Digital Communication using GNU Radio

Prof. Kumar Appaiah

Department of Electrical Engineering

Indian Institute of Technology Bombay

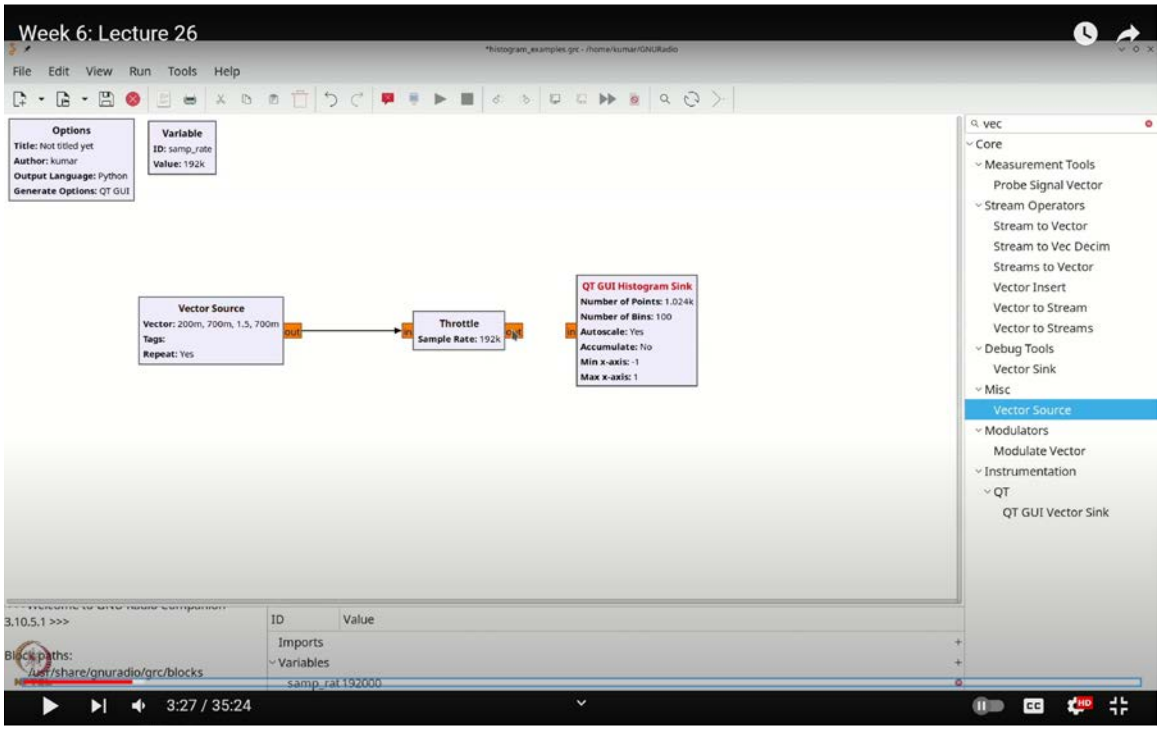
Week-06

Lecture-26

Histograms in GNU Radio

In this lecture, we will introduce the QT-GUI histogram sink in GNU Radio, a powerful tool that allows you to count and visualize the distribution of values within a specified range. This histogram sink is particularly useful for visualizing the probability distribution function (PDF) of a data source or a sequence, which can be invaluable when studying the effects of noise on signal processing, such as in evaluating symbol error rates.

(Refer Slide Time: 03:27)


Week 6: Lecture 26

*histogram_examples.grc - /home/kumar/GNURadio

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

Variable
ID: samp_rate
Value: 192k

Vector Source
Vector: 200m, 700m, 1.5, 700m
Tags:
Repeat: Yes

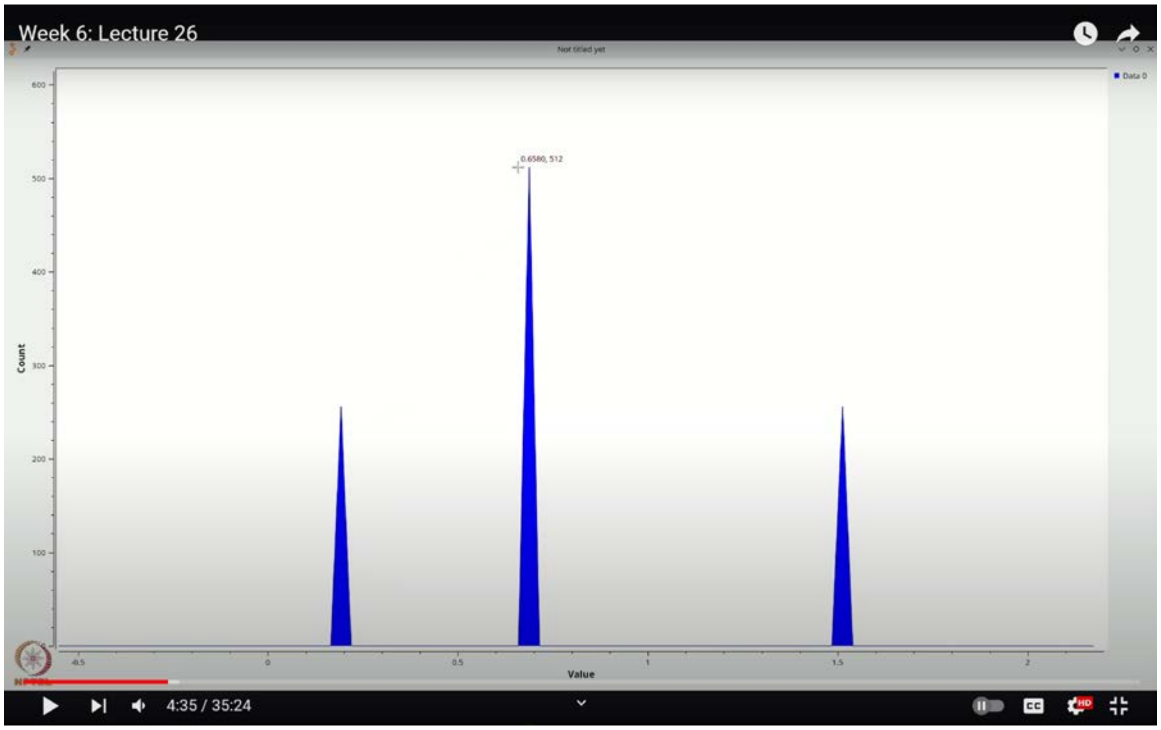
Throttle
Sample Rate: 192k

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

We'll leverage this tool to observe how different types of noise, such as Gaussian noise, manifest in terms of the range of values they produce. This will help us better understand the impact of noise on signal processing performance.

To start, let’s dive into the histogram sink’s evaluation, which will be essential when performing various measurements, including symbol error rate analysis. Begin by pressing Ctrl+F (or Cmd+F) and typing "histogram" or "HIST" to locate the histogram sink. Drag and drop it into your workspace.

(Refer Slide Time: 04:35)



The histogram sink comes with several key parameters. One important parameter is the number of points, which determines how the histogram bins are created and how the values are counted. You might already be familiar with parameters like grid and auto scale. There’s also an "accumulate" option, which we’ll discuss in more detail later. Additionally, you’ll find "min x-axis" and "max x-axis" settings, which define the range of values over which the histogram is calculated.

To better understand how this works, let's go through an example. We’ll start by adding a throttle block. Again, use Ctrl+F (or Cmd+F) to search for "throttle." Once found, add it to your workspace and configure it as a float type. I've made a small adjustment here: setting the sampling rate to 1920.00 to make the simulation run a bit faster.

Finally, we’ll introduce a vector source. Search for "VEC" using Ctrl+F (or Cmd+F), add the vector source, and double-click it to configure it as a float. I’ve selected a specific vector with values [0, 2, 0.7, 1.5, 0.7]. This four-length list will repeat continuously since we’ve set the repeat option to "yes."

With this setup, we'll be able to explore how the histogram sink in GNU Radio can effectively visualize and analyze the distribution of these values, giving us deeper insights into the behavior of signals under various noise conditions.

After configuring the vector source and clicking 'OK,' I’m ready to connect it to the throttle and then link the throttle to the histogram sink. Once I execute this flow graph, the resulting plot displays distinct peaks, one around the value 0.2, another at 0.7, and a third at 1.5, though the peak at 1.5 is barely visible.

The reason the 1.5 peak is less prominent is that the value range needs adjustment for better visibility. To address this, let's modify the minimum value to -0.5 and the maximum value to 2. Now, when I re-run the flow graph, you’ll observe three clear peaks: one near 0.2, another close to 0.7, and the final one at approximately 1.5.

Next, let's analyze the heights of these peaks. The peak near 0.2 reaches a height of about 256. If you zoom in, you’ll notice that this height is indeed around 256. The peak at 0.7 is taller, with a height of 512, while the peak at 1.5 returns to 256. But what does this signify?

The histogram sink processes 1024 values and counts how many fall within specific bins, or intervals along the x-axis. In our case, these bins span from -0.5 to 2, covering a range of 2.5 units. This range is divided into 100 equally spaced bins, each covering $\frac{2.5}{100} = 0.025$ units. Every time a value from our vector source falls within a bin's range, the count for that bin increases by one.

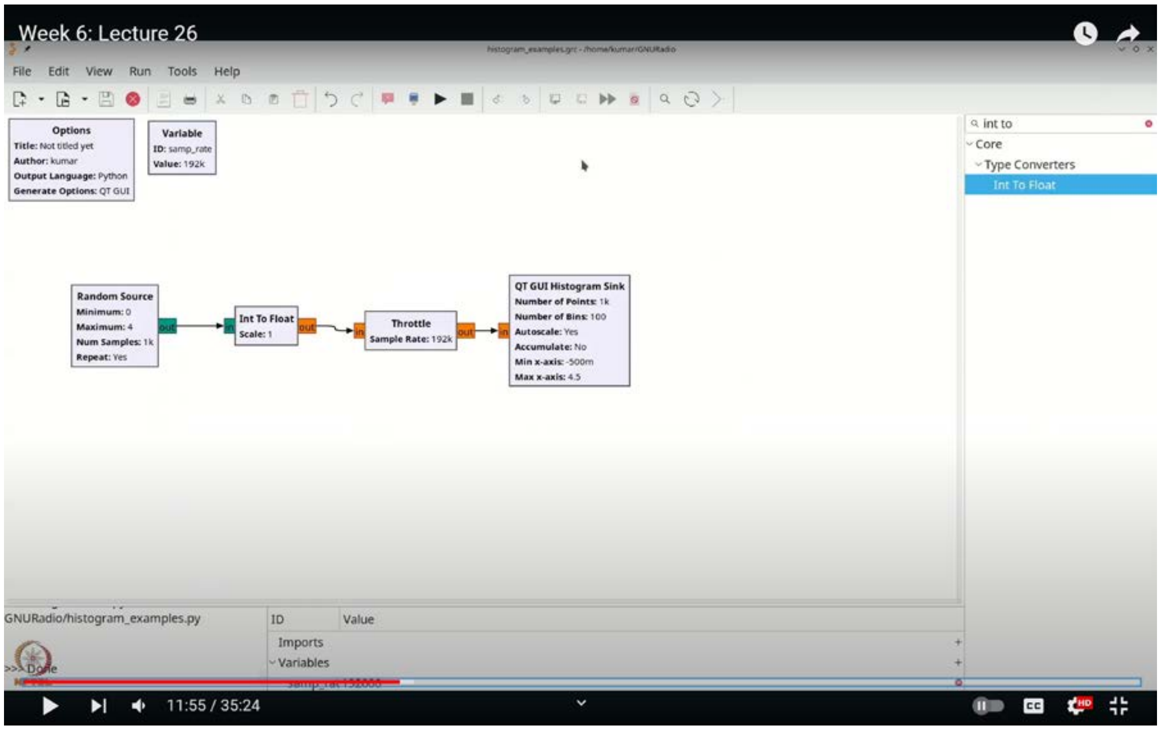
Given our input values, 0.2, 0.7, and 1.5, each appearance of 0.2 increases the count in its corresponding bin by one. The same applies to 0.7 and 1.5. Since our vector source repeats 256 times to produce a total of 1024 points, we end up with exactly 256 occurrences of 0.2, 256 of 1.5, and 512 of 0.7, because 0.7 appears twice as often as the other values.

Thus, the bin containing 0.2 shows a height of 256, the bin for 0.7 shows 512, and the bin for 1.5 shows 256, perfectly reflecting the distribution of values in our input data.

When you add these values, you get a total of 1024. Now, if you want to mix things up, you can change this number to something larger, let's say 5000. With 5000 samples, you would see that 2500 of them are 0.7, and 1250 each are 0.2 and 1.5. Essentially, the histogram counts how often each of these numbers occurs.

However, this particular vector source might not be very interesting because we already know in advance that it will send 0.2, 1.5, and 0.7, with 0.7 appearing twice as often as the others. The outcome is predictable and not particularly insightful. But where a histogram sink truly shines is in scenarios where you want to visualize the distribution of values when you don't have prior knowledge of what those values might be.

(Refer Slide Time: 11:55)



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

Variable
ID: samp_rate
Value: 192k

Random Source
Minimum: 0
Maximum: 4
Num Samples: 1k
Repeat: Yes

Int To Float
Scale: 1

Throttle
Sample Rate: 192k

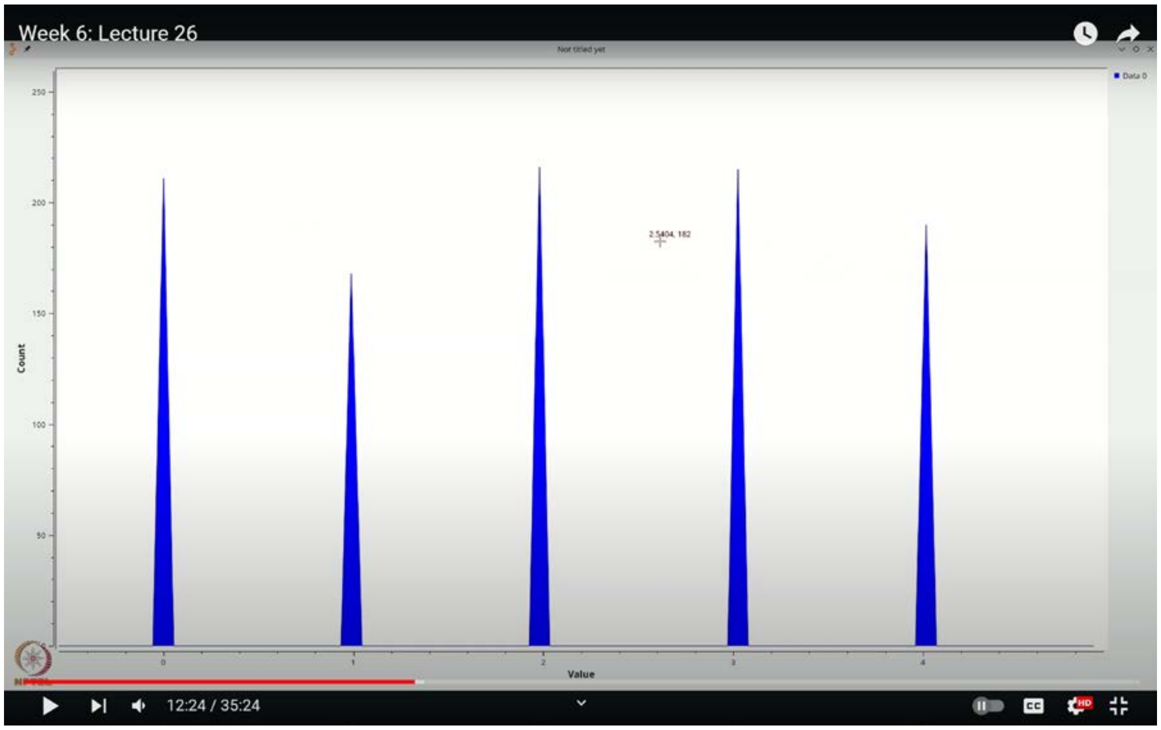
QT GUI Histogram Sink
Number of Points: 1k
Number of Bins: 100
Autoscale: Yes
Accumulate: No
Min x-axis: -500m
Max x-axis: 4.5

For instance, let’s say you have a random source. What kind of values does this random source generate? This could be really useful to visualize. So, let's remove the vector source by selecting it and hitting delete, and instead, we'll add a random source.

To do this, press Ctrl+F and type "random" (R, A, N, D). We'll select a random source, which outputs integers. The minimum value is set to 0, and the maximum value is set to 2. However, keep in mind that the outputs will always be 0 and 1 because the maximum value of 2 is not actually output. Instead, the source only emits 0s and 1s.

To verify whether we're getting an even distribution of 0s and 1s, let's create a histogram of this output. We'll set the x-axis range from -0.5 to 2, which will cover the values 0 and 1. Next, we need to convert this random source to the correct data type. Press Ctrl+F (or Cmd+F) and search for "int to float." Grab the "int to float" block, place it between the random source and the histogram sink, wire everything up, and now, when you observe the histogram, you'll see the distribution of 0s and 1s clearly.

(Refer Slide Time: 12:24)

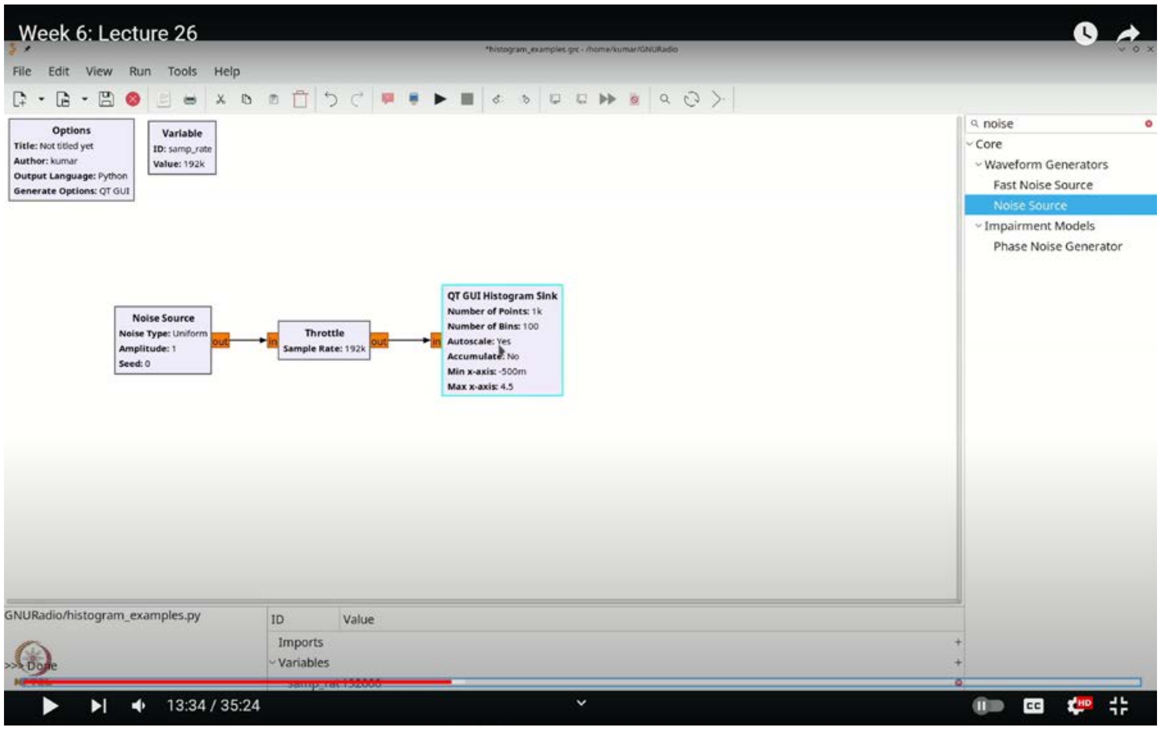


I estimate that you have approximately 2450 of one value and 2550 of another. These numbers aren't exact because, in any random simulation, even when values are equally likely, you won’t get an exact 50-50 split. The results will vary slightly. Here, we’re dealing with 1000 samples, but when we increase the number of points to 5000, let’s reset it back to 1000 and execute the flow graph.

You’ll notice that the distribution is about 500 for each value, which makes complete sense. Now, let’s experiment a bit. We’ll modify the random source to generate values from 0 through 4 and adjust our x-axis range from -0.5 to 4.5. Now, with 1000 points being fed into the histogram sink, when you execute the flow graph, you should see a structure emerge. Ideally, with 1000 points, each value should appear roughly 250 times, but in this realization, there were more 2s and 1s compared to 0s.

Let’s run the simulation again. This time, you might observe more 1s than before. This slight variation is something to be aware of, it’s due to the size of the sample set chosen. Another key point is that the random source generates 1000 samples and then keeps repeating them, which is why the values don’t change as more cycles are processed.

(Refer Slide Time: 13:34)



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

Variable
ID: samp_rate
Value: 192k

Noise Source
Noise Type: Uniform
Amplitude: 1
Seed: 0

Throttle
Sample Rate: 192k

QT GUI Histogram Sink
Number of Points: 1k
Number of Bins: 100
Autoscale: Yes
Accumulate: No
Min x-axis: -500m
Max x-axis: 4.5

GNURadio/histogram_examples.py
>>> Done

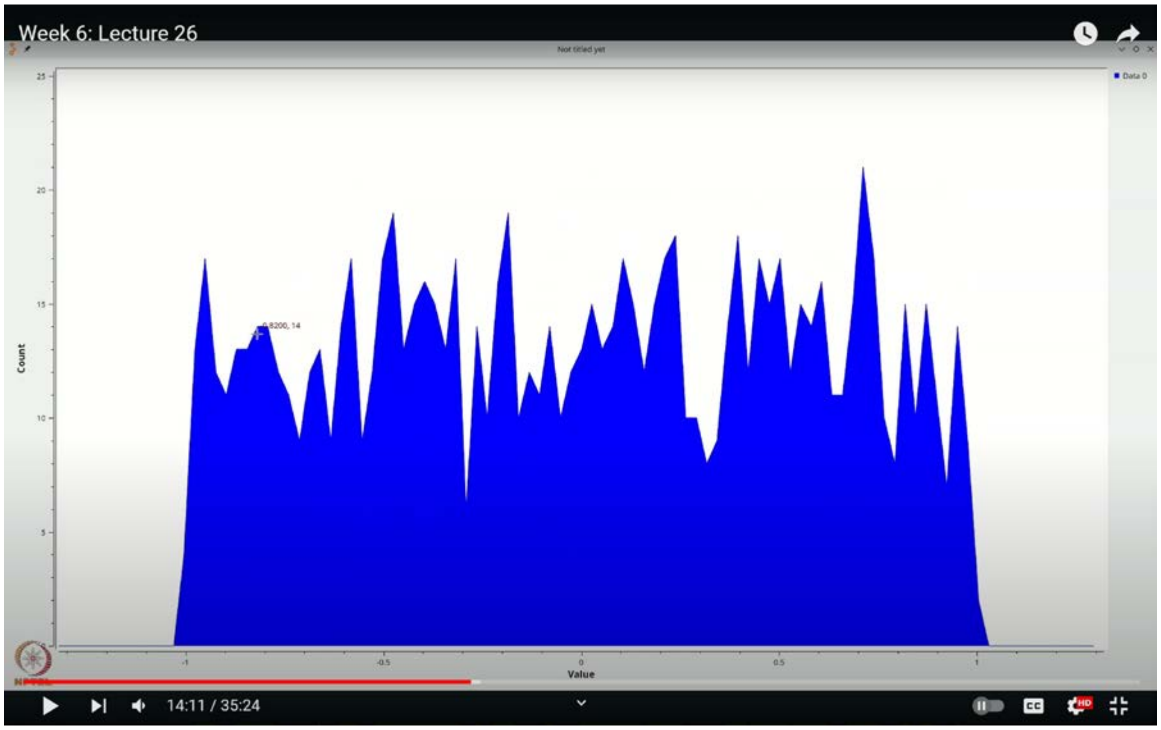
ID	Value
Imports	
Variables	

If you had set the maximum to 5 to include the fourth value, you would start noticing gaps in the distribution. While this can be explained through probability theory, the more critical insight here is that a random source isn't ideal for generating something like noise or a continuous pattern. It will only generate a fixed number of samples, like 1000, and keep repeating them.

In communication simulations, you often need a continuously varying set of random values, so using a random source as a noise generator isn't advisable. Let's now transition to using an actual noise source for our analysis. To do this, I'll delete the random source and the int-to-float block. Then, I'll press Ctrl+F (or Cmd+F) and type "noise" to bring in a noise source for our simulation.

Let's double-click on the noise source. We'll start by setting the output type to float and choose the uniform distribution. The amplitude is set to 1. In the context of GNU Radio, a uniform noise source with an amplitude of 1 will generate random values uniformly distributed between -1 and 1. Let's modify this range slightly, setting it to -1.2 to 1.2, and then run the flow graph.

(Refer Slide Time: 14:11)

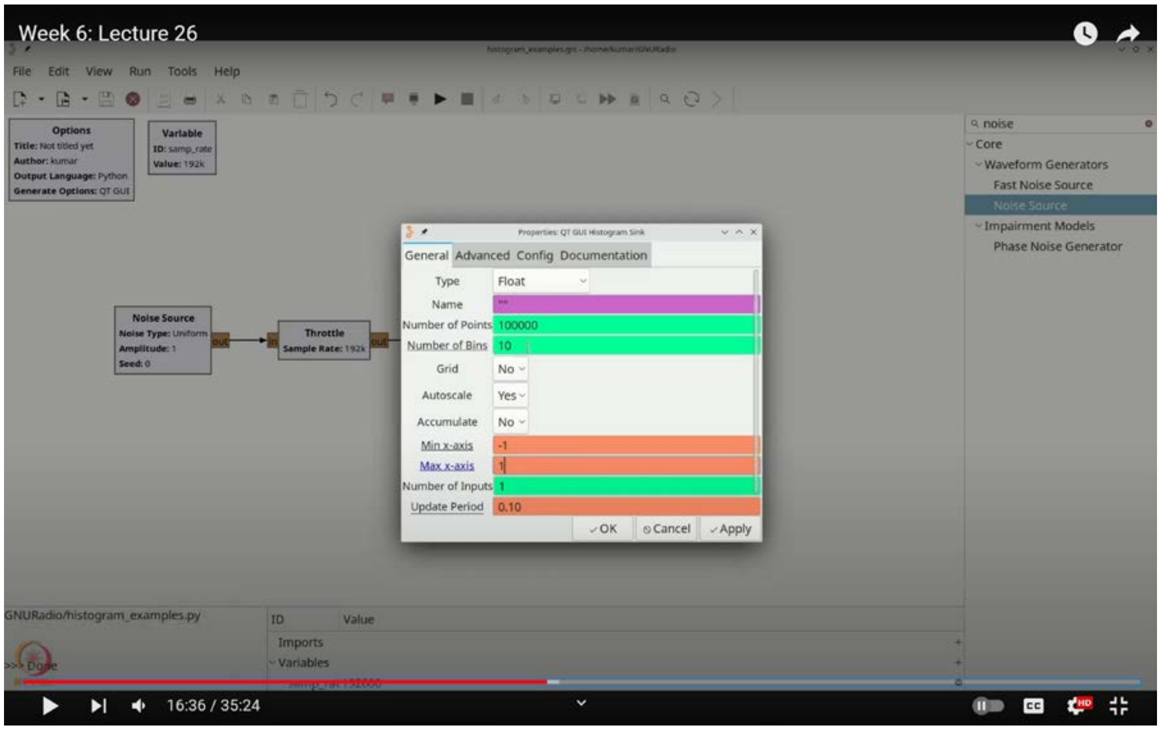


You'll notice that the histogram displays some movement. This movement occurs because each time the noise source generates a new set of values, a new histogram is constructed. Essentially, you're seeing one histogram for every 1000 points generated by the noise source. However, with only 1000 points, the histogram doesn't appear very stable or consistent.

To improve this, we can increase the number of points used to construct the histogram. Let's change it to 100,000 points. As expected, the histogram updates more slowly, but the result is much cleaner and more stable.

Another key detail to note is that the height of each bin in the histogram represents the actual count of points falling into that bin. To understand this concept more clearly, let's try something different. We'll reduce the number of bins to just two and then execute the flow graph again.

(Refer Slide Time: 16:36)



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

Variable
ID: samp_rate
Value: 192k

Noise Source
Noise Type: Uniform
Amplitude: 1
Seed: 0

Throttle
Sample Rate: 192k

Properties: QT GUI Histogram Sink

General | Advanced | Config | Documentation

Type	Float
Name	""
Number of Points	100000
Number of Bins	10
Grid	No
Autoscale	Yes
Accumulate	No
Min x-axis	-1
Max x-axis	1
Number of Inputs	1
Update Period	0.10

With only two bins, you'll observe that the values cluster into bins near -2 and 0, which isn't the desired outcome. So, let's adjust this. We'll increase the number of bins to four, or perhaps even ten, for better granularity. Now, when we run the flow graph, you'll see that while most bins show a uniform distribution of values, there are a couple of bins with noticeably different counts.

This discrepancy might seem surprising, especially given that we're using a uniform noise source where you would expect the values between -1 and 1 to be evenly distributed across the bins. So, what could be causing this?

Well, let's try a little trick to investigate further.

Let's double-click on the histogram sink and adjust the settings. First, set the minimum value to -1 and the maximum value to 1. We currently have 10 bins, which means that the interval between -1 and 1 is divided into 10 equal parts. Each of these bins should ideally contain about 10,000 values, assuming we have a total of 100,000 points.

Now, let’s examine the output. When you execute the flow graph, you'll notice that one bin appears to have a height that seems appropriate, but another shows a noticeable drop. What's going on here? To understand this, we need to take a closer look at the exact bin placements.

Double-click on the histogram sink again, go to the configuration settings, and activate the marker for line 1. Let's make this marker something clearly visible, like an "X" cross, and click "OK."

Remember, we asked GNU Radio to create 10 uniformly spaced bins between -1 and 1. However, when you execute the flow graph, you'll see that the crosses, which represent the bin centers, are not exactly where we expected them to be. For example, one cross might be at 0.88, another at 0.66, and so on. If you move to the first bin, it might be around -1.1. What GNU Radio is doing here is slightly different from what you asked for. Instead of strictly starting at -1 and ending at 1, it begins slightly to the left of -1 and extends slightly to the right of +1. As a result, the bin centers are not at the expected intervals.

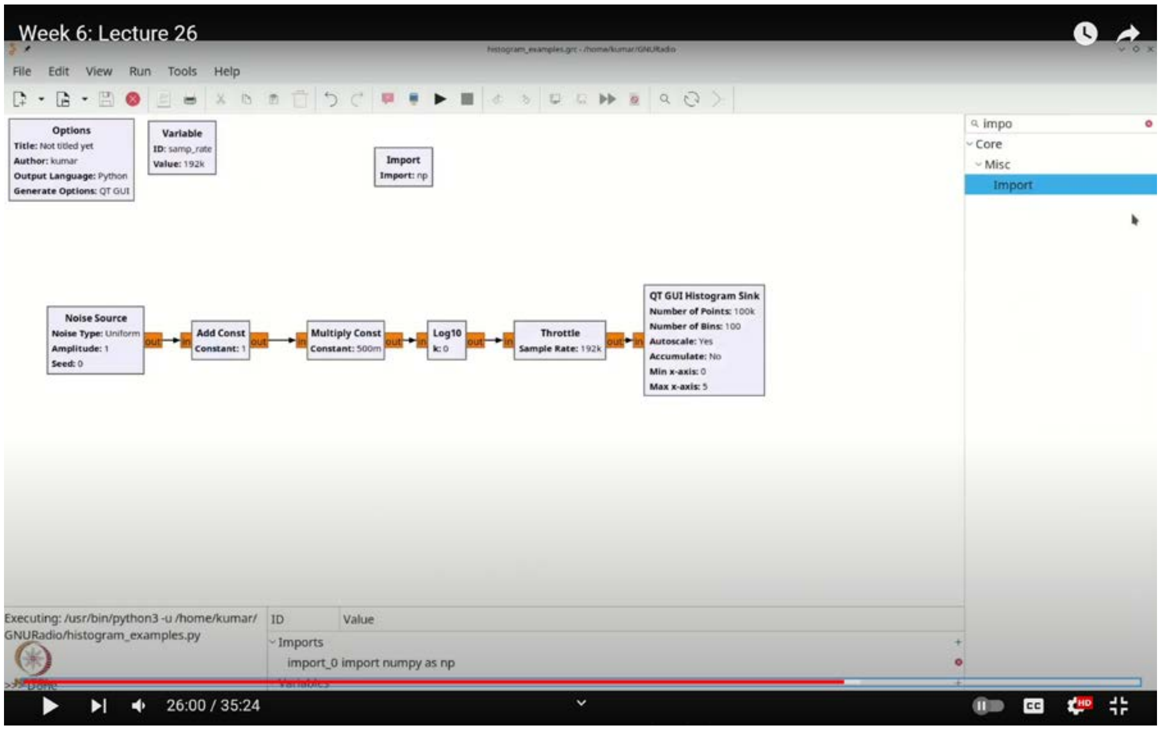
For instance, the bins might actually be at -1.09, -0.88, -0.66, -0.44, -0.22, and so on. This means that the bins are not neatly aligned between -1 and 1 as you would expect if each bin were exactly 0.2 units wide. Instead, GNU Radio’s bins are slightly offset, causing some of the unexpected results you’re seeing.

For example, if you look at the last bin, which might span from 0.88 to 1.11, you’ll notice that 1 falls right in the middle of this bin. Similarly, -1 might fall right in the middle of its respective bin, which can result in some strange behavior due to the way floating-point arithmetic works. This can cause the software to choose one bin over another, leading to the inconsistencies you've observed.

This situation is a result of the way GNU Radio handles bin partitioning, and it's important to be aware that the actual bin intervals might not align perfectly with what you intended. However, the qualitative plot is still useful, and this issue becomes less noticeable when you increase the number of bins, giving you a better approximation of the histogram.

Finally, there’s an option called "Accumulate." What this does is, after computing the histogram once with 100,000 points, it continues to add the counts from the next 100,000 points to the existing histogram, effectively accumulating the data over time.

(Refer Slide Time: 26:00)



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

Variable
ID: samp_rate
Value: 192k

Import
Import: np

Noise Source
Noise Type: Uniform
Amplitude: 1
Seed: 0

Add Const
Constant: 1

Multiply Const
Constant: 500m

Log10
k: 0

Throttle
Sample Rate: 192k

QT GUI Histogram Sink
Number of Points: 100k
Number of Bins: 100
Autoscale: Yes
Accumulate: No
Min x-axis: 0
Max x-axis: 5

```
Executing: /usr/bin/python3 -u /home/kumar/GNURadio/histogram_examples.py
>>> Done
```

ID	Value
Imports	
import_0	import numpy as np
Variables	

To put it another way, let's say out of 100,000 samples, a particular bin receives 2,000 values. In the next set of 100,000 samples, the same bin might receive 2,010 values. GNU Radio then adds these together, resulting in a total of 4,010 for that bin. Consequently, you'll notice that the y-axis on the histogram consistently shows an upward trend as values accumulate.

For instance, if you look closely at the y-axis, you'll see this continuous increase, clearly demonstrating the uniform nature of the source. It's apparent that this is a uniform source, which becomes particularly useful when you need a qualitative but more accurate representation. Since the counts are constantly being added, the accumulation leads to a more stable histogram. Now, let's proceed by removing the accumulation.

Next, we'll make a few adjustments. To illustrate, let's consider how you can create an exponential random variable from a uniform random variable. This is typically done by taking the logarithm of a uniform random variable whose values range from 0 to 1. To see if this can be achieved in our setup, we'll apply a little trick. The uniform noise source we're using currently generates values between -1 and 1.

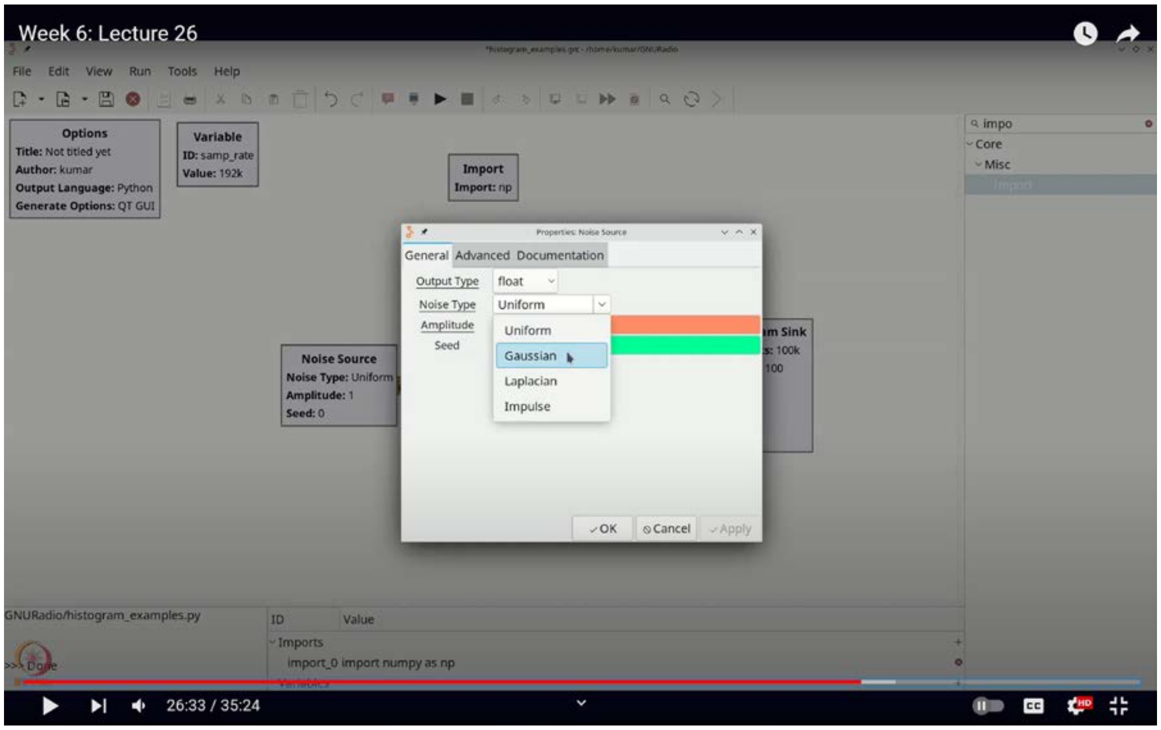
Let's shift these values so they fall between 0 and 1. To do this, press **`Ctrl + F`** to open the command finder and search for "Add." Select "Add Const," as we'll be adding a constant. Double-click to set the constant value to 1, and then multiply by a constant to divide the result by 2. Again, use **`Ctrl + F`** to search for "Multiply Const." Set the multiplier to 0.5.

Now, after these adjustments, we should have a uniform random variable with values between 0 and 1. As you can see, the values are now concentrated within this range. Next, let's take the negative natural logarithm of this modified random source. To do this, first, delete the existing wire. Then, press **`Ctrl + F`** and search for "Log." You might select "Log10" from the options.

Double-click on "Log10" to access the parameters, where you'll find options for "n" and "k." According to the documentation, the output is calculated as $n \times \log_{10}(\text{input}) + k$. Since we want the natural logarithm but only have the base-10 logarithm available, we need to divide this by the logarithm of e (base-10).

To do this, we could manually calculate the constant, but to avoid errors, let's use NumPy. Press **`Ctrl + F`** again, search for "Import," and select "import numpy." Double-click to import it as **`np`**. Now, in the log function, enter the constant as $-\frac{1}{\text{np.log10(np.exp(1))}}$. This operation effectively computes the natural logarithm and multiplies it by -1, which is what we need for an exponential distribution.

(Refer Slide Time: 26:33)



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

Variable
ID: samp_rate
Value: 192k

Import
Import: np

Noise Source
Noise Type: Uniform
Amplitude: 1
Seed: 0

Output Type: float
Noise Type: Uniform
Amplitude
Seed

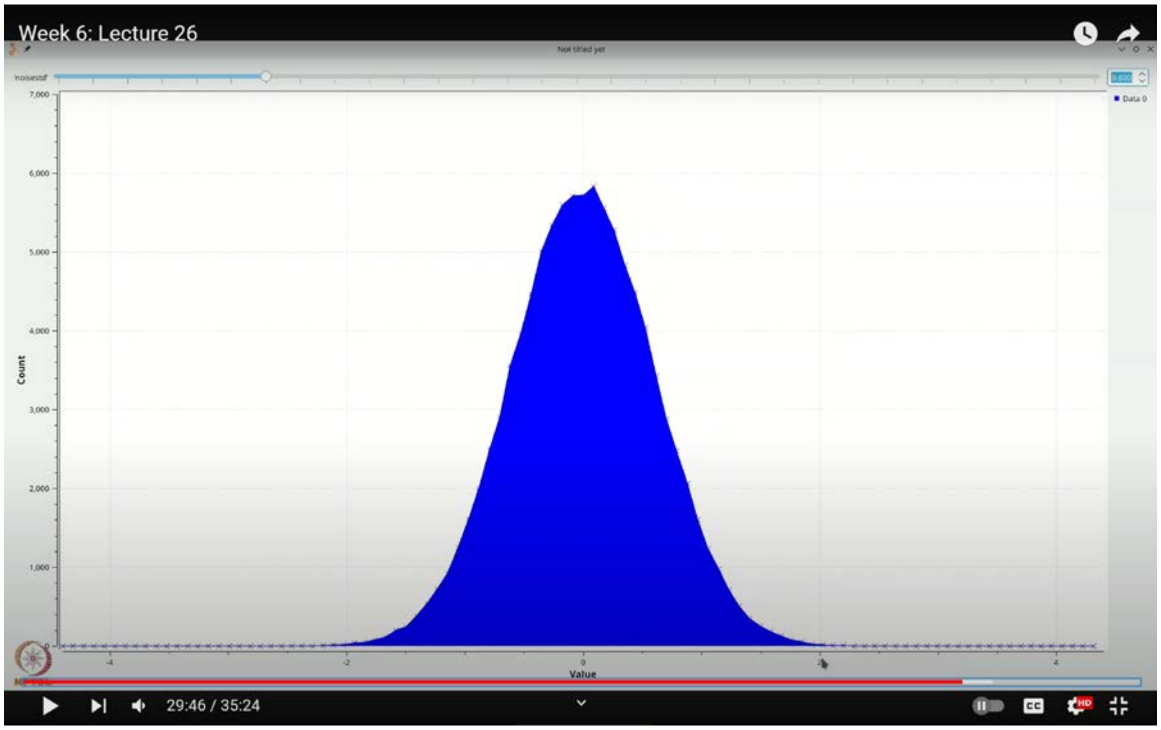
GNURadio/histogram_examples.py

ID	Value
Imports	
import_0	import numpy as np

After connecting everything and executing the flow graph, you should see a histogram that exhibits an exponential distribution. Adjust the intervals to range between 0 and 5, and upon re-execution, the histogram should clearly resemble the probability density function (PDF) of an exponential random variable. Finally, let's make a slight modification to explore other random sources, particularly to observe how Gaussian distributions appear.

Let's start by deleting the previous setup and reconnecting our noise source directly. This time, we’ll change the noise source to a Gaussian distribution. After selecting the Gaussian option, click "OK." We'll also double-click to check the settings and ensure that "Auto Scale" is enabled, which is fine for now. Let's set the number of bins to 100 and adjust the range from -4 to 4. Now, when we run the flow graph, you'll notice that we get a smooth, bell-shaped curve, characteristic of a Gaussian distribution.

(Refer Slide Time: 29:46)



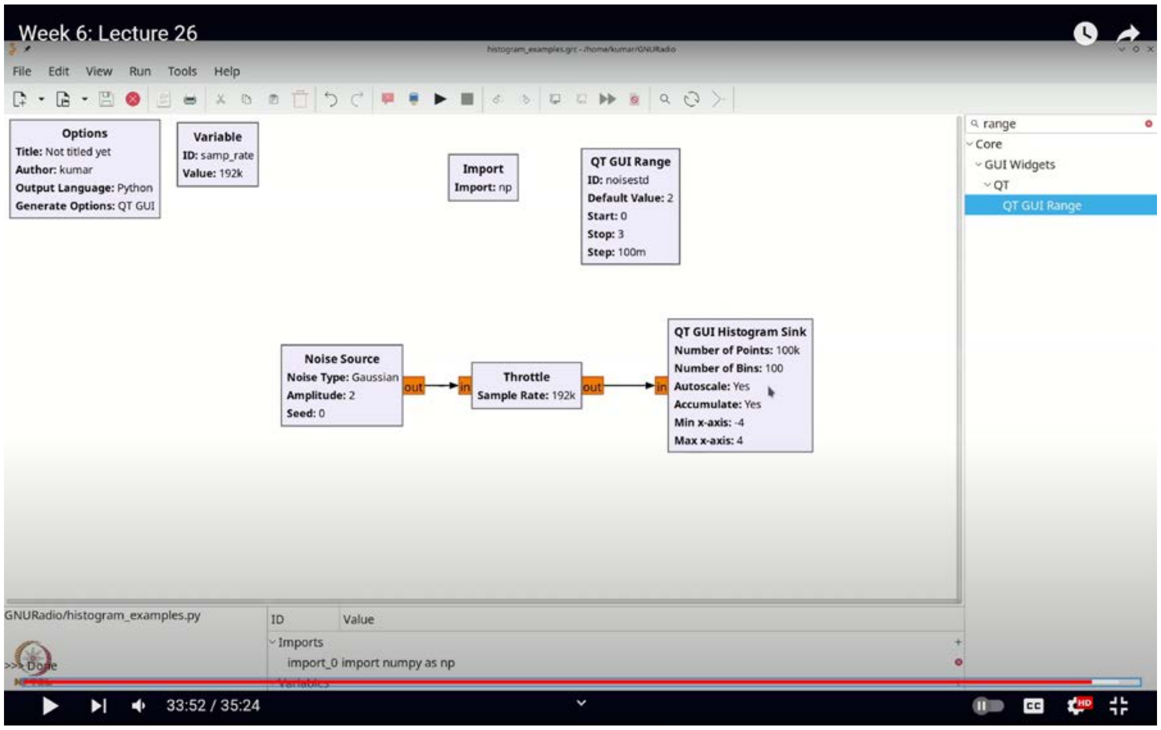
The reason the plot appears to fluctuate is that every time the Gaussian source generates new data points, a new histogram is created for every 100,000 points. Therefore, the bell shape isn't perfectly consistent, reflecting the variability in the sequence of random variables. It's typical to adjust some parameters of the Gaussian distribution, such as variance or standard deviation.

As you may recall from our previous discussions, when a Gaussian distribution has a mean of zero, as it does in this case, you can alter the standard deviation by adjusting the amplitude. Let's add a control for this. Press **`Ctrl + F`** to open the command finder and search for "Range." Select "QtGyRange," double-click, and label it "NOISESTD" for noise standard deviation. We'll set the default value to 1, with a start value of 0, a stop value of

3, and increments of 0.1. Before proceeding, let's ensure that the amplitude is set to "NOISESTD."

Now, when we execute the flow graph, you'll see a range control on the interface, allowing us to observe the same behavior as before. Let's experiment by setting the range to 0. When you do this, the Gaussian source effectively outputs its mean, which is 0, so all 100,000 values fall into a single bin at 0. As we begin to increase the noise standard deviation, say to 0.1, you'll notice that the plot becomes more stable, with only a few bins receiving values.

(Refer Slide Time: 33:52)



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

Variable
ID: samp_rate
Value: 192k

Import
Import: np

QT GUI Range
ID: noisestd
Default Value: 2
Start: 0
Stop: 3
Step: 100m

Noise Source
Noise Type: Gaussian
Amplitude: 2
Seed: 0

Throttle
Sample Rate: 192k

QT GUI Histogram Sink
Number of Points: 100k
Number of Bins: 100
Autoscale: Yes
Accumulate: Yes
Min x-axis: -4
Max x-axis: 4

ID	Value
Imports	
import_0	import numpy as np
Variables	

As we continue to increase the standard deviation, say to 0.6, the Gaussian distribution becomes more apparent, activating more bins. In the lower bins, the probability of receiving values is low but not zero, so occasionally, you'll see values in these bins. These are what we might refer to as "rare occurrences." As we further increase the standard deviation, you'll notice that these rare occurrences become more frequent. Typically, most of the Gaussian distribution is concentrated within the range of -3σ to 3σ.

If we keep increasing the standard deviation, you'll observe that values start falling into bins beyond 1 or 2σ, or simply beyond 1 or 2 in this setup, since we are scaling the standard deviation. For example, if we increase the standard deviation to 2, the variance becomes 4, and the Gaussian distribution spreads over a broader range of values. In the context of a communication system, this would correspond to a scenario with higher noise levels, increasing the likelihood of symbols being corrupted or shifted out of their correct decision regions due to this noise.

Thus, we can effectively visualize a Gaussian distribution using a histogram. However, with higher standard deviations, the distribution of values across bins becomes less stable as the values are spread out more widely. To address this, you can enable accumulation. Let's try increasing the standard deviation to 2 and observe the effect. But there's a potential issue, if you accumulate values with a standard deviation of 1 and then change it to 2, the histogram will no longer accurately represent the distribution.

To avoid this, it's best to set the default standard deviation to 2 before visualizing the distribution. This ensures that the histogram accurately reflects the Gaussian distribution under the specified conditions.

When you perform this operation, you'll notice that the output stabilizes, and the resulting shape corresponds to the equation $e^{-\frac{x^2}{2\sigma^2}}$. To determine the value of σ, you can leverage the fact that the peak value is proportional to $k \times e^{-\frac{x^2}{2}}$, where k represents a constant. By finding this constant k, you can use the equation $k \times e^{-\frac{x^2}{2\sigma^2}} = 60{,}000$ to obtain a reliable estimate of the standard deviation, σ.

Now, let’s illustrate this with an example. Suppose we determine k to be 129,560. If we then consider a point where x equals -2.7, we can substitute these values into the equation,

$$129{,}560 \times e^{-\frac{2.7^2}{2\sigma^2}} = 52{,}100.$$

Solving this equation will give us the value of σ, which can subsequently be used to estimate the variance.

In this manner, the histogram proves to be a valuable tool for visualizing the characteristics of a random source, allowing us to effectively count the ranges of values it can assume and derive an estimate of its probability distribution. In future lectures, we will continue to employ histograms to calculate various statistics related to the values received by a communication system's receiver.

In this lecture, we have explored the use of histograms in GNU Radio. Specifically, we have examined the size of the histogram sink in terms of the number of values it can accommodate, the impact of bin selection, and how this tool can be used to visualize random variables such as noise samples or received symbols. Your visual inspection of the histogram will provide critical insights into the performance of your system, a method we will continue to refine and utilize in upcoming discussions.