

Digital Communication using GNU Radio

Prof. Kumar Appiah

Department of Electrical Engineering

Indian Institute of Technology Bombay

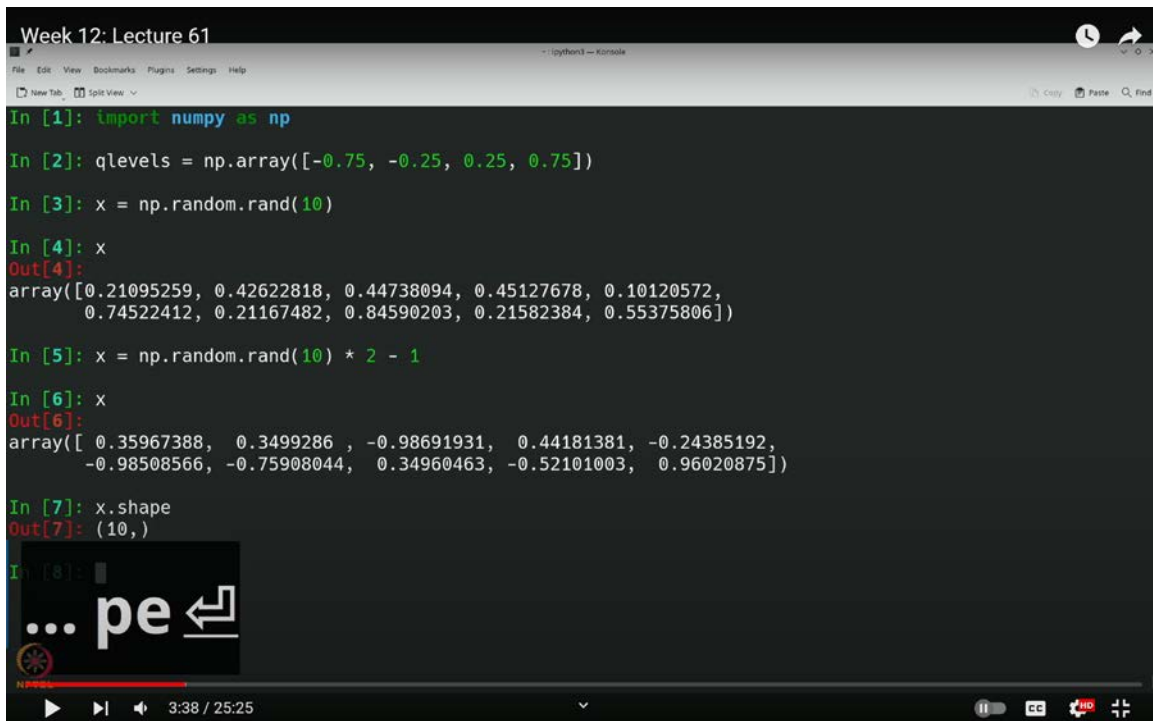
Week-12

Lecture-61

Visualising Quantisation in GNU Radio

Welcome to this lecture on Digital Communication Using GNU Radio. I'm Kumar Appiah from the Department of Electrical Engineering at IIT Bombay. In our previous lecture, we explored scalar quantization in detail. Specifically, we examined how to quantize a uniform random variable and a Gaussian random variable, analyzing the impact on mean squared error and how this relates to the choice of boundaries and quantization levels.

(Refer Slide Time: 03:38)



```
Week 12: Lecture 61
-- (python) -- Console

File Edit View Bookmarks Plugins Settings Help
New Tab Split View
Copy Paste Find

In [1]: import numpy as np
In [2]: qllevels = np.array([-0.75, -0.25, 0.25, 0.75])
In [3]: x = np.random.rand(10)
In [4]: x
Out[4]:
array([0.21095259, 0.42622818, 0.44738094, 0.45127678, 0.10120572,
       0.74522412, 0.21167482, 0.84590203, 0.21582384, 0.55375806])
In [5]: x = np.random.rand(10) * 2 - 1
In [6]: x
Out[6]:
array([ 0.35967388,  0.3499286 , -0.98691931,  0.44181381, -0.24385192,
       -0.98508566, -0.75908044,  0.34960463, -0.52101003,  0.96020875])
In [7]: x.shape
Out[7]: (10,)
In [8]:
```

In this lecture, we will implement these concepts in GNU Radio through simulation. We will write a small Python block that not only calculates the quantized values but also

computes the mean squared error based on the specified quantization levels.

Let's begin by implementing a simple quantization process in Python, which we can later embed into our GNU Radio block. First, we need to import the necessary library:

```
import numpy as np
```

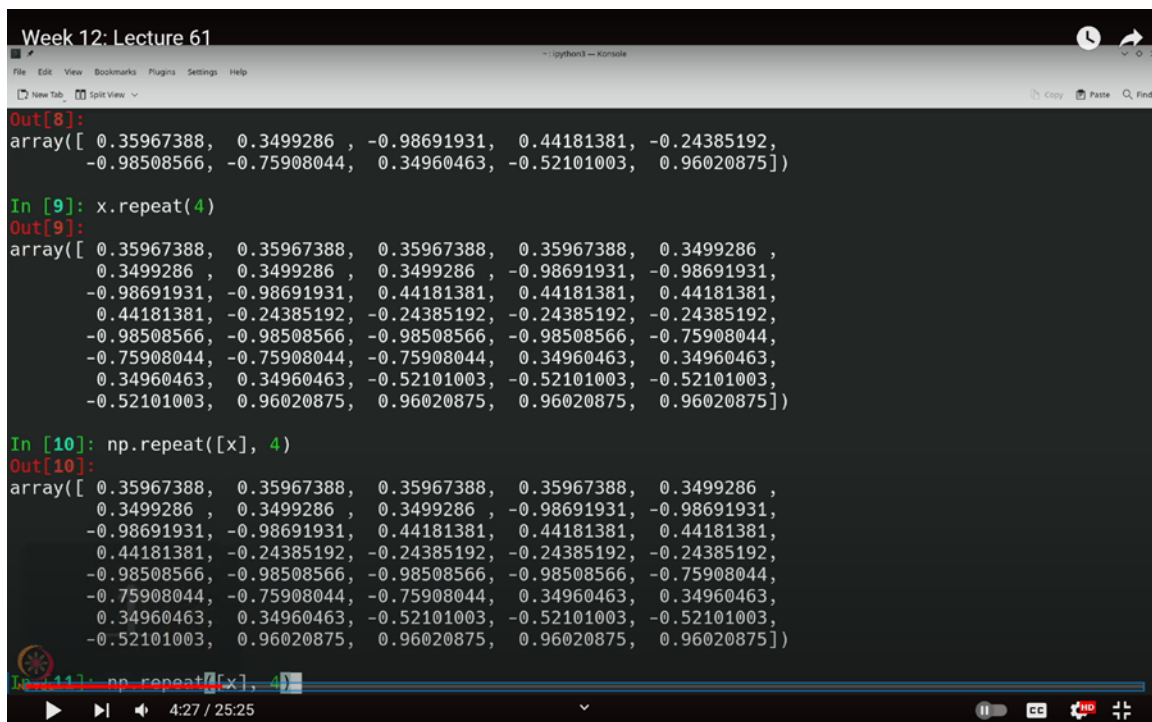
We will perform quantization using Python's and NumPy's built-in array features. To start, let's define our quantization levels. For example:

```
q_levels = np.array([-0.75, -0.25, 0.25, 0.75])
```

These levels are optimal for quantizing a uniform random variable ranging between -1 and 1. Next, let's generate 10 random values uniformly distributed between -1 and 1:

```
x = np.random.rand(10) * 2 - 1
```

(Refer Slide Time: 04:27)



```
Week 12: Lecture 61
Out[8]:
array([ 0.35967388,  0.3499286 , -0.98691931,  0.44181381, -0.24385192,
        -0.98508566, -0.75908044,  0.34960463, -0.52101003,  0.96020875])

In [9]: x.repeat(4)
Out[9]:
array([ 0.35967388,  0.35967388,  0.35967388,  0.35967388,  0.3499286 ,
        0.3499286 ,  0.3499286 ,  0.3499286 , -0.98691931, -0.98691931,
       -0.98691931, -0.98691931,  0.44181381,  0.44181381,  0.44181381,
        0.44181381, -0.24385192, -0.24385192, -0.24385192, -0.24385192,
       -0.98508566, -0.98508566, -0.98508566, -0.98508566, -0.75908044,
       -0.75908044, -0.75908044, -0.75908044,  0.34960463,  0.34960463,
        0.34960463,  0.34960463, -0.52101003, -0.52101003, -0.52101003,
       -0.52101003,  0.96020875,  0.96020875,  0.96020875,  0.96020875])

In [10]: np.repeat([x], 4)
Out[10]:
array([ 0.35967388,  0.35967388,  0.35967388,  0.35967388,  0.3499286 ,
        0.3499286 ,  0.3499286 ,  0.3499286 , -0.98691931, -0.98691931,
       -0.98691931, -0.98691931,  0.44181381,  0.44181381,  0.44181381,
        0.44181381, -0.24385192, -0.24385192, -0.24385192, -0.24385192,
       -0.98508566, -0.98508566, -0.98508566, -0.98508566, -0.75908044,
       -0.75908044, -0.75908044, -0.75908044,  0.34960463,  0.34960463,
        0.34960463,  0.34960463, -0.52101003, -0.52101003, -0.52101003,
       -0.52101003,  0.96020875,  0.96020875,  0.96020875,  0.96020875])

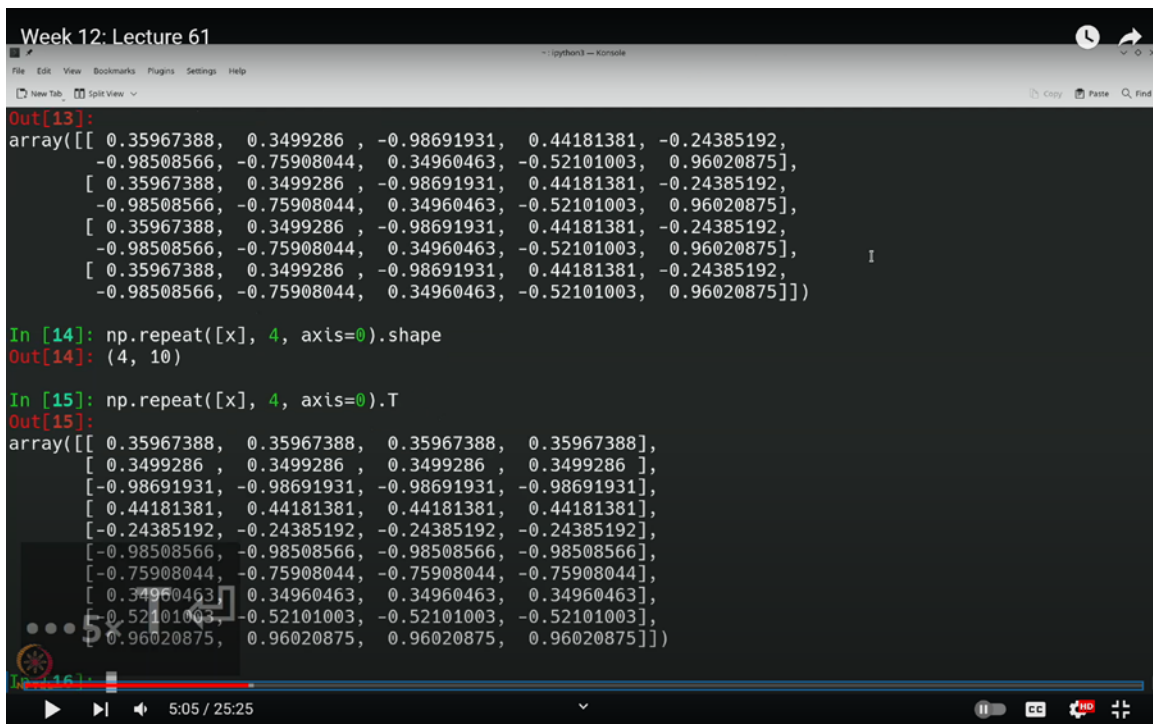
In [11]: np.repeat(x, 4)
```

The array `x` will contain 10 random values between -1 and 1. Our goal is to quantize each value to the nearest quantization level from `q_levels`.

For instance, consider the value 0.35. It is closest to 0.25, while -0.98 is obviously closest to -0.75 . One straightforward approach would be to loop through each value and compare it to each quantization level, selecting the minimum. However, a more efficient method utilizes NumPy's array features to accomplish this in one step.

Here's how we can do it: First, we will repeat the array `x` four times, matching the number of quantization levels. Simultaneously, we will repeat the quantization levels array ten times to align with each value in `x`. By performing a subtraction operation and then finding the minimum value in each row, we can achieve our goal.

(Refer Slide Time: 05:05)



```
Week 12: Lecture 61
Out[13]:
array([[ 0.35967388,  0.3499286 , -0.98691931,  0.44181381, -0.24385192,
        -0.98508566, -0.75908044,  0.34960463, -0.52101003,  0.96020875],
       [ 0.35967388,  0.3499286 , -0.98691931,  0.44181381, -0.24385192,
        -0.98508566, -0.75908044,  0.34960463, -0.52101003,  0.96020875],
       [ 0.35967388,  0.3499286 , -0.98691931,  0.44181381, -0.24385192,
        -0.98508566, -0.75908044,  0.34960463, -0.52101003,  0.96020875],
       [ 0.35967388,  0.3499286 , -0.98691931,  0.44181381, -0.24385192,
        -0.98508566, -0.75908044,  0.34960463, -0.52101003,  0.96020875]])

In [14]: np.repeat([x], 4, axis=0).shape
Out[14]: (4, 10)

In [15]: np.repeat([x], 4, axis=0).T
Out[15]:
array([[ 0.35967388,  0.35967388,  0.35967388,  0.35967388],
       [ 0.3499286 ,  0.3499286 ,  0.3499286 ,  0.3499286 ],
       [-0.98691931, -0.98691931, -0.98691931, -0.98691931],
       [ 0.44181381,  0.44181381,  0.44181381,  0.44181381],
       [-0.24385192, -0.24385192, -0.24385192, -0.24385192],
       [-0.98508566, -0.98508566, -0.98508566, -0.98508566],
       [-0.75908044, -0.75908044, -0.75908044, -0.75908044],
       [ 0.34960463,  0.34960463,  0.34960463,  0.34960463],
       [-0.52101003, -0.52101003, -0.52101003, -0.52101003],
       [ 0.96020875,  0.96020875,  0.96020875,  0.96020875]])
```

Let's break this down step-by-step. Suppose `x` is an array with shape `(10,)`. To repeat this array across four columns, we could use `x.repeat(4)`, but this method repeats the array linearly, which is not what we want. Instead, we'll use `np.repeat` with the `axis` parameter set to 0.

Here's how:

```
import numpy as np
```

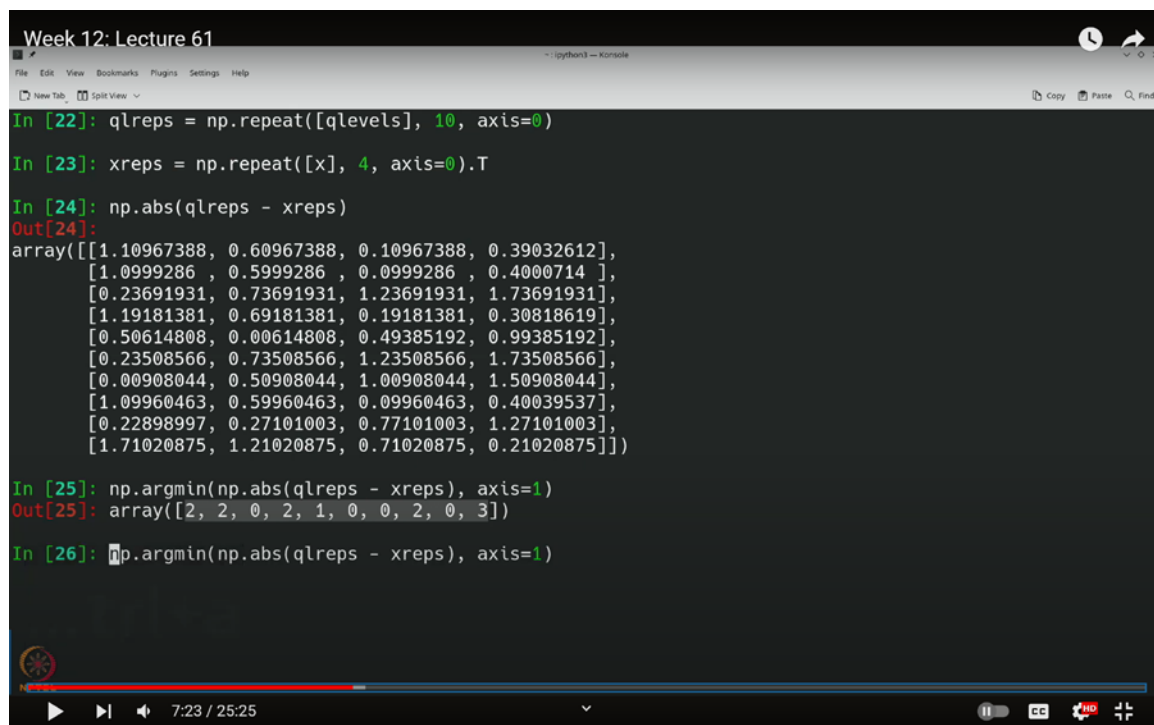
```
x = np.random.rand(10) * 2 - 1 # Example array of random values between -1 and 1
```

```
q_levels = np.array([-0.75, -0.25, 0.25, 0.75]) # Quantization levels
```

```
x_reps = np.repeat(x[:, np.newaxis], 4, axis=1) # Repeat x across columns
```

```
ql_reps = np.repeat(q_levels[np.newaxis, :], 10, axis=0) # Repeat q_levels across rows
```

(Refer Slide Time: 07:23)



```
Week 12: Lecture 61
In [22]: ql_reps = np.repeat([qlevels], 10, axis=0)
In [23]: x_reps = np.repeat([x], 4, axis=0).T
In [24]: np.abs(ql_reps - x_reps)
Out[24]:
array([[1.10967388, 0.60967388, 0.10967388, 0.39032612],
       [1.0999286 , 0.5999286 , 0.0999286 , 0.4000714 ],
       [0.23691931, 0.73691931, 1.23691931, 1.73691931],
       [1.19181381, 0.69181381, 0.19181381, 0.30818619],
       [0.50614808, 0.00614808, 0.49385192, 0.99385192],
       [0.23508566, 0.73508566, 1.23508566, 1.73508566],
       [0.00908044, 0.50908044, 1.00908044, 1.50908044],
       [1.09960463, 0.59960463, 0.09960463, 0.40039537],
       [0.22898997, 0.72101003, 1.22101003, 1.72101003],
       [1.71020875, 1.21020875, 0.71020875, 0.21020875]])
In [25]: np.argmin(np.abs(ql_reps - x_reps), axis=1)
Out[25]: array([2, 2, 0, 2, 1, 0, 0, 2, 0, 3])
In [26]: np.argmin(np.abs(ql_reps - x_reps), axis=1)
```

In this code, `x_reps` will have a shape of (10, 4), with each row containing the values of `x` repeated four times. The `ql_reps` will have a shape of (10, 4), with each column containing the quantization levels.

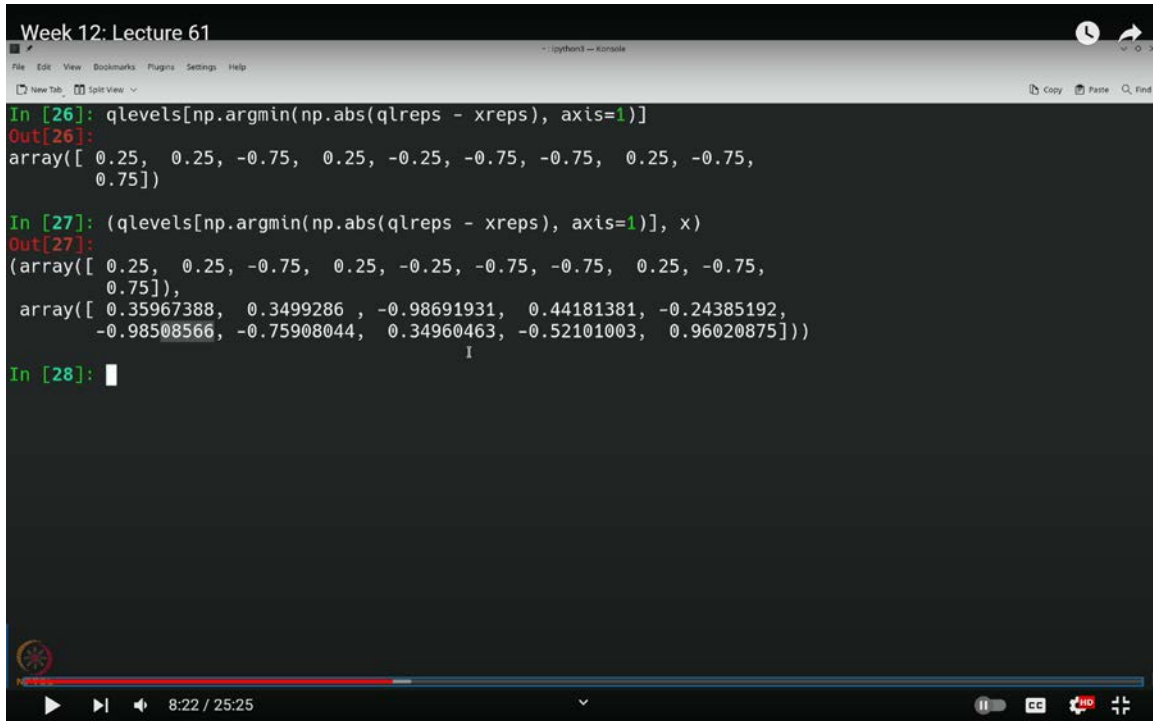
Next, subtract each quantization level from the corresponding value in `x_reps`, take the absolute value, and then find the minimum value across all columns to determine the closest quantization point:

```
diff = np.abs(ql_reps - x_reps) # Compute the absolute difference
```

```
min_indices = np.argmin(diff, axis=1) # Find the indices of the minimum values
```

```
quantized_values = q_levels[min_indices] # Map indices to quantization levels
```

(Refer Slide Time: 08:22)



```
Week 12: Lecture 61
In [26]: qlevels[np.argmin(np.abs(qlreps - xreps), axis=1)]
Out[26]:
array([ 0.25,  0.25, -0.75,  0.25, -0.25, -0.75, -0.75,  0.25, -0.75,
        0.75])

In [27]: (qlevels[np.argmin(np.abs(qlreps - xreps), axis=1)], x)
Out[27]:
(array([ 0.25,  0.25, -0.75,  0.25, -0.25, -0.75, -0.75,  0.25, -0.75,
        0.75]),
 array([ 0.35967388,  0.3499286 , -0.98691931,  0.44181381, -0.24385192,
        -0.98508566, -0.75908044,  0.34960463, -0.52101003,  0.96020875]))

In [28]:
```

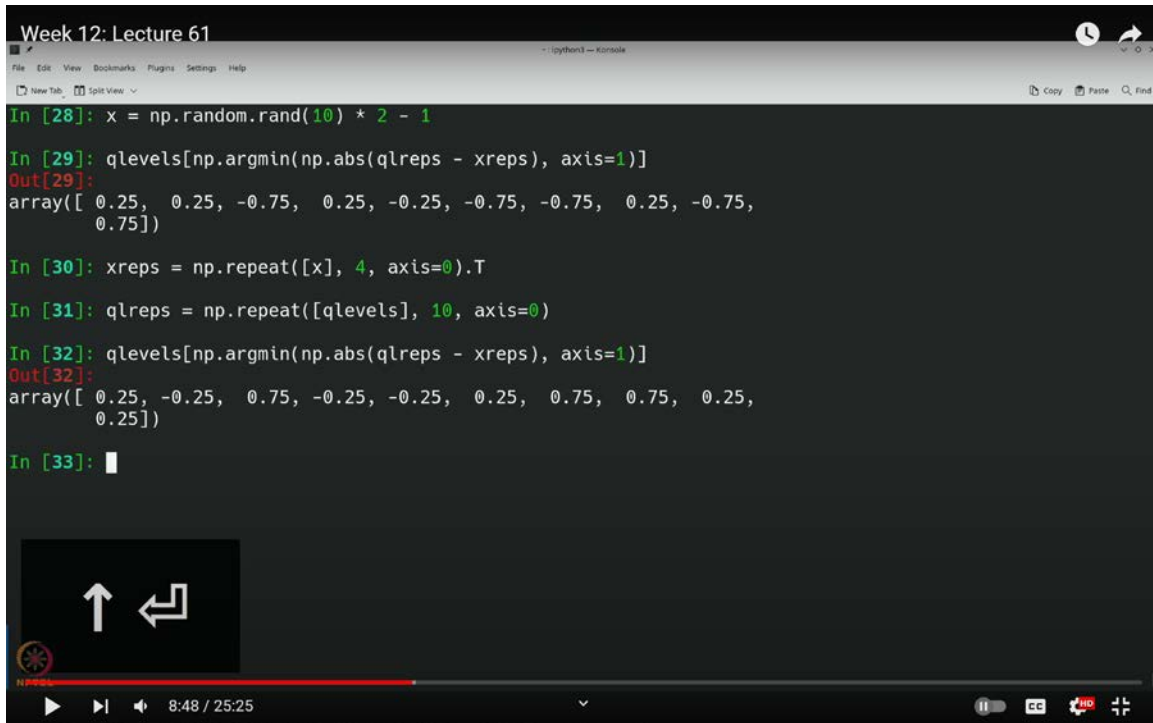
The result, `quantized_values`, contains the quantized version of each value in `x`. This approach efficiently quantizes each value to its nearest quantization level in one go.

To find the index of the minimum value in each row, you can use the `np.argmin` function. By specifying the axis parameter, you can determine whether you want to find the minimum across rows or columns. For example, setting the axis to 1 returns the indices of the minimum elements for each row. Using these indices with your quantization levels array will yield the corresponding quantized values.

To verify the correctness of this approach, compare the quantized results with the original values. For instance, you should see that 0.35 is indeed closest to 0.25, -0.98 is closest to -0.75, and so on. This demonstrates that with just a single line of code, you can effectively perform quantization. We will apply this method when implementing the quantization

block in GNU Radio.

(Refer Slide Time: 08:48)



The screenshot shows a video player interface with a dark theme. The title bar reads "Week 12: Lecture 61". The main content area displays a Python console window with the following code and output:

```
In [28]: x = np.random.rand(10) * 2 - 1
In [29]: qllevels[np.argmax(np.abs(qlreps - xreps), axis=1)]
Out[29]:
array([ 0.25,  0.25, -0.75,  0.25, -0.25, -0.75, -0.75,  0.25, -0.75,
        0.75])
In [30]: xreps = np.repeat([x], 4, axis=0).T
In [31]: qlreps = np.repeat([qllevels], 10, axis=0)
In [32]: qllevels[np.argmax(np.abs(qlreps - xreps), axis=1)]
Out[32]:
array([ 0.25, -0.25,  0.75, -0.25, -0.25,  0.25,  0.75,  0.75,  0.25,
        0.25])
In [33]:
```

At the bottom of the console window, there is a small video player control with a play button and a progress bar. The progress bar shows the current time as 8:48 / 25:25.

Now, let's move on to GNU Radio to put this into practice. To start, we'll review our setup:

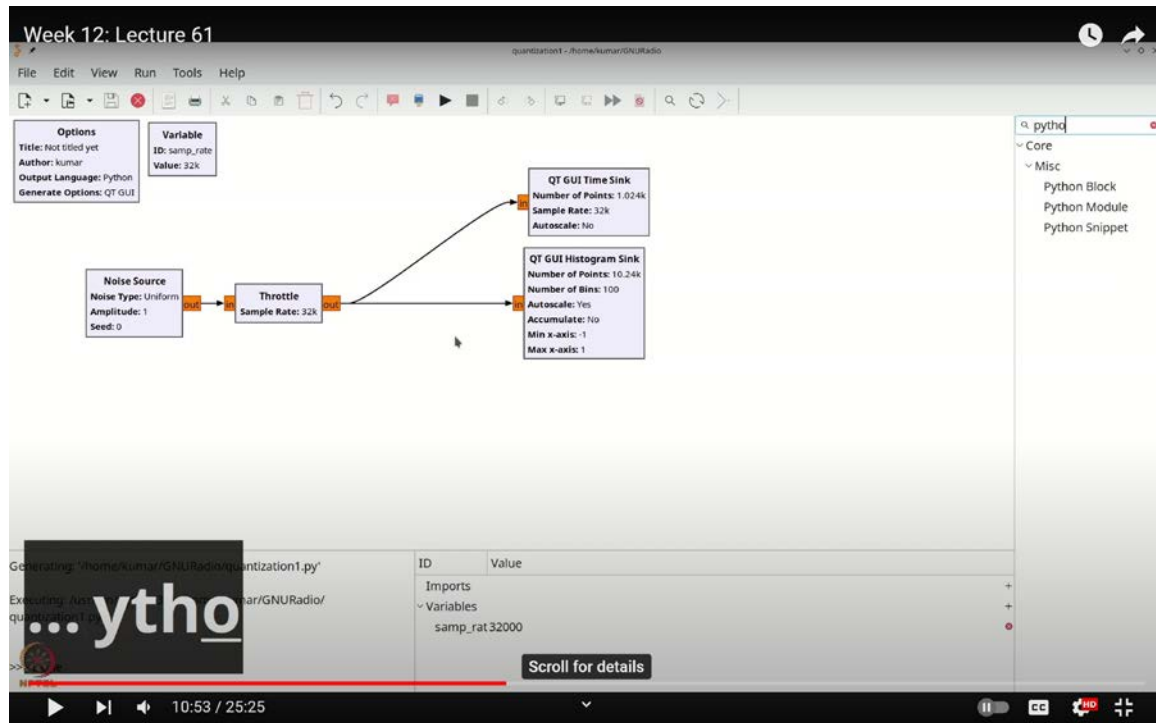
- Random variables
- Quantization levels
- ``x_reps`` and ``ql_reps``
- Quantization process

In GNU Radio, we'll first add a noise source. Use Ctrl + F to find it, and label it "uniform." Set the amplitude to 1 and choose "float" as the data type. To observe the values generated, add a histogram by searching for it with Ctrl + F.

Next, insert a throttle block (Ctrl + F) to control the data flow, and connect it to the histogram. Running this flow graph will produce values uniformly distributed between -1 and 1. To confirm the uniform distribution, use a larger number of points and verify that the histogram remains relatively flat, indicating that values are evenly spread across bins.

To proceed with quantization, add a time sync block (Ctrl + F) and set it to "float" to view the uniformly distributed values.

(Refer Slide Time: 10:53)



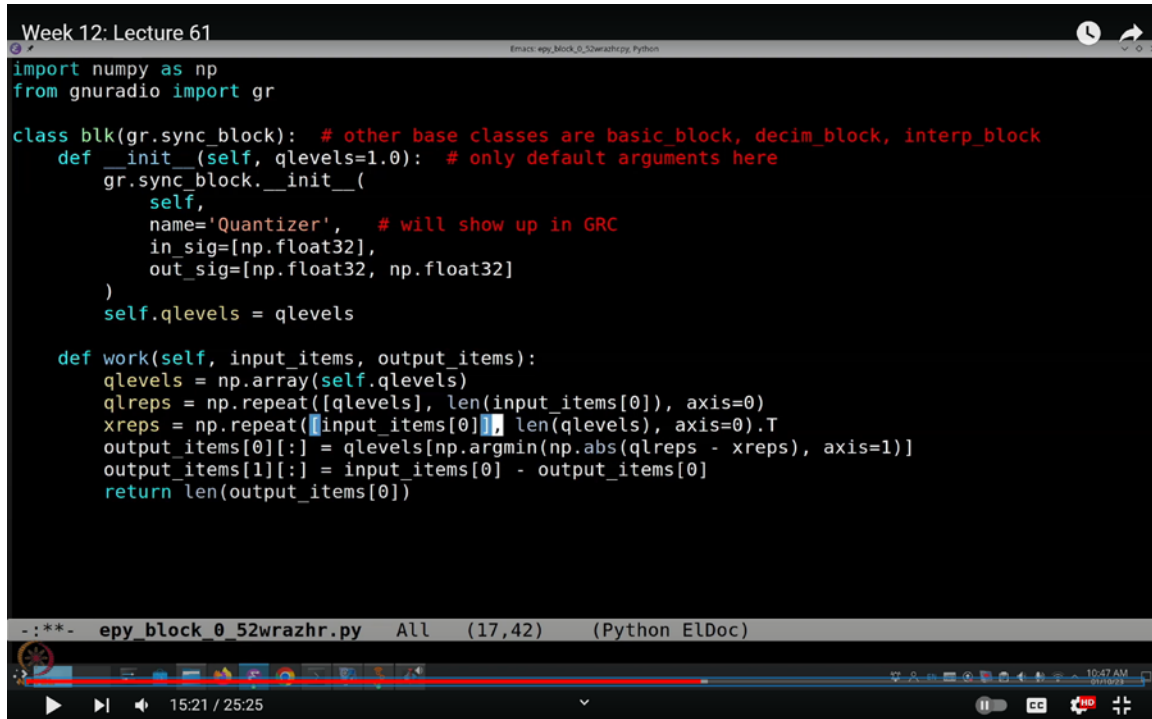
Now, let's implement the quantizer. Add a Python block (Ctrl + F) and configure it to accept quantization values as an argument. Modify the block's parameters to use a list of quantization levels, and connect it to the rest of your flow graph. This setup will allow you to perform quantization on your data within GNU Radio, applying the principles discussed earlier.

We'll start by opening the editor for modifications. The next step involves removing comments from the existing code and setting up our quantizer block. This block will accept a float input and produce two float outputs: one for the quantized value and the other for the quantization error. Including the error output allows us to histogram it and analyze its distribution.

Instead of using the placeholder "example_param," we will rename it to "Q_levels" for clarity. Now, let's proceed to implement the quantizer. To mirror the approach used in the

code, we'll first define ``Q_levels`` as ``self.QLevels``. This alias simplifies our code by preventing the need to repeatedly type ``self.QLevels``.

(Refer Slide Time: 15:21)



The screenshot shows a video player interface with a dark background. The title bar at the top reads "Week 12: Lecture 61". Below the title bar, the code is displayed in a light blue font on a dark background. The code defines a class `blk` that inherits from `gr.sync_block`. The `__init__` method takes `qlevels=1.0` as a parameter and sets up the block's name, input/output signal types, and the number of quantization levels. The `work` method performs the quantization by repeating the input values based on the number of quantization levels and then finding the minimum absolute difference between the input and the quantized values to determine the output.

```
Week 12: Lecture 61
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, qlevels=1.0): # only default arguments here
        gr.sync_block.__init__(
            self,
            name='Quantizer', # will show up in GRC
            in_sig=[np.float32],
            out_sig=[np.float32, np.float32]
        )
        self.qlevels = qlevels

    def work(self, input_items, output_items):
        qlevels = np.array(self.qlevels)
        qlreps = np.repeat([qlevels], len(input_items[0]), axis=0)
        xreps = np.repeat([input_items[0]], len(qlevels), axis=0).T
        output_items[0][:] = qlevels[np.argmin(np.abs(qlreps - xreps), axis=1)]
        output_items[1][:] = input_items[0] - output_items[0]
        return len(output_items[0])

-:***- epy_block_0_52wrazhr.py All (17,42) (Python ElDoc)
```

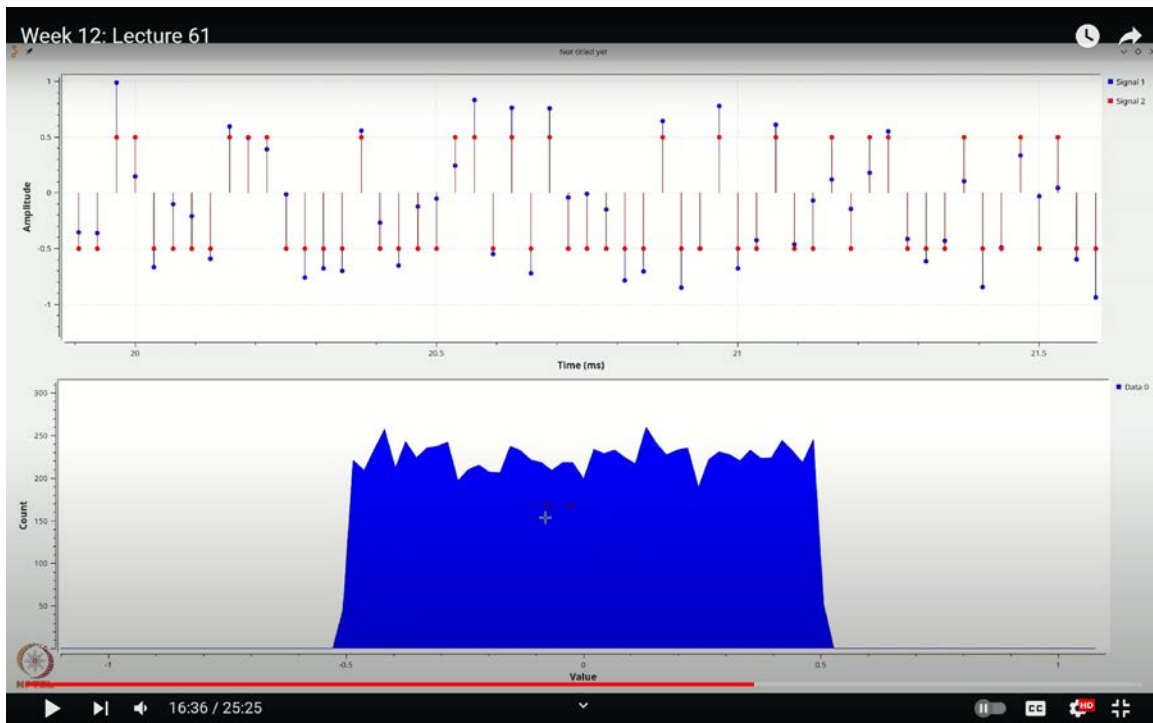
Next, we will create ``QLReps`` by repeating ``Q_levels`` as many times as there are input values. This is done using ``np.repeat(Q_levels, len(inputs.items[0]), axis=1)``. Similarly, ``xReps`` will be created by repeating the input values based on the number of quantization levels, using ``np.repeat(input.items[0], len(Q_levels), axis=0)``.

To ensure that ``Q_levels`` is properly formatted, we will convert it to a NumPy array. This avoids potential indexing issues that can arise with standard Python lists. We'll calculate the quantized values by finding the index of the minimum absolute difference between ``QLReps`` and ``xReps`` using ``np.argmin(np.abs(QLReps - xReps), axis=1)``.

The error output will be the difference between the input values and the quantized values. Thus, ``output.items[1]`` will be computed as ``input.items[0] - output.items[0]``, which represents ``x - x_hat``.

A few final adjustments are necessary: rename ``self.qLevels`` to ``self.QLevels`` to match the code, remove the redundant ``self.qLevels_param`` comment, and ensure that ``input.items`` is within square brackets to correctly replicate the array. With these changes, the code should be ready. Assuming we set the quantization levels to -0.5 and 0.5, executing the flow graph will demonstrate the quantization performance.

(Refer Slide Time: 16:36)



To verify the accuracy of our quantization, let's examine the stem plot and temporarily halt the execution. Zooming in, you will observe that when the blue values are negative (i.e., between 0 and -1), the quantized value correctly maps to -0.5. Conversely, when the blue values are positive (i.e., between 0 and 1), the quantized value appropriately maps to +0.5. This behavior is consistent with our expectations.

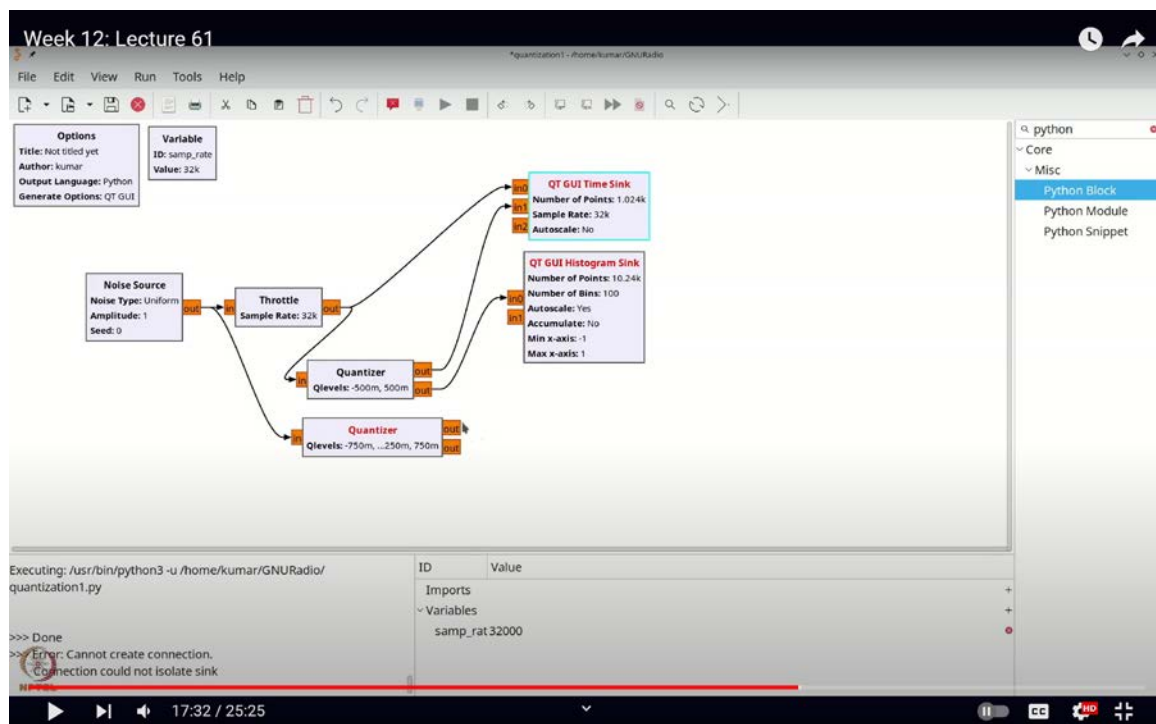
The quantization error ranges between -0.5 and +0.5. For example, if a random value between 0 and 1 is quantized to +0.5, the maximum possible error is ± 0.5 . If a value close to 0 is quantized to +0.5, the resulting error is -0.5. Conversely, if a value near 1 is quantized to +0.5, the error is +0.5. The error will always be within this range.

Now, let's observe how performance improves with more quantization levels. We can duplicate this quantizer by copying and pasting it, connecting the same input to this new quantizer. We'll add another time sink and histogram to analyze the output. In this setup, we'll configure the quantizer with levels of -0.75, -0.25, +0.25, and +0.75. These levels are optimal for a 2-bit quantizer applied to a uniform random variable ranging from -1 to 1.

By connecting the original output to the time sink and examining the error, you'll notice that the quantization error narrows significantly, now ranging between -0.25 and +0.25. Zooming in on the waveforms, if you hide the red trace, you'll see that the green trace offers a more accurate representation of the original waveform, with noticeably reduced error.

Next, let's add one more bit to further refine the quantization. The new levels will be -0.875, -0.625, -0.375, -0.125, +0.125, +0.375, +0.625, and +0.875. This additional bit narrows the error range further to between -0.125 and +0.125. As you verify this, you'll see that the green trace is even closer to the blue trace, and the error is minimized. You can plot this error by subtracting and visualizing it as an exercise.

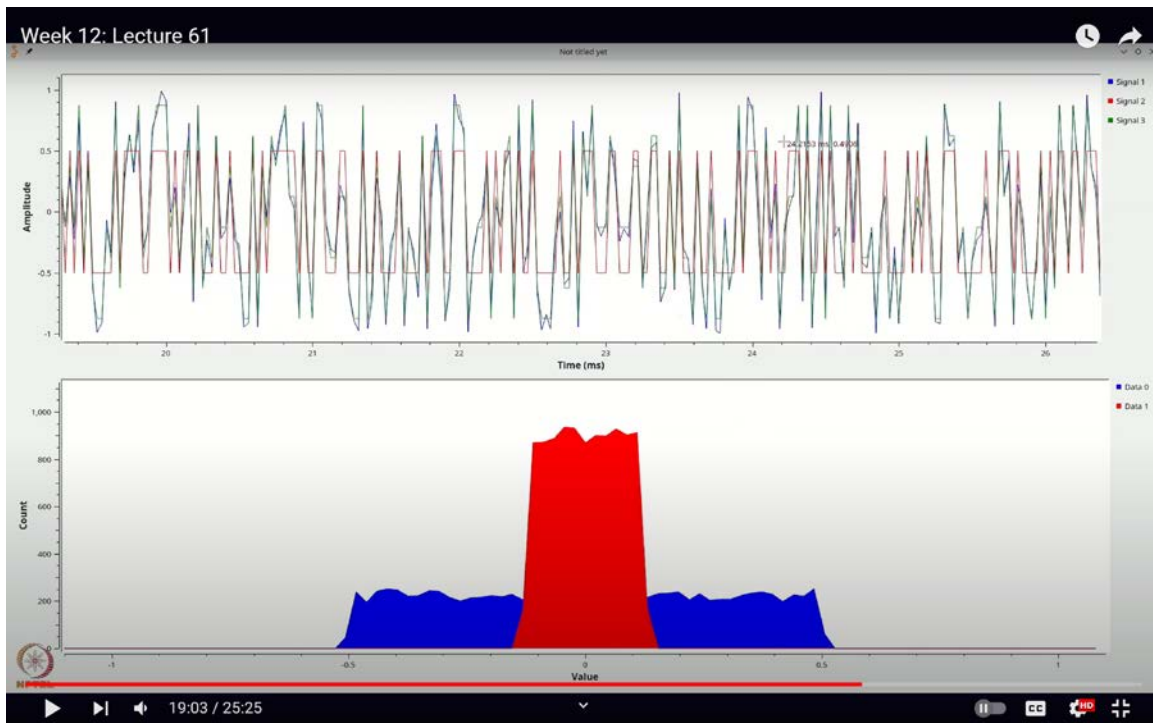
(Refer Slide Time: 17:32)



Lastly, let's investigate how the quantization performs with Gaussian noise. To do this, add a parallel noise source and replicate the previous setup. Change the noise source to Gaussian and update the quantizer levels to the optimal values for Gaussian noise, which are approximately ± 0.798 . This value is close to $\sqrt{2/\pi}$, as discussed in the lecture. Set the quantizer levels accordingly and observe how the performance and error compare.

Now, let's connect the time sink to the original source and execute the flow graph. To make things clearer, let's add some labels. We'll label one part as "Uniform" and the other as "Gaussian" for easy identification.

(Refer Slide Time: 19:03)



Upon running the flow graph for the Gaussian case, you will notice that the performance is as shown, and the histogram for the Gaussian source will look like this. Interestingly, the error distribution starts to resemble a uniform distribution, and the original error profile also begins to flatten out.

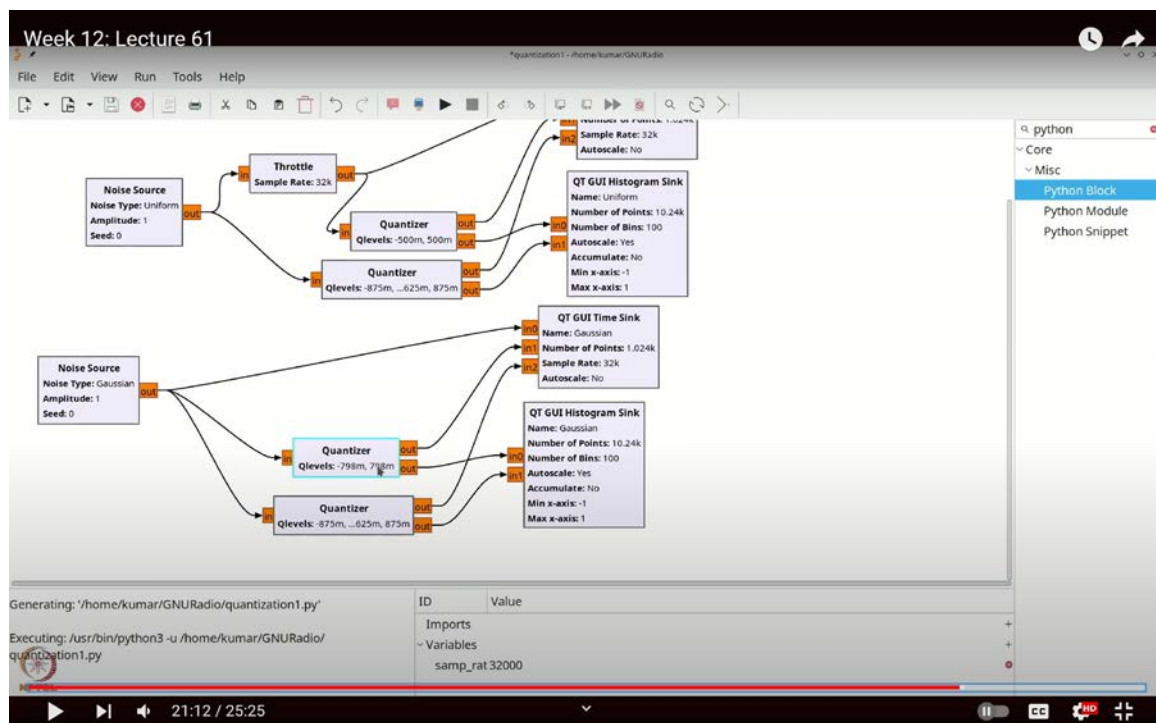
Let's experiment by setting the quantization levels to 0.1 just for demonstration. Although this is not an optimal quantizer, it's useful to observe its impact. You'll see that the

quantization error is quite substantial, with the quantized values being significantly different from the actual values. This results in a much wider blue curve, indicating a high error range. The amplitude of the error is notably larger, making the quantizer less effective compared to the uniform quantizer, which still performs better.

If we adjust the quantization range to -1 and 1 and use an optimal 1-bit quantizer with levels of -0.79, the performance improves significantly. The second quantizer shows a much lower error footprint, effectively reducing the quantization error compared to the first one. Although the green and red curves appear close, the green curve better tracks the blue curve, indicating a more precise quantization with lower amplitude errors.

For an even clearer comparison, set the range to -2 and 2. This results in a very poor quantizer, with the blue histogram showing a wide error range and noticeable overshoot in the red curve. Conversely, the green and blue curves perform much better in comparison.

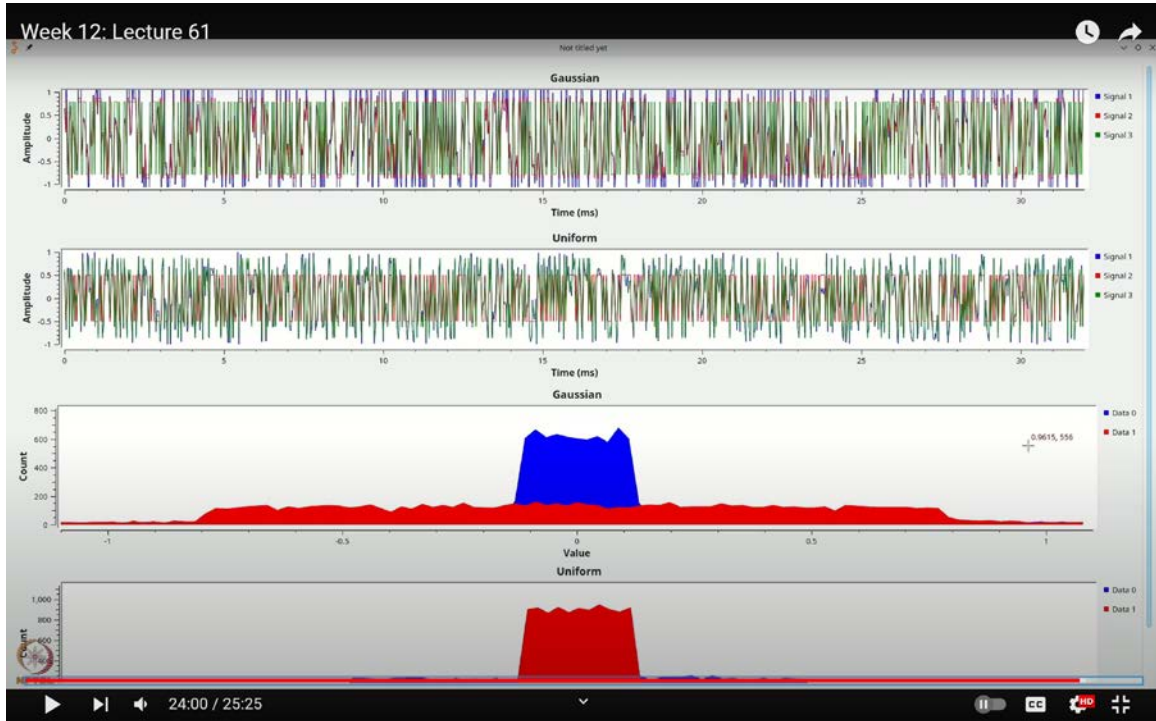
(Refer Slide Time: 21:12)



By using this approach, you can evaluate the performance of different quantizers. As you increase the number of bits, the performance improves, providing a more accurate

representation of the signal. Even with Gaussian noise, adding more bits will enhance the performance of the quantizer. A uniform quantizer, despite its simplicity, still performs reasonably well for Gaussian distributions, although it might not be the most efficient as it does not account for the distribution's characteristics.

(Refer Slide Time: 24:00)



In this manner, you can effectively assess the performance of various quantization algorithms. In this lecture, we demonstrated how GNU Radio can be used to perform quantization efficiently. We created a straightforward quantization block where you specify the quantization levels. This block in GNU Radio performs mapping to these levels using minimum distance mapping and allows you to observe the associated error.

We examined the performance for both uniform and Gaussian sources. It became evident that deviating from the optimal quantization points significantly worsens the error performance. Therefore, selecting the correct quantization levels is crucial for improving quantization performance by minimizing the quantization error. Thank you.