Digital Communication using GNU Radio

Prof. Kumar Appaiah

Department of Electrical Engineering

Indian Institute of Technology Bombay

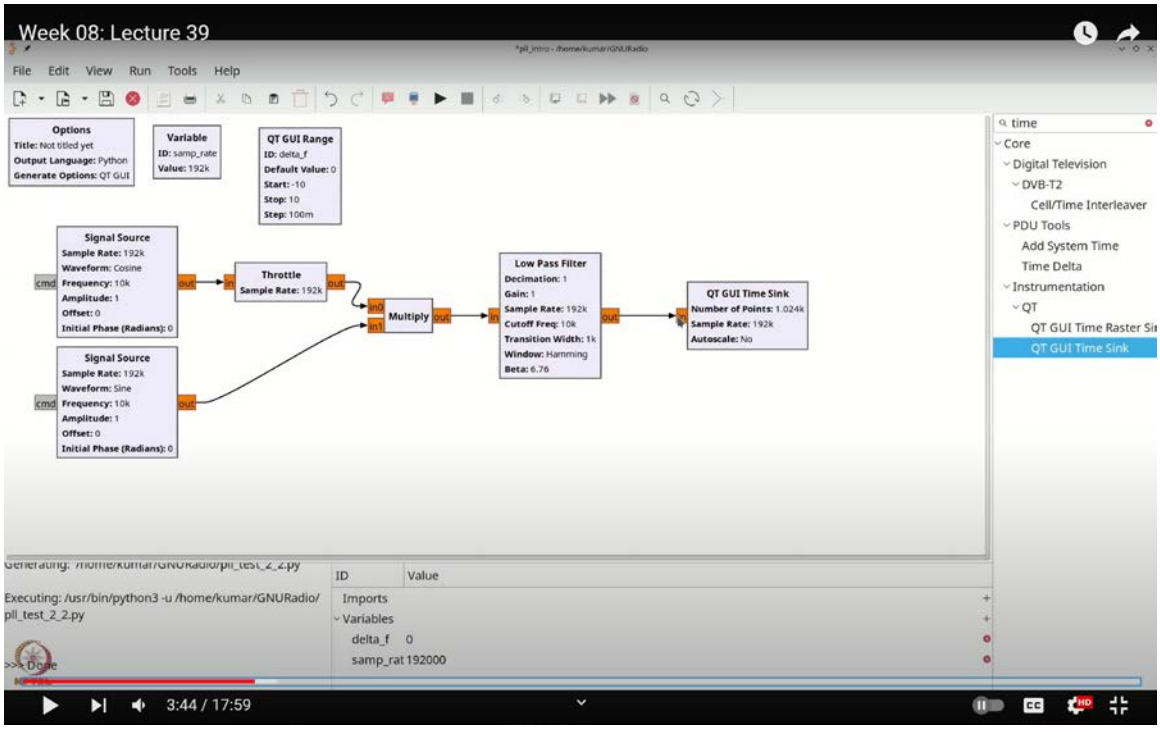
Week-08

Lecture-39

Phase Locked Loop in GNU Radio

Welcome to this lecture on Digital Communication using GNU Radio. My name is Kumar Appiah, and I am with the Department of Electrical Engineering at IIT Bombay. In the previous lecture, we explored some impairments and their effects on communication system performance. In this lecture, we will delve into frequency offsets, how to measure them using GNU Radio, and specifically how to utilize the Phase Locked Loop (PLL) blocks to determine frequencies. The PLL block is particularly valuable because it can detect frequencies even in the presence of noise, a capability we will explore as we work with GNU Radio.

(Refer Slide Time: 03:44)



Before we dive into the intricacies of the PLL, let's start with a simple conceptual exercise. We'll mix two sinusoids of different frequencies and observe the results. To begin, I'll set the sampling rate to 192,000. First, let's add a signal source by pressing Ctrl-F or Cmd-F. We'll select a signal source and configure it to generate a cosine wave.

We'll set the frequency of the first signal source to 1000 Hz, and ensure it's a float value. Next, we'll add another signal source by copying and pasting the first one. This new signal source will generate a sine wave. We'll adjust its frequency by adding a variable frequency offset, which we'll call Δf. To do this, we'll create a range for Δf.

Press Ctrl-F (or Cmd-F) and select "range." Double-click on this range and name it Δf. Set its range from -10 to 10 Hz with a step size of 0.1. For better visualization, we'll set the sampling rate to 10,000.

Now, we'll mix these two signals. First, add a throttle by pressing Ctrl-F (or Cmd-F) and selecting "throttle." Set the throttle to float and connect it to the signal sources. Then, add a multiplication block by pressing Ctrl-F (or Cmd-F) and choosing "multiply." Connect the multiplication block appropriately.

To visualize the results, we need to filter out the 2FC component. Add a low-pass filter by pressing Ctrl-F (or Cmd-F) and selecting "low pass filter." Configure the filter with real coefficients, a cutoff frequency of 10,000 Hz, and a transition width of 1,000 Hz.

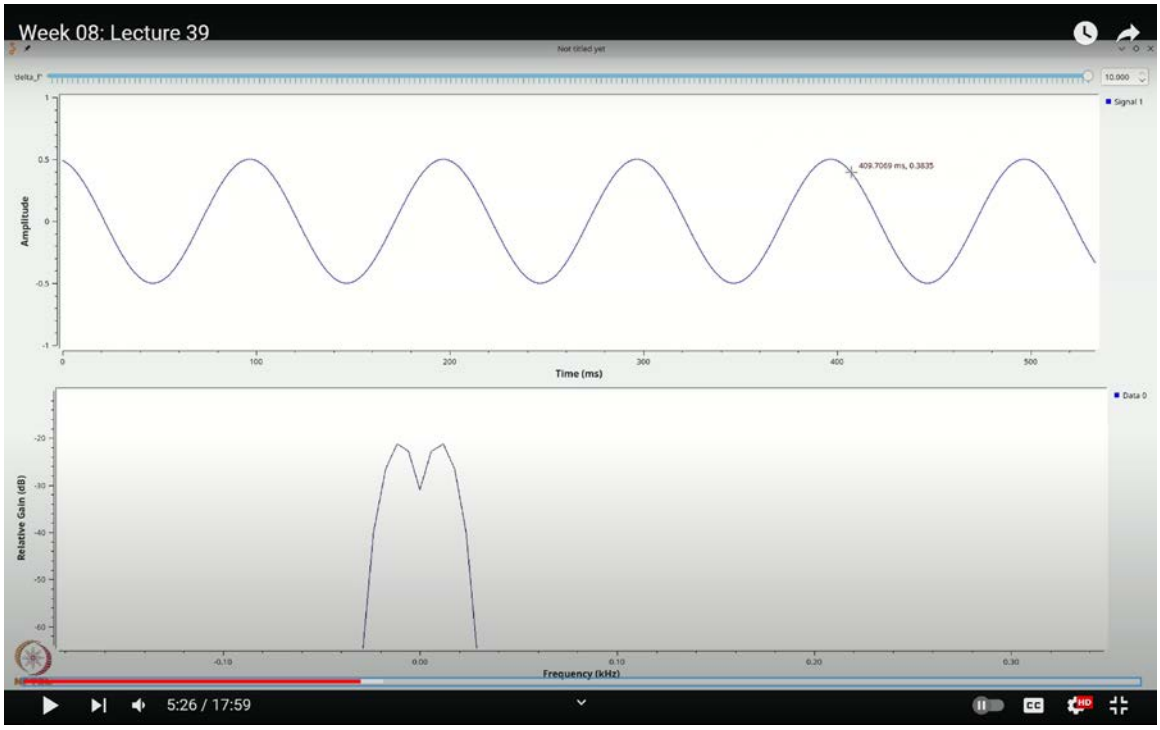
Finally, add a time sync block by pressing Ctrl-F (or Cmd-F) and selecting "time sync" to observe the output.

If there is no frequency offset, the result will be close to zero because the cosine multiplied by the sine will effectively cancel out. However, if we introduce a frequency offset Δf, you'll notice a shift or movement in the signal. To enhance the visualization, increase the number of points by a factor of ten. With a frequency offset, you should see a cyclic pattern. By measuring the frequency of this cyclic signal, you can determine the magnitude of the frequency offset.

Let's increase the frequency offset to 10 Hz and observe the results. To view this in the

frequency domain, we'll add a frequency sync. Press Ctrl-F (or Cmd-F) and add a "frequency sync" block. Set the number of points for the frequency sync to 32,768 for better resolution.

(Refer Slide Time: 05:26)

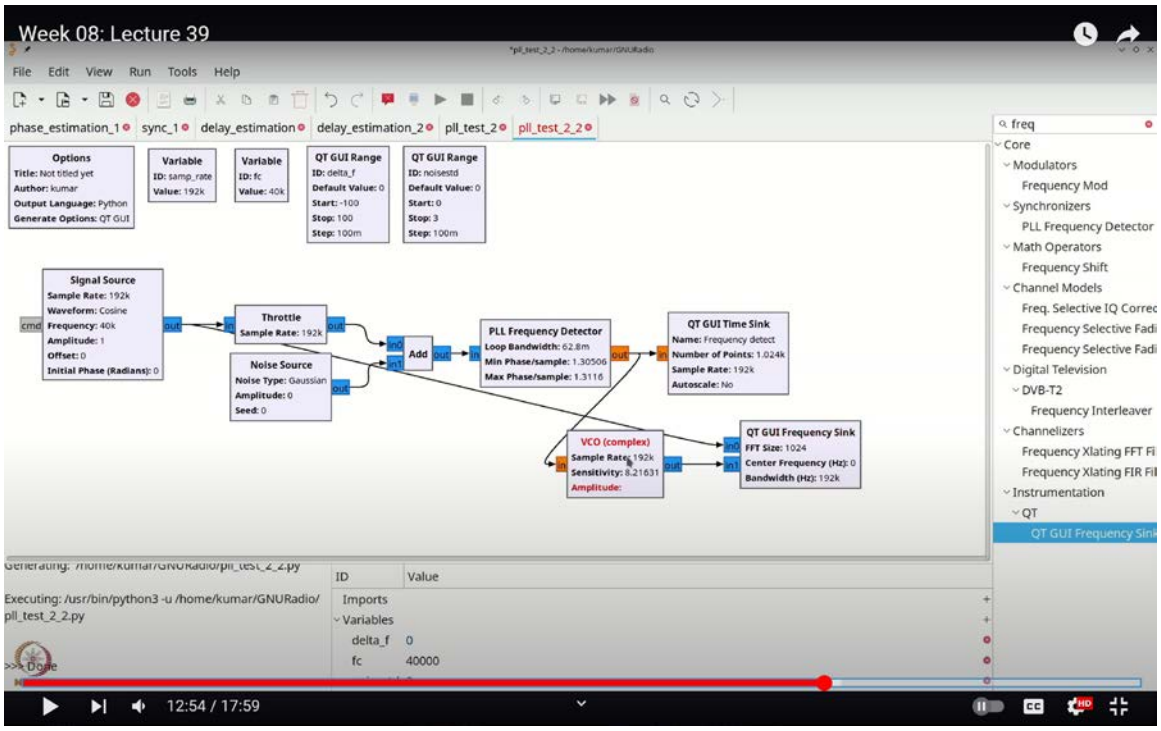


Initially, you may see zeros, which can be disregarded for now. When you increase the offset to 10 Hz, zoom in, and you'll notice a peak appearing at 10 Hz. This corresponds to the frequency offset we introduced, with peaks appearing at 396 Hz, 496 Hz, and so on, precisely reflecting the 10 Hz offset.

The fundamental idea here is that this method of frequency measurement by mixing can help you build a filter to adjust the frequency of the sine wave to match the desired phase. This approach is implemented in the Phase Locked Loop (PLL) blocks within GNU Radio.

Press Ctrl-F (or Cmd-F) and search for "PLL." You'll find the PLL carrier tracking blocks, including frequency detectors and regeneration modules. We'll focus on the PLL frequency detector, which provides a more advanced version of this technique and can be used to generate a carrier signal at the receiver, even in the presence of noise.

(Refer Slide Time: 12:54)



To get started, let’s make a few adjustments. First, set the sample rate to a more convenient value, such as 192,000 Hz. Next, configure the carrier frequency f_c to 40 kHz and set it to track nominal changes. Press Ctrl-F (or Cmd-F) and add a "variable" block. Double-click the block, name it f_c, and set its value to 40,000 Hz.

Additionally, we’ll add a range to manage the frequency offset and ensure accurate tracking. Press Ctrl-F (or Cmd-F) and search for "range" to add a QTGI range block.

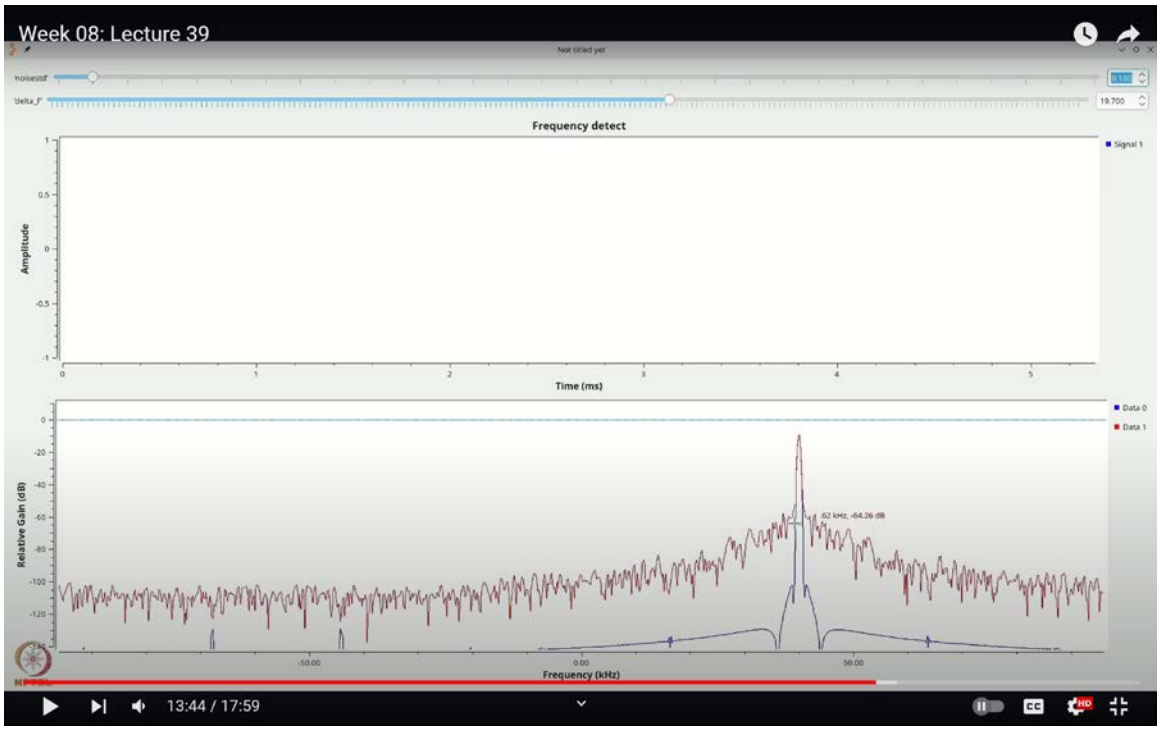
We’ll designate this range as **`delta_F`**, with a default value of 0. Set the range to vary from -100 to 100 Hz, with a step size of 0.1 Hz. Now we are ready to proceed.

Next, let's create a complex signal source. Press Ctrl-F (or Cmd-F) and search for "signal." Select the signal source block and double-click it. Set the frequency to $f_c + \Delta f$, with an amplitude of 1 and no initial phase.

Add a throttle block by pressing Ctrl-F (or Cmd-F) and typing "throttle." We will also add noise to our signal. First, create a range block for the noise standard deviation. Press Ctrl-

F (or Cmd-F), search for "range," and double-click it. Name this range **`noise_std`** and set its values to range from 0 to 3, with a step size of 0.1.

(Refer Slide Time: 13:44)



Add a noise block by pressing Ctrl-F (or Cmd-F) and searching for "noise." Insert this noise block, ensuring that the amplitude is set to **`noise_std`**. The **`noise_std`** should be a number ranging from 0 to 3, with a step size of 0.1.

Now, we need a PLL frequency detector. Press Ctrl-F (or Cmd-F) and search for "PLL." Select the PLL frequency detector block. Although we could also use a phase detector, we'll start with the frequency detector.

The PLL frequency detector takes an input with three parameters and provides a float output, which corresponds to the detected frequency. Let's configure these parameters:

- The loop bandwidth, which is the loop filter bandwidth, should be set to $\frac{6.28}{100}$. This is within the recommended range of 0 to $\frac{2\pi}{100}$.
- The minimum phase per sample corresponds to the minimum frequency we wish

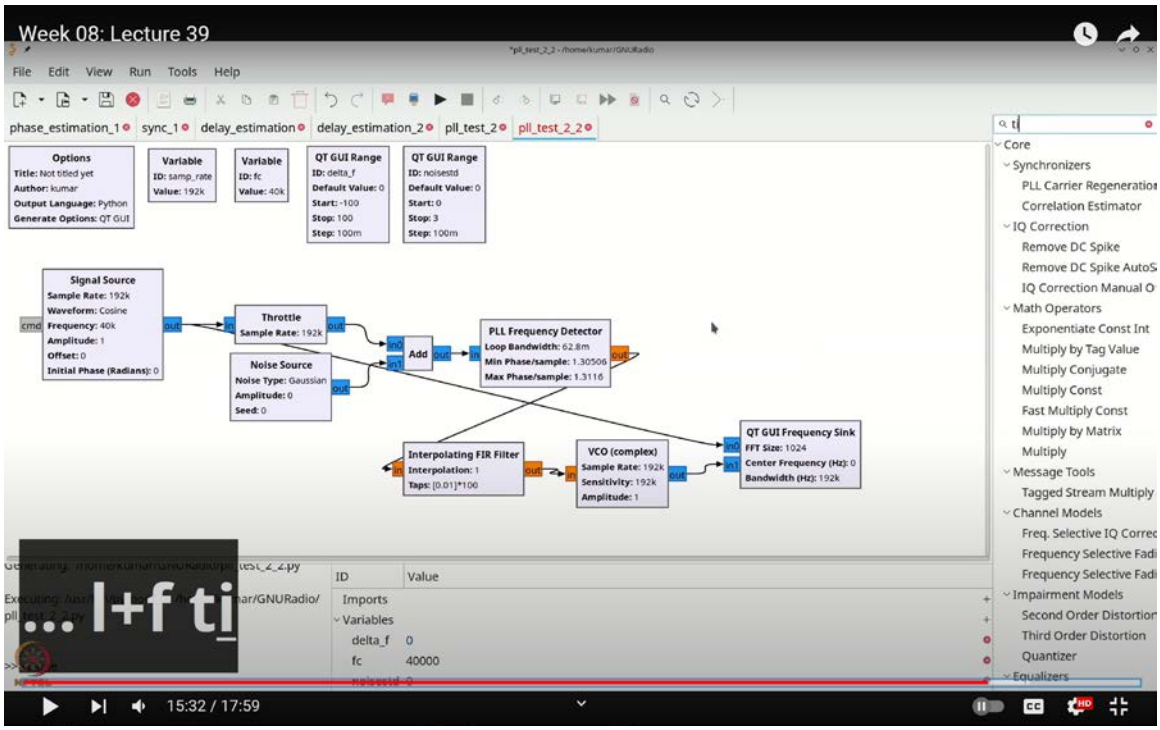
to detect. Since we expect frequency shifts from -100 to 100 Hz, we need to carefully calculate this. The value should be $6.28 \times \frac{f_c - 100}{\text{sampling rate}}$.

Similarly, the maximum frequency deviation we anticipate is 100 Hz, which corresponds to $6.28 \times (f_c + 100) \div \text{sampling rate}$. This value is essential for configuring our frequency detector. To verify that everything is functioning correctly, let's add a QTGI timesink. Press Ctrl-F (or Cmd-F), search for "QTGI timesink," and select a real-value timesink, naming it "frequency detector."

Now, let’s examine the output of this setup. When executing the flow graph, we observe an output close to 0.86. By adjusting the **`delta_F`**, you will notice slight variations. Zooming in on this, you can see that the output fluctuates up and down.

What’s happening here is that the PLL is estimating the frequency based on the input signal. After estimating the frequency, it translates this into an amplitude or voltage value. This amplitude can then be used to generate a corresponding frequency.

(Refer Slide Time: 15:32)



To achieve this, press Ctrl-F (or Cmd-F) and search for "VCO," which stands for Voltage Controlled Oscillator. Select the complex VCO block and set the sampling rate to **`SAMP rate`**. The sensitivity of this VCO converts the voltage to a frequency. In this case, the sensitivity value is between 1.30 and 1.31, with a midpoint around 1.308 corresponding to 40 kHz.

For precise calculation, use the formula $2\pi \times 40{,}000 \div 1.30833$, which gives approximately 1.308. Connect the VCO block to the flow graph. Next, add a QTGI frequency sync by pressing Ctrl-F (or Cmd-F) and search for "FREQ." Connect the output of the generated frequency to this sync.

Ensure that the sensitivity is adjusted to reflect a frequency of 40,000 Hz. Therefore, set it to $6.28 \times 40{,}000 \div 1.30833$. Verify that the amplitude is set to 1. When executing the flow graph, you should obtain a frequency close to 40 kHz, matching the expected value. If you adjust **`delta_F`**, you'll see that these changes are accurately reflected in the output.

In the presence of noise, you'll notice that while the original frequency serves as a very good carrier, the generated carrier is also affected by noise. This issue arises because the frequency measurement in the frequency detector introduces very minor inaccuracies. To address this, one effective method is to smooth out these variations using a low-pass filter.

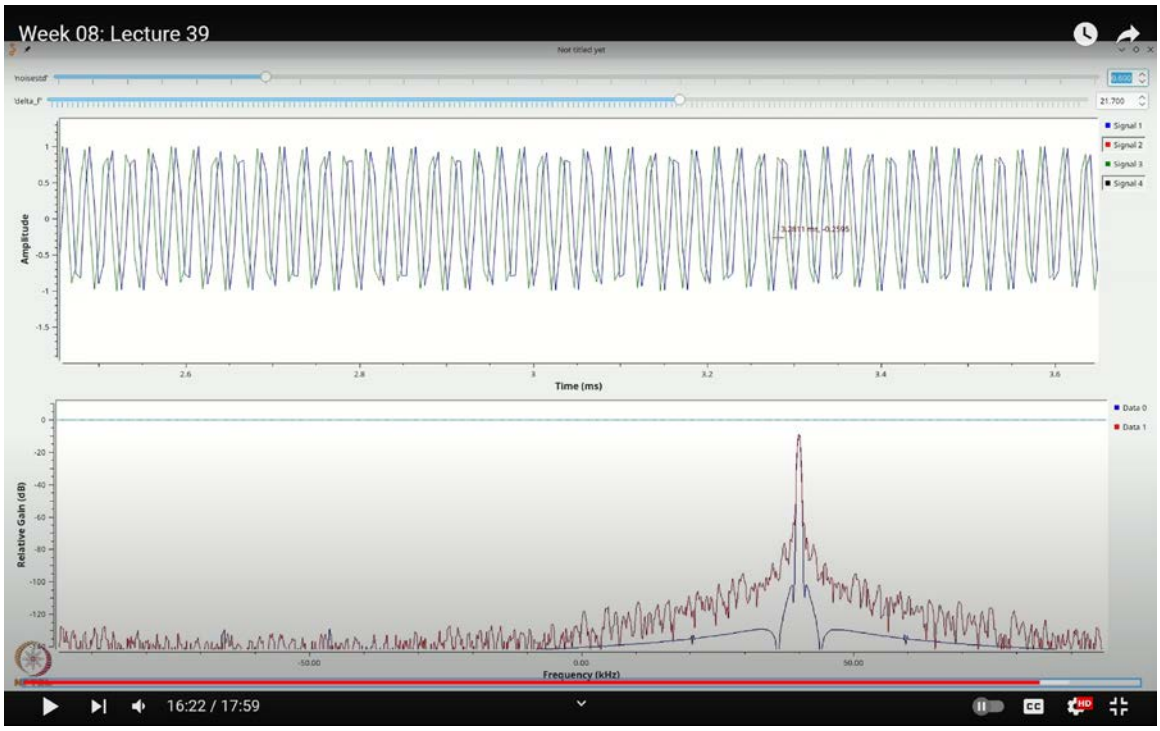
For instance, you could use a low-pass filter, which you can add by pressing Ctrl-F (or Cmd-F) and searching for "low pass filter." Alternatively, a simpler approach might be to use averaging. Let's demonstrate this approach by removing the low-pass filter and opting for an averaging technique.

Press Ctrl-F (or Cmd-F), search for "interpolating," and select an interpolating FIR filter. Set it to float-to-float, with an interpolation factor of 1. For the taps, configure it to perform averaging. For example, setting the taps to 0.1 times 10 will provide a 10-fold averaging effect, effectively creating a moving average of the last 10 samples.

By connecting the VCO through this averaging filter, you'll find that even with noise present, the output becomes much more stable. If you increase the averaging further, such as adjusting the taps to 0.01 times 100, you'll observe an even sharper and more stable

result. Averaging helps in accurately retrieving the frequency, making the output more consistent.

(Refer Slide Time: 16:22)



To visualize the results, let’s update our setup. Remove the previous time sync, and press Ctrl-F (or Cmd-F) to add a new time sync. This new time sync will have two inputs: the original signal and the generated signal. By comparing the real parts of these signals, you’ll see that they have very similar frequencies, with only minor offsets. These small offsets are due to minor differences in floating-point calculations and other minor discrepancies.

You can observe that the frequency remains consistent, but these offsets can be effectively managed. The PLL is adept at tracking frequency offsets with remarkable precision, even amidst noise. As you increase the noise level, the PLL continues to track the frequency closely, maintaining accuracy. When you incorporate feedback into the PLL and account for phase offsets as well, the locking mechanism becomes exceptionally reliable, eliminating these issues.

Thus, a PLL proves to be a valuable tool for frequency tracking. Although we didn't explore

scenarios involving modulation on the carrier in this lecture, the fundamental concept remains similar. For those interested in modulated carriers, further reading will provide additional insights.

In this lecture, you've gained an understanding of how phase-locked loops function within GNU Radio. It might initially seem challenging to grasp how frequency is extracted from a pure carrier signal. However, the key takeaway is that even with noise present, a PLL performs effectively. With some additional filtering, the PLL can accurately track frequency offsets. In the next lecture, we will explore alternative methods for managing phase and frequency without relying on a PLL. Thank you for your attention.