

# Digital Communication using GNU Radio

Prof. Kumar Appiah

Department of Electrical Engineering

Indian Institute of Technology Bombay

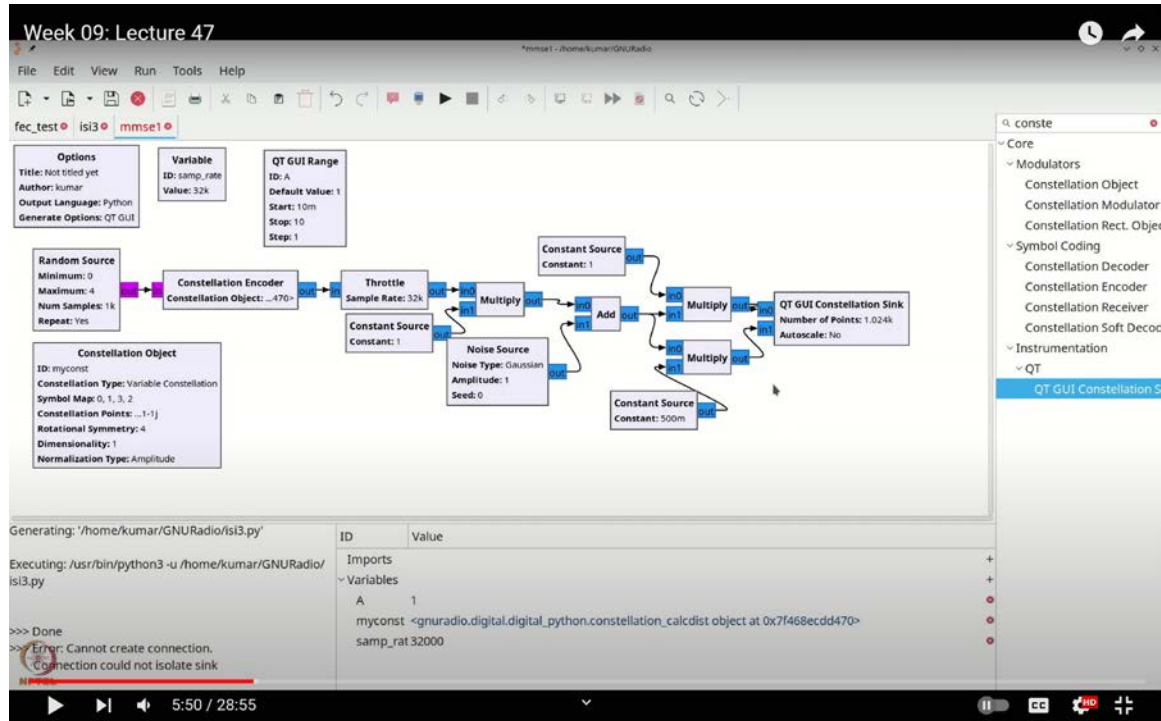
Week-09

Lecture-47

## LMMSE Receiver 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 the Minimum Mean Square Error (MMSE) equalizer and discussed its potential advantages over the Zero Forcing (ZF) equalizer. Although the MMSE equalizer is still a suboptimal approach, it offers a significant benefit by considering the impact of noise during the equalization process.

(Refer Slide Time: 05:50)



In this lecture, we will implement the MMSE equalizer in GNU Radio and compare its

performance with the ZF equalizer. We'll observe how each equalizer performs under different Signal-to-Noise Ratio (SNR) conditions, both in lower and higher SNR regions. To start, we will perform a simple MMSE equalization example using a straightforward channel model, where the channel effect is purely based on scaling.

So, let's get started. First, we will create a random source, which generates values ranging from 0 to 4, with each value represented by a byte. We'll then pass this random source through a constellation encoder. For this example, let's use the default QPSK constellation. We'll define this as ``myconst`` and link it to our constellation object.

Next, since we are running a simulation, we'll incorporate a throttle block. Additionally, we'll introduce a simple gain factor for the channel. To do this, we'll create a range control, which we'll label as ``A``. This gain factor ``A`` will vary between 0.01 and 10. Given that we're focusing on a high SNR scenario, we'll set the default value of ``A`` to 1.

Now, let's move forward. We'll multiply our signal by this gain factor ``A`` using a multiplier block. To set this up, we'll add a constant source block with a value equal to ``A``, ensuring our signal is scaled appropriately. Once we've scaled the signal, we'll add noise to it, specifically, unit energy noise. We can do this by adding a noise source block to our flowgraph.

With these elements in place, we can now analyze the performance of two equalizers: the Zero Forcing (ZF) equalizer and the MMSE equalizer.

The ZF equalizer operates on a simple principle: it undoes the effect of the channel by applying the inverse of the channel's gain. In this case, since our channel is represented by the gain factor ``A``, the ZF equalizer will apply a scaling of  $\frac{1}{A}$ .

On the other hand, the MMSE equalizer requires a more nuanced approach. Instead of applying  $\frac{1}{A}$ , the MMSE equalizer scales the signal based on the SNR. Specifically, the scaling factor is given by:

$$\text{Scaling Factor} = \frac{A}{A^2 + 1}$$

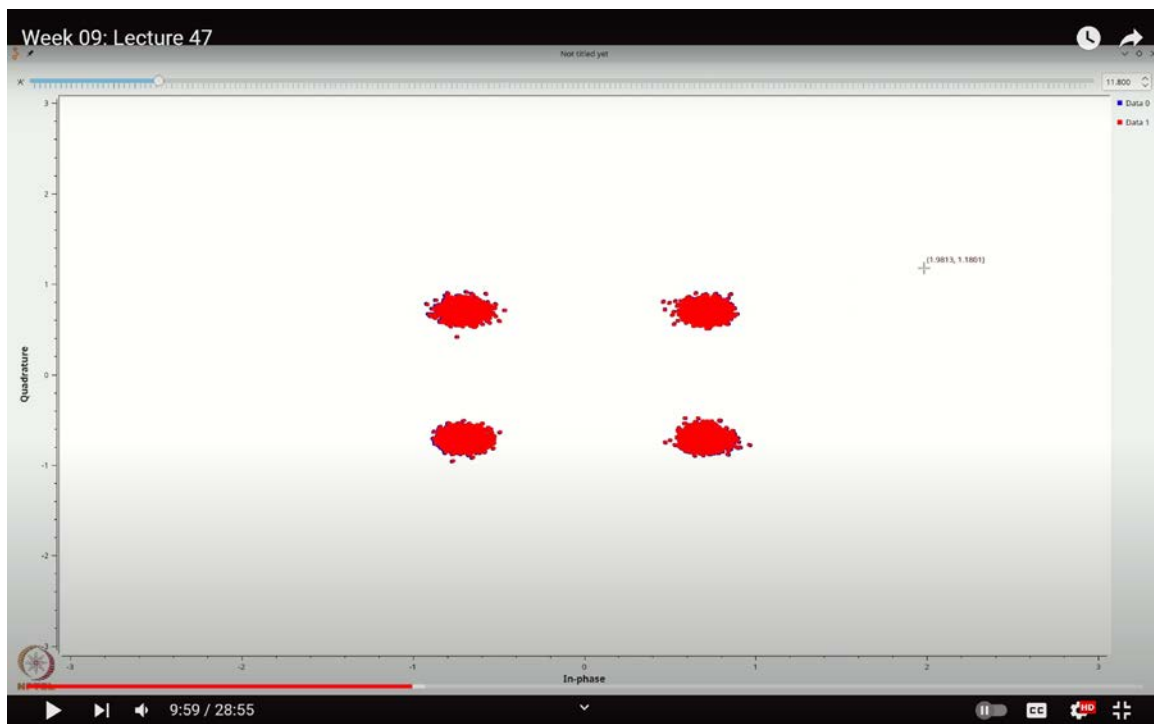
If you're uncertain about this formula, I encourage you to substitute the values into the equations we discussed in class to verify it. You'll find that the correct scaling factor indeed is  $\frac{A}{A^2+1}$ .

Now, let's see how this all works out in our implementation.

To begin, we'll add a constellation sink to our flow graph. This sink will take two inputs: the first input will pass through a scaling factor of  $\frac{1}{A}$ , and the second input will go through a scaling factor of  $\frac{A}{A^2+1}$ . To implement this, we'll add two multiplier blocks. We can simply copy and paste the first multiplier ( $\hat{\mathbf{C}}\mathbf{r}\mathbf{l} + \mathbf{C}, \mathbf{C}\mathbf{r}\mathbf{l} + \mathbf{V}$ ), and connect it to a constant source block that multiplies by  $\frac{1}{A}$ .

For the second multiplier, we'll again copy and paste ( $\hat{\mathbf{C}}\mathbf{r}\mathbf{l} + \mathbf{C}, \mathbf{C}\mathbf{r}\mathbf{l} + \mathbf{V}$ ) and connect it to another constant source, but this time, the constant will be  $\frac{A}{A^2+1}$ .

(Refer Slide Time: 09:59)



Now, let's analyze what happens as we vary the value of A. When A becomes very large,

indicating a high SNR regime, the expression  $\frac{A}{A^2+1}$  simplifies to approximately  $\frac{1}{A}$  because, in such cases, the term 1 in the denominator becomes negligible. Alternatively, you could rewrite this as  $\frac{1}{1+\frac{1}{A}}$ , which also leads to the same result. However, for simplicity, we'll stick with  $\frac{A}{A^2+1}$ .

Let's execute the flow graph and observe the results. As the value of A increases, and the corresponding noise levels change, you'll notice that the constellation points start becoming more distinct. To illustrate this, let's set A to a higher value, perhaps around 100, and adjust the step size to 0.1 for finer control.

With A set to this high value, we're clearly in a high SNR regime. Here, you can observe that the outputs from both the Zero Forcing and MMSE equalizers are nearly identical, which aligns with our expectations. However, as we gradually reduce A, effectively lowering the SNR, you'll start to notice that the blue points in the constellation, representing the Zero Forcing equalizer, begin to deviate and spread out more.

To further enhance our observation, let's increase the number of points generated by the random source and also increase the sampling rate to achieve faster results. Now, as we lower the SNR further by decreasing A, it becomes increasingly evident that the blue points are spreading out, moving further away from the red points, which represent the MMSE equalizer.

This behavior highlights the key difference between the two equalizers: the MMSE equalizer takes noise into account, moderating the division by A and instead using a factor like  $A + \frac{1}{A}$  to achieve better performance, especially in low SNR scenarios. On the other hand, the Zero Forcing equalizer, regardless of how small A becomes, continues to divide by  $\frac{1}{A}$ , often resulting in nonsensical outputs, especially as A approaches zero.

As we increase the SNR, let's say by setting A to 0.1, the red blob, representing the MMSE output, becomes more discernible, but still, you can see that the MMSE equalizer restricts the red points from spreading too far.

(Refer Slide Time: 10:34)

Week 09: Lecture 47

## Linear equalization

- Consider a block of five consecutive samples:

$$\mathbf{r}[k] = b[k-1] \begin{bmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ 0 \\ 0 \\ 0 \end{bmatrix} + b[k] \begin{bmatrix} 0 \\ 1 \\ \frac{1}{2} \\ -\frac{1}{2} \\ 0 \end{bmatrix} + b[k+1] \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ \frac{1}{2} \end{bmatrix} + \mathbf{w}[k] = \mathbf{U}\mathbf{b}[k] + \mathbf{w}[k]$$

where  $\mathbf{w}[k]$  are Gaussian noise samples and  
 $\mathbf{b}[k] = [b[k-1], b[k], b[k+1]]^T$ .

6

10:34 / 28:55

(Refer Slide Time: 10:54)

Week 09: Lecture 47

## Linear equalization

- (Contd.) for five consecutive samples:

$$\mathbf{r}[k] = \mathbf{U}\mathbf{b}[k] + \mathbf{w}[k]$$

where  $\mathbf{w}[k]$  are Gaussian noise samples and  
 $\mathbf{b}[k] = [b[k-1], b[k], b[k+1]]^T$  and

$$\mathbf{U} = \begin{bmatrix} \frac{1}{2} & 0 & 0 \\ -\frac{1}{2} & 1 & 0 \\ 0 & \frac{1}{2} & 0 \\ 0 & -\frac{1}{2} & 1 \\ 0 & 0 & \frac{1}{2} \end{bmatrix}$$

7

10:54 / 28:55

In contrast, the Zero Forcing equalizer allows the points to spread significantly, particularly in low SNR conditions, illustrating its susceptibility to noise.

As the SNR continues to increase, the differences between the two equalizers diminish, but it remains clear that the MMSE approach consistently offers a more robust performance across varying SNR levels.

This intuitive explanation highlights why the Zero Forcing (ZF) approach, which aims to cancel out the channel effects regardless of the noise, is not ideal for low SNR (Signal-to-Noise Ratio) scenarios. At very low SNRs, simply inverting the channel, even if it's a single-tap channel, can lead to suboptimal results. However, as the SNR increases and moves into higher regimes, the difference between the ZF and MMSE (Minimum Mean Square Error) equalizers diminishes. In fact, when the SNR is very high, say, close to 10, you'll notice that the ZF and MMSE equalizers behave almost identically, with only a slight difference, if any. This observation confirms our intuition that while ZF and MMSE equalizers converge in performance at high SNRs, they employ significantly different strategies to handle the problem at low SNRs.

This was a simplified overview. Let's now shift our focus to the running example we've been discussing, where the channel has coefficients of 1, 0.5, and -0.5. We'll examine how the ZF and MMSE equalizers compare in this specific scenario. The example involves symbol transmission at a rate of half a symbol per second, and due to the channel characteristics, the received symbols are essentially convolved with  $P(t)$ , which lasts between 1 and 4 samples with values of 1, 0.5, and -0.5.

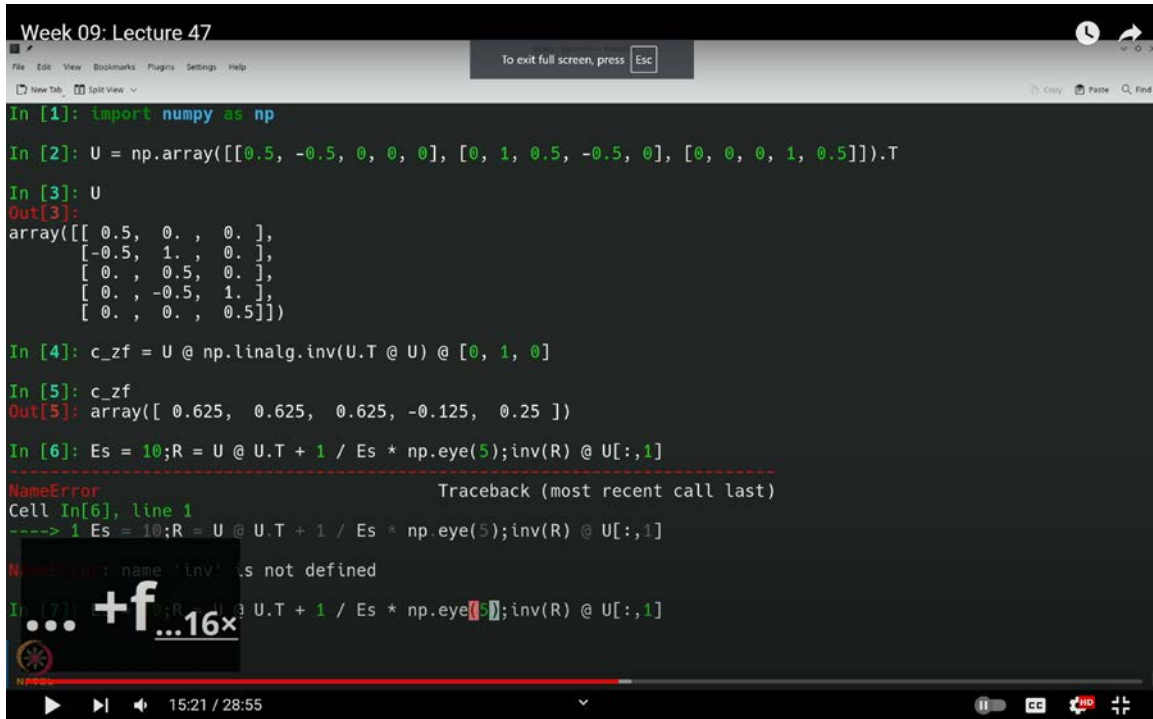
You may recall that we expressed this in the form of a matrix equation, which can be compactly written as:

$$U = \begin{pmatrix} 0.5 & -0.5 & 0 & 0 & 0 \\ 0 & 0.5 & -0.5 & 0 & 0 \\ 0 & 0 & 0.5 & -0.5 & 0 \\ 0 & 0 & 0 & 0.5 & -0.5 \end{pmatrix}$$

Here,  $R_k$  can be written as  $UB_k + W_k$ , where the symbol being detected,  $B_k$ , corresponds to the middle column of  $U$ .

With this in mind, let's open a Python prompt and piece this together to compare the actual MMSE and ZF equalizers. We'll begin by importing the necessary libraries, such as `numpy`.

(Refer Slide Time: 15:21)



```

Week 09: Lecture 47
File Edit View Bookmarks Plugins Settings Help
To exit full screen, press Esc
In [1]: import numpy as np
In [2]: U = np.array([[0.5, -0.5, 0, 0, 0], [0, 1, 0.5, -0.5, 0], [0, 0, 0, 1, 0.5]]).T
In [3]: U
Out[3]:
array([[ 0.5,  0. ,  0. ],
       [-0.5,  1. ,  0. ],
       [ 0. ,  0.5,  0. ],
       [ 0. , -0.5,  1. ],
       [ 0. ,  0. ,  0.5]])
In [4]: c_zf = U @ np.linalg.inv(U.T @ U) @ [0, 1, 0]
In [5]: c_zf
Out[5]: array([ 0.625,  0.625,  0.625, -0.125,  0.25 ])
In [6]: Es = 10; R = U @ U.T + 1 / Es * np.eye(5); inv(R) @ U[:,1]
-----
NameError                                Traceback (most recent call last)
Cell In[6], line 1
----> 1 Es = 10; R = U @ U.T + 1 / Es * np.eye(5); inv(R) @ U[:,1]

NameError: name 'inv' is not defined
In [7]: ... +f ...16x

```

First, let's write our matrix  $U$ . Remember that the columns of  $U$  correspond to 0.5, -0.5, and so on, but the first column specifically contains 0.5, -0.5, followed by three zeros. The second column is [0, 0.5, -0.5, 0, 0], and the last column is three zeros followed by 1 and 0.5. We'll write these columns in row form and then transpose the matrix.

So,  $U$  can be defined as follows:

$$U = \text{np.array} \left( \begin{bmatrix} 0.5 & -0.5 & 0 & 0 & 0 \\ 0 & 0.5 & -0.5 & 0 & 0 \\ 0 & 0 & 0.5 & -0.5 & 0 \\ 0 & 0 & 0 & 0.5 & -0.5 \end{bmatrix} \right)^T$$

This transpose ensures that we correctly format the matrix as needed. If you inspect your matrix  $U$  now, you can verify that it matches the one we've discussed in the slides.

Next, let's revisit the ZF equalizer that we evaluated earlier. You might recall that the ZF equalizer, denoted as  $C_{ZF}$ , can be calculated using the formula:

$$C_{ZF} = U^{-1} \times \text{np.array}([0,1,0])$$

Here, the `@` operator in recent versions of Python allows for matrix multiplication. Additionally, we can take the transpose because the Hermitian and transpose are equivalent for real matrices.

If you compute  $C_{ZF}$ , you'll find that it results in values such as  $\frac{5}{8}, \frac{5}{8}, \frac{5}{8}, -\frac{1}{8}, \frac{2}{8}$ , which matches the solution we derived earlier.

Now, let's evaluate the MMSE equalizer using a similar approach. Remember, to compute the MMSE equalizer, we'll need the  $E_s$  term, which corresponds to the signal-to-noise ratio (SNR). Let's proceed to write out the formula for the MMSE equalizer.

Let's assume a signal energy  $E_s$ , say  $E_s = 10$ , which corresponds to 10 dB. To calculate the matrix  $R$ , we will use  $R = U \times U^T + \frac{1}{E_s} \times C_w$ , where  $\frac{1}{E_s}$  is multiplied by the noise covariance matrix  $C_w$ . Given that our noise is independent and identically distributed (iid) across samples, we can take the noise covariance matrix to be the identity matrix. Next, we need to compute  $R^{-1} \times P$ , which in this case is  $R^{-1} \times U_0$ .

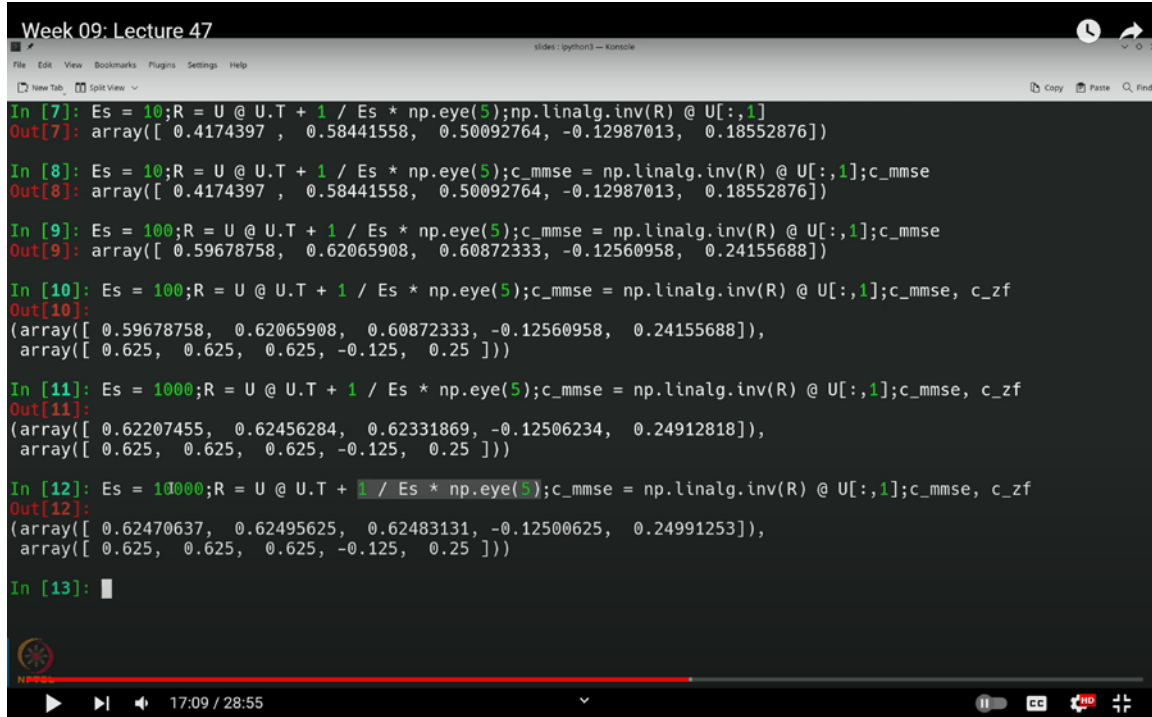
Let's proceed with this calculation. To get  $U_0$ , which corresponds to  $U_k$ , remember we are interested in  $b_k$ . In our matrix  $U$ , the middle column multiplies  $b_k$ , the preceding column multiplies  $b_{k-1}$ , and the following column multiplies  $b_{k+1}$ . Therefore, we need to extract the middle column of  $U$ . This can be done by specifying  $U[:, 1]$ , where the colon selects all rows and the 1 indicates the second column.

Once we have  $U_0$ , we can evaluate the inverse of  $R$  using `np.linalg.inv()`. Let's store this result as `cmmse` and display it.

Now, let's increase the SNR to 100, which corresponds to 20 dB. You'll notice that the coefficients start to change, they begin to resemble those of the Zero Forcing equalizer. To illustrate this, let's compare the Zero Forcing equalizer directly below. As you can see, the

coefficients start to converge. If we further increase the SNR to 1000 (30 dB), you'll observe that the coefficients become almost indistinguishable. And if we push the SNR to 10,000 (40 dB), the coefficients are nearly identical to those of the Zero Forcing equalizer.

(Refer Slide Time: 17:09)



The screenshot shows a video player with a Jupyter Notebook interface. The title bar indicates 'Week 09: Lecture 47'. The notebook contains several code cells and their outputs:

```

In [7]: Es = 10; R = U @ U.T + 1 / Es * np.eye(5); np.linalg.inv(R) @ U[:,1]
Out[7]: array([ 0.4174397,  0.58441558,  0.50092764, -0.12987013,  0.18552876])

In [8]: Es = 10; R = U @ U.T + 1 / Es * np.eye(5); c_mmse = np.linalg.inv(R) @ U[:,1]; c_mmse
Out[8]: array([ 0.4174397,  0.58441558,  0.50092764, -0.12987013,  0.18552876])

In [9]: Es = 100; R = U @ U.T + 1 / Es * np.eye(5); c_mmse = np.linalg.inv(R) @ U[:,1]; c_mmse
Out[9]: array([ 0.59678758,  0.62065908,  0.60872333, -0.12560958,  0.24155688])

In [10]: Es = 100; R = U @ U.T + 1 / Es * np.eye(5); c_mmse = np.linalg.inv(R) @ U[:,1]; c_mmse, c_zf
Out[10]: (array([ 0.59678758,  0.62065908,  0.60872333, -0.12560958,  0.24155688]),
          array([ 0.625,  0.625,  0.625, -0.125,  0.25 ]))

In [11]: Es = 1000; R = U @ U.T + 1 / Es * np.eye(5); c_mmse = np.linalg.inv(R) @ U[:,1]; c_mmse, c_zf
Out[11]: (array([ 0.62207455,  0.62456284,  0.62331869, -0.12506234,  0.24912818]),
          array([ 0.625,  0.625,  0.625, -0.125,  0.25 ]))

In [12]: Es = 10000; R = U @ U.T + 1 / Es * np.eye(5); c_mmse = np.linalg.inv(R) @ U[:,1]; c_mmse, c_zf
Out[12]: (array([ 0.62470637,  0.62495625,  0.62483131, -0.12500625,  0.24991253]),
          array([ 0.625,  0.625,  0.625, -0.125,  0.25 ]))

In [13]:

```

The video player interface at the bottom shows a progress bar at 17:09 / 28:55.

Why does this happen? The reason is that the expression  $R^{-1} \times P$ , where  $P = U_0$ , essentially simplifies to the same expression as  $U^T \times U^{-1} \times E_0$  in the Zero Forcing case, as the contribution of the noise term diminishes to zero. You can verify this numerically or intuitively, when there is no noise, the optimal strategy is simply to cancel the interference entirely.

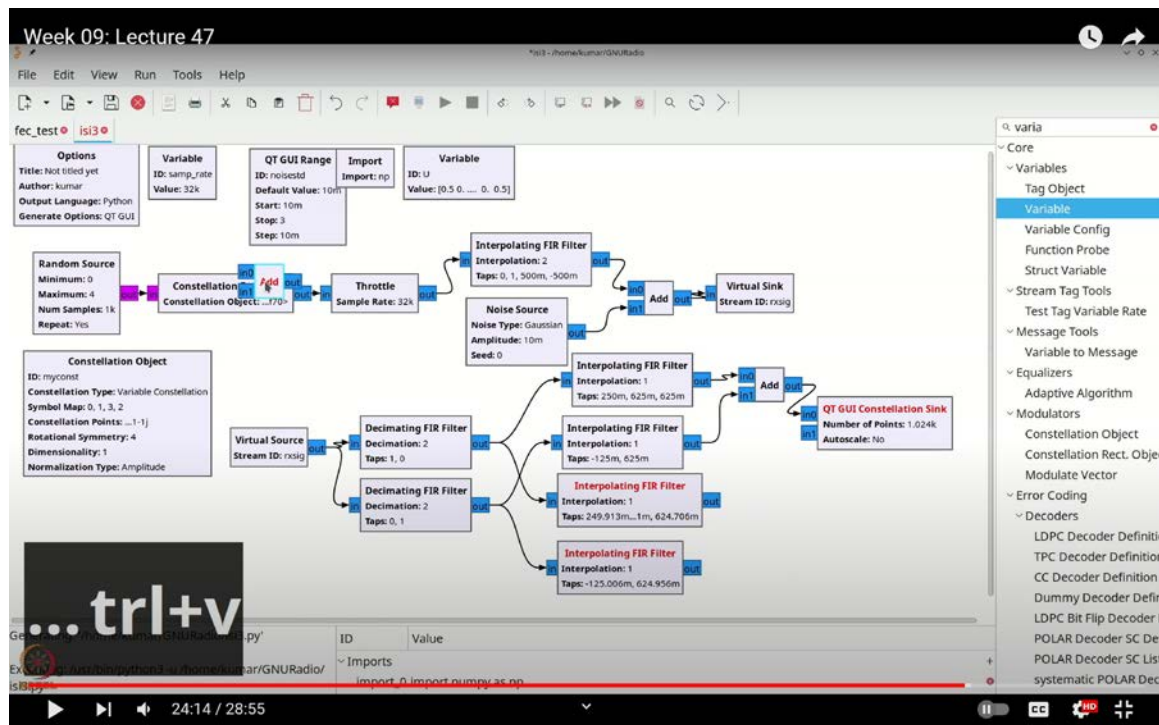
Thus, you can use this numerical approach to confirm that the Zero Forcing and MMSE equalizers are nearly identical at very high SNR levels. To push this even further, let's add another zero to the SNR, increasing it to 50 dB. At this level, the difference between the two equalizers is so minimal that their coefficients differ only in the fifth or sixth decimal place.

Our next task will be to implement this in a flow graph to directly compare the performance

of the Zero Forcing and MMSE equalizers. We will build on the flow graph we used earlier for the Zero Forcing equalizer. If you don't have this flow graph, I encourage you to revisit the related GNU Radio lecture on Zero Forcing and follow along to construct it. This will allow us to expand upon it for the comparison with the MMSE equalizer.

The first step here will be to add a variable corresponding to the matrix  $\mathbf{U}$ . Let's use Control-F (or Command-F on Mac) to find where to insert this variable. But before proceeding, let's ensure we've imported `numpy`, as it will be very useful.

(Refer Slide Time: 24:14)



Let's begin by importing the necessary libraries. We'll start with the import block, specifically `import numpy as np`. With `numpy` imported, the first step is to create the variable. You can do this by using Control-F (or Command-F on a Mac) to search for the variable insertion point. We'll name this variable `u` and assign it a value using `np.array`, following the same format we used earlier for the matrix columns. After defining the first, second, and third columns, we'll transpose the matrix to complete the setup. Now, our matrix is ready.

The next task is to actually construct the filter. To achieve this, we'll need to account for the SNR (Signal-to-Noise Ratio). The noise standard deviation (noise std) will serve as our proxy for SNR, providing a measure of noise power. Given that we are using a normalized constellation, the signal power is 1. Therefore, our  $E_s$  will be simply  $\frac{1}{\text{noise std}^2}$ .

Now, to perform a comparison between the Zero Forcing and MMSE equalizers, we'll first remove any extra constellation points. We'll adjust the inputs so that there are only two: the first corresponding to the Zero Forcing based constellation, and the second to the MMSE based constellation. For the MMSE based constellation, we'll create another pair of interpolating FIR filters that use coefficients determined by the MMSE equalizer.

We'll start by duplicating the current setup with Control-C and Control-V. These coefficients, however, will be calculated at runtime. Double-click to edit the filter block, and instead of static coefficients like 2/8, 5/8, 5/8, we'll calculate them by performing  $R^{-1} \times P$ . Specifically, we'll compute this with `np.linalg.inv()` on  $U \times U^T + \frac{1}{\text{noise std}^2} \times \text{np.i5}$ . The result is the inverse multiplied by  $U_1$ .

Let's see how this works. If you encounter an issue, such as a singular matrix when the noise std is 0, don't worry, this can be resolved by setting the noise std to a small value like 0.01, which will avoid zero-related errors. Now, everything should run smoothly.

However, we don't need all the coefficients for this particular filter. We only need the first, third, and fifth coefficients. To do this, we'll first reverse the sequence and then select the coefficients in reverse order. To reverse the sequence, simply use `[::-1]`. After that, to select every other coefficient, use `::2`.

As a sanity check, when the SNR is very low, you'll notice that the coefficients should approximate 0.25 for the first and 0.625 for the last. Now, I'll duplicate this interpolating filter with Control-C and paste it. Then, by double-clicking to edit, I'll adjust the selection to include the first, third, and fifth coefficients instead of the first, third, and fourth.

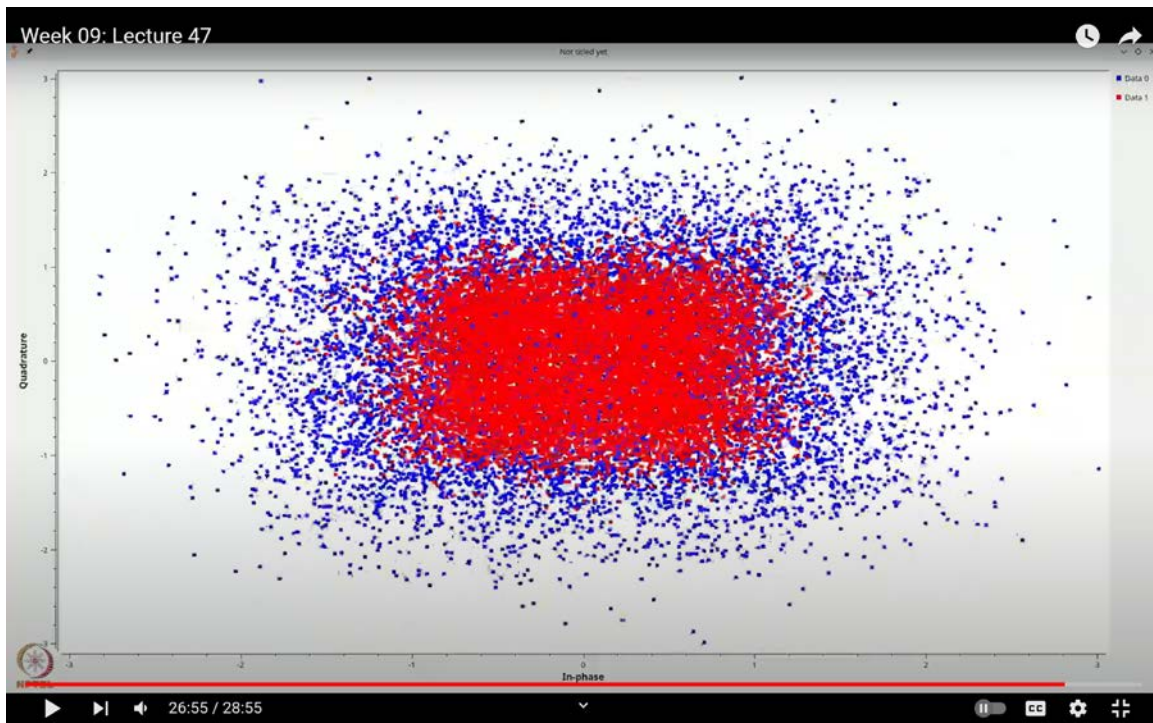
I'm going to start by setting a value of one here. Using `1::2` ensures that I get the second and fourth coefficients. This results in values of -0.125 and -0.625, which simplifies the

process of constructing the MMSE equalizer significantly.

Let's make everything visible for better clarity. Now that we have the MMSE equalizer set up, we need to connect the corresponding filters. To do this, I'll add an adder block using Control-C and Control-V to sum these two signals. We can then proceed to compare the constellations.

Next, let's make the noise standard deviation (**`noise std`**) a variable. We'll delete the current setting, use Control-F (or Command-F), and search for "variable." We'll define this variable as **`noise std`** and set its initial value to 0.1. At this setting, you'll notice that the constellations look somewhat similar. However, the real insight comes when the SNR degrades further.

(Refer Slide Time: 26:55)



Let's increase the **`noise std`** to 0.3 and also raise the number of samples to 10,000. You'll observe that both constellations exhibit some spread. On closer inspection, the red constellation (Zero Forcing) shows slightly less spread compared to the blue constellation (MMSE). By increasing the number of constellation points and examining the results,

you'll see that the spread for the red constellation is indeed slightly less.

This difference arises because the MMSE equalizer considers the impact of noise. Increasing the ``noise std`` to 0.4 places us in a lower SNR range. At this level, the blue points (MMSE) spread much farther than the red points (Zero Forcing), indicating that the MMSE equalizer performs somewhat better. While the improvement might not be dramatic, it's noticeable. With an even higher noise level, such as 0.8, the performance of both equalizers deteriorates, but the MMSE equalizer still manages to keep the red points from spreading as much as the blue ones. Increasing the sample rate helps visualize this effect more clearly, revealing that at high noise levels, the blue points scatter widely due to significant noise amplification, while the red points remain somewhat more concentrated.

It's important to note that at very low SNRs, the MMSE equalizer also struggles because it doesn't fully mitigate the noise impact. However, it still generally outperforms the Zero Forcing equalizer. One point to remember is that minimizing the symbol error rate optimally requires maximum likelihood sequence estimation, such as the Viterbi algorithm. The MMSE equalizer minimizes the mean squared error, which isn't the same as minimizing the symbol error rate directly.

In summary, Zero Forcing equalizers work well at very high SNRs, MMSE is effective at medium SNRs, and both approaches converge at high SNRs. At very low SNRs, both perform poorly, and when SNR is extremely low, you might need to accept reduced data rates or performance. In this lecture, we explored the MMSE equalizer for both single-tap and more complex channel responses. The MMSE equalizer has notable advantages over Zero Forcing in low SNR scenarios due to its ability to limit noise enhancement, even though it's suboptimal. At high SNRs, the MMSE and Zero Forcing equalizers perform similarly. Using suboptimal equalizers is a practical approach that simplifies receiver implementation, and you can further explore this through various GNU Radio blocks. Thank you.