**Digital Communication using GNU Radio**

**Prof. Kumar Appaiah**

**Department of Electrical Engineering**

**Indian Institute of Technology Bombay**

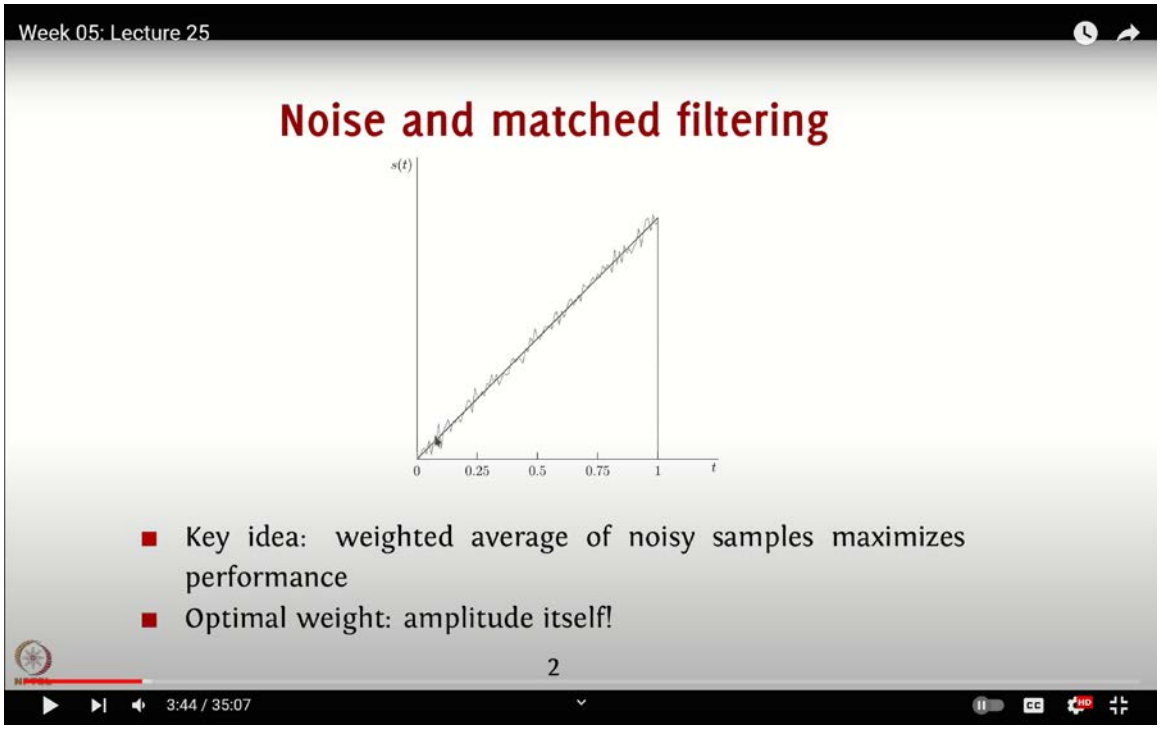
**Week-05**

**Lecture-25**

**Understanding Matched Filtering using GNU Radio**

Welcome to this lecture on digital communication with GNU Radio. My name is Kumar Appiah, and I am part of the Department of Electrical Engineering at IIT Bombay. In this session, we'll explore how noise and related concepts can be effectively managed using GNU Radio. Specifically, we'll examine the noise source block within GNU Radio, its various properties, and its behavior with both real and complex noise.

(Refer Slide Time: 03:44)




We'll also introduce the concept of matched filtering, which is a technique used to mitigate the impact of noise on different types of pulses, ultimately enhancing the efficiency of your

receiver. Our focus will be on analyzing the effect of matched filtering from the perspective of noise reduction.

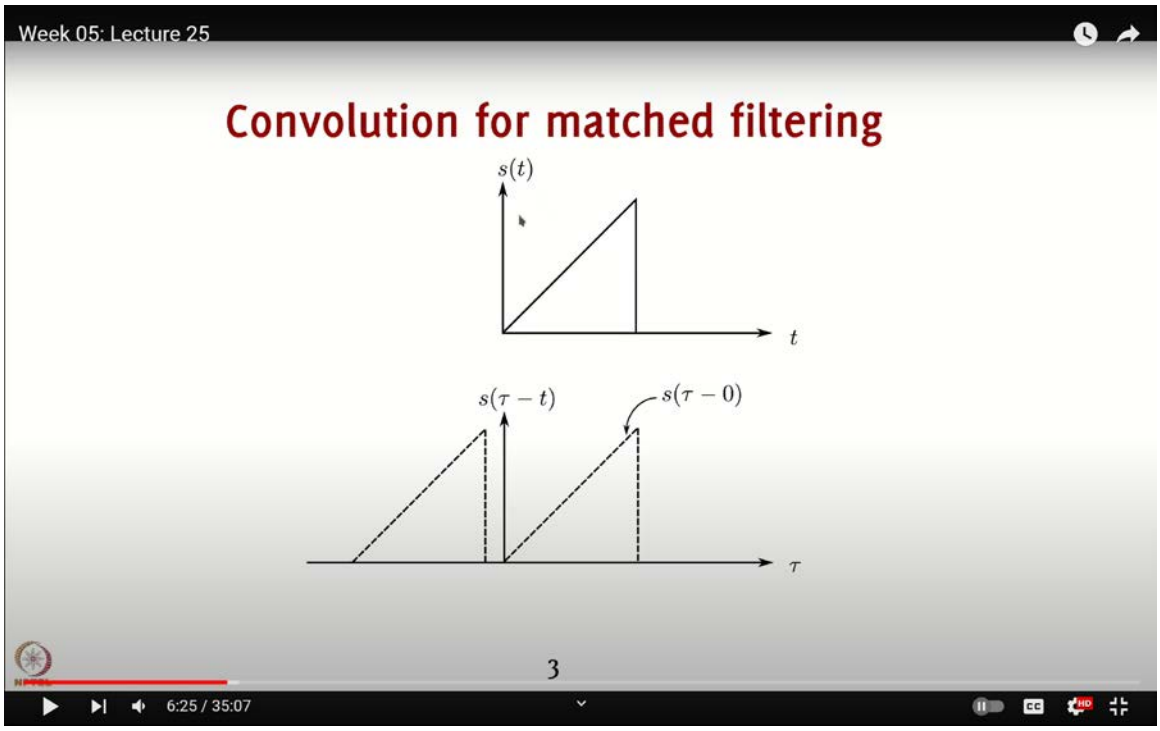
A key point to understand is that any signaling pulse, such as the partial ramp signal we will use as an example, is susceptible to noise. When this ramp signal is used to transmit data, noise can distort the signal, resulting in a wavy pulse at the receiver. Ideally, to determine the transmitted data, you would examine the point with the highest amplitude. However, if noise is particularly high at that point, it could lead to incorrect data interpretation.

You are also making a mistake by not considering all possible values in your analysis. The key idea is to account for all these values to accurately determine which amplitude was actually transmitted. For instance, if you send a ramp signal from 0 to 1, another ramp with an amplitude from 0 to 2, and so forth, you need to evaluate the entire duration to identify the exact value sent. Essentially, you should calculate the weighted average of the noisy samples rather than a simple average.

Why use the weighted average instead of just averaging? The reason lies in the amplitude of the ramp signal. When the ramp amplitude is high, even if there is some noise, it will not significantly deviate the signal from its true value. However, at lower amplitudes, such as near 0, noise can cause substantial shifts. For example, if the amplitudes are 0, 1, 2, and 3, and these points are very close together, noise might push the signal to an incorrect amplitude, leading to more errors.

In this context, each measurement represents the actual value but with varying signal-to-noise ratios (SNRs). According to matched filter theory, the filter or the combination of these values should be weighted such that lower SNR points receive less weight, while higher SNR points receive more weight. To maximize the signal-to-noise ratio by optimally combining these measurements, you should weight each value by the amplitude of the ramp. This means you multiply each noisy sample by the corresponding amplitude, then sum these products. Essentially, this translates to multiplying the noisy ramp by the clean ramp and averaging the result. The optimal weighting is thus proportional to the amplitude of the ramp signal.

(Refer Slide Time: 06:25)



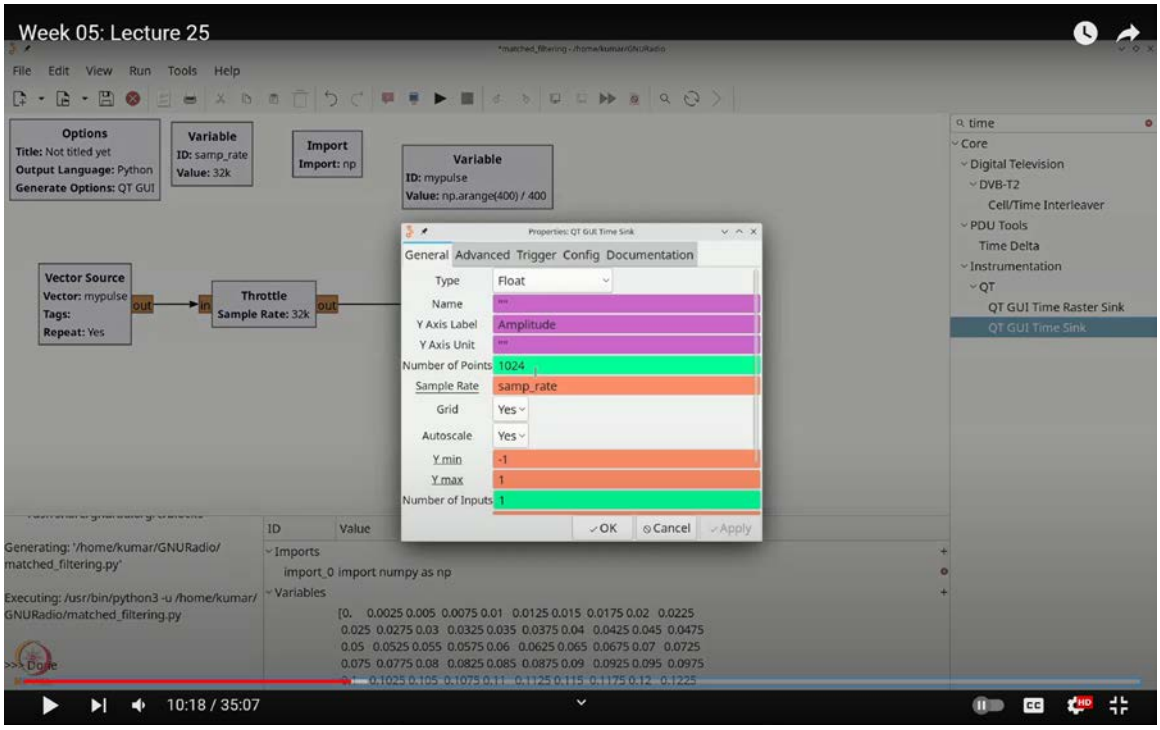

This is why matched filter theory indicates that this approach provides the optimal solution for maximizing the signal-to-noise ratio (SNR), as it effectively accounts for all values with their respective SNRs. For example, if you were to use a rectangular pulse, which we will explore shortly, you could weight all points equally since each point would have the same SNR.

Another important aspect to consider is the implementation of matched filtering, which essentially involves correlation performed through convolution. So how do we achieve this? Let's use our signal S(t) as an example. To perform convolution, we need to consider S(t - τ) and multiply it accordingly.

However, S(t - τ) will be a flipped version of the original signal, specifically, it will be a flipped ramp, going from increasing to decreasing. Instead, we should look at S(τ - t), which aligns with the desired signal. If you take τ as the reference, S(τ - t) represents the signal shifted by t along the time axis. Setting t to zero gives you S(τ), which matches the signal you want. The overlap of S(t) with S(-t) sampled at zero represents the convolution of S(t)

with S(-t). In essence, this means you take S(t), flip it, convolve it with S(t), and observe it at a zero offset. While a causal implementation may introduce additional delays, this is the conceptual basis for implementing correlation as convolution.

(Refer Slide Time: 10:18)




Now, let’s briefly explore this in GNU Radio. We will set up a straightforward example of matched filtering to send pulse amplitude modulated (PAM) data and then recover it using matched filtering in the presence of noise. Before diving into this, let's look at how to create a specific pulse and retrieve data from it. To streamline the process, I will use some NumPy functions. To begin, I’ll press Ctrl-F (or Cmd-F) and type **`import`**, then double-click to enter **`import NumPy as np`** for convenience, so we don’t need to repeatedly type **`NumPy.`**

For simplicity, let’s start by using a vector source. To do this, press Ctrl-F (or Cmd-F) and select "vector source." We'll use a real simulation, so we’ll set the data type to float.

Next, we need to incorporate our pulse. I’ll use a relatively long pulse for easier visualization and to apply the pulse in multiple locations. Instead of hardcoding the pulse

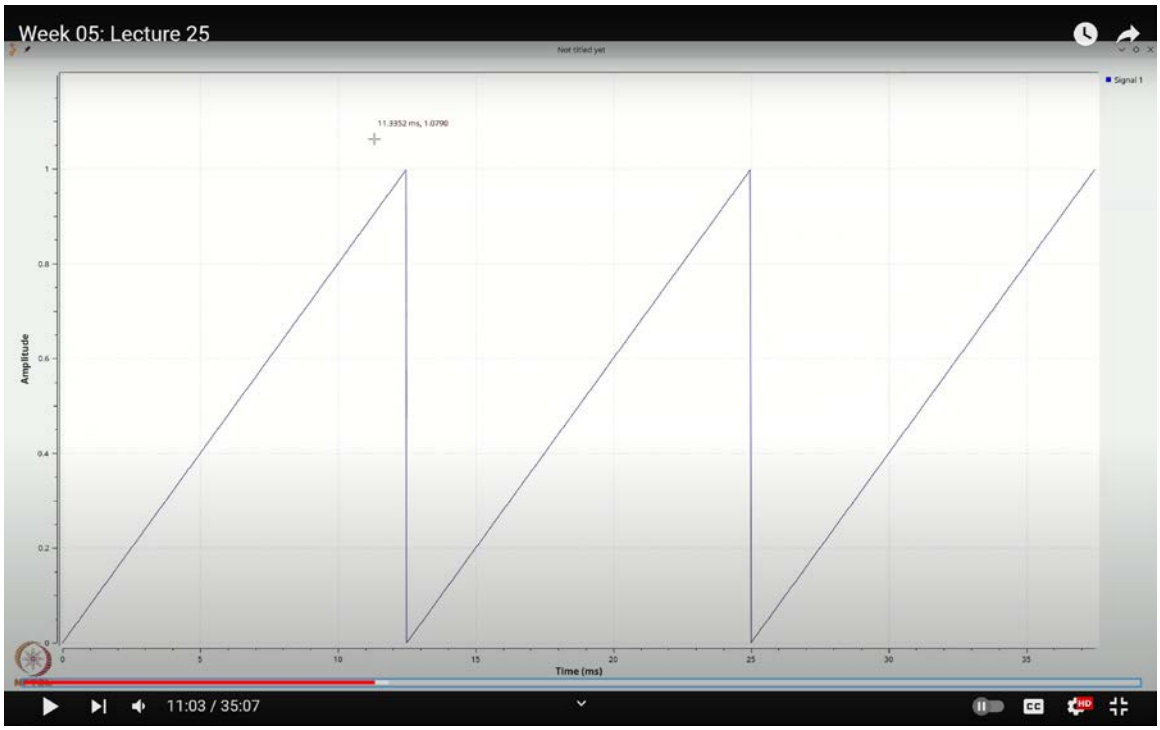
within the vector source, we'll use a variable. Press Ctrl-F (or Cmd-F), type "variable," and double-click to create a variable. Name this variable "my_pulse." The pulse we'll use is a ramp.

To generate the ramp, we'll create a sequence of numbers using NumPy. I'll use **`np.arange`** to create a ramp from 0 to 1 over 400 samples. This means the ramp will be from 0 to nearly 1, with the sample range set to 399, as **`np.arange`** produces numbers from 0 to 399. This should be sufficient for our needs.

For context, with a sample rate of 32 kHz, each sample is spaced 1/32,000 seconds apart. So, a 400-sample ramp will last about 400/32,000 seconds, which simplifies to 1/80th of a second. Let's confirm this calculation to ensure accuracy.

We'll now use "my_pulse" as the variable name for our pulse. Add a throttle by pressing Ctrl-F (or Cmd-F), select "throttle," and double-click to add it. Set this to float.

(Refer Slide Time: 11:03)




Next, we'll add a time sink. Press Ctrl-F (or Cmd-F), type "time," and select the "qt-gui

time sink." Double-click to add it and set it to float. This time sink will help us visualize our pulse and verify that it’s correct.

If I execute this flow graph, the resulting pulse appears as follows. To enhance clarity, let’s adjust the time sink settings for convenience. Double-click on the time sink to open its properties. Enable the grid and set the scale to make the visualization more manageable. For consistency, let’s set the scale to a multiple of 400, which will make it static. For instance, set it to display 1200 points.

With this configuration, you can now observe the pulse. To understand the spacing, note that the pulse starts at 0 and the next noticeable point is at 12.5 milliseconds. Performing the calculation, 400 samples divided by 32,000 samples per second equals 4/320 seconds, which simplifies to 1/80 seconds. Converting this to milliseconds, 1/80 of a second is equivalent to 12.5 milliseconds. This spacing represents a sequence similar to sending continuous '1111' values. Our pulse ramps from 0 to 1, then descends, and repeats this pattern.

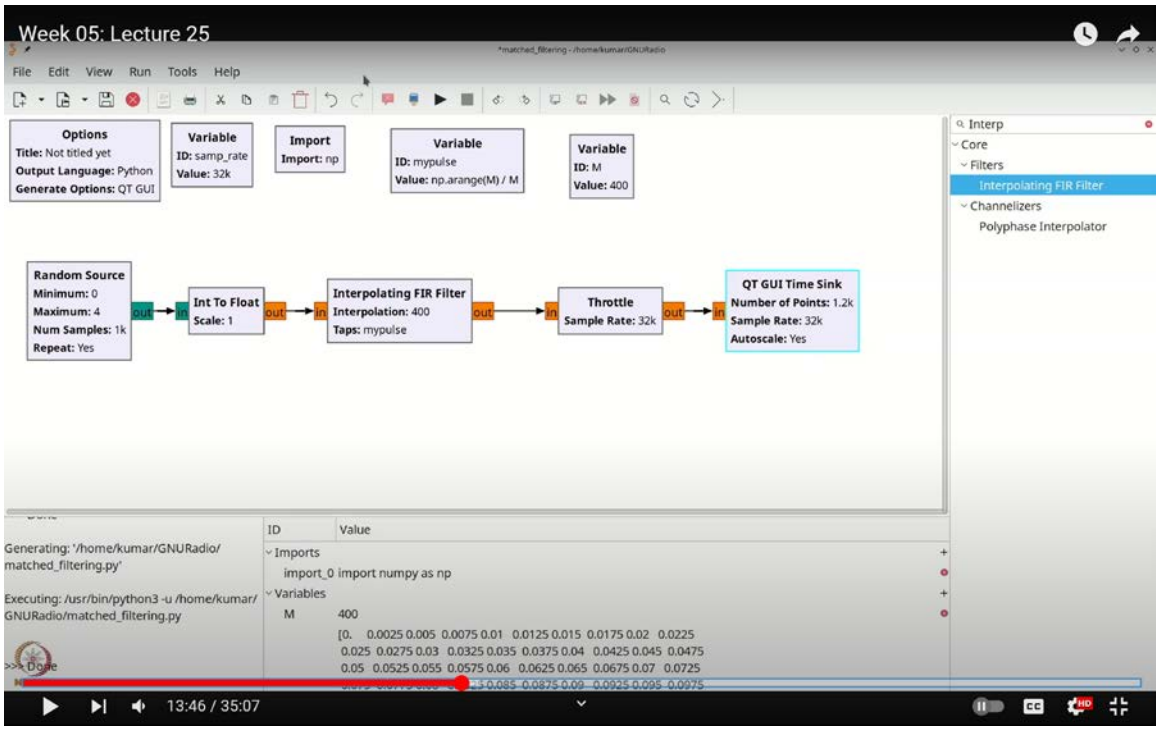
Now, let’s consider matched filtering. Our goal is to convolve this pulse with its flipped version. Notice that I specifically chose a ramp because it is an asymmetric pulse. For symmetric pulses like a sinc function, a root raised cosine, or a rectangular pulse, convolution with itself is sufficient. However, with an asymmetric pulse like this ramp, we need to flip the pulse for proper convolution. Let’s proceed with this exercise: we will take some data and perform the convolution to apply matched filtering.

I will now remove the vector source from the flow graph. To replace it, use Ctrl-F or Cmd-F to search for "random," and add a Random Source block. Configure this Random Source to produce four possible values: 0, 1, 2, and 3. We will treat these values as floats, so we need to convert integers to floats. For this, use the Int to Float block.

Search for "int to float" using Ctrl-F or Cmd-F, select the block, and connect it to the Random Source. Next, we need to apply our pulse to this data. I’ve chosen a pulse length of 400 samples, which I’ll define as a variable to avoid any issues. Let’s create a variable named **`M`** with a value of 400. Use Ctrl-F or Cmd-F to search for "variable," double-click

to create a new variable, and set its name to **`M`** with a value of 400.

(Refer Slide Time: 13:46)




Now, we need to insert zeros between these samples and convolve them with the pulse. For this purpose, we'll use the Interpolating FIR Filter. Search for "interpolating" with Ctrl-F or Cmd-F, select the Interpolating FIR Filter block, and configure it for float-to-float operations.

We will set the interpolation amount to 400 and use the pulse defined by our variable as the filter taps. What this will yield is a sequence of ramps, each multiplied by one of the values: 0, 1, 2, or 3, appearing one after another. Initially, this number of points might be sufficient, but let’s test it and adjust if necessary.

If we increase the number of points to, say, 12,000, you will see a clearer result. Stop the simulation, and you should observe a sequence where, for example, a 0 is followed by a 2, then a 1, a 2, another 1, and a 3, and so on. To better understand this, let’s measure the gap between successive peaks. You should find that the gap is consistently around 12.5 milliseconds, which aligns with the pulse duration we designed. Each ramp in the

modulated data is indeed 12.5 milliseconds long.

The next step is to retrieve the original data. To do this, we need to apply the reverse operation using a Decimating FIR Filter. This filter will process the output and help recover our random source values. Search for "decimating" using Ctrl-F or Cmd-F, select the Decimating FIR Filter block, and configure it for float-to-float processing. Set the decimation factor to the variable **`M`**. It’s crucial to note that, for the decimating filter, we should initially set the decimation factor to 1.

We are not going to apply any decimation in this setup. Let’s proceed by setting the taps to **`mypulse[::-1]`**. In Python notation, **`[::-1]`** reverses a sequence. This step is crucial because convolution inherently involves reversing the sequence, and for correlation, you need to reverse the pulse.

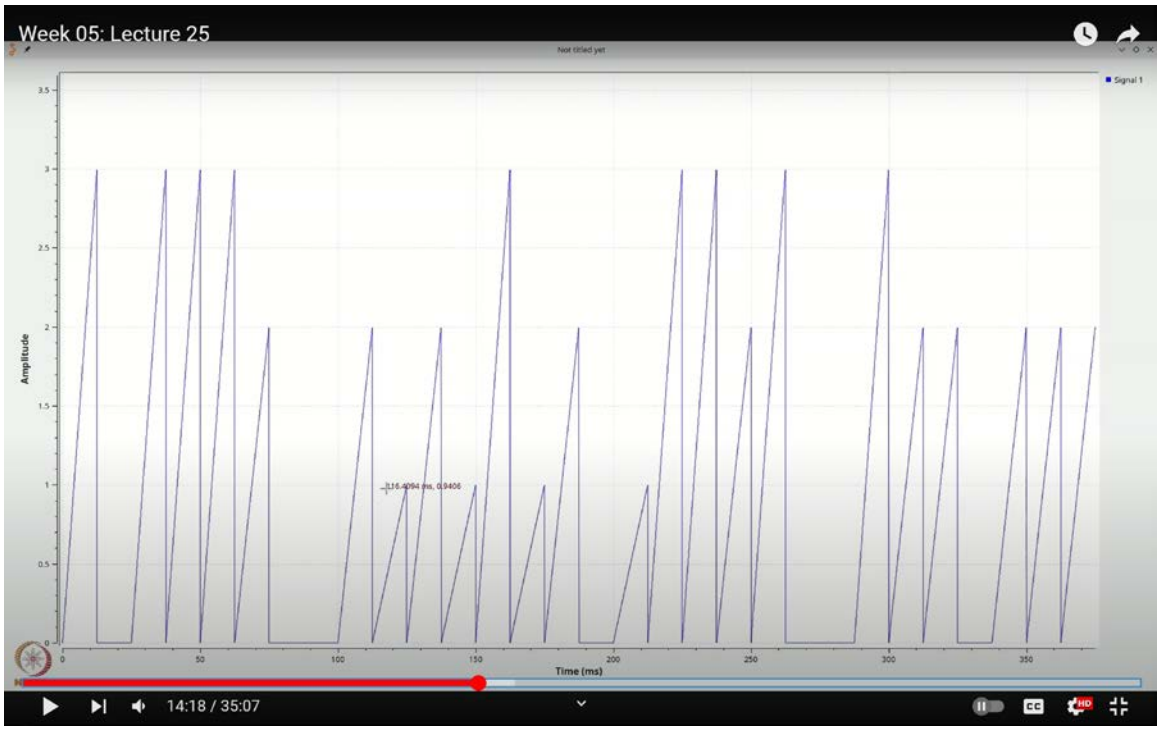
Next, let’s double-click to configure this filter and name it **`m`**. Now, let’s add another QT GUI Time Sink to visualize the results. Double-click this new sink and name it **`interpolated`**. Then, add yet another QT GUI Time Sink by pressing Ctrl-F (or Cmd-F), search for "time," and select it. Double-click this one and name it **`decimated`**. While it's not technically decimated yet, we’ll adjust that shortly. Set it up with a grid and enable auto-scaling, and click OK.

Now, connect the output of the Interpolating FIR Filter to the QT GUI Time Sink labeled **`interpolated`**. You might see some peaks appearing in the output. These peaks result from convolving the pulse with the FIR filter, which produces these prominent features.

Notice the discrepancy in time scales: the time scale on one sink ranges from 0 to 350, while the other ranges from 0 to 30. This difference arises due to the factor multiplication and upcoming decimation. This is primarily because we have chosen different time scales for each sink. Keep this in mind as you interpret the results.

Here, I have chosen a value of 12,000 for one setting and 1,024 for another. Ideally, to ensure consistency, especially since we are decimating by a factor of 400, I should adjust this to 12,000 divided by 400. However, for simplicity, I am maintaining the current number of points. This choice is not crucial for our demonstration.

(Refer Slide Time: 14:18)

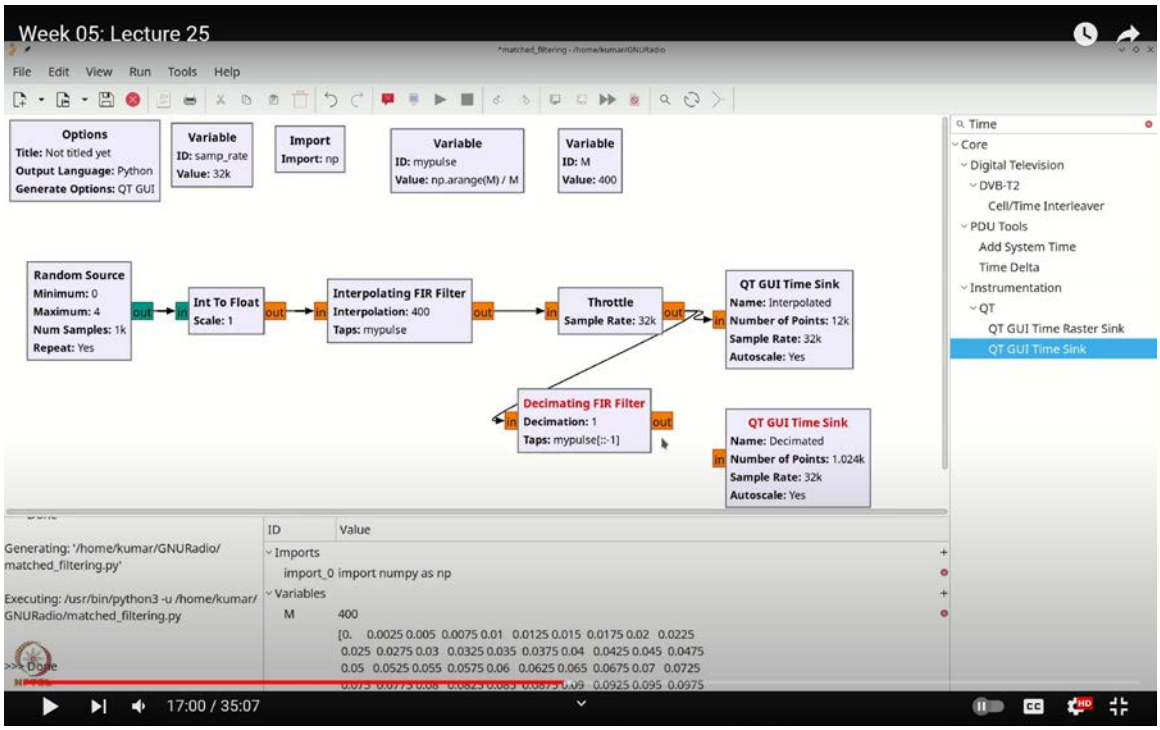



Now, if I stop this and examine the values, let's extend the length to about 3,000 points and run the simulation again. Upon stopping, you will observe values close to 400, 138, and 267. These values are approximately 0, 130, 266, and 400. The reason for these values is that when you overlap the original pulse with the decimating filter, what you're effectively calculating is the integral or autocorrelation of the function evaluated at zero. In essence, you are computing the integral of the square of the signal.

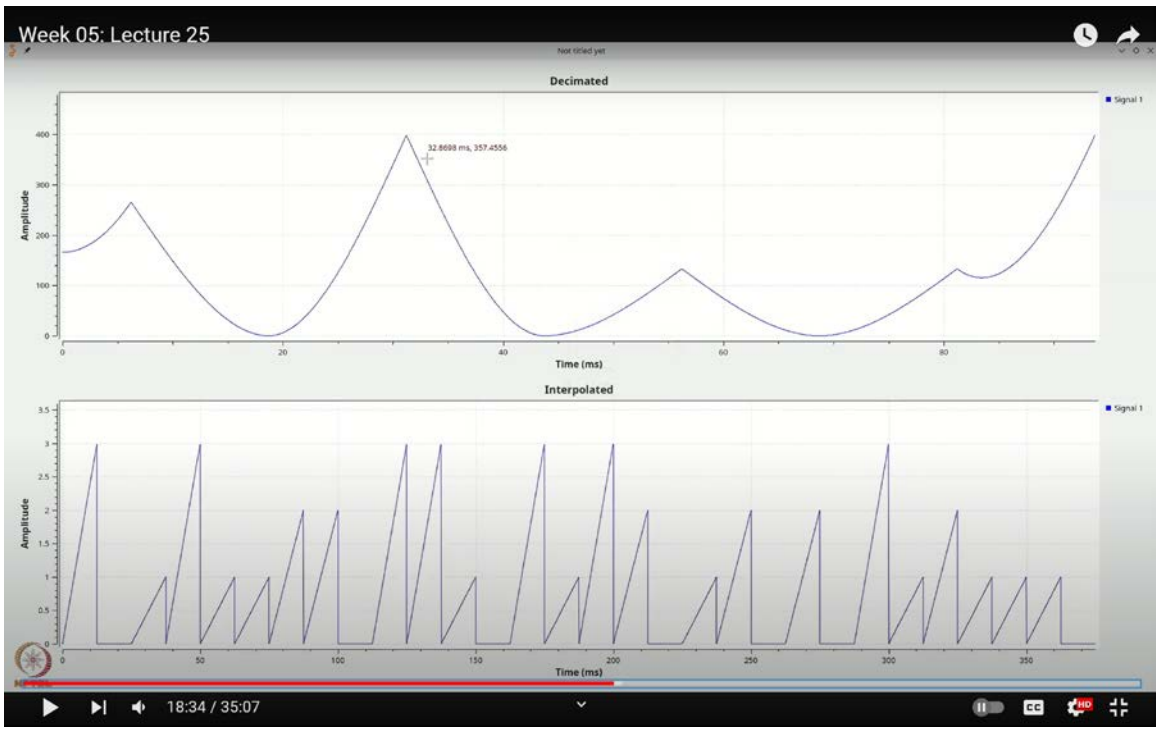
To verify this, let's make a temporary adjustment. We'll disconnect the current setup and add a Vector Source. Press Ctrl-F (or Cmd-F) and search for "Vector Source." In the Vector Source, input our pulse, specify it as a float, and name it "my pulse." This operation will essentially involve multiplying the pulse by itself and shifting it in time.

Upon running this setup, you'll see a pattern emerge. To stabilize and make this result more consistent, double-click on the Vector Source and adjust the length. Since the pulse length is around 400, increasing this length to 800 should provide a more stable result.

(Refer Slide Time: 17:00)

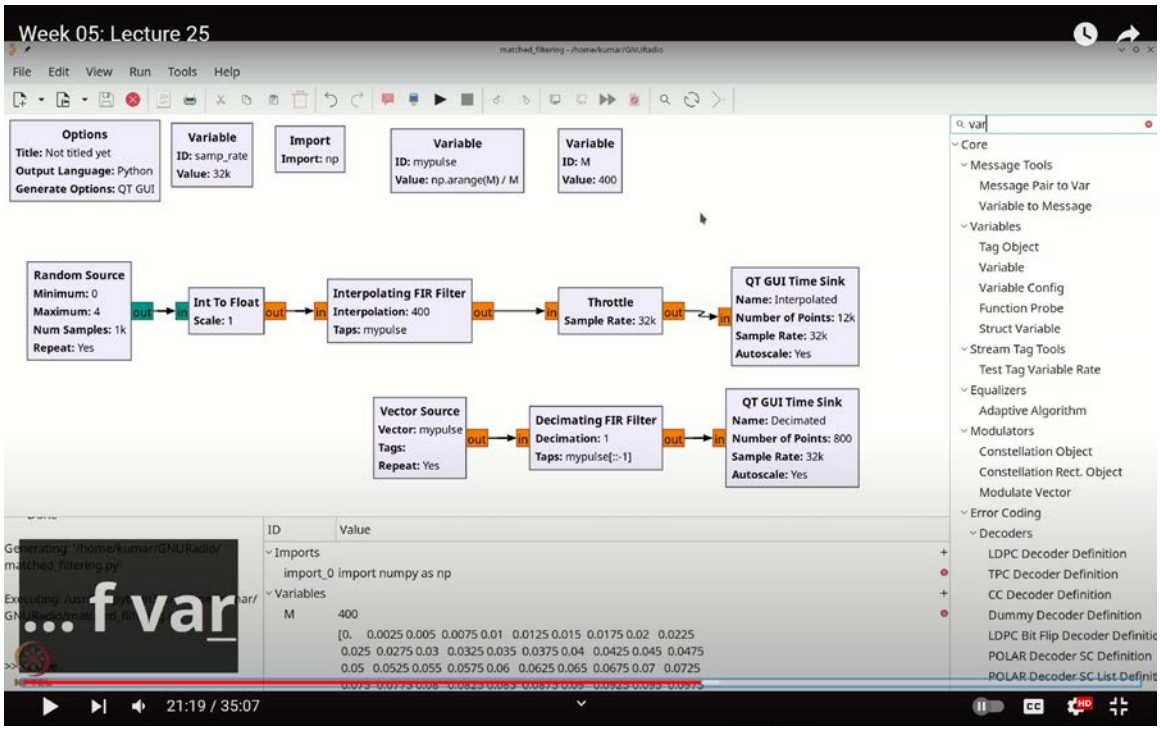



(Refer Slide Time: 18:34)



So, let's take a closer look at this peak value. If I zoom in here, you'll see it's around 133 or so. But why is this value 133? What does it signify? This peak value arises because we are overlapping the ramp with itself. To recap, overlapping the ramp with itself means that we're essentially multiplying each sample of the ramp by itself and summing up the results. Consequently, this peak value represents the sum of the squares of the coefficients of **`myPulse`**.

(Refer Slide Time: 21:19)




If you're skeptical, let's perform a quick verification. Press Ctrl-F (or Cmd-F), type "variable," and select it. Double-click to add a variable block, and then use **`np.sum(myPulse ** 2)`** to calculate the sum of the squares of the elements in **`myPulse`**. This operation yields a result of approximately 132.834. If you compare this with the output peak we observed, it should be very close to 132.8, which aligns with our calculation. The slight difference is due to the exact positioning of the peak, but it confirms our result.

To recover the original samples from this, you can scale the result. Simply add a scaling factor by dividing by **`np.sum(myPulse ** 2)`**. Once you execute this adjustment, the

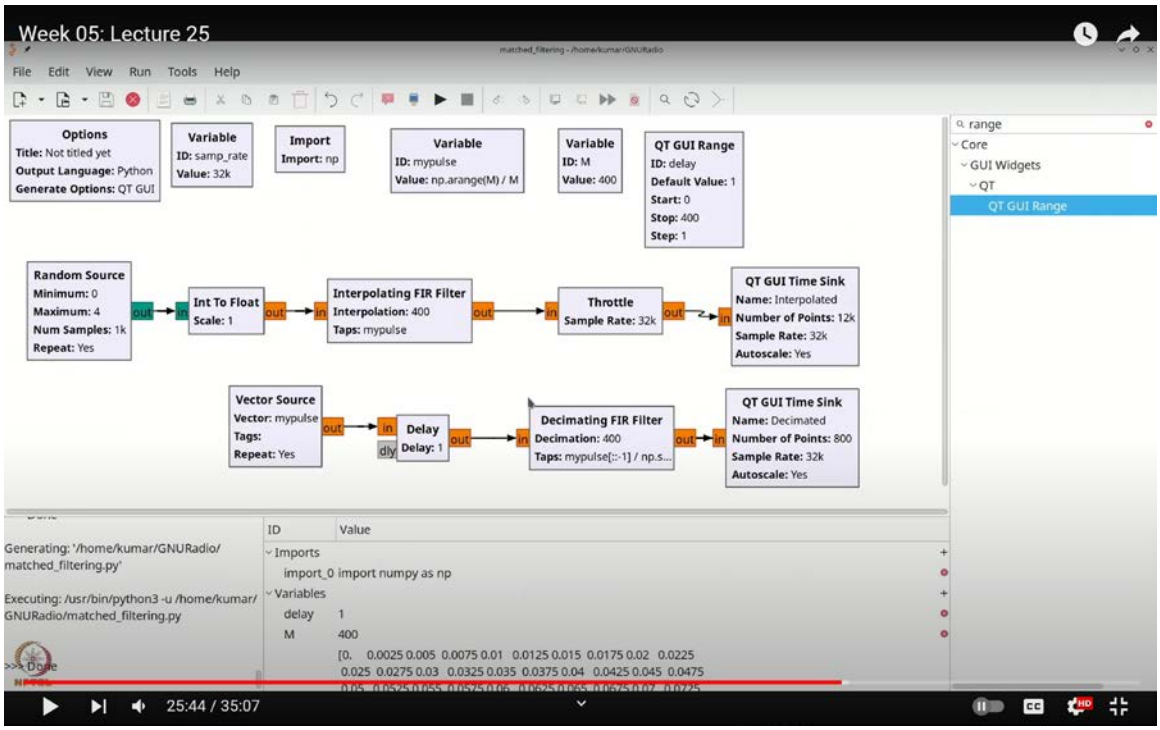
result should normalize to 1.

Our next step involves adding noise and then recovering the original pulse. But before we proceed, observe that there's a periodic pattern here: you see a value of 1, followed by a decrease, then another 1, and so on. This pattern is a result of periodic convolution effects.

We need to address this issue carefully. Before moving forward, let's incorporate decimation into our process. Set the decimation factor to 400 or **`m`**. When applying this decimation, you should ideally get a value very close to 1. However, you might notice a slight discrepancy; it should be exactly 1, given the precision we achieved earlier.

To correct this, one approach is to introduce a sample delay and observe if that resolves the issue. Alternatively, you might adjust or remove the sample delay to see if it improves accuracy. For this, let's add a delay block. Press Ctrl-F (or Cmd-F), type "delay," and select the delay block. Double-click to configure it as a float, and set it as a variable delay.

(Refer Slide Time: 25:44)



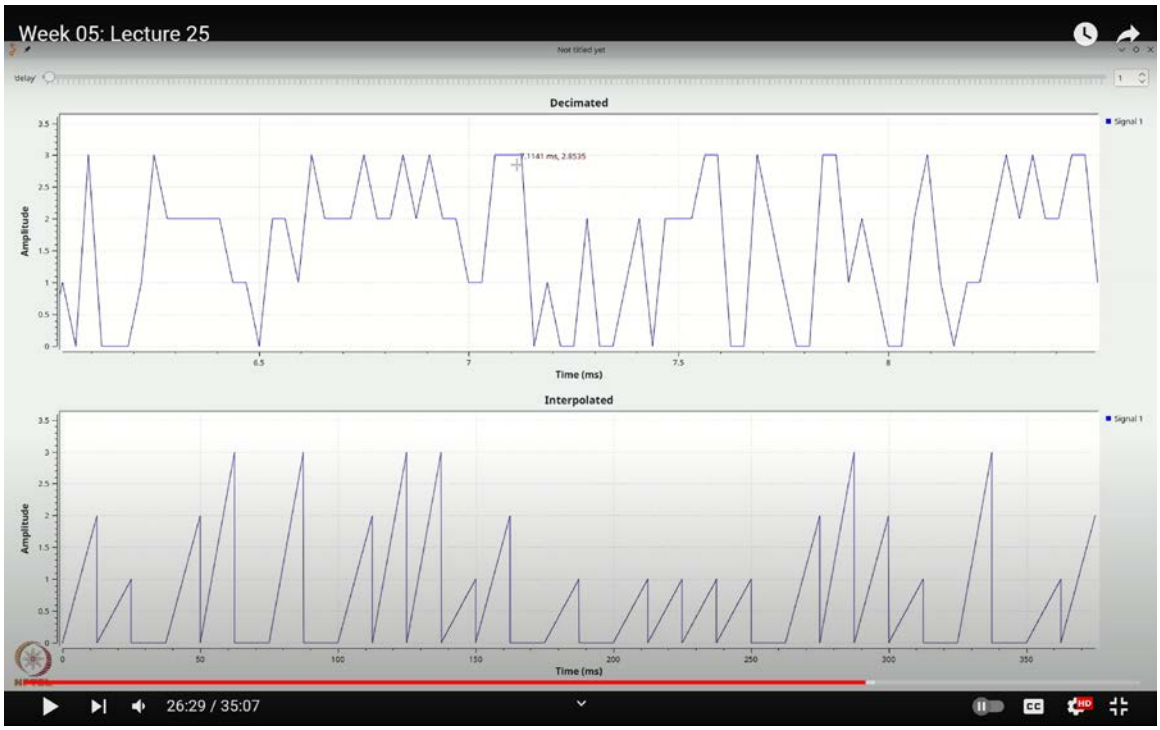

Next, connect this delay block and set up a range. Press Ctrl-F (or Cmd-F), type "range,"

and add a Qt GUI Range block. Set the ID as "delay" and configure it to range between 1 and **`m`**, with a default value of 0. The step should be 1 and the type should be set to int.

Running this flow graph should yield a result close to 1, though not precisely. By adjusting the delay, you can change the sampling point. As you approach a delay of 400, the result should get closer to 1 again. Setting the delay to 1 should match the exact value.

The exact match of this value is achieved when the delay is correctly set. Without a delay, there is a slight offset. This offset occurs because, without adding a delay, when you convolve my pulse with itself, the resulting peak begins at 1, which is the first point. To accurately capture this point, you need to add a delay of 1. If you skip this delay, the sampling point will be slightly misaligned. This is why adding decimation requires including a delay.

(Refer Slide Time: 26:29)




To correct this, simply add a delay of 1. For our current setup, we will set this delay with a default value of 1. Now, let's proceed to recover our original data. We'll remove the vector source and connect the output to the appropriate point in the flow graph to see if we can

retrieve the original data.

When you run this flow graph, you'll need to allow some time for the samples to accumulate. Once the wait is over, you should observe that the samples are coming through. The values will appear as 0, 1, 2, 3, and so on. This delayed observation is due to the multi-rate nature of the flow graph: this section operates more slowly because it processes 400 samples before decimating.

The reason it takes time to obtain the samples is due to the buffer needing to be fully populated before displaying the results. To facilitate direct comparison, let's adjust the number of samples to 1000. Additionally, we'll add another input and connect the output of the **`int to float`** block directly here. This will allow us to verify that everything is functioning correctly.

Once we execute this flow graph, we need to wait for the buffer to fill up before the samples are displayed. After the process completes, we'll pause and stop the execution. Upon zooming in, you'll notice an interesting effect: there is a noticeable delay. The pulses themselves appear very similar, with the red pulse representing the original and the blue pulse showing the recovered version.

While the samples seem reasonably accurate, there is an observable delay. This delay arises because your ramp is causal. When you convolve the ramp with itself and sample, the resulting sampling point is shifted by one sample. Referring back to our presentation, you'll see that this sample is one symbol away from the start of the pulse. Therefore, to compensate for this additional delay, you need to introduce an extra delay of 1 in your setup.

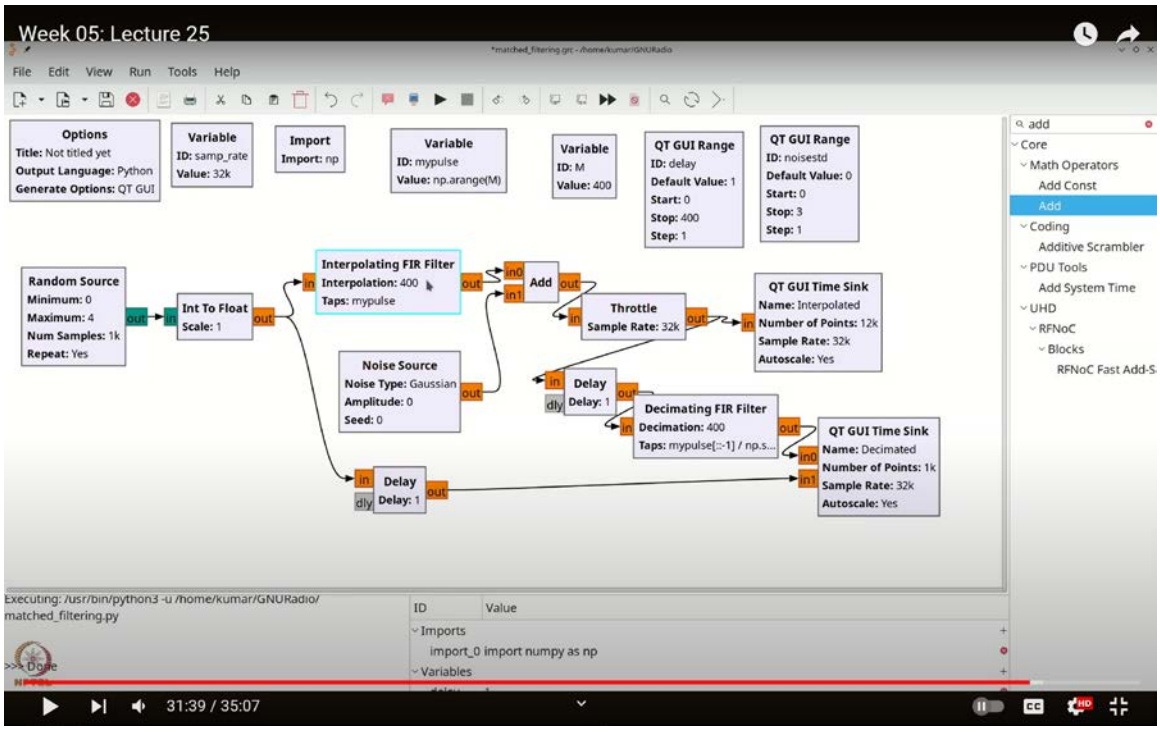
To address the delay issue, we'll add an additional delay to the flow graph. First, remove the existing connection and reposition the delay block. Copy the delay block using **`Ctrl+C`** and **`Ctrl+V`**, then reconnect them to facilitate a clearer understanding of the process. Arrange these blocks accordingly and execute the flow graph. After allowing the plot to stabilize, you should see that the delay problem is resolved, effectively addressing the causal nature of the matched filter.

With the delay issue resolved, we now turn our attention to analyzing the impact of noise. Noise influences performance through the Signal-to-Noise Ratio (SNR), which determines how noise affects the system. In our waveform, values range between 0 and 1200 due to the ramp's construction. We need to observe how noise affects the matched filter's performance, particularly how it handles noise through averaging.

To incorporate noise, add a noise source between the interpolating FIR filter and the decimating FIR filter responsible for the matched filtering. Begin by adding a range for the noise standard deviation.

Press **`Ctrl+F`** or **`Command+F`**, type "range," and select the QTGUI Range block. Name it **`noise_std`** and set it to range from 0 to 3. This range is suitable for our purposes. Next, add a noise source to the flow graph. Remove the previous noise component, then search for "noise" using **`Ctrl+F`** or **`Command+F`** and insert the noise source block.

(Refer Slide Time: 31:39)

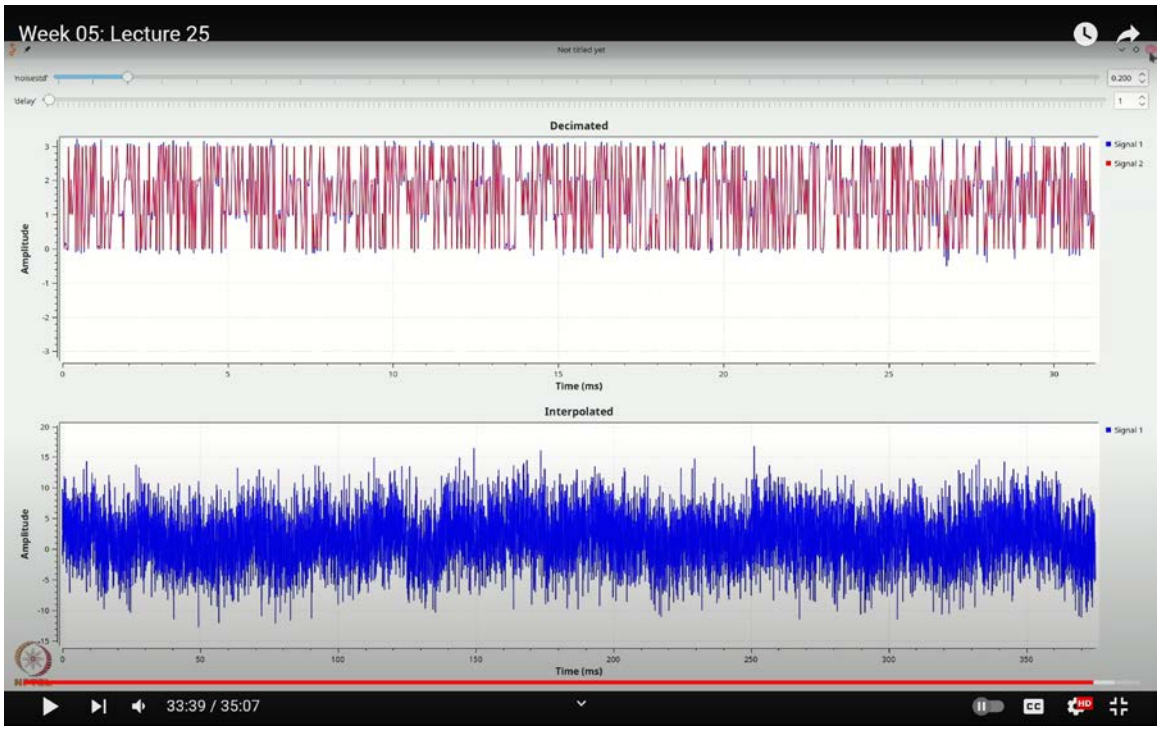



To normalize the noise with respect to the pulse energy, multiply the noise standard deviation by the square root of the pulse energy. Use the formula

**`np.sqrt(np.sum(my_pulse ** 2))`** to achieve this normalization. Set the noise source block to float, then add an Add block by searching "add" with **`Ctrl+F`** or **`Command+F`**, and configure it to float as well.

Connect the noise source to the Add block. Now, the setup is complete. Note that the impact of noise may not be immediately visible.

(Refer Slide Time: 33:39)




Let's set the noise level and observe its effects. You'll notice that the noise significantly impacts the signal, making its presence quite clear in the results. This demonstrates a high noise level.

To refine our noise analysis, let's adjust the noise step size to 0.1 instead of the previous setting. Execute the flow graph with this new step size. Although a noise level of 0.2 seemed high due to normalization, 0.1 should be more appropriate. After running the graph with the updated setting, you'll see that the noise impact is still evident, but the averaging process effectively mitigates it. Despite the waveform becoming less discernible, the matched filter manages to average out the noise substantially, demonstrating its capability

to recover the signal even in the presence of significant noise.

Next, let's test the system with a different pulse type, a rectangular pulse. For this, use **`np.ones`** to create a rectangular pulse consisting of 400 ones. When you examine this pulse, you'll see a series of rectangles. Adding noise to this rectangular pulse will show a notable effect of the noise on the signal. Let's see the outcome with this new pulse configuration.

On the right side of the display, you can observe variations caused by noise, though these variations remain relatively small, allowing you to largely discern the signal.

However, if you increase the noise level, the signal becomes much harder to distinguish. Despite this, the averaging effect of the matched filter still helps to reveal the signal to some extent. This demonstrates how the matched filter performs a weighted averaging of waveform samples, even in the presence of noise, to provide the clearest possible result. This ability to effectively process signals amidst noise is a fundamental aspect of the matched filter's design as a correlator.

In this lecture, you've seen how the matched filter operates and the effects of causal delay through a GNU Radio experiment. Using GNU Radio, you simulated noise effects, added noise to signals, and examined how this noise impacts signal clarity. The matched filter's role in combining received symbols, affected by noise, to recover the original signal with minimal error was also highlighted. We will delve deeper into measuring symbol error in upcoming lectures. Thank you for your attention.