Digital Communication using GNU Radio

Prof. Kumar Appaiah

Department of Electrical Engineering

Indian Institute of Technology Bombay

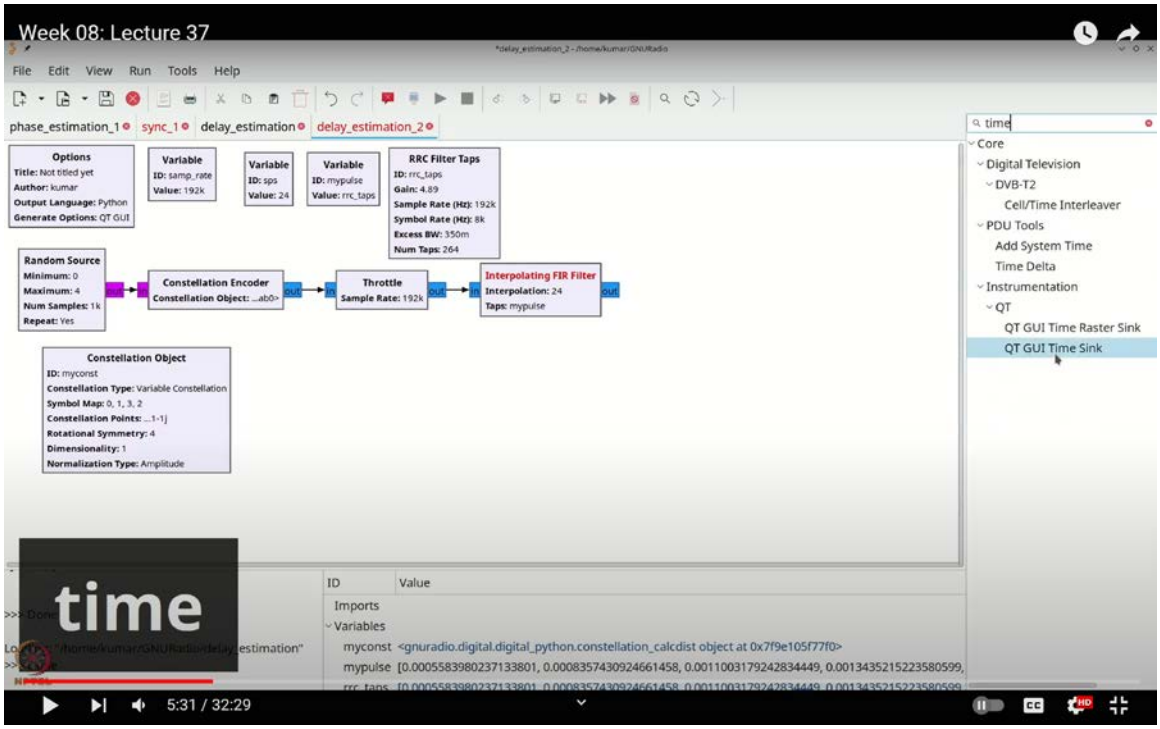
Week-08

Lecture-37

Maximum Likelihood Delay Estimate for Multiple Symbols 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 this lecture, we will continue our exploration of delay estimation using GNU Radio.

(Refer Slide Time: 05:31)



Previously, we confirmed that the maximum likelihood estimate of delay, when dealing with a single signal s(t), can be obtained by identifying the peak location in the matched filtering process. However, when multiple symbols or signals are involved, we encounter a combination effect. For instance, if you use a root-raised cosine pulse, the previous

analysis will no longer be directly applicable, and adjustments are necessary.

In this session, we will use simple template functions, such as a ramp and examples based on root-raised cosine pulses, to measure delays. We will relate this to the theoretical discussions and practical adjustments required for accurate delay estimation, even in the presence of noise and more complex modulation systems.

Let’s begin our second delay estimation exercise. We will follow a similar approach to what we've used before, sending an 8k symbols per second data stream, and examine the effects of delays. First, we’ll set the sampling rate to a convenient value of 192,000.

Let's get started. First, we'll add our random source. So, press **`Ctrl+F`** or **`Cmd+F`**, type **`RND`**, and select the random source block. We'll set it to output bytes and adjust the maximum value to 4.

Next, we need to add the constellation encoder. Press **`Ctrl+F`** or **`Cmd+F`**, type **`constellation encoder`**, and select it. The constellation encoder requires a constellation object, which we’ll set up shortly. Press **`Ctrl+F`** or **`Cmd+F`**, type **`CONST`**, and choose the constellation object. By default, this object is set to a QPSK constellation. We’ll name it **`myconst`**, and that will be our constellation setup.

Now, let’s add a throttle to control the flow rate. Press **`Ctrl+F`** or **`Cmd+F`**, type **`throttle`**, and add it to the flow graph.

We also need to introduce oversampling systematically. To do this, we’ll define our parameters. As mentioned, we want 8000 samples per second. Press **`Ctrl+F`** or **`Cmd+F`**, type **`variable`**, and add a variable block. Double-click it and name it **`SPS`** (samples per symbol). Since we want a sample rate of 8000, set it to **`samrate / 8000`**, which gives us 24.

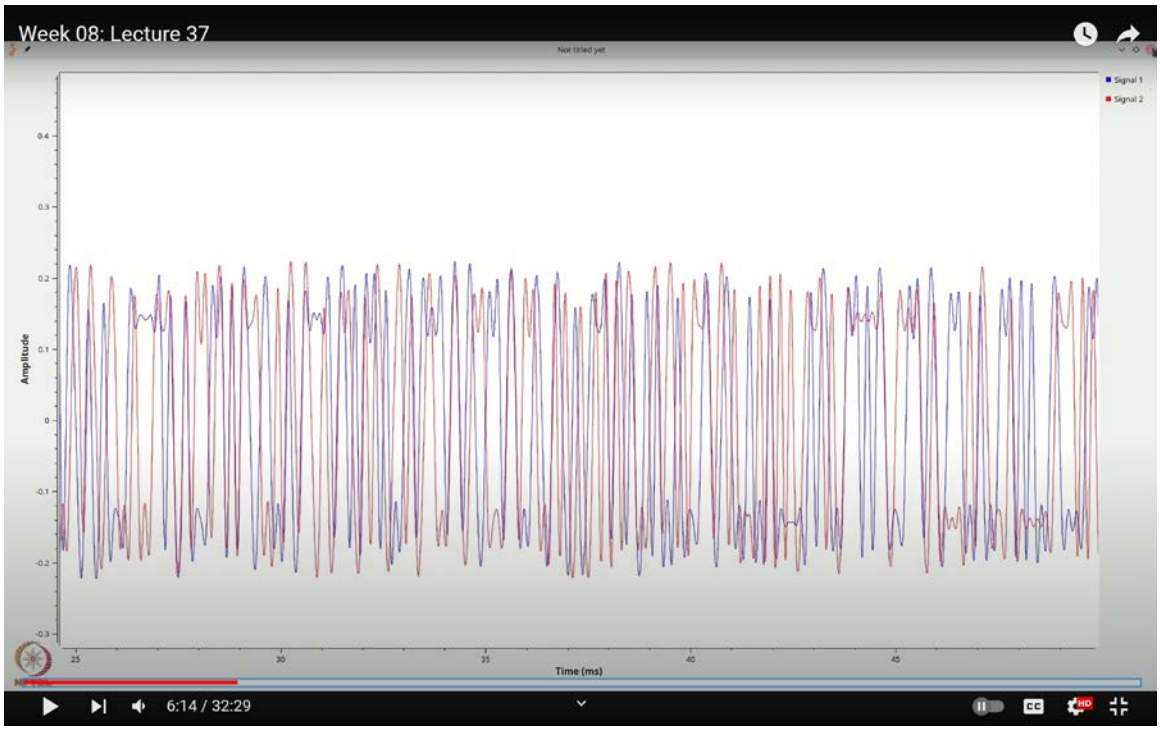
Next, we’ll create a variable for the pulse, named **`mypulse`**, so we can easily swap it out if needed. Press **`Ctrl+F`** or **`Cmd+F`**, type **`variable`**, and add another variable block. Double-click it and name it **`mypulse`**.

For **`mypulse`**, we need to define a pulse shape. Let’s use a root-raised cosine (RRC) pulse.

We'll create the RRC taps. Press **`Ctrl+F`** or **`Cmd+F`**, type **`RRC filter taps`**, and add the corresponding block. Name this **`RRC_taps`**.

For the gain setting, I'm going to adjust it to the square root of **`SPS`**, which is the square root of 24. The symbol rate is set to 8000, and my RRC taps are now ready. Next, we'll add an interpolating FIR filter. Press **`Ctrl+F`** or **`Cmd+F`**, type **`interpolating FIR filter`**, and add it to the flow graph. This filter will use the **`SPS`** interpolation, which corresponds to the sample-to-symbol ratio, and **`mypulse`** as the pulse shaper. We're all set up now.

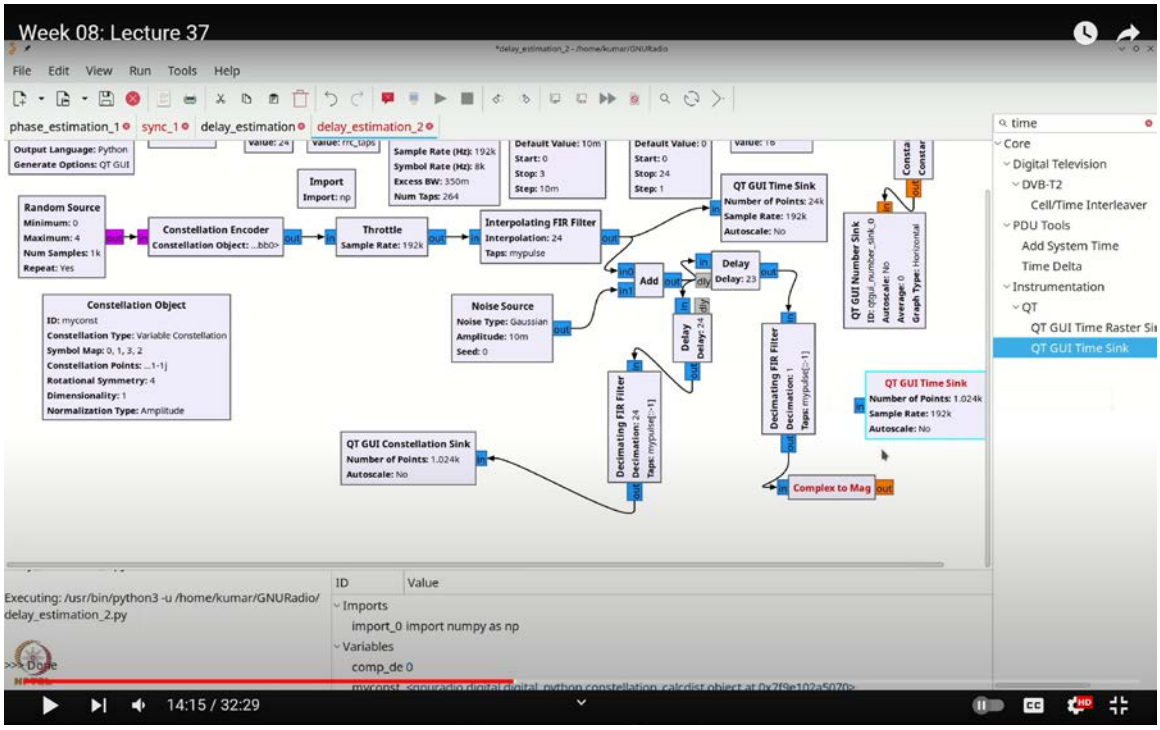
(Refer Slide Time: 06:14)



To visualize this, we'll add a GUI time sink. Press **`Ctrl+F`** or **`Cmd+F`**, type **`time`**, and select the QTGUI Time Sink. Place it in the flow graph. When you run the flow graph, you'll see a moving display, but I want this to appear as a static image to match the 1000 samples. To achieve this, I'll set the number of points to 1000 times the length of **`mypulse`** times **`SPS`**, which should yield a nice static plot of the upsampled symbols. I'll also add a grid for better visibility.

Next, let's introduce noise. Press **`Ctrl+F`** or **`Cmd+F`**, type **`add`**, and select the noise source. To configure the noise standard deviation, press **`Ctrl+F`** or **`Cmd+F`**, type **`range`**, and add it. Name it **`NOISESTD`** for noise standard deviation. Set the default value to 0.01, start at 0, stop at 3, and use a step of 0.01.

(Refer Slide Time: 14:15)



With the noise standard deviation set, configure the amplitude of the Gaussian noise to match **`NOISESTD`**. The noise is added at the baseband to avoid unnecessary complexity, so there's no frequency conversion involved here. Now that we have the interpolating FIR filter followed by the noise source, we'll add a delay in the future. Finally, the signal should be processed through a decimating FIR filter, which will take the reverse of **`mypulse`** and sample at the correct location.

Let's proceed by adding a decimating FIR filter. Press **`Ctrl+F`** or **`Cmd+F`**, type **`decimating FIR filter`**, and add it to the flow graph. For convenience, you can use the arrow keys to rotate the filter to the desired orientation. Set the decimation to **`SPS`**, and for the taps, use **`mypulse`** with the notation **`square bracket colon colon minus 1`** to

reverse it.

Next, let’s add a QTGUI Time Sink. Press **`Ctrl+C`** to copy the previous time sink and **`Ctrl+V`** to paste it. Rotate it twice using the right arrow key and connect it to the flow graph. Set the number of points to 1000 for a clear display.

When you execute the flow graph, you’ll notice some shaking due to the noise. Removing the noise will stabilize the display. To verify that your constellation is correct, replace the time sink with a constellation sink. Press **`Ctrl+F`** or **`Cmd+F`**, type **`CONST`**, and select the QTGUI Constellation Sink. Rotate it twice using the right arrow key, connect it, and run the flow graph. You should see a well-defined QPSK constellation with minor noise.

If there are no delay-related issues at this point, but you want to account for delays, consider the following approach: Instead of using a decimating filter, use a filter without decimation. Compute the absolute value and identify the peak location. Recall from previous discussions that you need to convolve with the matched filter and locate the peak of the magnitude.

To implement this, copy the existing filter setup (**`Ctrl+C`** and **`Ctrl+V`**), and remove the decimation so you can properly assess the delay. Then, add a compensatory delay. Press **`Ctrl+F`** or **`Cmd+F`**, search for **`range`**, and add a QTGUI Range object. Name it **`comp_delay`** for the compensatory delay. This compensatory delay is essential to offset the delay encountered in the signal processing.

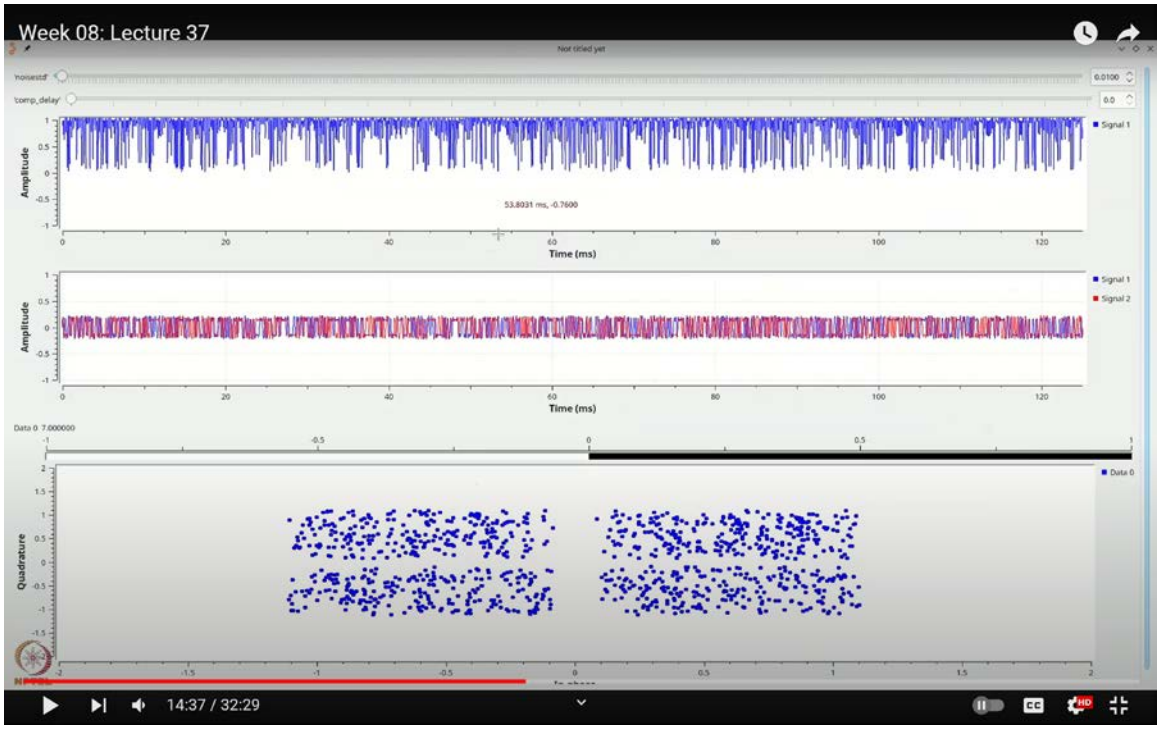
This compensatory delay will be an integer value between 0 and SPS. To implement this, we need to add a delay block. Press **`Ctrl+F`** or **`Cmd+F`**, search for **`delay`**, and add the delay block to the flow graph. Connect it, name it **`comp`**, and set its value to **`SPS minus comp delay`**.

The reason for this setup is that when we measure the delay, it will be a number between 0 and SPS. To counteract this delay, we need to apply a negative delay. Since negative delays aren't feasible, we effectively shift by the delay amount starting from the next symbol. Additionally, we'll introduce a second delay to handle randomness.

Let’s add another delay block. Press **`Ctrl+F`** or **`Cmd+F`**, then **`Ctrl+C`** and **`Ctrl+V`** to duplicate the existing delay block. Rotate it to the left and name it **`random_delay`**. This delay will be a random value. To generate this random delay, press **`Ctrl+F`** or **`Cmd+F`**, search for **`import`**, and include the import block with **`import numpy as np`** for convenience.

Next, add a variable for the random delay. Press **`Ctrl+F`** or **`Cmd+F`**, add a variable block, and configure it to be **`random_delay`**. Set this variable to **`np.random.randint(0, SPS)`**. Considering delays up to SPS is sufficient because delays of SPS plus 1 are equivalent to zero delay due to circular indexing. This approach handles the random delay effectively within the given range.

(Refer Slide Time: 14:37)



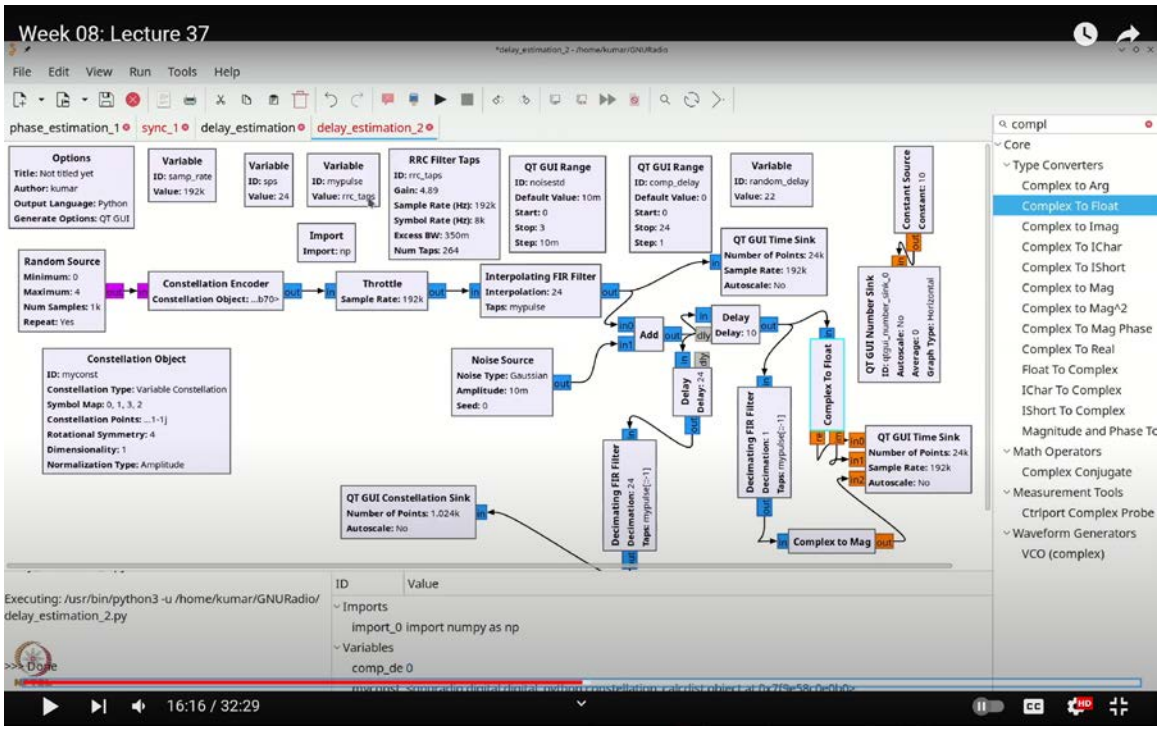
This block should generate a random delay. Double-click it to confirm that it produces random delays, and then connect it accordingly. To view the random delay, add a GUI number sink. Press **`Ctrl+F`** or **`Cmd+F`**, search for **`QT GUI Number Sink`**, and add it to your flow graph. Next, create a constant source to connect to this number sink. Press

`Ctrl+F` or **`Cmd+F`**, search for **`CONST`**, and select the constant source block. Set this source to a float type, double-click it, and assign it the value of **`random_delay`**.

Now, connect the constant source to the number sink. This should be properly set up. Let's check if anything is missing. We need to ensure the filter is visualized correctly after taking the absolute value. Press **`Ctrl+F`** or **`Cmd+F`**, search for **`complex to abs`**, and choose **`complex to mag`**. Add a time sync block by pressing **`Ctrl+F`** or **`Cmd+F`**, searching for **`QT GUI Time Sync`**, and connecting it accordingly. Set the number of points to **`1000 times SPS`** and verify if it works as expected. Make sure this block is set to float.

We are now ready to execute the flow graph. First, let's verify the compensatory delay. If the delay is 7, adding a compensatory delay of 1, 2, 3, 4, 5, 6, 7 should correctly align the constellation points. Since we cannot directly advance by 7, we adjust by setting **`SPS - 7`** for the compensatory delay. This approach ensures accurate alignment. If you reduce the delay, you should observe a clear constellation.

(Refer Slide Time: 16:16)



For further clarity, let's compare the original waveform with the absolute values obtained

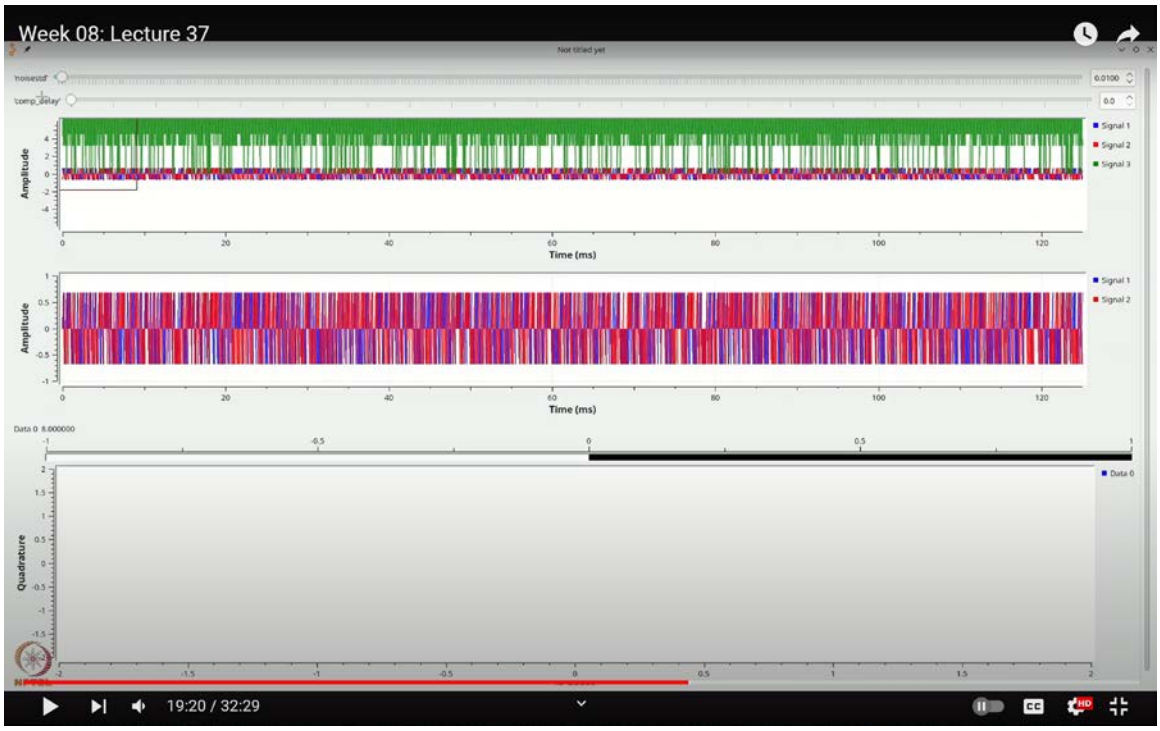
after processing. Initially, analyze these in the same context to understand the differences. To simplify things, I will increase the number of inputs in my time sink to 3.

I want to view the real and imaginary parts of the received signals directly. For example, consider the delayed output; we need to split this into its real and imaginary components and display them separately. To do this, we'll use the **`Complex to Float`** block. Press **`Ctrl+F`** or **`Cmd+F`**, search for **`Complex to Float`**, and rotate the block accordingly.

Connect the output of the delayed signal to this block, with the real part going to one output and the imaginary part to the other. By visualizing these components, we can analyze the real and imaginary parts separately.

Additionally, let’s simplify the process by using a basic pulse. Instead of using the RRC pulse, I’ll switch to a simpler approach. I'll use **`np.arange`**, with a ramp defined as **`sps divided by sps`**. This approach will simplify our pulse to a straightforward ramp. Once you execute the flow graph with this setup, you should observe distinct peaks in the output. These peaks are crucial for our analysis.

(Refer Slide Time: 19:20)

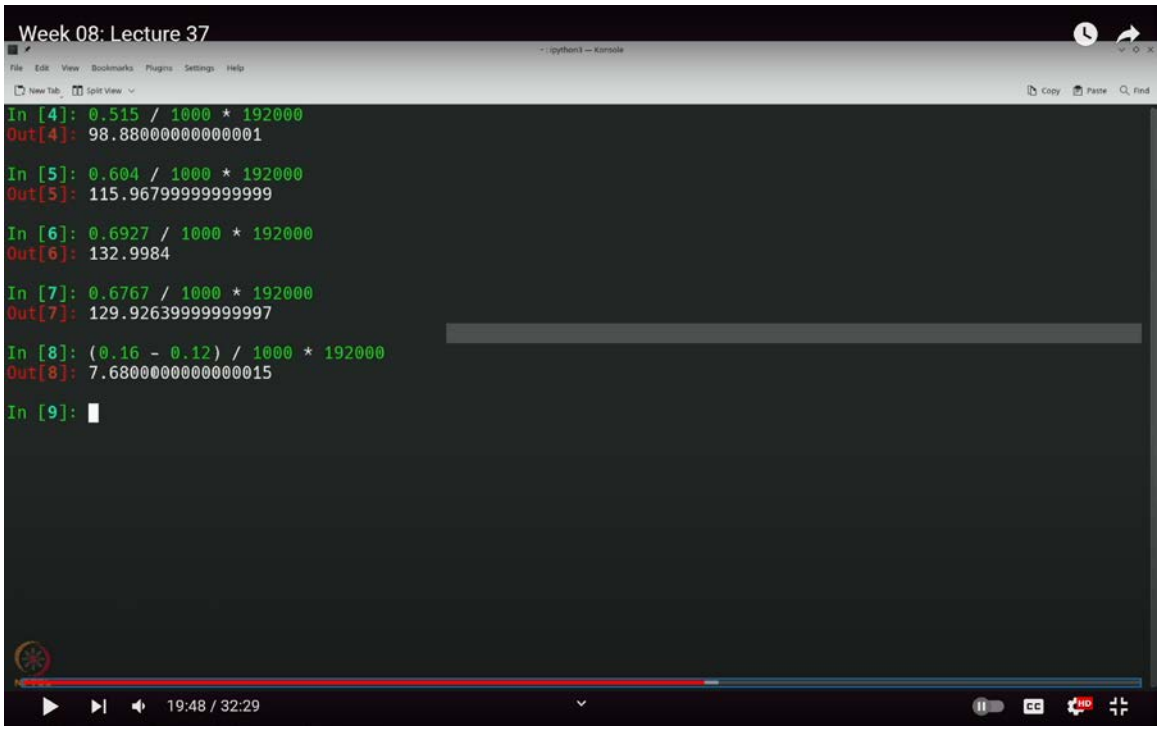


To further validate, let’s perform a calibration assuming there is no delay. We can remove the random delay by setting it to zero. This allows us to focus on identifying the peaks accurately.

With no random delay, you should see a peak at the beginning (0 milliseconds) and subsequent peaks at approximately 0.1195 milliseconds, 0.2456 milliseconds, and so on. This pattern aligns with what we expect. Given that we chose a transmission rate of 8000 symbols per second, the 0.125 millisecond interval corresponds to the gap between two symbols.

Let’s verify this by assuming the first peak is at 0 and observe how the subsequent peaks align with the expected intervals. To improve clarity, remove any noise and add a grid to the visualization.

(Refer Slide Time: 19:48)



Let's reset and review our results. Initially, the first peak appears around 0.12 milliseconds. Now, if we introduce a random delay and execute the flow graph, we observe that the first peak shifts to approximately 0.1471 milliseconds. For our analysis, let’s consider this

value.

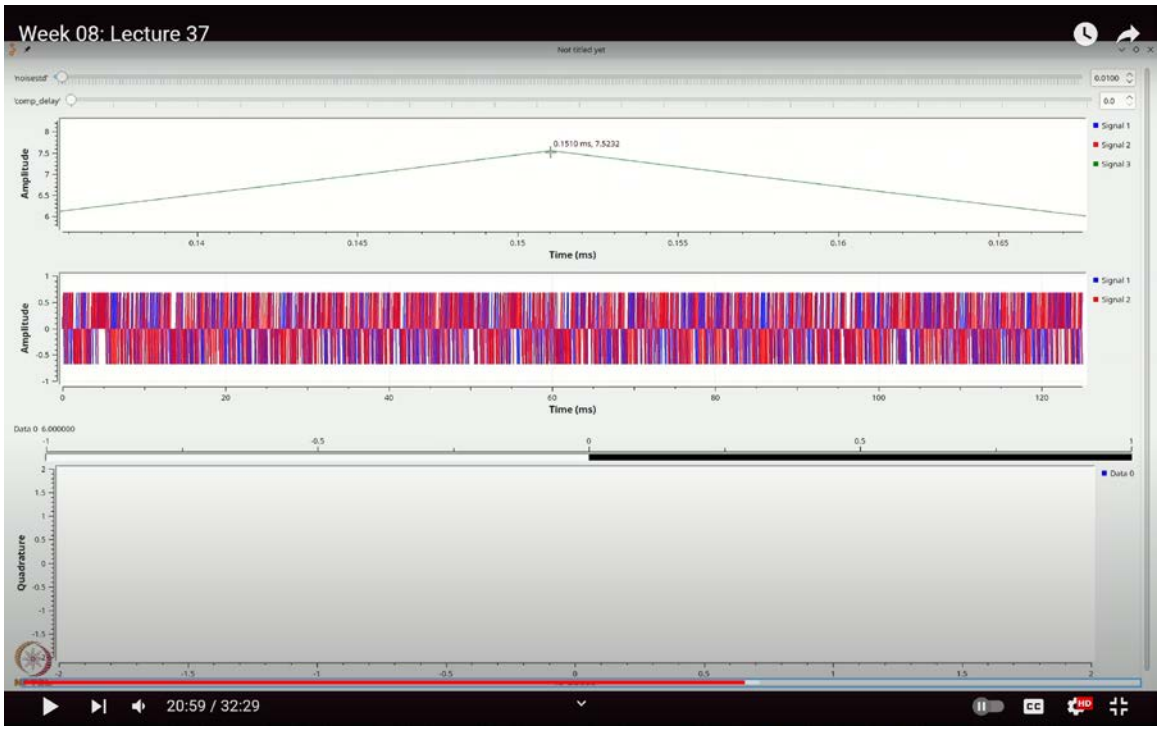
Let’s compare this with the reference peak observed without any delay, which was around 0.12 milliseconds. With the random delay applied, we need to determine the peak location relative to our reference. For instance, if we find a peak at 0.116 milliseconds with the delay, we calculate the difference from the reference peak as follows:

$$0.16 - 0.12 = 0.04 \text{ milliseconds}$$

Converting this to samples:

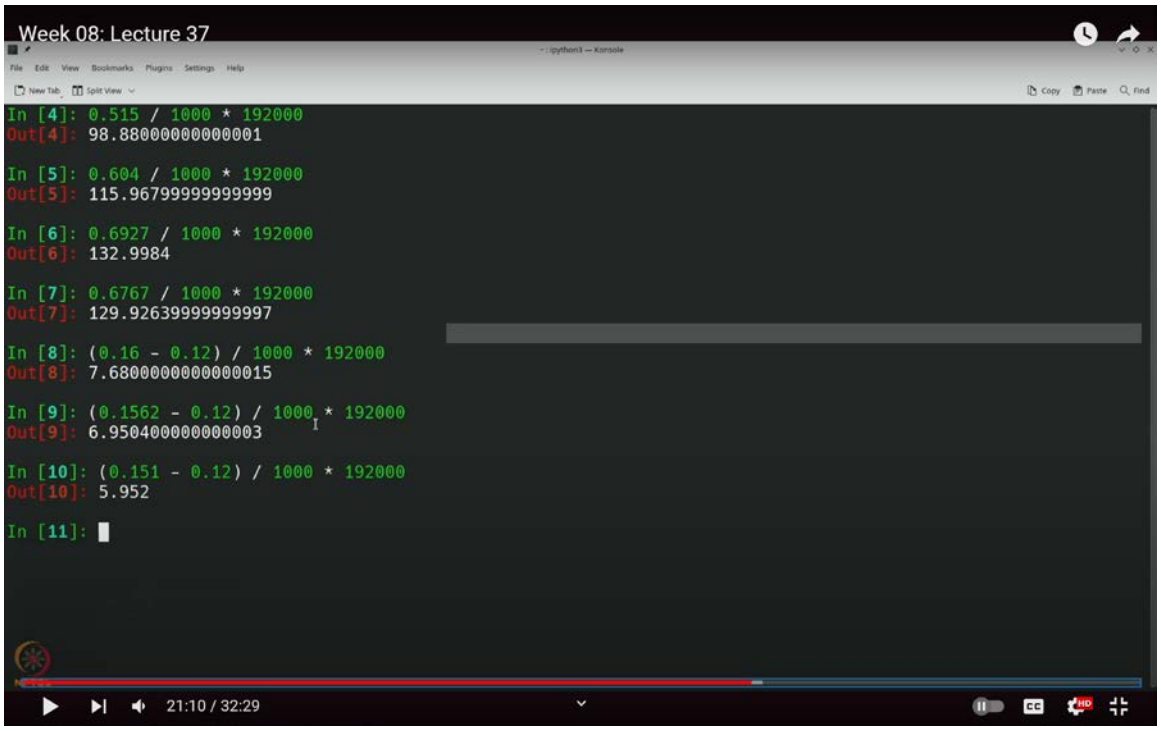
$$0.04 \text{ milliseconds} \times \frac{192{,}000}{1000} = 7.68$$

(Refer Slide Time: 20:59)



This suggests a delay of approximately 7 or 8 samples. Checking the data, it indicates 8 samples, which aligns with our observation. Setting the compensatory delay to 8 samples should correct the constellation alignment. Let’s verify this: executing the flow graph again and observing the peak at around 0.1562 milliseconds confirms a delay of 8 samples.

(Refer Slide Time: 21:10)



If the calculated delay is 7, rounding may have caused the discrepancy. Testing again with a peak value close to 0.151 milliseconds yields:

$$0.151 - 0.12 = 0.031 \text{ milliseconds}$$

$$0.031 \text{ milliseconds} \times \frac{192{,}000}{1000} = 5.952$$

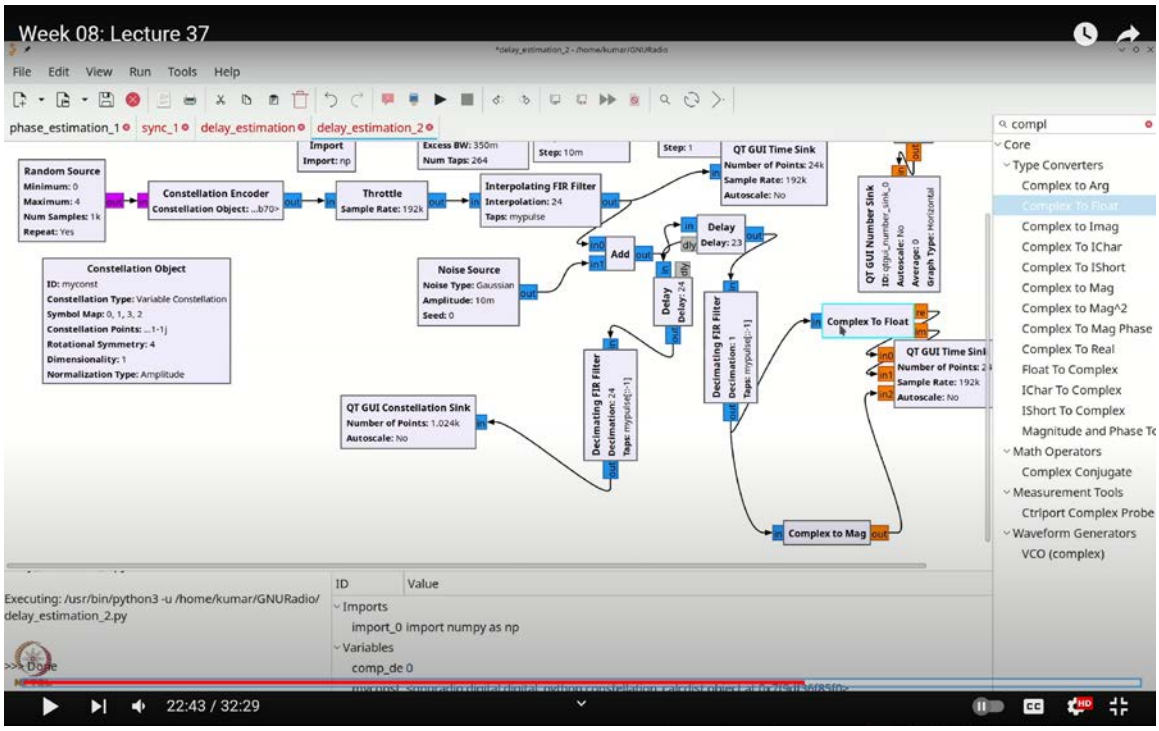
Thus, the delay should be approximately 6 samples. Setting the compensatory delay to 6 samples correctly aligns the constellation.

This method, involving the absolute value and peak location, accurately determines the delay and compensates for it. Remember, we use $\text{SPS} - \text{compensatory delay}$ to adjust for the delay. This approach shifts the signal forward by one symbol and a few samples to align the constellation properly.

Now, switching back to the RRC (Root Raised Cosine) pulse taps introduces more complexity. The peaks with the RRC taps may not align as neatly, making it slightly more

challenging to interpret the results.

(Refer Slide Time: 22:43)



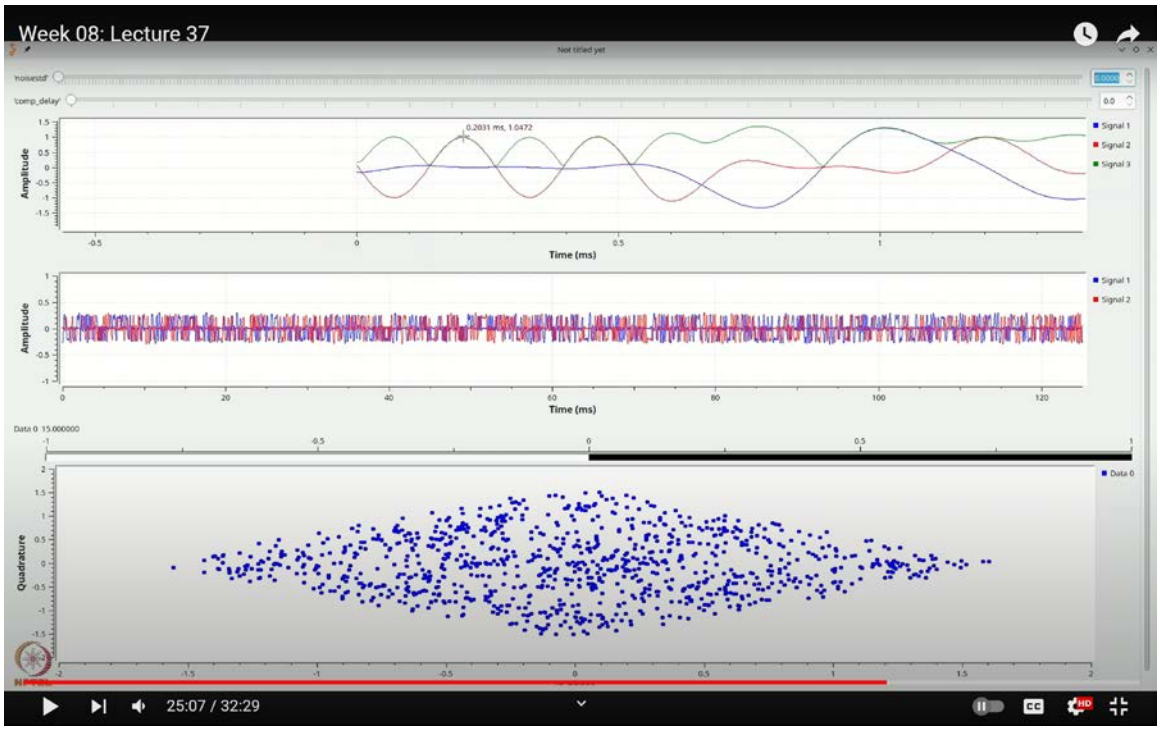
Before proceeding further, let's ensure we correctly visualize the delay by adjusting our setup. Disconnect the current connections and reconnect them in such a way that we observe the same delay across the system. It helps to rotate the components for clarity and to organize the flow graph neatly. I will rearrange the elements to provide some space and make the flow graph more comprehensible.

Now that everything is clearer, execute the flow graph to verify that the results are indeed more apparent. In this setup, you should observe distinct peaks, though they may seem a bit unusual. To accurately comprehend the delay in this scenario, remember that while the peak detection method, such as absolute value maximization, is effective for a single S(t), it requires a more nuanced approach when dealing with complex combinations.

Let's add a grid to facilitate our analysis. We're working with a QPSK constellation, which has specific properties, such as 1 + 1j, and so on. However, for simplicity, we will switch to a more straightforward QPSK constellation with the symbols: 1, j, -1, and -1j. This

simpler constellation makes interpretation easier because, in this configuration, when the i part is +1 or -1, the j part is always 0, and vice versa.

(Refer Slide Time: 25:07)



With this setup, it becomes much simpler to reason about the constellation. For instance, let's identify the locations of the symbols clearly. To do this, temporarily remove any noise to better focus on the symbols themselves.

Here, you can observe that where the red line, representing the q-axis, is at 0, the i-axis is at -1. Similarly, where the i-axis is 0, the q-axis is at -1. And where the q-axis is 0, the i-axis is at +1. This pattern simplifies the analysis. In this QPSK constellation, whenever either the i-axis or q-axis is exactly ±1, the magnitude is precisely 1. Thus, at these points, the magnitude remains constant.

To determine the peaks, focus on the zero crossings of the green line, which is sufficient for identifying these points. Let's apply this method now. I'll execute the flow graph to identify one of these zero crossings. For clarity, I'll add a grid and temporarily remove the noise.

For example, let’s say a good candidate for zero crossing occurs at around 0.204 milliseconds. We’ll use this as our reference time of 0.204 milliseconds. Before proceeding, we need to ensure that the random delay is set to zero.

So, remove the random delay by setting it to 0 and execute the flow graph. This allows us to calibrate the system accurately. Add a grid to the view, and observe that the zero crossing at 0.12 milliseconds serves as our reference.

Next, reintroduce the random delay and observe where the zero crossing now occurs. With 0.12 milliseconds as our reference, check the new crossing point. Add a grid for better visibility and note that the crossing at 1 happens around 0.12 milliseconds.

To confirm, let's repeat this process without any delay to ensure our calibration is accurate. Remove the random delay again, perform the calibration, and identify the zero crossing of the magnitude. Add the grid to the sink for better precision. Double-click on the grid settings and ensure it is enabled for accurate measurement.

Now, observe the crossing of the magnitude of 1. So, the signal crosses the magnitude of 1 initially and then crosses it again around 0.125 milliseconds. This makes perfect sense because with 8000 samples per second, the expected period between these crossings is 0.125 milliseconds.

Now, let’s reintroduce the random delay and check where the signal crosses 1 again. Temporarily remove the noise to get an accurate reading. The crossing now occurs at approximately 0.1825 milliseconds. To determine the delay, calculate the difference between this time and our reference of 0.125 milliseconds. Converting this difference to seconds and multiplying by 1920 gives us about 11 samples as the predicted delay.

Setting the compensatory delay to 11 samples aligns the constellation correctly. Let’s try this process blind to the delay. Disable the delay blocks to hide the actual delay from view and execute the flow graph.

Without seeing the delay, our task is to figure it out. Observing the crossing at approximately 0.396 milliseconds, we calculate this as follows:

$$0.396 - 0.125 = 0.271 \text{ milliseconds}$$

(Refer Slide Time: 29:49)

Convert this to seconds and multiply by 1920, giving us 52 samples. To get a number within the sample range, use modulo SPS (where SPS is 24):

$$52 \; mod \; 24 \; = \; 4$$

So, the delay is about 4 samples. If you count 4 samples from the reference, you align the constellation correctly.

For another test, observe the crossing at 0.3222 milliseconds. Calculating:

$$0.3222 - 0.125 = 0.1972 \text{ milliseconds}$$

Convert to seconds and multiply by 1920, giving about 14 samples. Setting the delay to 14 samples also correctly aligns the constellation.

By carefully analyzing the constellation properties, you can accurately determine the delay.

Note that this approach does not rely on the specific data, making it robust for various situations.

We've utilized the properties of the constellation to estimate delay. To summarize, the approach involves enabling the necessary components to accurately find and calibrate the delay in your system. While more sophisticated methods are available, this example demonstrates a practical way to extract delays for various types of modulations and pulses. These concepts are integral to practical system implementations.

In this lecture, we explored a more complex example of delay estimation. Unlike simpler cases with a single signal s(t), we worked with a sequence of modulated symbols, represented as $\sum b_k g(t - kT)$. Despite the added complexity, a thorough understanding of the waveform structure allows for precise delay determination.

It is important to note that our method was visual. In the presence of noise, visual inspection alone is insufficient. To address this, you would typically develop a system that systematically detects zero crossings and averages results to achieve a more accurate delay estimate. Such a system can be implemented quite straightforwardly.

In the next class, we will continue our discussion, exploring other aspects of receiver impairments, such as frequency offsets. We will integrate these concepts to develop a practical receiver adjustment system. Thank you.