

# ELEN90051 ADVANCED COMMUNICATION SYSTEMS

## Digital Communication Systems in GNU Radio

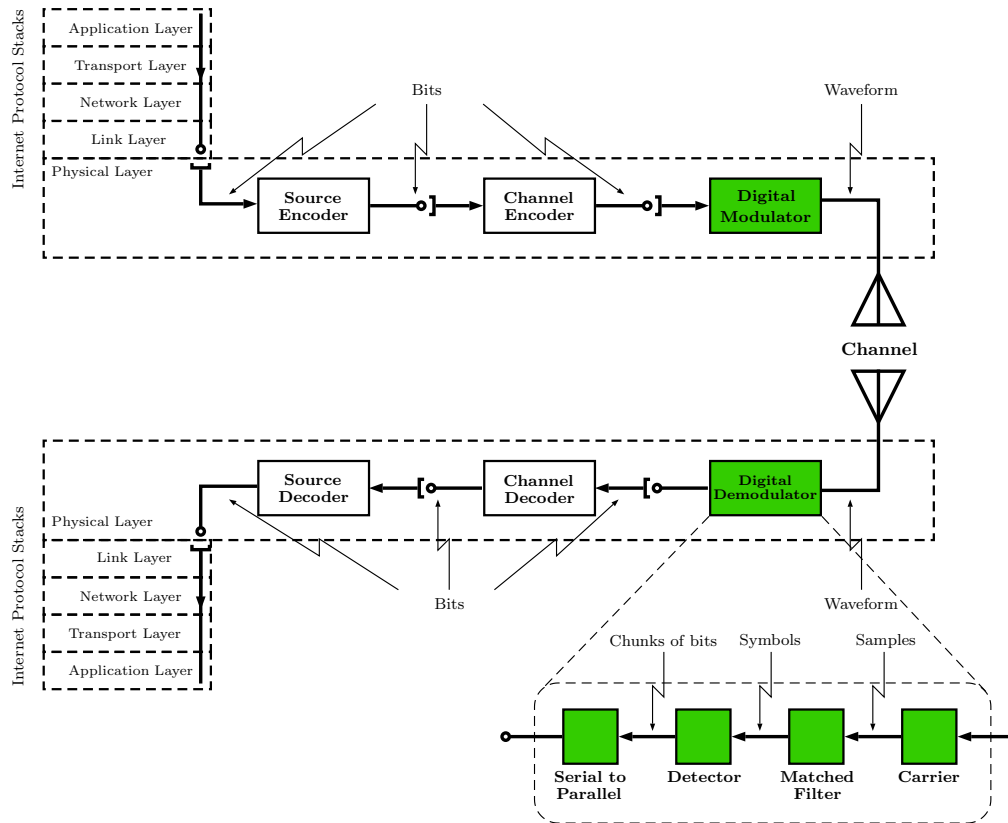
Workshop week 6 and 7 (*=week of 9 and 16 April*):

## QPSK Demodulation and Error Performance

Department of Electrical and Electronic Engineering  
The University of Melbourne

Created by Xiangyue Meng and Margreta Kuijper  
updated on April 2, 2018

In digital communication systems, demodulation is the process that recovers the information bits from the modulated carrier waves. A demodulator is usually specifically designed for a particular type of modulator so as to maximize the probability of extracting the correct information out of the received signals.



The green blocks are going to be implemented in this workshop.

Figure 1: The road map of implementing a digital communication system.

Different modulation schemes have different performances. The bit error probability is often used as a performance metric when we compare modulation schemes. The bit error probability can either be derived theoretically or obtained via measurements. It is very important to understand the performance in order to make trade-offs during the design process. For example, the header of a file is usually modulated by a low-order modulation scheme, such as BPSK, since the header contains important information about the file and we want to transmit the header reliably. On the other hand, the body of a file is usually modulated by a high-order modulation scheme, such as 64-QAM, to achieve a high transmission rate.

## I Objectives and Logistics

This is a two-session workshop, which is held **in Week 6 as well as in Week 7**. In this workshop, a practical QPSK demodulator will be implemented in GNU RADIO. Specifically, you will

- understand the principles of QPSK demodulation;
- implement a QPSK matched filter;
- implement a QPSK maximum-likelihood detector in Python;
- calculate the theoretical error probability of  $M$ -ary PSK modulation schemes;
- simulate the error performances of  $M$ -ary PSK modulation schemes in PYTHON using GNU RADIO.

You are expected to **be prepared before attending the workshop sessions**. For this, read through the workshop manual. An **individual pre-workshop report**, answering Question 1 to Question 4, worth 15 marks, is to be submitted before the start of the workshop session in week 6 (*=week of 9 April*). The second **individual pre-workshop report**, answering Question 5, worth 10 marks, is to be submitted before the start of the workshop session in week 7 (*=week of 16 April*).

There are **TWO in-workshop check-off points** upon successfully finishing Section III Task 1 and Task 2, worth 15 marks respectively, before the end of the workshop in week 7.

After the workshop in week 7, you are asked to write a group project report, following the instructions in Section IV. This report is worth 45 marks and should be submitted by **4pm on Friday of week 8 (27 April)**.

Please read the document “Rules on workshops & report submission” for more information, see LMS/Workshops/Rules on Workshops and Report Submission/  
RulesWorkshopsReportsELEN90051.pdf.

## II Background

### 1 Optimal Detection for the AWGN Channel

#### 1.1 Maximum A Posteriori Probability Detector

From the lectures, we know that the maximum a posteriori probability (MAP) detector is an optimal detector for minimizing the probability of error, which is given by

$$\hat{m} = \arg \max_{1 \leq m \leq M} [P(\mathbf{s}_m)p(\mathbf{r}|\mathbf{s}_m)], \quad (1)$$

where  $M$  is the number of symbols,  $\mathbf{s}_m$  is the sent vector signal, and  $\mathbf{r}$  is the received vector signal. We assume that the AWGN channel is modeled by

$$\mathbf{r} = \mathbf{s}_m + \mathbf{n}, \quad 1 \leq m \leq M, \quad (2)$$

where  $\mathbf{n}$  is a Gaussian distributed white noise random vector, with zero mean and variance of  $\frac{N_0}{2}$ .

ToDo: **Question 1.** Show that the MAP detector for the AWGN channel is given by

$$\hat{m} = \arg \max_{1 \leq m \leq M} [\eta_m + \mathbf{r} \cdot \mathbf{s}_m], \quad (3)$$

where  $\eta_m = \frac{N_0}{2} \ln P(\mathbf{s}_m) - \frac{1}{2} \|\mathbf{s}_m\|^2$ .

#### 1.2 Maximum Likelihood Decision Rule

If all symbols are equally likely to be transmitted, i.e.,  $P(\mathbf{s}_m) = \frac{1}{M}$  for all  $1 \leq m \leq M$ , the MAP criterion simplifies to the maximum-likelihood (ML) criterion. Figure 2 shows a constellation diagram of QPSK.

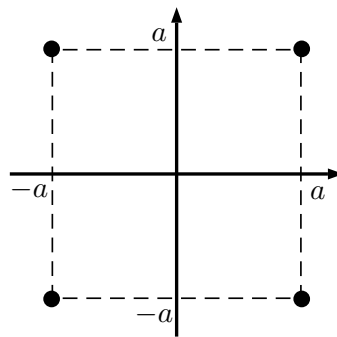


Figure 2: QPSK constellation diagram.

ToDo: **Question 2.** Give mathematical expressions of the decision regions for ML detection in terms of the constant  $a$  in Figure 2.

ToDo: **Question 3.** Write a piece of pseudo codes using `if...elif...elif...else...` statements that implement your decision rules of **Question 2**, given that the inputs are complex numbers.

## 2 Implementation of the Optimal Receiver for the AWGN Channel

### 2.1 Correlation-Type Receiver

Recall from the lectures that both the correlation-type and the matched filter-type demodulator implement the receiver for the AWGN channel. They provide the same input to the detector, namely the input that maximizes the SNR.. Mathematically, they have the same expression when sampled at  $t = T$ , i.e.,

$$r_k = \int_0^T r(\tau) \phi_k(\tau) d\tau, \quad (4)$$

where  $r(t)$  is the received signal and  $\phi_k(t)$  is the  $k$ -th orthonormal basis signal. A natural question is: what are the differences between correlation-type and matched filter-type demodulators?

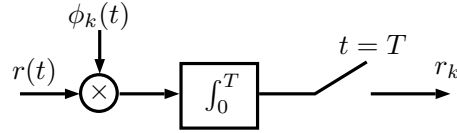


Figure 3: Core structure of the correlation-type demodulator.

Figure 3 shows one of the branches of a correlation-type demodulator. The demodulator performs the following sequence of actions:

- (a) Generates the  $k$ -th orthonormal basis signals  $\phi_k(t)$ ;
- (b) Multiplies the input signal  $r(t)$  with  $\phi_k(t)$ ;
- (c) Integrates this product for a symbol period  $T$ ;
- (d) Samples at  $t = T$ ;
- (e) Dumps the integrator and starts accumulating again.

In practice, the multipliers and integrators can be easily implemented on a circuit. However, a correlation-type receiver does not lend itself easily to a digital design.

### 2.2 Matched Filter-Type Receiver

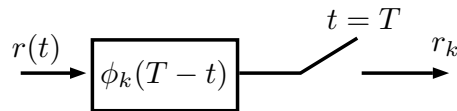


Figure 4: Core structure of the matched filter-type demodulator.

Figure 4 shows one of the branches of a matched filter-type demodulator. The demodulator performs the following sequence of actions:

- (a) Convolve the input signal  $r(t)$  with the time-reversed orthonormal basis signal  $\phi_k(T - t)$ ;
- (b) Samples at  $t = T$ .

Clearly, the matched filter-type receiver has fewer processing steps. More importantly, it lends itself easily to a digital design via a DSP chip. This is the reason that we prefer a matched filter-type receiver in digital communication systems.

**ToDo: Question 4.** Give a sketch of the proof of the following statement in your own words:

*If a signal  $s(t)$  is corrupted by AWGN, the filter with impulse response matched to  $s(t)$  maximizes the output SNR.*

### 3 Error Performance

#### 3.1 Symbol-Error Rate vs. Bit-Error Rate

In digital communication systems, a **symbol** refers to a point on the complex plane that represents a group of bits. In order to compare error performances among different modulation schemes, the symbol-error rate (SER), denoted by  $P_e$ , is usually normalized to the bit-error rate (BER), denoted by  $P_b$ . In general, determining the bit error probability requires detailed knowledge of how different bit sequences are mapped to the signal points. For an arbitrary modulation scheme, SER and BER satisfy

$$P_b \leq P_e \leq P_b \log_2 M \quad (5)$$

or, equivalently,

$$\frac{P_e}{\log_2 M} \leq P_b \leq P_e \quad (6)$$

where  $M$  is the order of the constellation.

#### 3.2 Error Probability of $M$ -ary PSK

From the lecture notes, for BPSK ( $M = 2$ ), the average symbol error probability is the same as for binary PAM, i.e.,

$$P_e = Q\left(\sqrt{\frac{2\mathcal{E}_b}{N_0}}\right), \quad (7)$$

where  $Q(\cdot)$  is the tail distribution function of the standard normal distribution.

For QPSK ( $M = 4$ ), the average bit error probability is

$$P_e = 1 - (1 - P_{\text{BPSK}})^2 = 2Q\left(\sqrt{\frac{2\mathcal{E}_b}{N_0}}\right) - \left(Q\left(\sqrt{\frac{2\mathcal{E}_b}{N_0}}\right)\right)^2. \quad (8)$$

For  $M$ -ary PSK ( $M \geq 8$ ), the symbol error probability calculations involve numerical integration. Read the section about error probability of  $M$ -ary PSK on page 319~321 of the reference book *M. Rice Digital Communications: A Discrete-Time Approach, 1st. Edition* or on page 194~195 of the reference book *J. Proakis Digital Communications, 5th Edition*. Assume that carrier phase is perfectly synchronized and symbol timing is also synchronized.

ToDo: **Question 5.** It can be shown that, for large SNR,

$$\int_{-\pi/M}^{\pi/M} e^{-\gamma_s \sin^2 \theta} \left( \int_0^\infty v e^{-\frac{(v - \sqrt{2\gamma_s} \cos \theta)^2}{2}} dv \right) d\theta \approx 1 - 2Q \left( \sqrt{2\gamma_s \sin^2 \frac{\pi}{M}} \right), \quad (9)$$

where the symbol SNR  $\gamma_s = \frac{\mathcal{E}}{N_0}$ . Use the above to show that the  $M$ -PSK symbol error probability  $P_e$  is approximated as

$$P_e \approx 2Q \left( \sqrt{2 \log_2 M \sin^2 \left( \frac{\pi}{M} \right) \frac{\mathcal{E}_b}{N_0}} \right). \quad (10)$$

Hint: First describe the decision region using polar coordinates.

### III In-Workshop Tasks (30 Marks)

In this workshop, you are asked to complete the following tasks:

- (1) Implement a QPSK Demodulator in GRC;
- (2) Investigate the performances of the QPSK demodulator under different situations;
- (3) Investigate the error performances of different modulation schemes in PYTHON using GNU RADIO.

After this workshop, you will then have acquired the ability to implement simple demodulators in GNU RADIO.

#### Task 1: Implement a QPSK Demodulator in GNU Radio (15 Marks)

Assuming that you have successfully implemented a QPSK modulator in the previous workshop, in this section, we will implement a QPSK demodulator in GNU RADIO.

##### Step 1: Preparation

Before implementing the demodulator, let us add some auxiliary variables and GUI components to the flowgraph.

ToDo: Open `.grc` file of the QPSK modulator that you implemented in the previous workshop. Create and setup the variables and GUI components as shown in Figure 5. Can you think of where and why we need those variables and GUI components judging from their IDs?



The demodulator is built upon the previous `.grc` file. You may want to save the previous `.grc` as another file for this workshop.

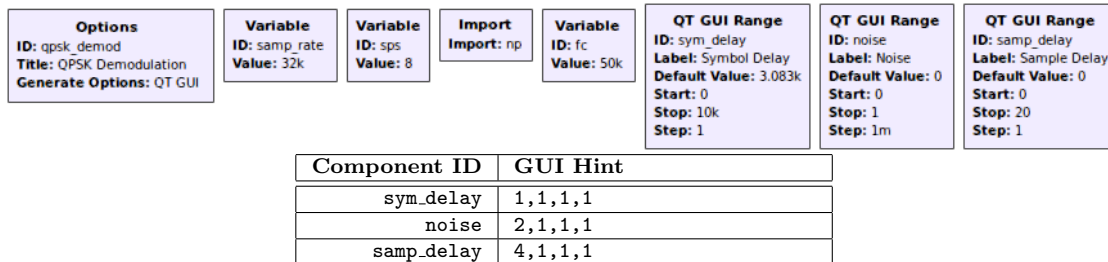


Figure 5: Setting variables and GUI components in GRC.

Next, let us organize the flowgraph by using Virtual Source and Virtual Sink blocks.

ToDo: Drag and drop a pair of Virtual Source and Virtual Sink blocks into the flowgraph. Set the Stream ID to 'mod\_symbol' (without the quotes), and connect the blocks as shown in Figure 6.

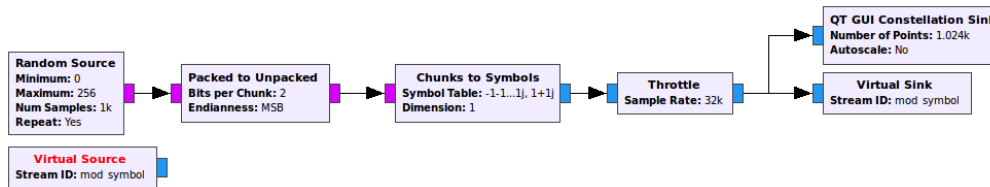


Figure 6: Connections of the blocks for symbols.

As we did in the previous workshop, a pulse shaping filter as well as carriers are added to the flowgraph. Let us build both the in-phase and quadrature branches for the modulator.

ToDo: Drag and drop two pairs of Virtual Source and Virtual Sink blocks into the flowgraph. Set the Stream ID to 'chn\_in\_inphase' and 'chn\_in\_quadrature' (without the quotes) respectively, and connect the blocks as shown in Figure 7. You could also get some hints about the parameters in the figure.

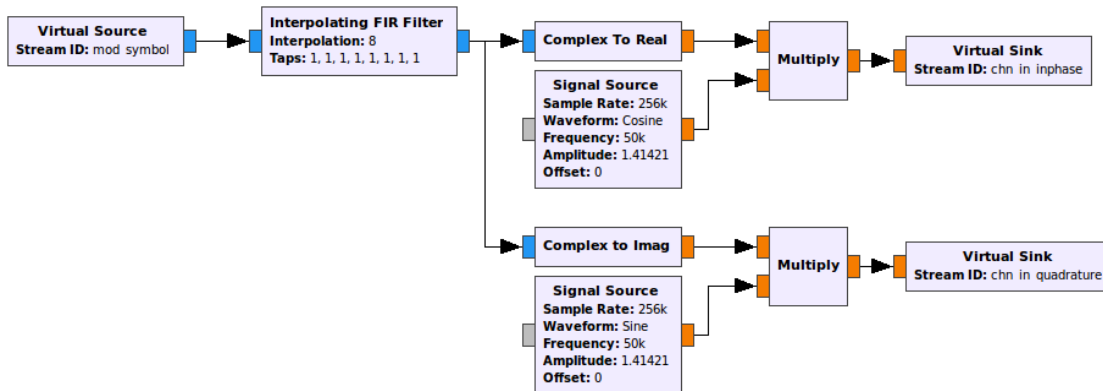
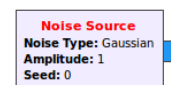


Figure 7: Connections of the pulse shaping and carrier blocks.

## Step 2: Build the Channel Model

For prototyping purposes, it is usually a good idea to first simulate the transmission channel as an infinite-bandwidth AWGN channel in the flowgraph. In doing so, we only need to add a Noise Source block.

ToDo: Drag and drop a Noise Source block into the flowgraph. Double click on the block and set the parameters as hinted in Figure 8.



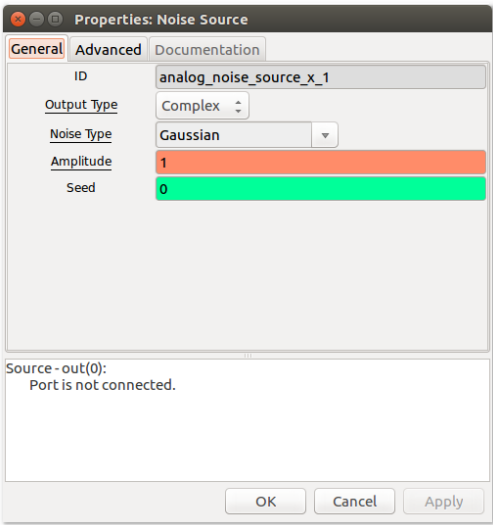
A pair of Virtual Source and Virtual Sink creates a virtual data tunnel that connects output to input. It is useful when the flowgraph becomes complicated and hard to visualize the connections.




In the early versions of Python,  $1/8$  equals to 0 instead of .125. In this case, use  $1./8$ . Notice the period after 1.



All blocks display integer values abbreviated with k or M for thousand or million in the flowgraph. But in the Properties dialogues, you have to enter the full decimal digits of the integer.



 This is where we use the auxiliary variable `noise`.

Name	Value
Output Type	<i>A proper selection</i>
Amplitude	<code>noise</code>

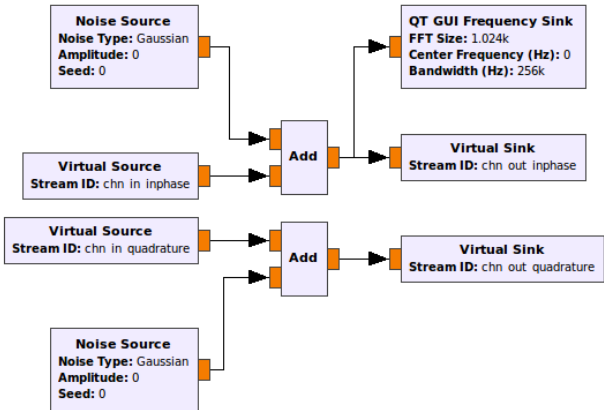
Figure 8: Setting parameters for Band Pass Filter block in GRC.

ToDo: Add the output of Virtual Source and Noise Source block together by using the Add block. Then build the other branch for the quadrature component.



It is always a good practice to stop and check whether the flowgraph is working as expected.

ToDo: Add a QT GUI Frequency Sink block with GUI Hint set to 1,0,4,1. The following flowgraph is provided for your reference. You might want to use the Null Sink to properly terminate a flowgraph.



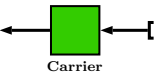
The Null Sink block is used to terminate a flowgraph without which the flowgraph will not run.

Figure 9: Connections of the channel model.

Execute the flowgraph and observe the spectrum. Change the value of `Noise` to see the effectiveness of the channel model.

Step 3: Demodulate Carriers

Since you have went through the usage of many blocks, the following guide will be less detailed. To demodulate the carriers, simply multiply the incoming signals with the carrier again.



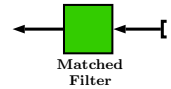


ToDo: Multiply the output of the channel model with the carrier again to get the demodulated signals. Then assemble the in-phase and quadrature channels into a complex number. You might want to use multiple **Signal Source**, **Multiply**, and **Float to Complex** blocks. Replace the **Null Sink** when necessary.

#### Step 4: Implement Matched Filter

Recall from the lectures that if a signal is corrupted by AWGN, the matched filter maximizes the output SNR. Let us observe the effectiveness of the matched filter.

ToDo: Continue to build the flowgraph as hinted in Figure 10. Set the GUI Hint of the QT GUI Time Sink and the QT GUI Constellation Sink block to 5,0,1,2 and 0,1,1,1 respectively. Connect the input of the upper Delay block to the output of the mod\_symbol Virtual Source block.



The first Decimating FIR Filter is the matched filter.

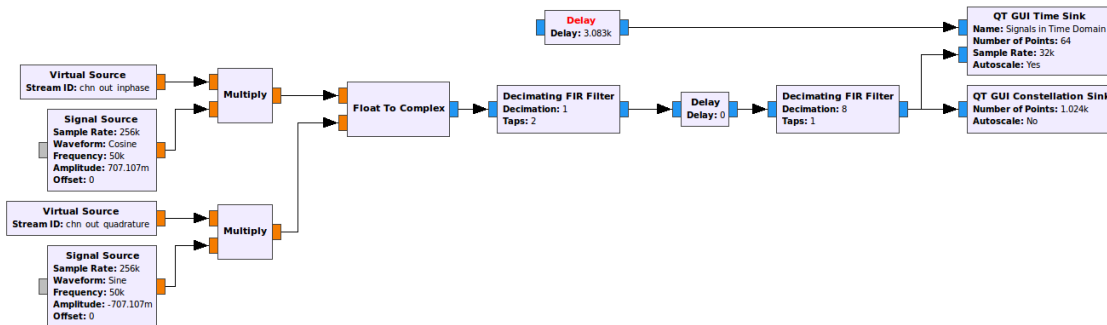


Figure 10: Connections of the matched filter.

From the lecture notes, the expression of the matched filter is  $h(t) = ks(T-t)$ , where  $s(t)$  is the transmitted signal and  $k$  is a normalization constant. Assuming that our channel model is AWGN, what is the value of  $k$  for this flowgraph such that on average the output signals are normalized to 1? Second, how many taps should the matched filter have and what are the values of the taps?

ToDo: Set the **Taps** of the matched filter to `'1./sps*np.ones(sps)'` (without the quotes). Confirm that this FIR filter matches to the rectangular pulse filter at the beginning of the flowgraph.

ToDo: There are two **Delay** blocks in the flowgraph. Set the field **Delay** with either `samp_delay` or `sym_delay`.

ToDo: Run the flowgraph and you should see an output window similar to Figure 11.



`'np.ones(sps)'` is actually a Python script that generates a list `[1,1,1,1,1,1,1,1]`.



You might notice that the legends of the time-domain plots in your flowgraph are different from Figure 11. You can change the name of each trace in the **Properties** dialogues of the GUI component.

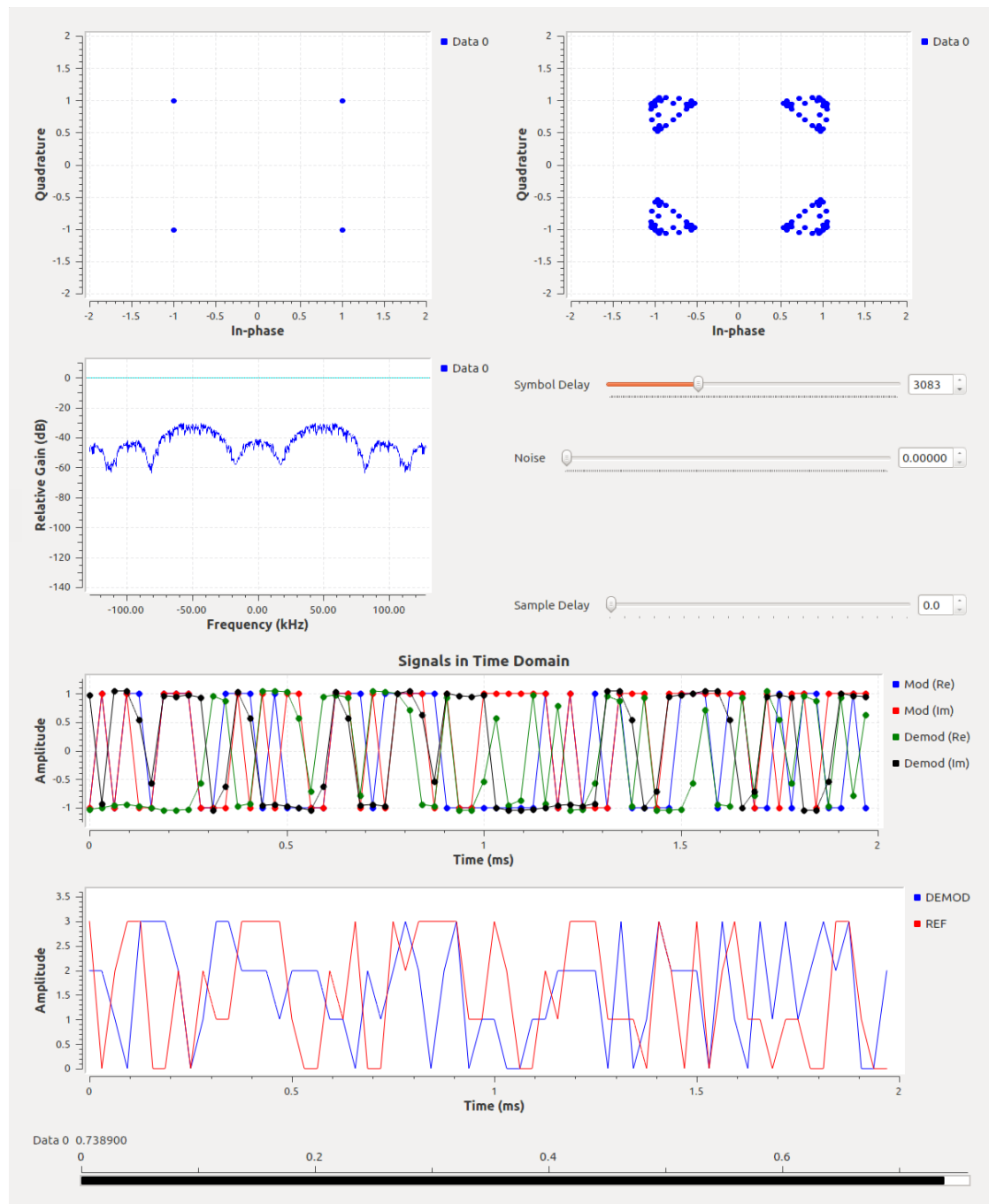


Figure 11: Flowgraph output.

As shown in Figure 12, the variable **Sample Delay** controls the delays between the matched filter and the downsampler, where the function of the downsampler is to downsample the incoming signals for every Decimation signals.

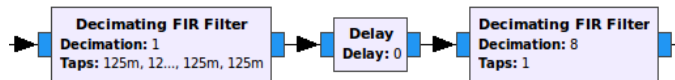


Figure 12: Matched filter and the downsampler.

- ToDo: Change the value of **Sample Delay** and observe the result. Which value of the **Sample Delay** is the best and why?
- ToDo: In the time plot, turn off the traces of the **imaginary part** of the modulated and demodulated signals. Assuming that you have chosen the correct value for **Sample Delay**, try to find the value of **Symbol Delay** such that the traces of the real part of the modulated and demodulated signals almost perfectly overlap with each other, as shown in Figure 13.
- ToDo: Change the value of **noise**, starting from 0 (the noiseless case). Observe the spectrum and the modular T plot.

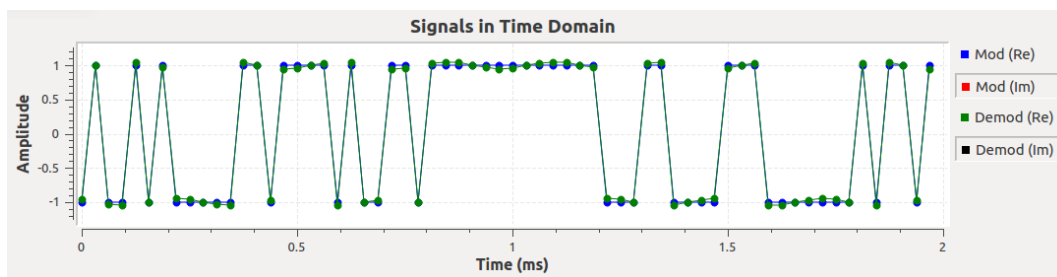


Figure 13: Modulated and demodulated signals.

This is one of the many places where designing a receiver is much more challenging than designing a transmitter. Without the knowledge of the structure of a transmitter, how does the receiver know where exactly the first sample starts?

**Step 5:** Let us now observe the outputs without the matched filter.

- ToDo: Set the **Taps** of the matched filter to `'1./np.sqrt(sps)'` (without the quotes) and run the flowgraph. Observe the constellation patterns with different values of **Sample Delay**. Can you still find a value for **Sample Delay** such that the constellation is comparably good to the one with the matched filter? Figure 14 shows the time plot of modulated and demodulated signals for a particular value of **Sample Delay**. We can still observe a certain level of correlation between the signals, but not as good as when we use the matched filter.
- ToDo: Change the value of **noise**. Observe the spectrum and modular T plot.

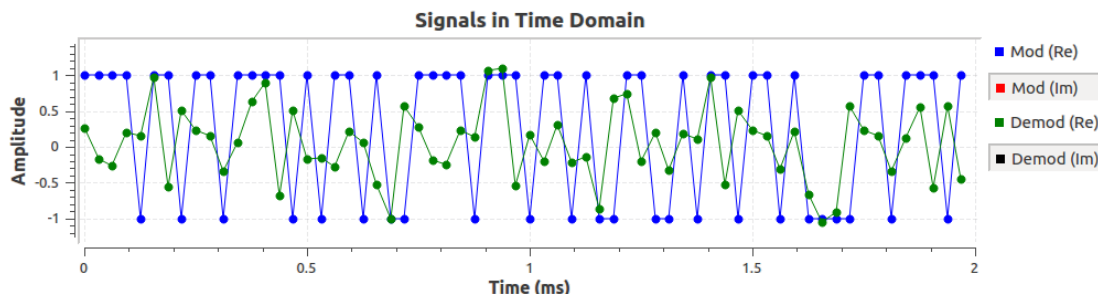


Figure 14: Non-overlapping of the modulated and demodulated signals.

### Step 6: Implement Detector

In this step, we will use the maximum-likelihood (ML) criterion to implement the detector. You will need the pseudo codes of the decision regions from the solution of **Question 3** to complete this step.



- ToDo: (1) Download the skeleton code archive file `gr_elen90051.tar.gz` from LMS and extract the folder to the Ubuntu desktop.
- (2) Open `qpsk_detector_cb.py` in the folder `gr_elen90051/python`. You will see a few lines of skeleton codes as shown below,

```
def get_minimum_distances(self, sample):
    # TODO: Implement your QPSK detector between the dashed lines.
    #       Do not modify other codes. (~8 lines)

    # -----
    return 0
    # -----
```

You are supposed to implement the QPSK detector between the dashed lines. The expected number of lines of codes is also given.

- (3) The input parameter of the function `get_minimum_distances()` is the current complex sampling point, stored in the variable `sample`. You can access the real and imaginary part of `sample` by using `sample.real` and `sample.imag`. The output of the function gives the indexes of the complex points. Your objective is to map the complex points to the indexes.
- (4) **STOP**. Check your codes with the demonstrator.
- (5) Upon completing the previous step, open **Terminal of Ubuntu** by clicking on the icon on the launch bar. Assuming that you put the folder `gr_elen90051` on the Ubuntu desktop, type in the follow commands in the **Terminal**,

```
cd Desktop
cd gr-el90051
mkdir build
cd build
cmake ../
make
```

If your codes get compiled successfully, you will see something similar to



In Python, code indentation is very important. Make sure your codes are indented according to the syntax.



Every time you change the codes, you should rebuild and re-install the program. Then reload the block library in GRC.

```
[ 33] Generating __init__.pyc, qpsk_detector_cb.pyc
[ 33] Generating __init__.pyo, qpsk_detector_cb.pyo
[ 66] Build target pygen_python_959d6
Scanning dependencies of target pygen_apps9a6dd
[ 66] Built target pygen_apps_9a6dd
Scanning dependencies of target doxygen_target
[100] Generating documentation with doxygen
[100] Built target doxygen_target
```

without any warning or error. If not, check the error message and debug your codes.

(6) Install the module by typing in

```
sudo make install
```

**ToDo:** Refresh the block library in GRC. Drag and drop your newly implemented QPSK detector `qpsk_detector_cb` to the flowgraph, as shown in Figure 15.



Click the **Reload Blocks** button on the top-right of the toolbar to reload all blocks.

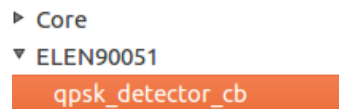
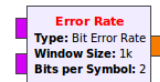
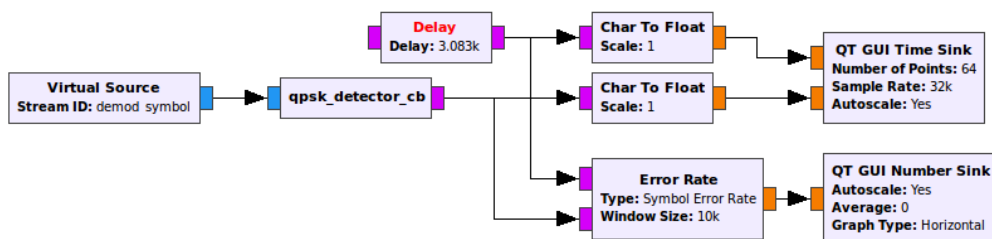


Figure 15: A user-defined QPSK detector in GRC.

**Step 7:** The outputs of the `qpsk_detector_cb` block are QPSK symbols. And we want to see the symbol-error-rate (SER) (not the bit-error-rate (BER)) under different situations.

**ToDo:** Add the necessary blocks and connect the flowgraph as hinted in Figure 16. You might need the **Char To Float**, **Error Rate**, and **QT GUI Number Sink** block. Connect the input of the **Delay** block to the output of the **Packed to Unpacked** block in the modulator.



The **Error Rate** block compares its inputs and gives the error rate within **Window Size**. Larger values of **Window Size** give higher accuracy of the error rate but introduce more latency.

Figure 16: Reference flowgraph of QPSK demodulation.

**ToDo:** Re-enable the matched filter of **Step 4** and run the flowgraph. Find the value of **Symbol Delay** and **Sample Delay** such that the modulated and demodulated samples are perfectly aligned. Change the value of **Noise** to see the change in value the SER.

### Step 8: Verification

Show your working flowgraph to the demonstrator to check off the **in-workshop assessment**. You might be asked some questions relating to your flowgraph.

## Task 2: Investigate the error performance of different modulation schemes (15 Marks)

In the past few weeks, we saw that the GNU RADIO COMPANION (GRC) is an effective tool that can build a system flowgraph in GNU RADIO. A flowgraph is actually a Python script that connects the compiled C++ processing blocks. The GRC is just a graphical user interface (GUI) for constructing a Python script. However, the functionality of the GRC is limited by the graphic model. In this workshop, you will learn how to construct a GNU RADIO application directly from Python script, without using the GRC.

### Step 1: Preparation

Download `ber_simulation_skeleton.py` from LMS and open it in the text editor, such as `gedit`. You will see a skeleton of a python script.

ToDo: Read the script and understand its structure.

You will see a few snippets like

```
### START CODE HERE ### (~# lines)

### END
```

It tells you where you should put your own codes in the script and how many lines of codes are expected.

### Step 2: Expressions of the Theoretical $P_e$ (Exact/Approximate) of $M$ -ary PSK.

```
### Step 2 ###
def berawgn(M, EbN0):
    """ Calculates the theoretical bit error rate in AWGN
        (for MPSK and given Eb/N0) """
    if M == 2:
        ### START CODE HERE ### (~1 line)
        return None
        ### END
    elif M == 4:
        ### START CODE HERE ### (~1 line)
        return None
        ### END
    else:
        ### START CODE HERE ### (~1 line)
        return None
        ### END
```

In the script, the function `berawgn(M, EbN0)` returns the BER of  $M$ -ary PSK when the SNR per bit  $\gamma_b = EbN0$ .

ToDo: Referring to the pre-workshop question, implement the theoretical calculations for the case when  $M = 2$ ,  $M = 4$ , and  $M \geq 8$ . You will need the following python functions.



`gedit` is the built-in GNOME text editor in Ubuntu.



Do NOT modify other parts of the script unless you know what you are doing.

- `np.log2()`
- `np.sqrt()`
- `np.sin()`
- `erfc()`



Click on the function to see the manual.

Note that the relationship between the  $Q$  function and the `erfc` function is given by

$$Q(x) = \frac{1}{2} \operatorname{erfc} \left( \frac{x}{\sqrt{2}} \right). \quad (11)$$

### Step 3: Construct the Flowgraph.

Instead of constructing the flowgraph in GRC, we will use Python scripts to implement the flowgraph shown in Figure 17.

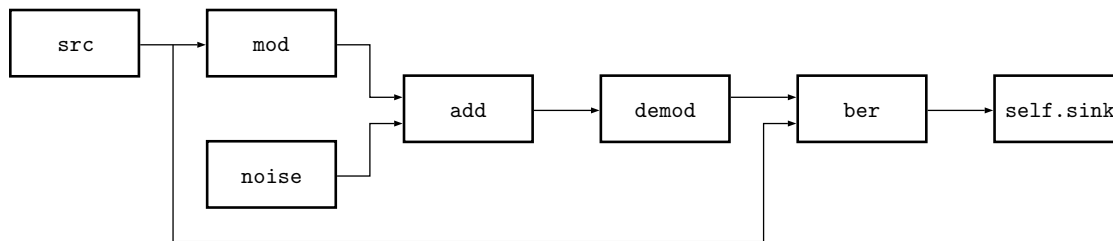


Figure 17: The flowgraph to be constructed.

The blocks have been implemented for you, stored in the variables `src`, `mod`, `add`, `noise`, `demond`, `ber` and `self.sink`. The general syntax for connecting the blocks is

```
self.connect(block1, block2, block3, ...),
```

which would connect the output of `block1` with the input of `block2`, the output of `block2` with the input of `block3` and so on. If you want to specify a specific port that a block connects to, use the tuple `(destination, 1)` instead of the block name, meaning that the block connects to the port 1 of `destination`. By default, if you do not specify the port number, `self.connect` will connect to the port 0.

**ToDo:** Inside the class `BERAWGNSimu()`, implement the flowgraph shown in Figure 17 in the following snippet.

```

### Step 3 ###
### START CODE HERE ### (~3 lines)

### END

```

### Step 4: Evaluate the Theoretical Expressions

**ToDo:** In the `main` function, write codes for evaluating the theoretical expressions defined in the function `berawgn()` for BPSK, QPSK, 8PSK, 16PSK, and 32PSK in the following snippet. Store the result in the list `ber_theory`.

```
### Step 4 ###
### START CODE HERE ### (~5 lines)

### END
```

Hint:

- `ber_theory` is a Python list that stores the plotting data of different modulation schemes. Use `ber_theory.append()` to append a new item to `ber_theory`.
- Use the function `berawgn(M, EbN0)`, implemented in **Step 2**, to evaluate the theoretical expressions, where the list `EbN0_range` has been provided.
- You can use the compound expression  
`[f(x) for x in list]`  
 instead of a `for` loop to calculate `f(x)` for every `x` in the list `list`.

### Step 5: Generate the Simulated Values

**ToDo:** Similar to the previous step, in the `main()` function, write codes for simulating BER of BPSK, QPSK, 8PSK, 16PSK, and 32PSK in the following snippet. Use the `simulate_ber()` function that has been implemented. Store the result in the list `ber_simu`.

```
### Step 5 ###
### START CODE HERE ### (~5 lines)

### END
```

### Step 6: Execute the Script

**ToDo:** Open Terminal of Ubuntu by clicking on the icon on the launch bar. Assuming that you put the `ber_simulation_skeleton.py` file on the desktop, type in the follow commands in the Terminal,

```
cd Desktop
python ber_simulation_skeleton.py
```

### Step 7: Verification

Show your working PYTHON simulation to the demonstrator to check off the **in-workshop assessment**. You might be asked some questions relating to your script.

## IV Report Tasks (45 marks)

### Task 1: Investigate the Performance of the QPSK Demodulator

**ToDo:** Investigate the performance of the QPSK demodulator in different situations: Use an **infinite-bandwidth** channel model. Change the value of `noise`, starting from 0 (noiseless case), and compare the SERs with and without the matched filter of at least 6 signal-to-noise ratios (SNR) and plot on the same graph.



**Task 2: Discussion and Reflection**

You should structure your report as follows:

1. Introduction: Introduce your report, e.g., what is the function of demodulation; what are the motivations of doing analysis of bit error performance.
2. Background: Give necessary knowledge and mathematical background related to your topic, e.g., what are the mathematical and other tools of doing the analysis.
3. Theoretical analysis: Similar to the pre-workshop questions, derive theoretical expressions of the MAP/ML decision rule and  $M$ -ary PSK bit error performance.
4. Implementation: Describe your implementation.
5. Simulation: Describe your simulation. Illustrate and compare the results.
6. Conclusion: Draw conclusions based on your analysis and observations. In particular, compare your theoretical analysis with the simulation results that you found and discuss the effect of increased noise levels. What are the difficulties that a receiver might encounter in PSK systems.
7. Reference: Use IEEE reference style. See `LMS/Workshops/Workshop week 3: source coding 2018/ieeecitationref.pdf` for details.

**End of Workshop**