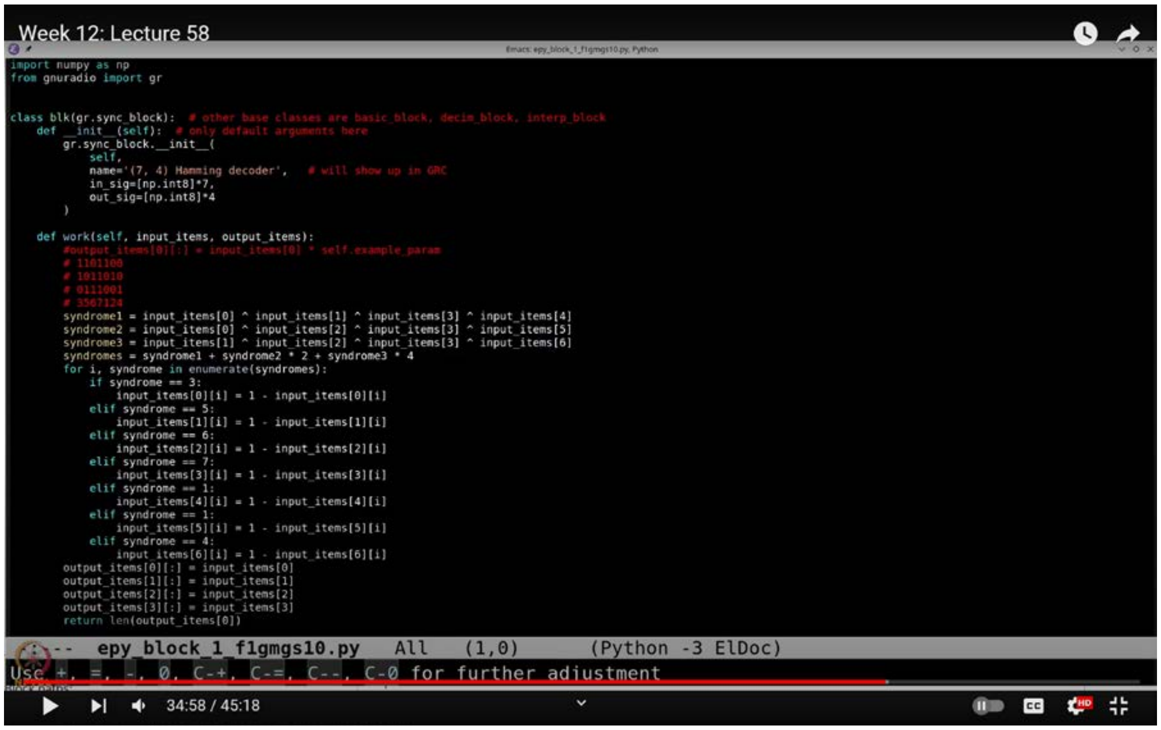
```
        # 1011010
        # 0111001
        # 3567124
        syndrome1 = input_items[0] ^ input_items[1] ^ input_items[3] ^ input_items[4]
        syndrome2 = input_items[0] ^ input_items[2] ^ input_items[3] ^ input_items[5]
        syndrome3 = input_items[1] ^ input_items[2] ^ input_items[3] ^ input_items[6]
        syndromes = syndrome1 + syndrome2 * 2 + syndrome3 * 4
        for i, syndrome in enumerate(syndromes):
            if syndrome == 3:
                input_items[0][i] = 1 - input_items[0][i]
            elif syndrome == 5:
                input_items[1][i] = 1 - input_items[1][i]
            elif syndrome == 6:
                input_items[2][i] = 1 - input_items[2][i]
            elif syndrome == 7:
                input_items[3][i] = 1 - input_items[3][i]
            elif syndrome == 1:
                input_items[4][i] = 1 - input_items[4][i]
            elif syndrome == 1:
                input_items[5][i] = 1 - input_items[5][i]
            elif syndrome == 4:
                input_items[6][i] = 1 - input_items[6][i]
        return len(output_items[0])
```

```
-:--  epy_block_1_f1gmgs10.py   Bot   (39,0)     (Python ElDoc)
```

We'll begin by setting up a new Python block for Hamming decoding. This block will be configured with 7 inputs and 4 outputs, reflecting the 7,4 Hamming code structure. We don't need the example parameters, so we will remove those. Our main task is to calculate the syndrome and flip the appropriate bit. Given that our generator matrix places the original 4-bit sequence in the first part, correcting the identified bit allows us to directly read the decoded data from the first 4 bits.

Let’s outline the decoding process:

(Refer Slide Time: 34:58)



```
import numpy as np
from gnuradio import gr


class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    def __init__(self):  # only default arguments here
        gr.sync_block.__init__(
            self,
            name='(7, 4) Hamming decoder',   # will show up in GRC
            in_sig=[np.int8]*7,
            out_sig=[np.int8]*4
        )

    def work(self, input_items, output_items):
        #output_items[0][:] = input_items[0] * self.example_param
        # 1101100
        # 1011010
        # 0111001
        # 3567124
        syndrome1 = input_items[0] ^ input_items[1] ^ input_items[3] ^ input_items[4]
        syndrome2 = input_items[0] ^ input_items[2] ^ input_items[3] ^ input_items[5]
        syndrome3 = input_items[1] ^ input_items[2] ^ input_items[3] ^ input_items[6]
        syndromes = syndrome1 + syndrome2 * 2 + syndrome3 * 4
        for i, syndrome in enumerate(syndromes):
            if syndrome == 3:
                input_items[0][i] = 1 - input_items[0][i]
            elif syndrome == 5:
                input_items[1][i] = 1 - input_items[1][i]
            elif syndrome == 6:
                input_items[2][i] = 1 - input_items[2][i]
            elif syndrome == 7:
                input_items[3][i] = 1 - input_items[3][i]
            elif syndrome == 1:
                input_items[4][i] = 1 - input_items[4][i]
            elif syndrome == 1:
                input_items[5][i] = 1 - input_items[5][i]
            elif syndrome == 4:
                input_items[6][i] = 1 - input_items[6][i]
        output_items[0][:] = input_items[0]
        output_items[1][:] = input_items[1]
        output_items[2][:] = input_items[2]
        output_items[3][:] = input_items[3]
        return len(output_items[0])
```

```
-:---  epy_block_1_f1gmgs10.py   All   (1,0)      (Python -3 ElDoc)
Use +, =, -, 0, C-+, C-=, C--, C-0 for further adjustment
```

1. Determine the Syndromes: We'll calculate three syndromes based on XOR operations involving the received inputs. For clarity, we'll reference the Hamming parity check matrix as follows:

1101100

1011010

0111001

We will use this matrix to compute the syndromes:

- Syndrome 1: XOR input items according to the first row of the parity matrix. For example, **`syndrome1 = input[0] XOR input[1] XOR input[3] XOR input[4]`**.
- Syndrome 2: XOR input items based on the second row of the matrix, such as **`syndrome2 = input[0] XOR input[1] XOR input[2] XOR input[5]`**.
- Syndrome 3: XOR input items according to the third row, for instance, **`syndrome3 = input[1] XOR input[2] XOR input[3] XOR input[6]`**.

2. Calculate Overall Syndrome: Combine these syndromes into a single value where:

- Syndrome 1 represents the least significant bit,
- Syndrome 2 represents the middle bit,
- Syndrome 3 represents the most significant bit.

This combined syndrome is calculated as: **`syndrome = syndrome1 + (syndrome2 * 2) + (syndrome3 * 4)`**.

For example:

- **`001`** results in 1,
- **`010`** results in 2,
- **`011`** results in 3,
- **`100`** results in 4,
- **`101`** results in 5,
- **`110`** results in 6,
- **`111`** results in 7.

Using these values, you can determine the specific bit to correct.

Now, we are ready to perform the bit corrections based on the computed syndromes.

I will now directly correct the input items based on the syndrome values. Here's the approach:

First, we'll loop through each syndrome value using the **`enumerate`** function. This provides both the index and the syndrome value. Based on the syndrome, we will perform the appropriate correction.

Let's start with a syndrome value of 3. If the syndrome value is 3, it indicates an error in the first bit. To correct this, we will flip the first bit. Specifically, if the syndrome is 3, we swap the value of input item 0 with its negation: **`input_items[0] = -input_items[0]`**.

Next, for a syndrome value of 5, the error is in the second bit. To flip the second bit, we simply change the second bit: **`input_items[1] = -input_items[1]`**.

Similarly, for a syndrome of 6, the error is in the third bit. We perform the correction by setting the third bit accordingly: **`input_items[2] = -input_items[2]`**.

For a syndrome value of 7, which indicates an error in the fourth bit, we will modify the fourth bit: **`input_items[3] = -input_items[3]`**.

We need to handle all these syndromes appropriately, but we'll just focus on the first four bits since they directly affect the message. Here's a summary:

- Syndrome 3 affects the first bit.
- Syndrome 5 affects the second bit.
- Syndrome 6 affects the third bit.
- Syndrome 7 affects the fourth bit.

If the syndrome is 0, no corrections are necessary, as it indicates no errors.

Finally, we'll copy the corrected values back to the outputs. For instance:

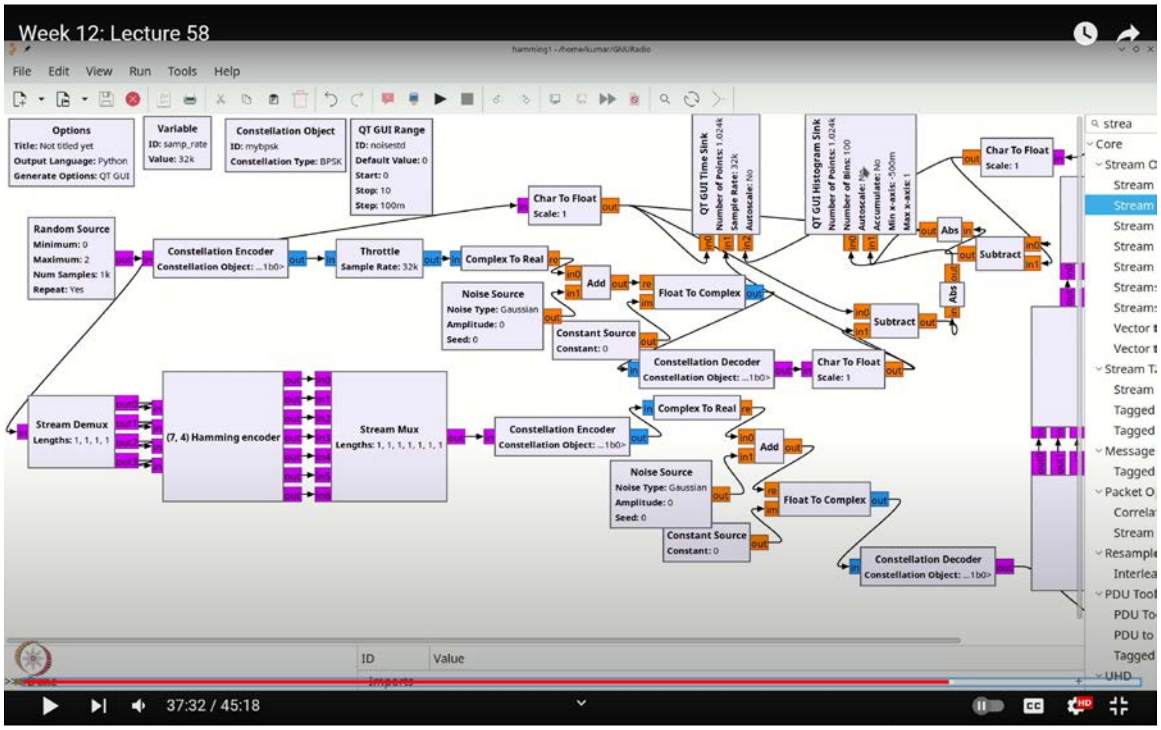
- **`output_item[0] = input_item[0]`**
- **`output_item[1] = input_item[1]`**

These outputs now represent the corrected message, with the first four bits corresponding to the actual data.

With our decoder configured and the corrections implemented, let's finalize this by setting up the flow graph. We need to split the constellation by using a stream demuxer. Use **`Ctrl+F`** or **`Cmd+F`** to find and add the stream demux block. Rotate the demux block as needed, and ensure it handles byte data with 7 inputs.

We’ll also need a stream mux block for combining the data. Add this block, ensuring it’s set to handle byte data with 1 input and 4 outputs. Connect the stream mux to the decoder, then use **`char to float`** and **`subtract`** blocks to process the output. Connect these to a time sink for visualization.

(Refer Slide Time: 37:32)


Week 12: Lecture 58
hamming1 - /home/kumar/GNURadio
File Edit View Run Tools Help
37:32 / 45:18

Options
Title: Not titled yet
Output Language: Python
Generate Options: QT GUI

Variable
ID: samp_rate
Value: 32k

Constellation Object
ID: mybpsk
Constellation Type: BPSK

QT GUI Range
ID: noisestd
Default Value: 0
Start: 0
Stop: 10
Step: 100m

Random Source
Minimum: 0
Maximum: 2
Num Samples: 1k
Repeat: Yes

Constellation Encoder
Constellation Object: ...1b0>

Throttle
Sample Rate: 32k

Complex To Real

Noise Source
Noise Type: Gaussian
Amplitude: 0
Seed: 0

Add

Constant Source
Constant: 0

Float To Complex

Constellation Decoder
Constellation Object: ...1b0>

Char To Float
Scale: 1

QT GUI Time Sink
Number of Points: 1.024k
Sample Rate: 32k
Autoscale: No

QT GUI Histogram Sink
Number of Points: 1.024k
Number of Bins: 100
Autoscale: No
Accumulate: No
Min x-axis: -500m
Max x-axis: 1

Abs

Subtract

Stream Demux
Lengths: 1, 1, 1, 1

(7, 4) Hamming encoder

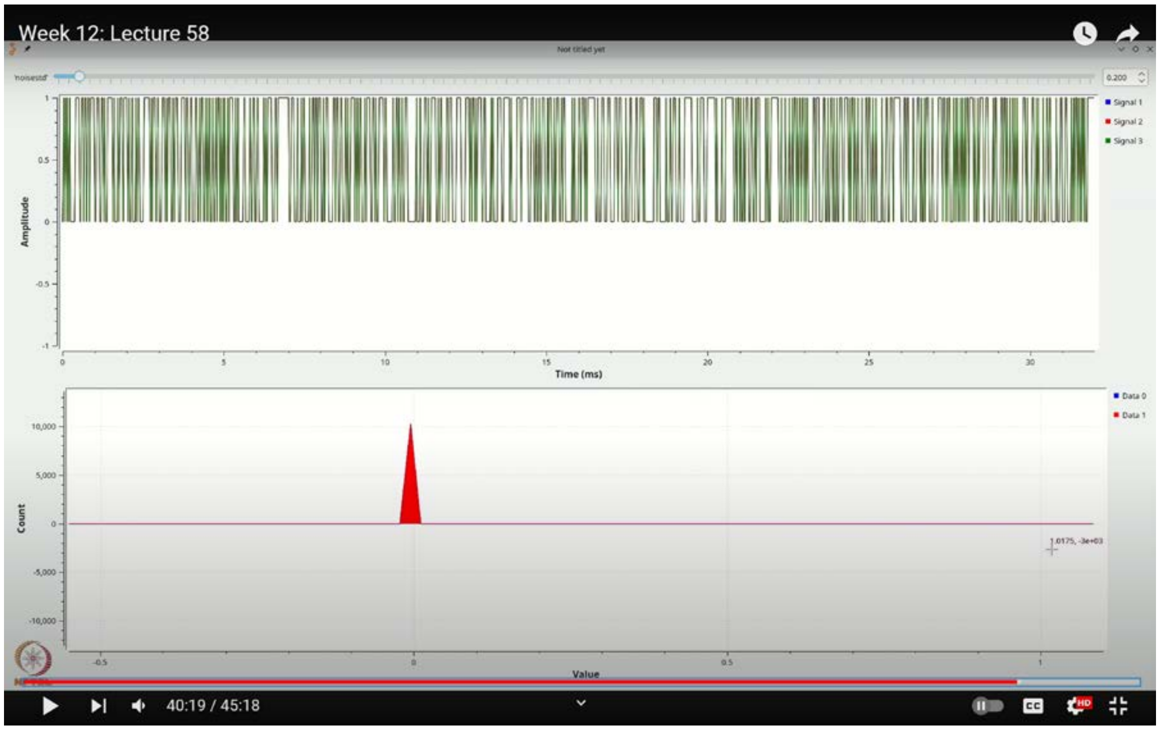
Stream Mux
Lengths: 1, 1, 1, 1, 1, 1, 1

Add a histogram sink to observe error correction, setting it to 2 inputs. Use **`Ctrl+C`** and **`Ctrl+V`** to duplicate and configure **`subtract`** and **`abs`** blocks. Connect these to compare the original data with the corrected output.

Execute the flow graph to verify error correction. To test its effectiveness, increase the histogram points and sample count, then evaluate the performance with varying levels of noise.

Let's proceed with integrating the absolute value block. First, copy and paste the block using **`Ctrl+C`** and **`Ctrl+V`**. Add the absolute value block here and connect it appropriately. With these adjustments made, we're ready to test the flow graph.

(Refer Slide Time: 40:19)



When you execute the flow graph, you should observe some level of error correction taking place. To get a clearer picture, increase the number of histogram points and samples, and introduce some noise into the system.

As you observe the results, you may notice that the blue trace starts to rise slightly, while the red trace remains a bit lower. This indicates that the system is successfully correcting single-bit errors. Adjusting the display might be necessary for the peak of the red trace to settle, which will show a lower peak compared to the blue trace, confirming that one-bit errors are being corrected.

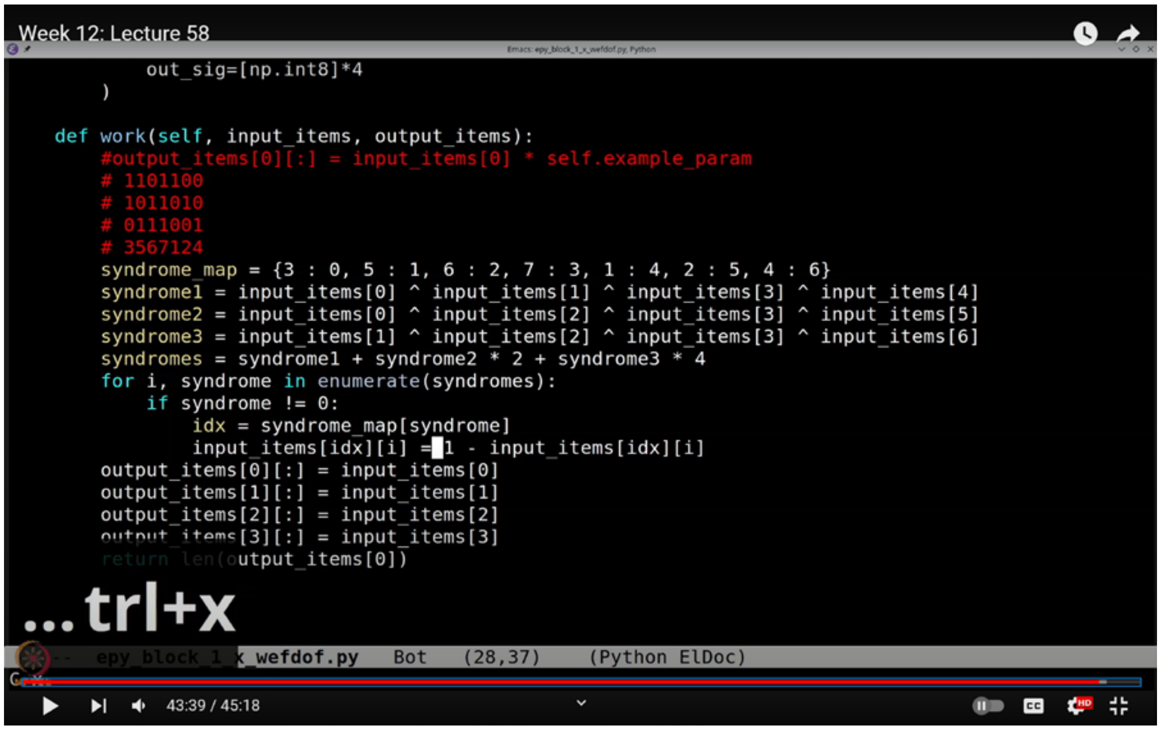
For further verification, you can manually introduce a bit flip to ensure that all one-bit errors are corrected. Here’s how to do it:

- Use **`Ctrl+F`** or **`Cmd+F`** to find and add a constant block.
- Set this constant block to byte format and configure it to add a value of -1 to the

bit.

- Multiply this by -1 to introduce a bit error.

(Refer Slide Time: 43:39)



```
            out_sig=[np.int8]*4
        )

    def work(self, input_items, output_items):
        #output_items[0][:] = input_items[0] * self.example_param
        # 1101100
        # 1011010
        # 0111001
        # 3567124
        syndrome_map = {3 : 0, 5 : 1, 6 : 2, 7 : 3, 1 : 4, 2 : 5, 4 : 6}
        syndrome1 = input_items[0] ^ input_items[1] ^ input_items[3] ^ input_items[4]
        syndrome2 = input_items[0] ^ input_items[2] ^ input_items[3] ^ input_items[5]
        syndrome3 = input_items[1] ^ input_items[2] ^ input_items[3] ^ input_items[6]
        syndromes = syndrome1 + syndrome2 * 2 + syndrome3 * 4
        for i, syndrome in enumerate(syndromes):
            if syndrome != 0:
                idx = syndrome_map[syndrome]
                input_items[idx][i] = 1 - input_items[idx][i]
        output_items[0][:] = input_items[0]
        output_items[1][:] = input_items[1]
        output_items[2][:] = input_items[2]
        output_items[3][:] = input_items[3]
        return len(output_items[0])

-- epy_block_1_x_wefdof.py   Bot   (28,37)    (Python ElDoc)
```

Alternatively, you can edit the Hamming encoder directly to introduce a bit error by flipping a bit in the encoding process. Execute the flow graph after making these changes.

If you find that the Hamming code correctly corrects single-bit errors even after introducing these errors, it confirms that the implementation is accurate. If you need to remove the manually introduced error, make sure to reset your setup.

This exercise demonstrates that the Hamming encoder and decoder can effectively correct one-bit errors even in the presence of noise. You can observe the high efficiency of error correction by waiting for the results to stabilize.

Finally, it's worth noting that while this implementation of the Hamming decoder is functional, it could be optimized. Currently, we manually check each syndrome and flip the corresponding bit. GNU Radio offers other advanced error coding options, such as

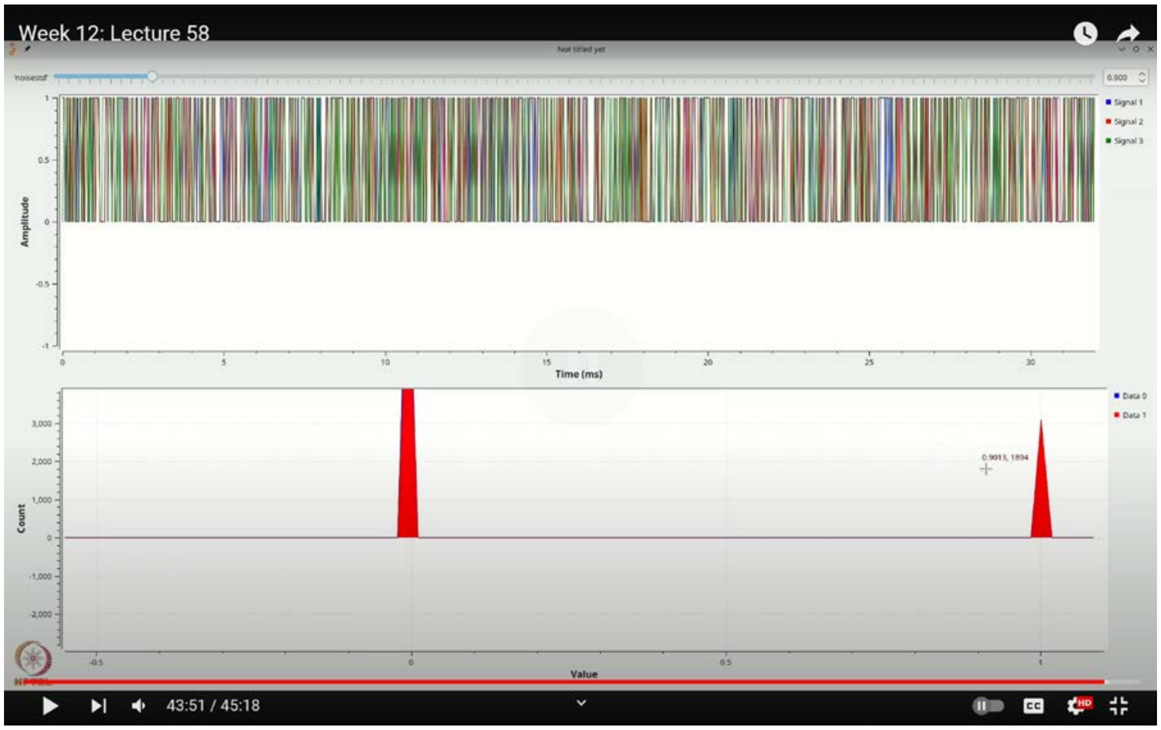
Reed-Solomon coding and polar codes. By exploring these options, you can leverage more sophisticated techniques for error correction. For instance, you can use **`Ctrl+F`** to search for "coding" to find various coding blocks like **`FECAsync decoder`**, **`FECAsync encoder`**, and more, which can be combined to achieve better results.

There is an alternative method that leverages Python for a more efficient implementation of error correction. Instead of manually checking and flipping bits for each syndrome, you can use a mapping approach. For instance, you can create a Python dictionary to map each syndrome value to the corresponding bit that needs to be flipped.

Consider this mapping: If the syndrome is 3, the bit to flip is 0. If the syndrome is 5, the bit to flip is 6. You can set up a dictionary like this:

syndromemap = { 3: 0, 5: 1, 6: 2, 7: 3, 1: 4, 2: 5, 4: 6}

(Refer Slide Time: 43:51)



With this dictionary, you no longer need multiple lines of code. Instead, you can simply use:

bit_to_flip = syndromemap.get(syndrome)

This streamlined approach means that if the syndrome is 3, **`syndromemap[3]`** automatically gives you 0, indicating that the 0th bit should be flipped. To make the code even clearer, you can use:

idx = syndromemap.get(syndrome)

This reduces the complexity of your code, making it more readable and maintainable by directly mapping each syndrome to the corresponding bit flip.

Additionally, remember to confirm that the syndrome is not 0. If the syndrome equals 0, no bit flip is needed. After implementing these changes, execute the flow graph again. You should observe the original performance, and by adjusting the noise levels, you will see that the Hamming code performs effectively.

In this lecture, we demonstrated one method to implement the binary Hamming code using GNU Radio's embedded Python block. This method offers a simple yet powerful approach for realizing the Hamming code. We verified the single-error correction capability of the Hamming code by deliberately introducing a bit error and confirming that the code could correct it. In future lectures, we will explore how error correction codes integrate into communication systems and how they can enhance overall performance. Thank you.