Digital Communication using GNU Radio

Prof. Kumar Appaiah

Department of Electrical Engineering

Indian Institute of Technology Bombay

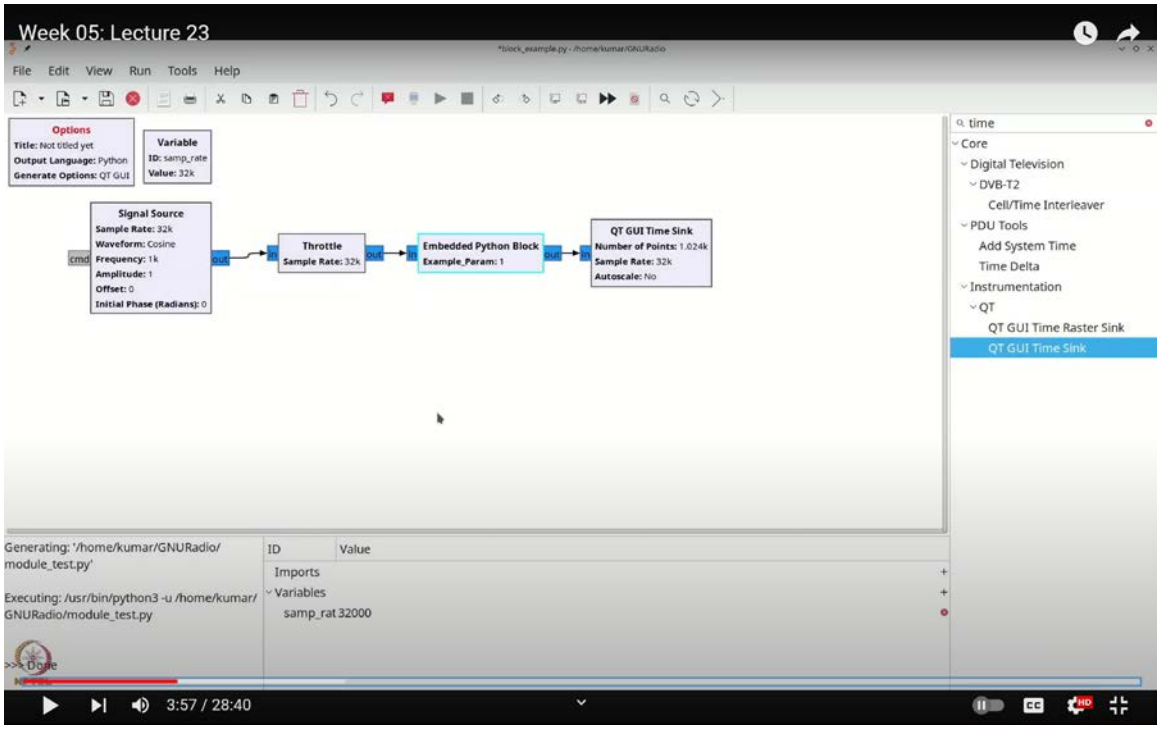
Week-05

Lecture-23

Extending GNU Radio Features using Python

Welcome to this lecture on Digital Communication using GNU Radio. Up to this point, you've been working with various blocks in GNU Radio to accomplish different functionalities. However, we haven't delved deeply into the inner workings of these blocks. While we have a general idea, such as filtering blocks performing convolution or amplification blocks multiplying samples, we haven't explored their details.

(Refer Slide Time: 03:57)



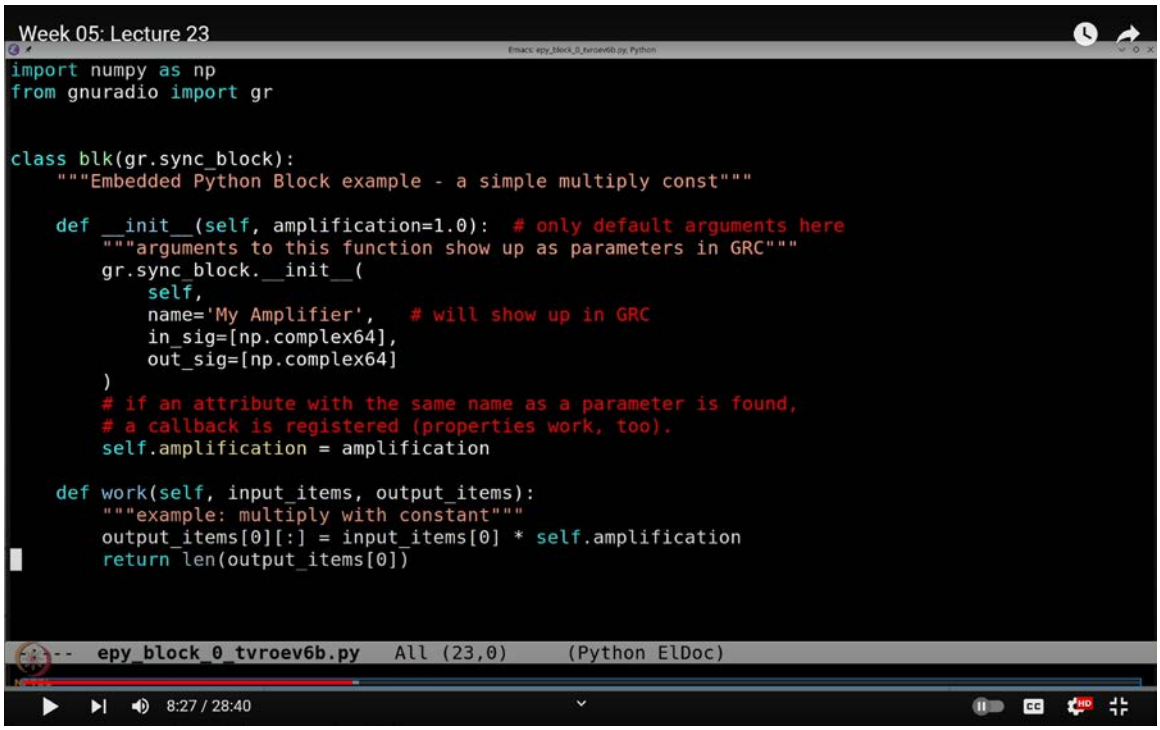
Sometimes, you might find that implementing a specific functionality using GNU Radio's built-in blocks can be cumbersome. In such cases, it might be more efficient to express that

functionality with a bit of Python code. This lecture will introduce you to the concept of extending GNU Radio with Python blocks, focusing on how Python can be used to achieve functionality that may be complex or inefficient to implement with built-in blocks alone.

GNU Radio blocks are generally written in C++ for performance reasons, but Python can be a powerful tool for adding simpler functionality. By incorporating Python blocks, you can extend GNU Radio's capabilities with minimal effort and just a few lines of code.

In this session, we will explore one of GNU Radio's most powerful features: the embedded Python block. This feature allows you to create custom blocks with Python, providing a compact and flexible way to achieve powerful transformations and operations without relying solely on pre-built GNU Radio blocks. This approach is especially useful for operations that are challenging to implement with built-in blocks.

(Refer Slide Time: 08:27)



Let's dive into how you can use embedded Python blocks to extend GNU Radio's functionality effortlessly.

Let's walk through using the embedded Python block in GNU Radio. Start by pressing **`Ctrl-F`** (or **`Command-F`** on a Mac) and typing "block" to locate the Python block found under Core > Miscellaneous. Drag this block onto your workspace; it’s labeled as the "Embedded Python Block."

Now, let’s set up a basic flowgraph. Add a throttle by using **`Ctrl-F`** (or **`Command-F`**) and searching for "throttle." Next, add a signal source and a time sink in a similar fashion. Connect these components as follows: the signal source to the embedded Python block, and then the embedded Python block to the throttle and the time sink.

Double-click the embedded Python block to open its configuration window. Set the example parameter to 1.0 and try to run the flowgraph. You’ll encounter an error stating, "Block example.py must begin with a letter and may contain letters, numbers, and so on." This indicates that the current block setup is not correctly configured.

To resolve this, double-click the embedded Python block again and select "Open in Editor." This action will open the block in your default Python editor. Here, you'll see boilerplate code, including a string at the beginning used for documentation purposes. In Python, such strings are typically used for documentation within code.

The embedded Python block's code is designed to be instantiated by GNU Radio Companion. When you save this file, GNU Radio Companion will use the first class it finds to determine the block’s ports and parameters. These parameters should all have default values.

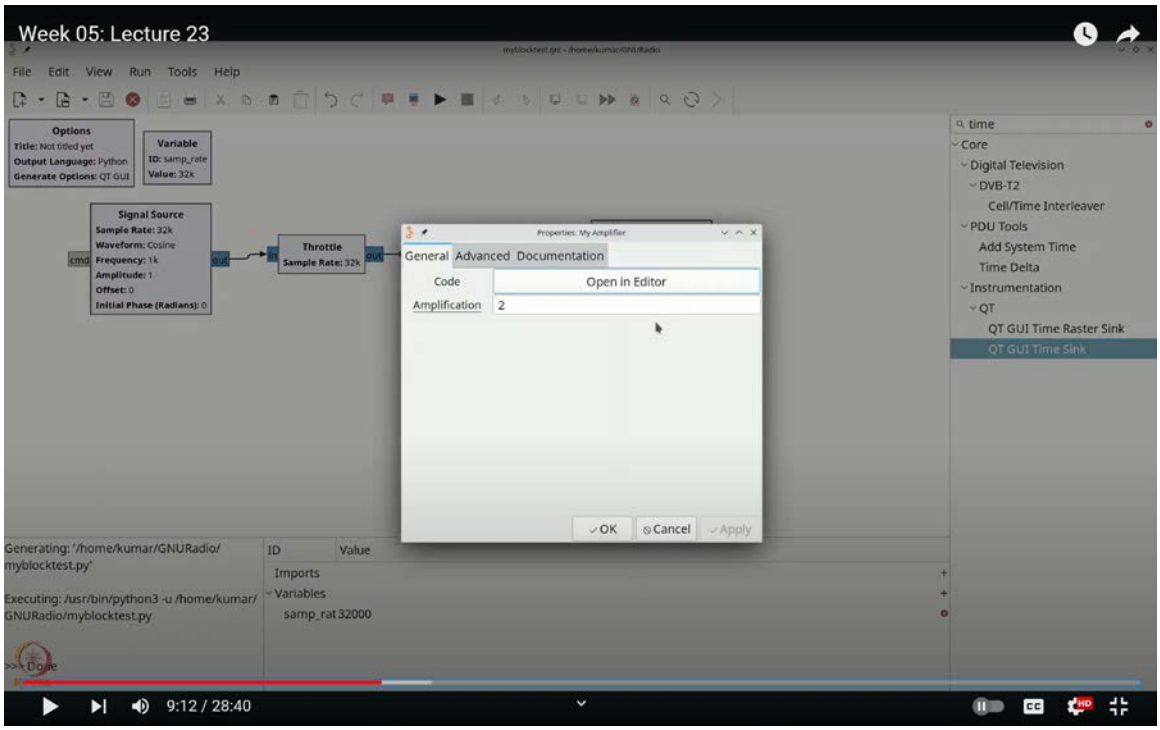
You can now remove the initial documentation string. The key point here is that the class defined in this file serves as the Python object that GNU Radio Companion uses to interact with your block. This class definition is essential for the functionality of the embedded Python block.

The embedded Python block is a subclass of **`gr.sync_block`**, which means it processes samples at a consistent rate: it takes in samples and produces outputs at the same rate. In GNU Radio, there are other base classes such as **`basic_block`**, **`decim_block`**, and **`interp_block`**. The **`basic_block`** class is more general, while **`decim_block`** and

`interp_block` are used for decimation and interpolation, respectively, which involve producing fewer or more samples than the input.

Let's clean up the initial boilerplate code. The block currently includes a string labeled "embedded Python block example," and a basic example for multiplying by a constant. If we dive into the code, we see the **`__init__`** method, also known as the constructor. This method is crucial as it initializes the block when GNU Radio first creates the object.

(Refer Slide Time: 09:12)



You need to use **`gr.sync_block.__init__`** and **`self`** to call the sync block's initialization function. For clarity, let’s rename "embedded Python block" to "myamplifier."

The **`insig`** and **`outsig`** attributes specify the data types for input and output samples, respectively. In this case, both **`insig`** and **`outsig`** are set to **`np.complex64`**, which we'll keep unchanged.

The **`self.param`** assignment is used to store parameters passed from GNU Radio Companion. For instance, if you have a parameter named **`example_param`** with a value

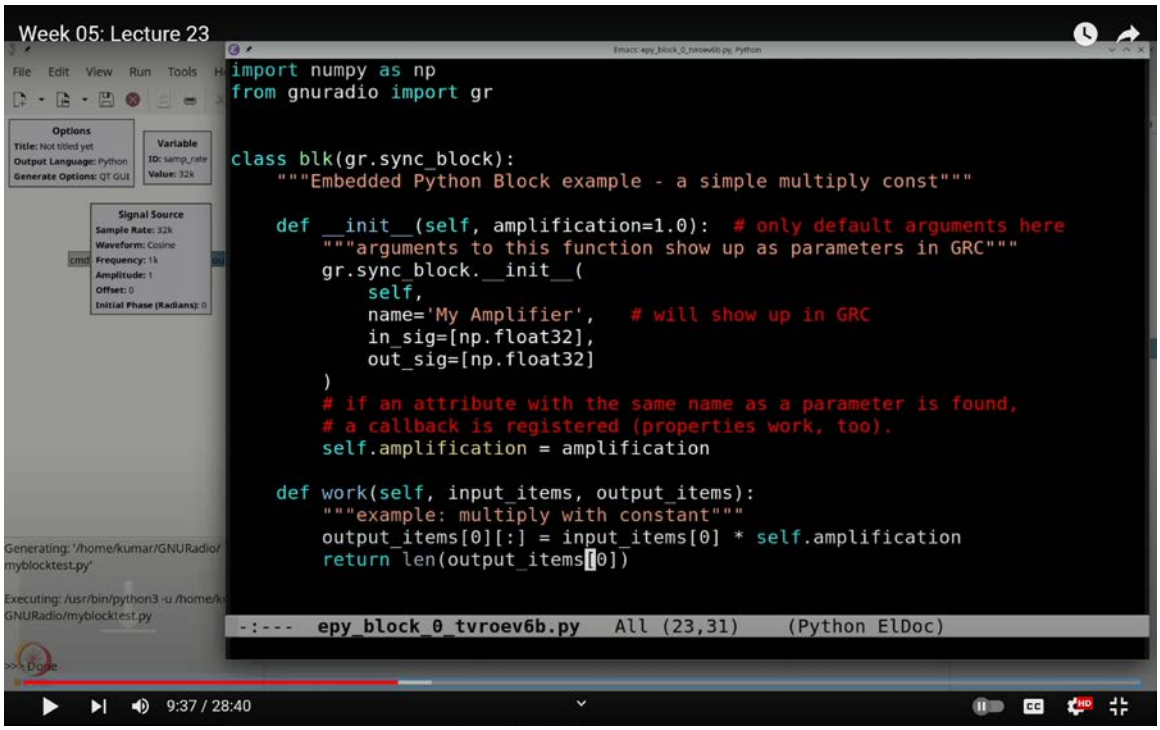
of 1.0, you should store it as **`self.example_param = example_param`**. This approach retains the parameter value within the object instance.

Now, let's adjust the parameter name from **`example_param`** to **`amplification`**. Make the corresponding changes throughout the code: update **`example_param`** to **`amplification`** in the class definition, and ensure all references to the parameter are updated accordingly.

Finally, we have the **`work`** method. This method is central to the functionality of your block, handling all the processing tasks.

In GNU Radio, input and output items are not strictly numpy arrays; rather, they are like lists of numpy arrays. Each list contains elements corresponding to the block's dimensions. For example, if **`insig`** is specified as **`np.complex64`**, it implies that the input items contain a single element, which is a numpy array. Similarly, if **`outsig`** is **`np.complex64`**, it means the output items also contain a single element, which is a numpy array.

(Refer Slide Time: 09:37)



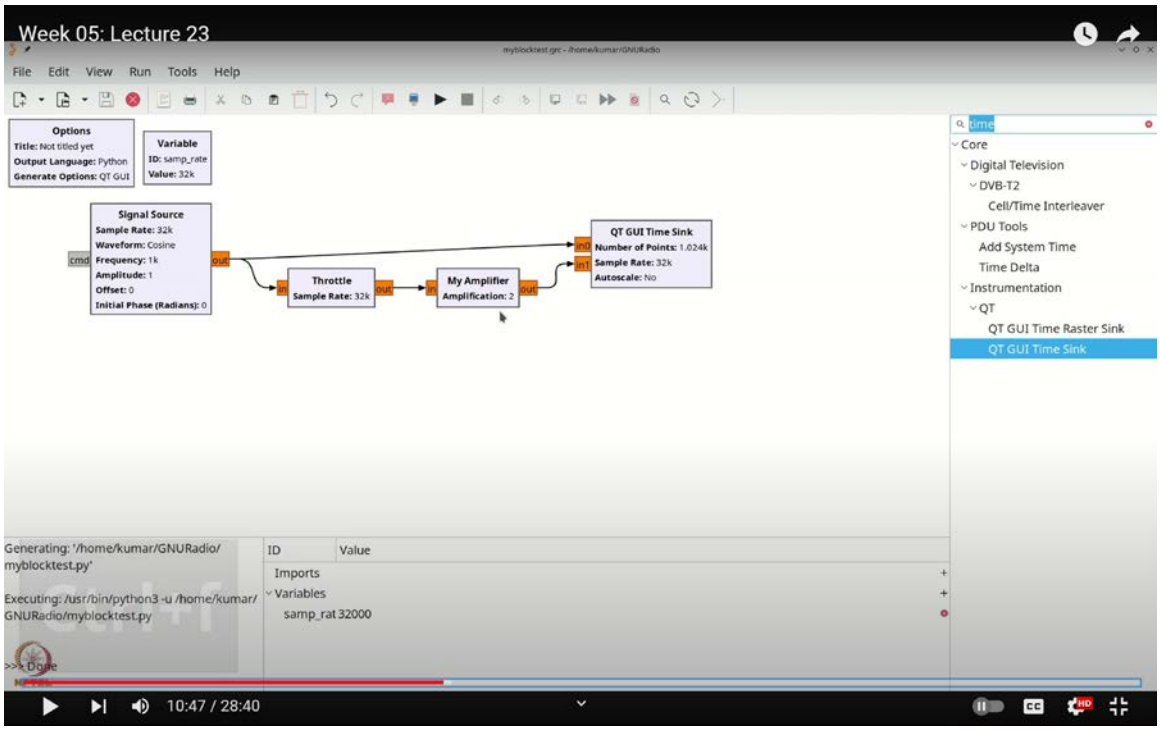
This notation allows you to access and modify the first element of these lists directly. For instance, the line:

output_items[0][:] = input_items[0] * self.amplification

This line writes into the first output item by multiplying the first input item by the **`self.amplification`** parameter. The **`[:]`** notation is used to replace the content of the output item with the computed values. The **`return len(output_items[0])`** statement specifies to GNU Radio how many items are being returned. As a sync block, it is expected that the number of output items matches the number of input items for each stream.

After making these changes, save and exit the editor, and execute the block. You'll observe that it processes the complex signal correctly, as indicated by the red and blue traces. If you adjust the amplification parameter to 2, you should see the output appropriately amplified.

(Refer Slide Time: 10:47)



However, if you want the amplifier to work with real samples instead of complex ones,

you need to modify the block accordingly. This adjustment ensures that the amplification is applied to real-valued signals rather than complex ones, avoiding the combined effects of cosine and sine components.

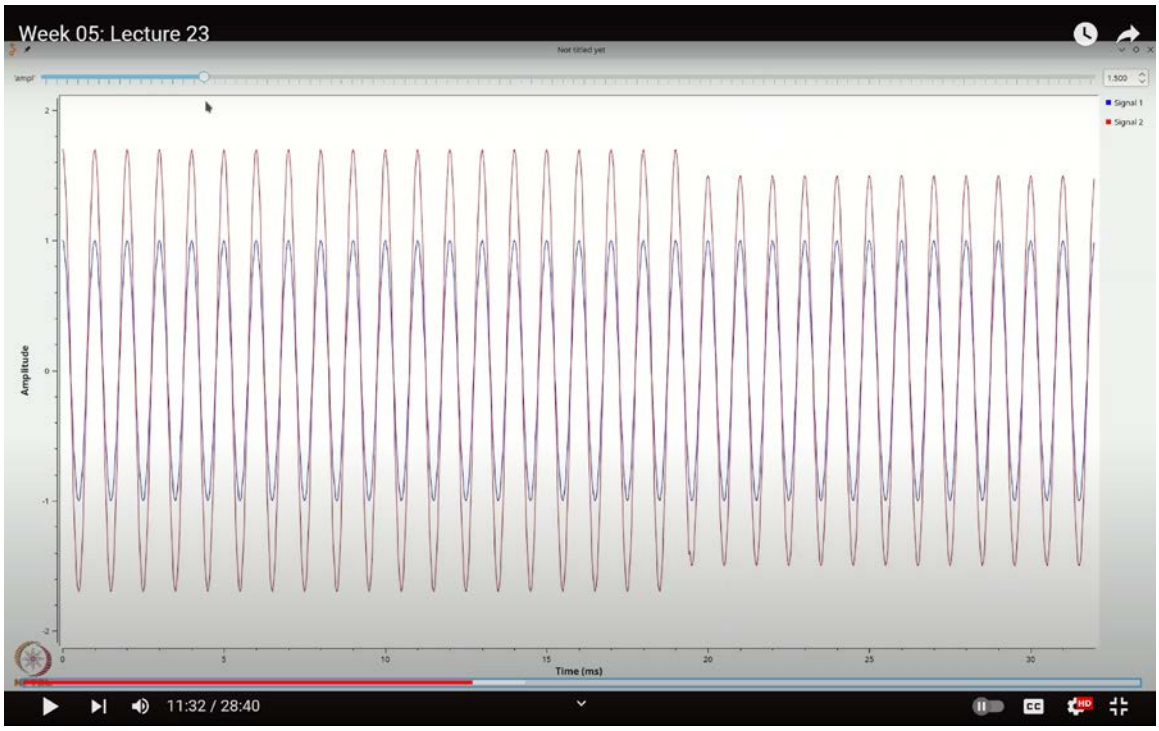
To adapt the block for real signals, follow these steps: double-click the block, select "Open in Editor," and change the data types from **`np.complex64`** to **`np.float32`**:

self.insig = np.float32

self.outsig = np.float32

You don't need to modify the rest of the code, as GNU Radio will handle the conversion of these numpy arrays to float arrays instead of complex arrays. Save the changes and click "OK." Notice that GNU Radio has disconnected the blocks due to the type change, with the color now showing orange to indicate that the data type is float.

(Refer Slide Time: 11:32)



Next, update all the related blocks to use float types: start with the Signal Source, then adjust the Throttle, and finally modify the Time Sink. For the Time Sink, add a second

input. Connect the original input to the first and the amplified signal to the second. When you run the flow graph, you'll observe that the second signal is amplified by a factor of 2 as expected.

To further enhance functionality, let's introduce a variable amplification factor. Press Ctrl+F (or Command+F), search for "range," and select a QTGUI Range widget. Double-click it and rename it to "AMPL" for convenience. Set its default value to 1, with a range from 0 to 10 and a step size of 0.1. Update the block code to use "AMPL" instead of a fixed amplification value:

self.ampl = self.AMPL

Execute the flow graph again. When you adjust the amplification using the QTGUI Range, you'll see that the signal's amplitude changes accordingly. If you set the amplification below 1, the signal's amplitude decreases. To add more parameters, simply double-click the "My Amplifier" block and make further adjustments as needed.

You can open the block in the editor. Next, let’s add a feature to introduce an offset into the signal. To achieve this, we need to modify the **`__init__`** function to accept an additional parameter called **`offset`**, which should default to 0. We’ll store this parameter with **`self.offset = offset`**. Finally, we will incorporate the offset into the signal by adding it to the output.

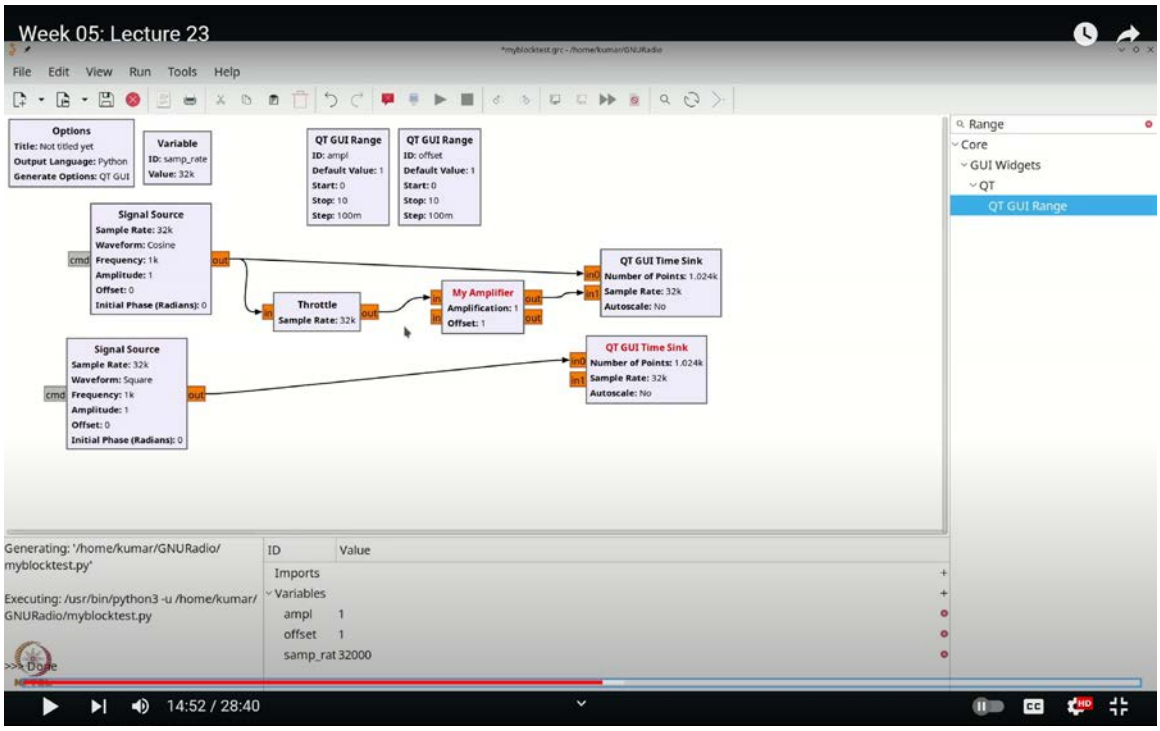
What does this change accomplish? It not only amplifies your signal by the amplification factor but also adds the specified offset. This means you can boost the signal by a certain amount, effectively raising its baseline level.

After making these adjustments, save the changes and exit the editor. We will now add a variable for the offset. Click to create a new variable, name it **`offset`**, and set its default value to 0.

If you execute the flow graph, you'll see that with the offset set to 1, the signal is shifted accordingly. Adjust the offset to 0, and the signal will match the original baseline. As you increase the offset, you’ll see the signal move up or down. You can also make the offset

negative if needed. By reducing the amplification and introducing an offset, you can now handle multiple parameters effectively.

(Refer Slide Time: 14:52)

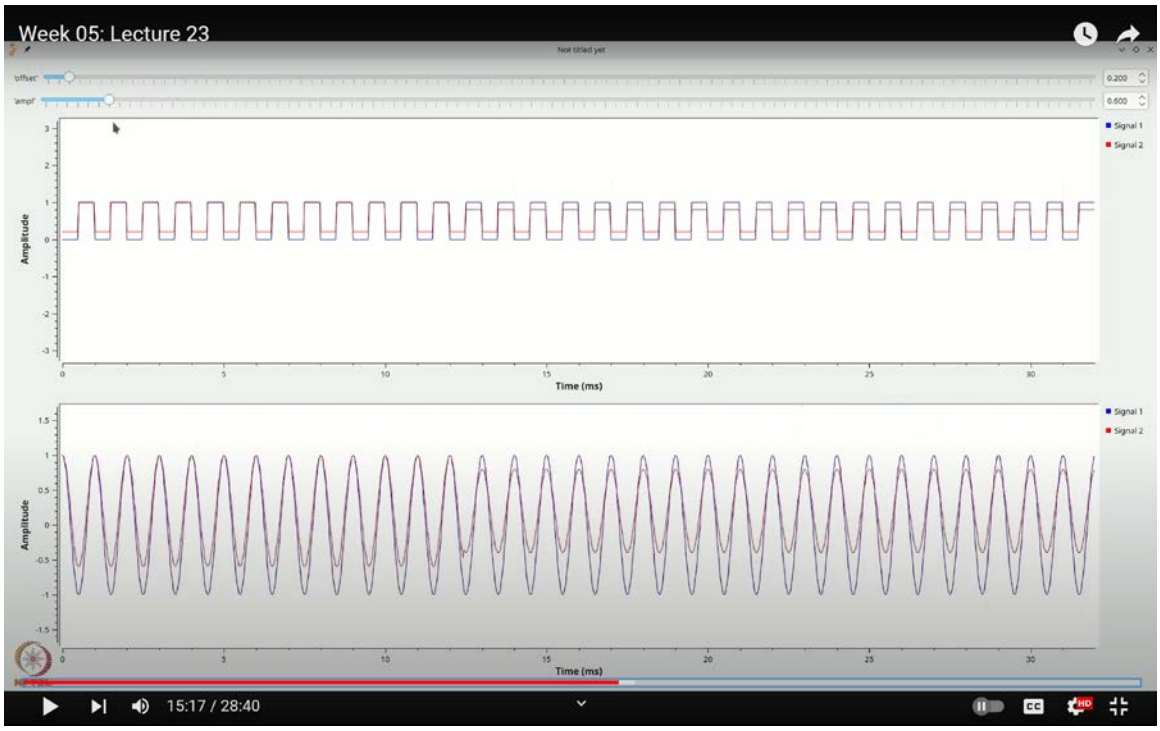


Let me show you how to modify this amplifier to handle two signals, just out of curiosity. Suppose you want to use your amplifier for both two input signals and two output signals. It's important to note that the inputs and outputs don't have to be identical, but for simplicity, let's assume they are. You'll use **`float32`** for both the input and output.

In this scenario, if you configure the amplifier with two input signals and two output signals, you can amplify both signals simultaneously. By adding a line to handle two signals and setting it to 1, you will enable the amplifier to process both signals in parallel.

Save your changes and apply them. You'll now see that your setup supports two signals. Let's take it a step further by expanding to four inputs. To do this, create a copy of the signal source by pressing **`Ctrl-C`** and **`Ctrl-V`**, and modify it to produce a square wave instead of a sine wave. Connect the original square wave to one input and the square wave with amplification and offset to another.

(Refer Slide Time: 15:17)



When you run the flow graph, both signals will experience the same offset and amplification. For example, increasing the offset will shift both the sine and square waves together, and increasing the amplification will boost both signals simultaneously.

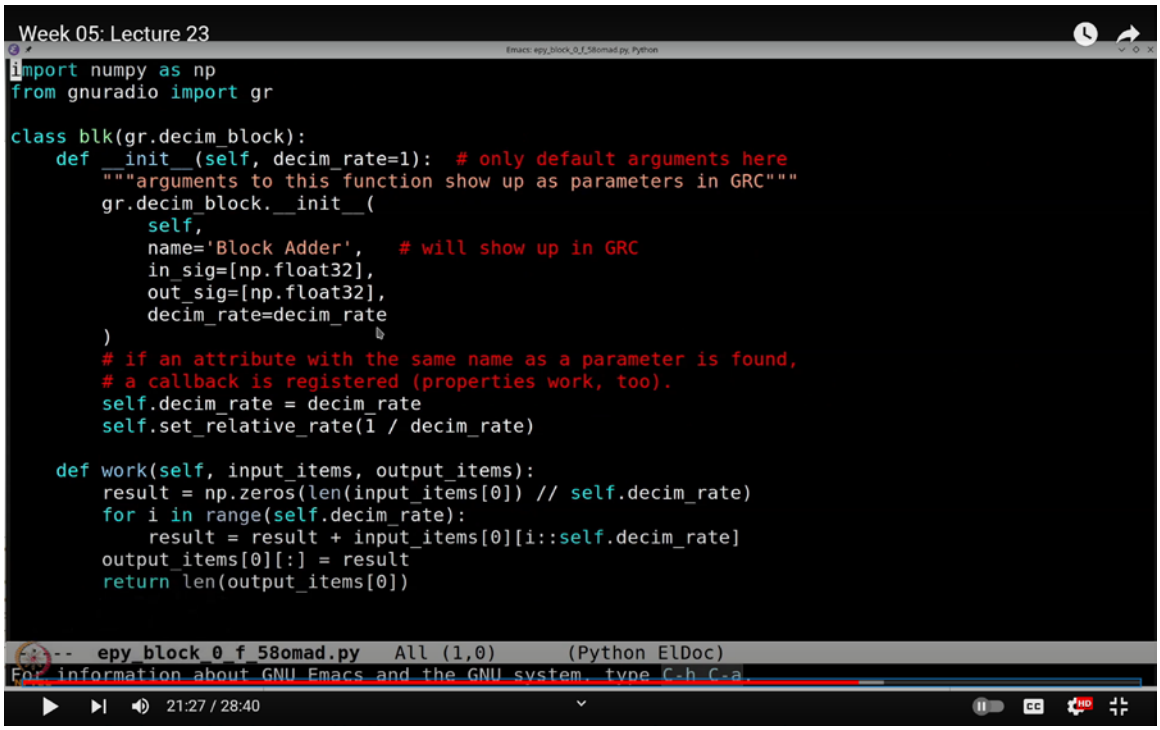
This demonstrates how you can create and use Python blocks with minimal effort. The key takeaway is that you can write custom Python functionality to process your arrays by simply opening the editor and coding as needed. While it's not mandatory to know, using Python blocks in GNU Radio can significantly simplify your implementation compared to managing code on your own.

There's an important caveat I'd like to mention. If you write inefficient or slow code within your Python blocks, it can adversely affect the overall performance of the entire block. So, it's crucial to ensure that your code is both concise and efficient.

To illustrate this further, let's consider an example with a decimation block to add some variety. We'll create a decimation block that performs block-based addition: for instance, it will take three numbers, sum them, and output the result; or take four numbers, sum

them, and output the result. Essentially, it will replace every set of four numbers with a single summed number, achieving the decimation.

(Refer Slide Time: 21:27)



Here’s how we can set this up:

1. Press **`Ctrl-F`** (or **`Command-F`**) to search for "block" and select the Python block from the core miscellaneous section.

2. Double-click the Python block, select "Open in Editor," and open it in your preferred Python editor.

Since we’re creating a decimation block, we need to make a few modifications:

- Instead of inheriting from **`gr.sync_block`**, we will derive from **`gr.decim_block`**.
- We’ll add one parameter: **`decimRate`**, which will be an integer representing the decimation rate.
- We will name this block **`adder`**, as it will perform addition based on blocks.

Keep the input and output types as **`float32`**, but add a new parameter **`decimRate`** that we need to pass to the superclass, **`gr.decim_block`**.

In the constructor, **`self.exampleParams`** should be replaced with **`self.decimRate`**. Also, specify to GNU Radio that the relative rate of this block is $(\frac{1}{\text{decimRate}})$. This tells GNU Radio that for every sample input, only one sample will be output. For example, if the decimation rate is 2, you'll receive one output sample for every two input samples.

This relative rate information is used by GNU Radio to determine the output length in the **`work`** function. Consequently, the **`input_items`** and **`output_items`** will automatically be sized according to the decimation rate.

In the **`work`** function, create an empty array and populate it with the results. For instance, initialize the result array with zeros, where the length of the array should be the number of input items divided by the decimation rate. Here's how you might do it:

```
result = np.zeros(len(input_items[0]) // self.decimRate)
```

Next, use a loop to accumulate the array elements. For example, if the decimation rate is 4, you'll sum elements from the zeroth index, the fourth index, the eighth index, and so on. This loop will add blocks of data and place the summed results into the result array.

This approach allows you to handle block-based operations efficiently while demonstrating how you can extend GNU Radio's capabilities with custom Python code.

Next, I will access the first element, the fifth element, the ninth element, and so on. To achieve this, I will use the slice notation in the code. Specifically, I'll set it to start at the **`i`**-th element, continue taking elements at intervals of the decimation rate, and then assign this to the output array.

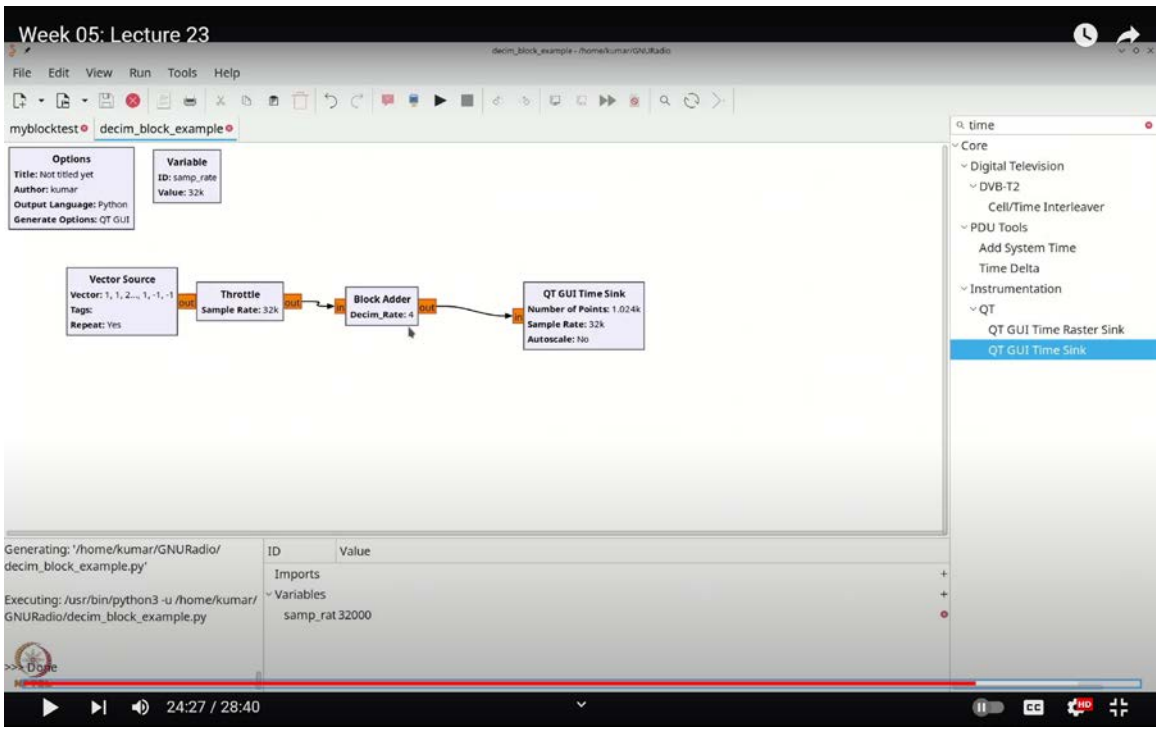
So, what does this particular slice notation mean? It means to start at the **`i`**-th element, retrieve all the elements, and step through them at the interval specified by the decimation rate. For example, in the first iteration of this loop, you will gather the zeroth element, the fourth element, the eighth element, and so on, assuming the decimation rate is 4. In the

subsequent iteration, you will retrieve the first element, the fifth element, the ninth element, and so forth.

In the result array, the first element will contain the sum of the 0th, 1st, 2nd, and 3rd elements from the input array. The second element will sum the 4th, 5th, 6th, and 7th elements, and so on. This process ensures that every group of four successive entries in the input items is summed up.

Let’s verify this setup. I’ll save the changes and exit the editor. If you encounter an "unexpected keyword" error, check the code for mistakes. For instance, the keyword should be **`decim`** instead of **`decimation`**, as per GNU Radio documentation.

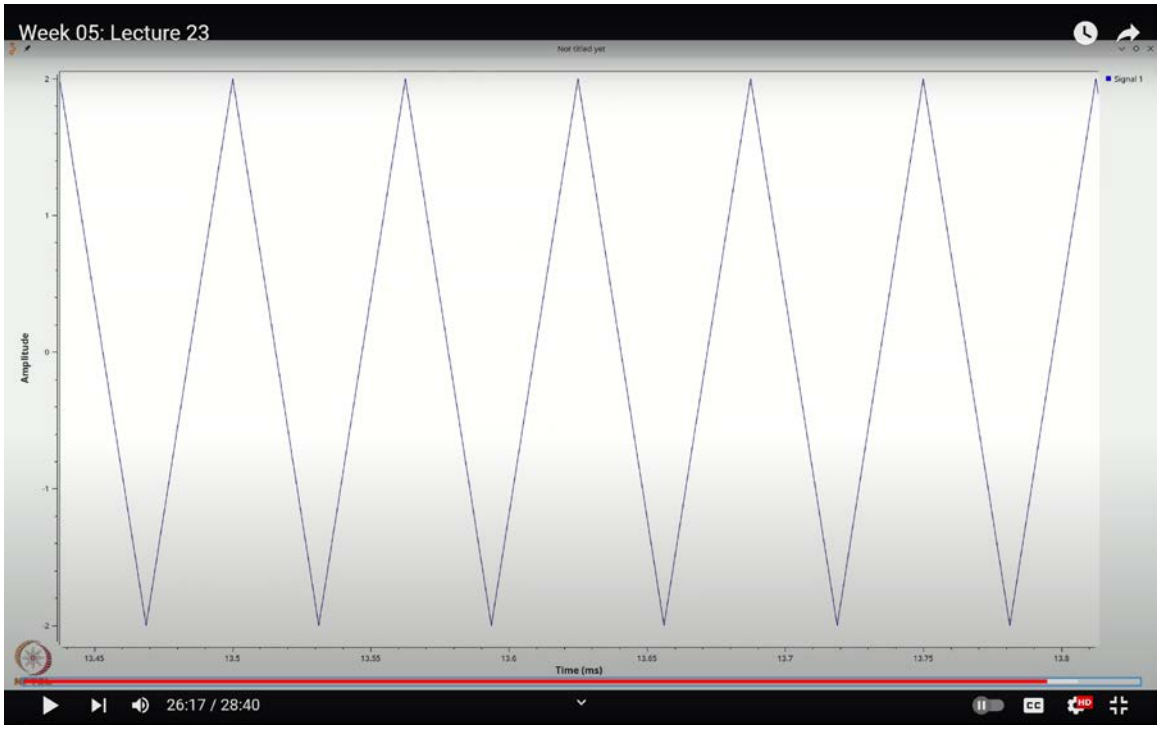
(Refer Slide Time: 24:27)



Now, let’s add a throttle block. We’ll set the throttle to operate with float inputs and outputs, then click "OK". Next, add a vector source block for easier control and to observe how the block behaves. Also, add a time sink block. Ensure that both the vector source and the time sink are set to float.

For the vector source, we'll provide elements as a Python list. For example, you might use **`[1, 1, 2, 3]`**. GNU Radio will handle this list and repeat it as needed. The resulting output will be the summation of these values, allowing us to see the effects clearly.

(Refer Slide Time: 26:17)



Let's run this setup and observe the results.

We will connect the blocks as follows: one connection here, another one here, and so on. Before we proceed, remember that with the input values 1, 1, 2, and 3, the total sum should be 7. Keep that in mind.

Oops, it seems there was a minor syntax error, sorry about that.

Actually, there was no syntax error; we just needed to specify the decimation rate. We'll set it to 4 for this example. Now, let's execute the flow graph. We get a result of 7. Why is it 7? Because the block is summing every 4 elements.

To verify, let's modify the input to 8 elements with the values 1, 1, -1, -1. If you add these 4 elements, you get 0. So, the output will be 7, followed by 0, then 7 again, and 0 again.

This pattern occurs because we are decimating by 4, meaning each cycle of 8 samples is reduced to a single output sample.

To further illustrate, let's compute the frequency corresponding to this setup. With 32kHz divided by 8, we get 4kHz, which translates to about 0.25 milliseconds. However, due to decimation by 4, the effective interval becomes 0.25 milliseconds divided by 4, resulting in 0.0625 milliseconds.

Let’s check the graph again. For a more detailed view, we can add more data points. For instance, using the sequence 1, 1, 2, 3, 1, 1, -1, -1, we get similar results where the sum of the first 4 values is 7 and the sum of the next 4 values is 0.

Zooming in, the distance between two peaks is 0.0625 milliseconds, confirming our calculations. So, in summary, with 8 samples and a decimation rate of 4, the resulting interval between peaks aligns with our expected value of 0.0625 milliseconds.

We see that one peak is around 16.25 and the next is at 16.31, which corresponds to an interval of 0.0625 milliseconds. This illustrates the efficiency of decimation.

Let's experiment with different inputs and parameters. Suppose we use the sequence 1, 1, followed by -1, -1, and set the decimation rate to 2. With this setup, you'll notice that the output values are 2, 0, 2, 0. If you adjust the decimation rate to 1, the block no longer performs addition, and you end up with a pattern resembling a square wave, where you see 2 samples of 1, 2 samples of -1, and so on.

This highlights how critical understanding decimation is to utilizing GNU Radio effectively. You can create blocks with various decimation rates and tailor them to your needs. The GNU Radio documentation provides valuable information on leveraging this feature to enhance your signal processing tasks.

What we’ve just covered is a fundamental introduction to extending GNU Radio's functionalities using Python. Specifically, you’ve learned how to create embedded Python blocks, where you can insert Python code snippets to perform efficient signal processing. This capability allows for significant customization and efficiency in handling your signal

processing requirements.

While creating custom blocks can be quite appealing, it's essential to strike a balance. Writing overly complex processing tasks in blocks can potentially slow down your flow graph. If you implement numerous Python computations inefficiently, you might adversely impact the performance of your simulation.

However, in many cases, using a few lines of Python code to express certain functionalities or processes can be more straightforward and efficient than assembling numerous blocks and wiring them together. Embedded Python blocks are an excellent way to extend GNU Radio's capabilities in such scenarios.

I encourage you to explore the GNU Radio documentation on creating your own blocks in both Python and C++. This knowledge will enable you to implement custom blocks efficiently when the need arises. Thank you.