TELE 4651 Lab experiment guide

Yixuan Xie and Lei Yang

Abstract

This guide provides sufficient materials for students to complete their pre-labs for the course TELE 4651. It is important to note that this guide ONLY provide extra hints from lab demonstrators outside the lab manual. Please refer to the student lab manual for detailed explanations of tasks.

I. Introduction

In this course, you will be designing and implementing a software defined radio (SDR) using National Instruments (NI) software and hardware. Through the implementation of this SDR you will investigate practical design issues and absorb theoretical knowledge in wireless digital communication.

For each lab in you manual, you are required to read the lab manual thoroughly and complete the Pre-Lab session in your own time before come to the lab session. This includes both answering questions and designing sub-VI files. Your Pre-lab work will be marked in front of lab demonstrator at the start of each lab. In the next section, we demonstrate how VI files will be marked accordingly.

We also provide you with hints of how to design VI files for each Pre-lab. Note that these hints will be provided weekly, and the hints for Pre-lab 1 is given in Section III. Through out this document, word/phrase in **bold** refers to the name of component, word/phrase in *italic* refers to the name of VI file.

II. How to Verify Your VIs

In each lab except the lab 0, you are required to implement and verify your own VI files before come to the lab. We now show you the flow of how to verify a VIs. Just bear in mind, this will also be the flow for lab demonstrators to mark your VI files.

There are 4 steps for verifying your VIs in the LabVIEW.

Step1: Open the simulator for your VIs.

Step2: Open the simulator and change to the block diagram window.

Step3: Find the original VIs in the block diagram and replace them with your implementations.

Step4: Run the simulator to verify your VIs.

Now, we will use the VI file *source.vi*, which is required to implement in the lab 1 - part 1, to demonstrate the flow.

Step1: Find the simulator for your VIs.

For lab 1 - part 1, we can find the simulator *awgn_simple_sim.vi* in folder lab 1.1 as show in fig. 1. For lab 2 to lab 8, we can find the simulator VI, namely *simulator.vi*, in the folder as show in fig. 2.

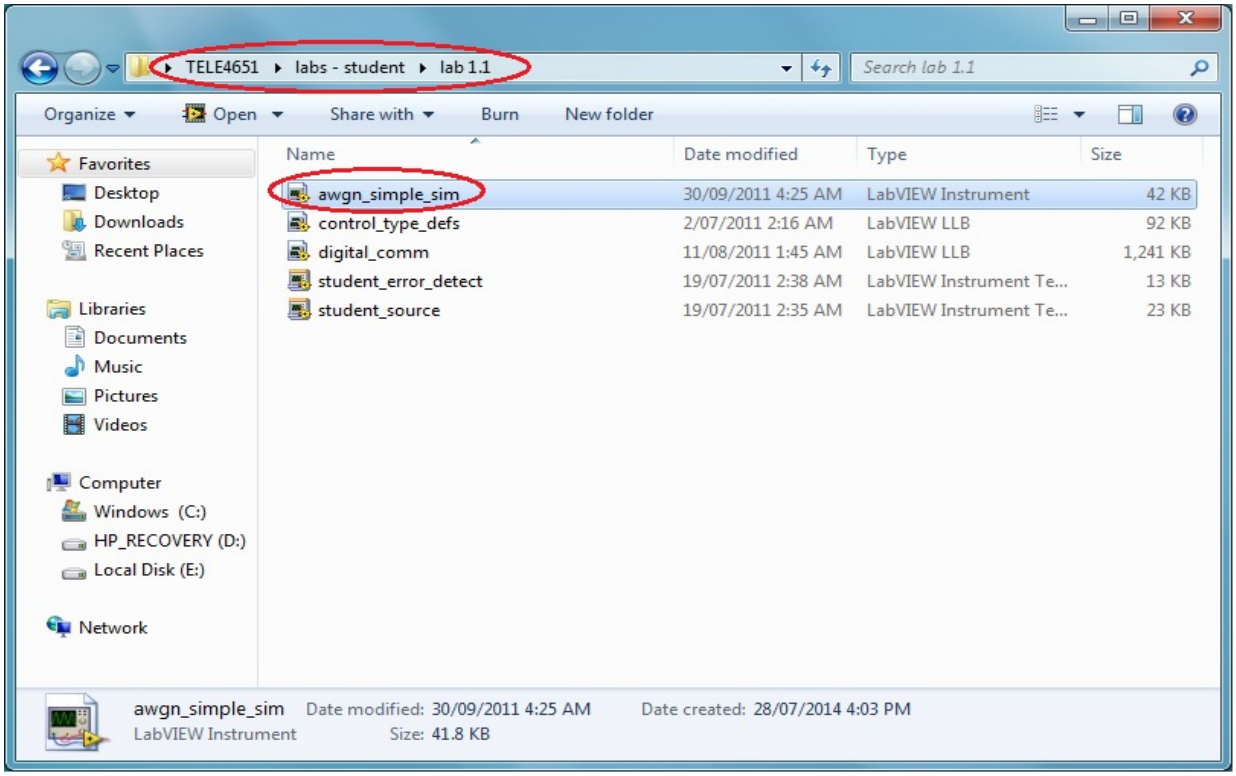

Fig. 1. Simulator for lab 1 - part 1.

Step2: Open the simulator and change to the block diagram window.

Double click on the simulator file to open the simulator, which is show in fig. 3.

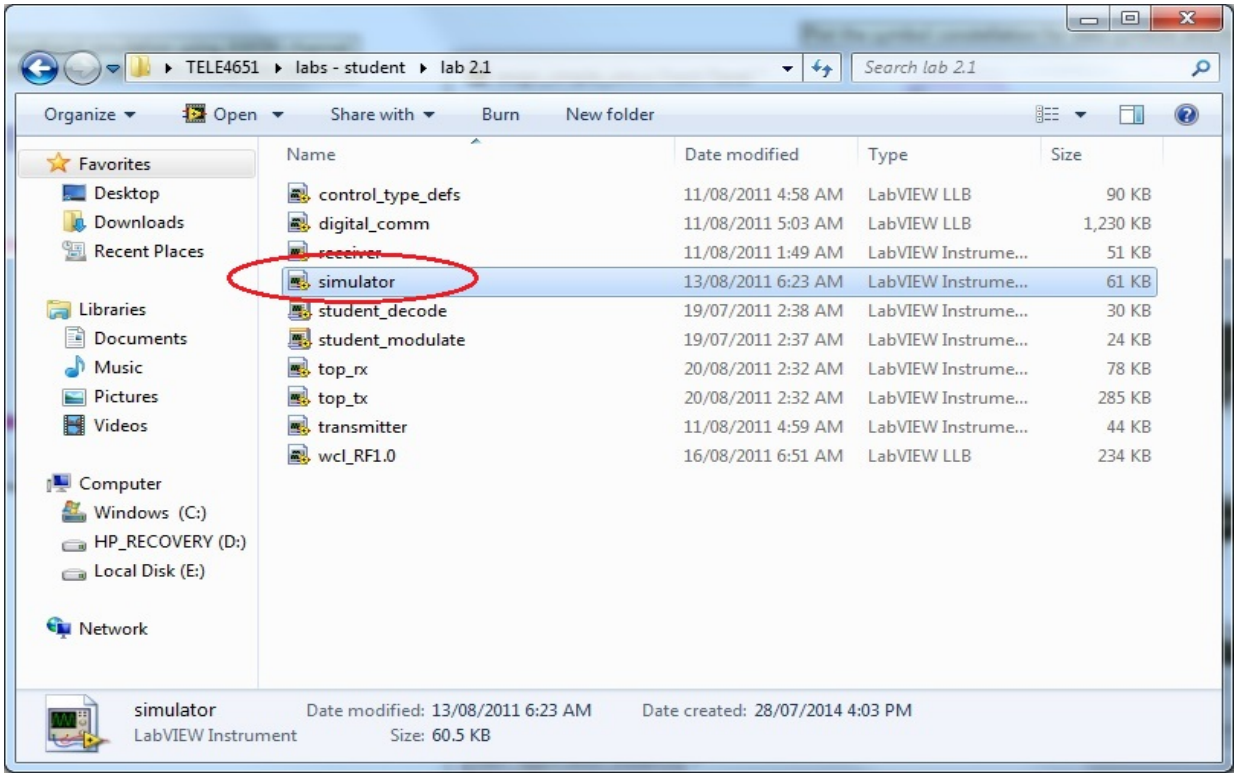

Fig. 2. Simulator for lab 2 to lab 8.

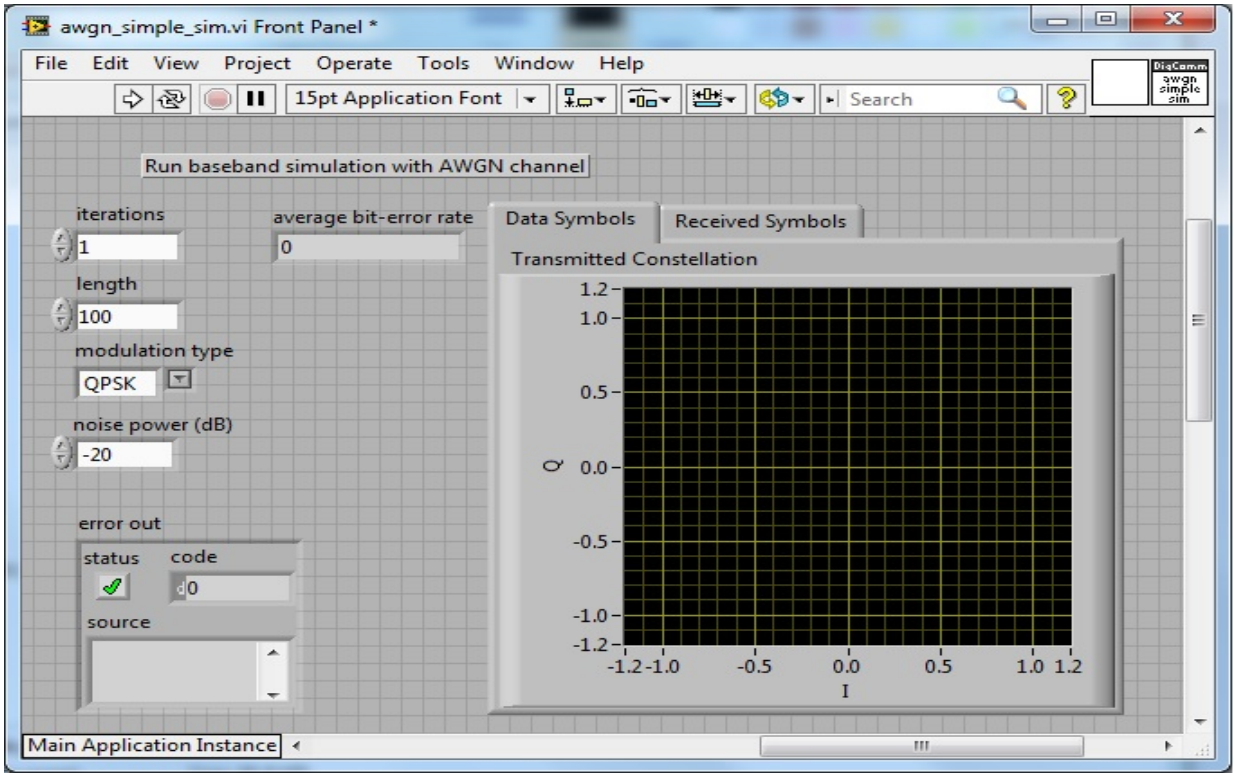

Fig. 3. Simulator front panel for lab 1 - part 1.

By clicking “Window” and choose ”Show Block Diagram” (or CTRL +E), the associated block diagram for this VI is open. See figures 4 and 5.

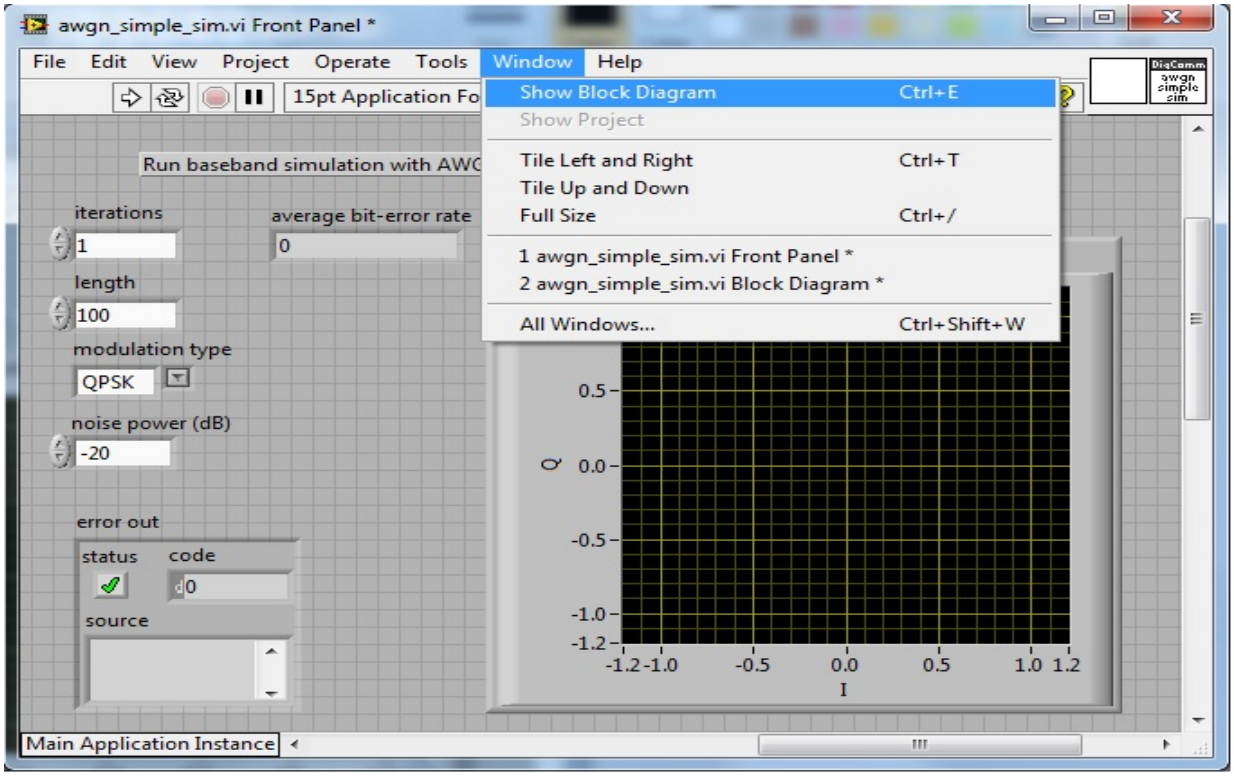

Fig. 4. Command for showing block diagram.

Step3: Find the original VIs in the block diagram and replace them with your implementations.

Find the *source.vi* in the block diagram window, right click on the module and select “Replace – All Palettes – Select a VI...” command from the pop-up menu. Select the VI file from the pop-up window.

After you have replaced the VIs with your implementation, please confirm you have replaced the VIs correctly. You can find out if the VIs are replaced by your implementation from the name of the VIs. In our example, after we replaced the *source.vi* with *student_source.vi*, the name of the module in the block diagram is also replaced with the name of our VIs. See figures 6, 7 and 8.

Step4: Run the simulator to verify your VIs.

After replacing the VIs with your implementation, input the parameters required for the simulation according to the lab manual and click the “Run” (white arrow on top left corner of either block diagram or front panel) button to run the simulation. A trial simulation is shown in fig. 9.

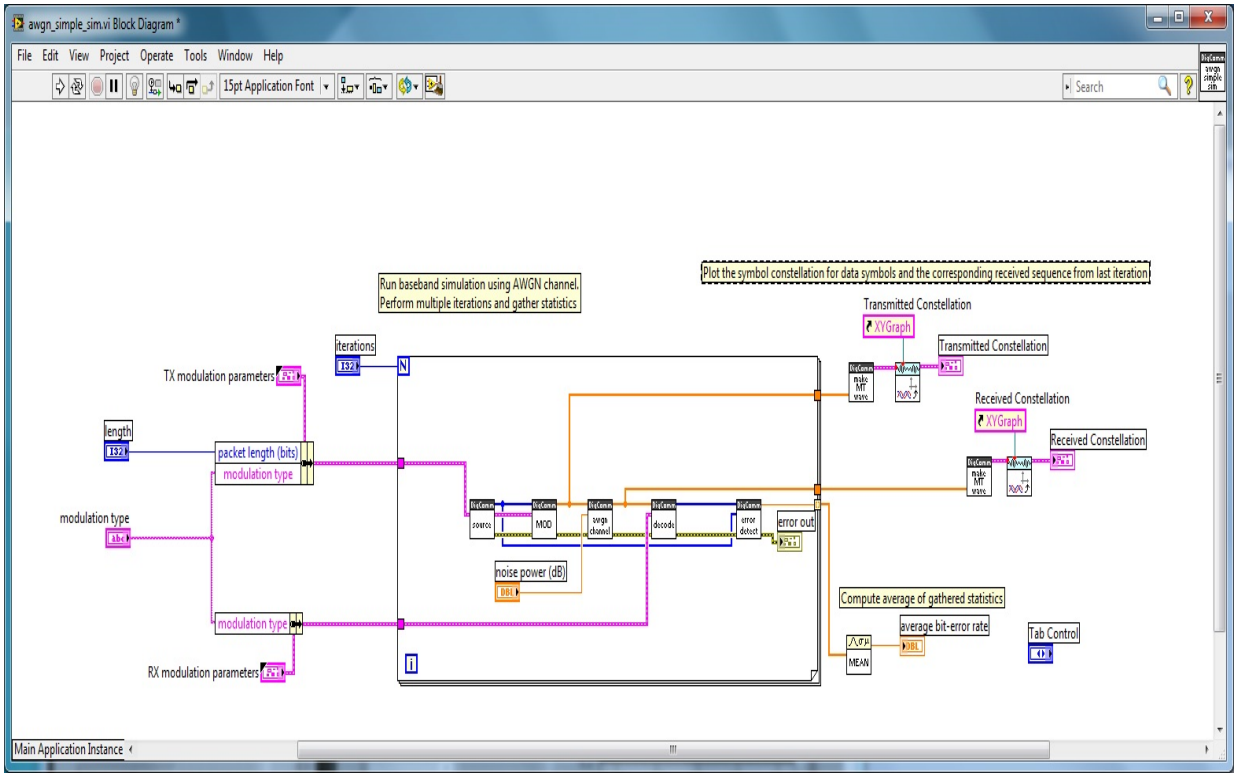

Fig. 5. Simulator block diagram for lab 1 - part 1.

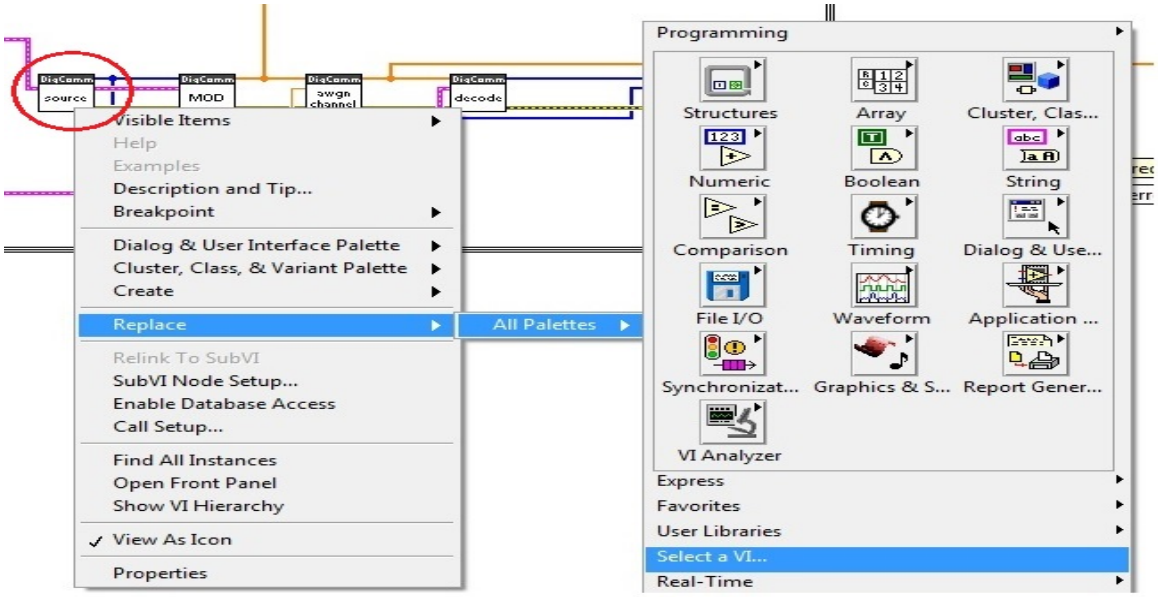

Fig. 6. Command for replacing VIs with your implementation.

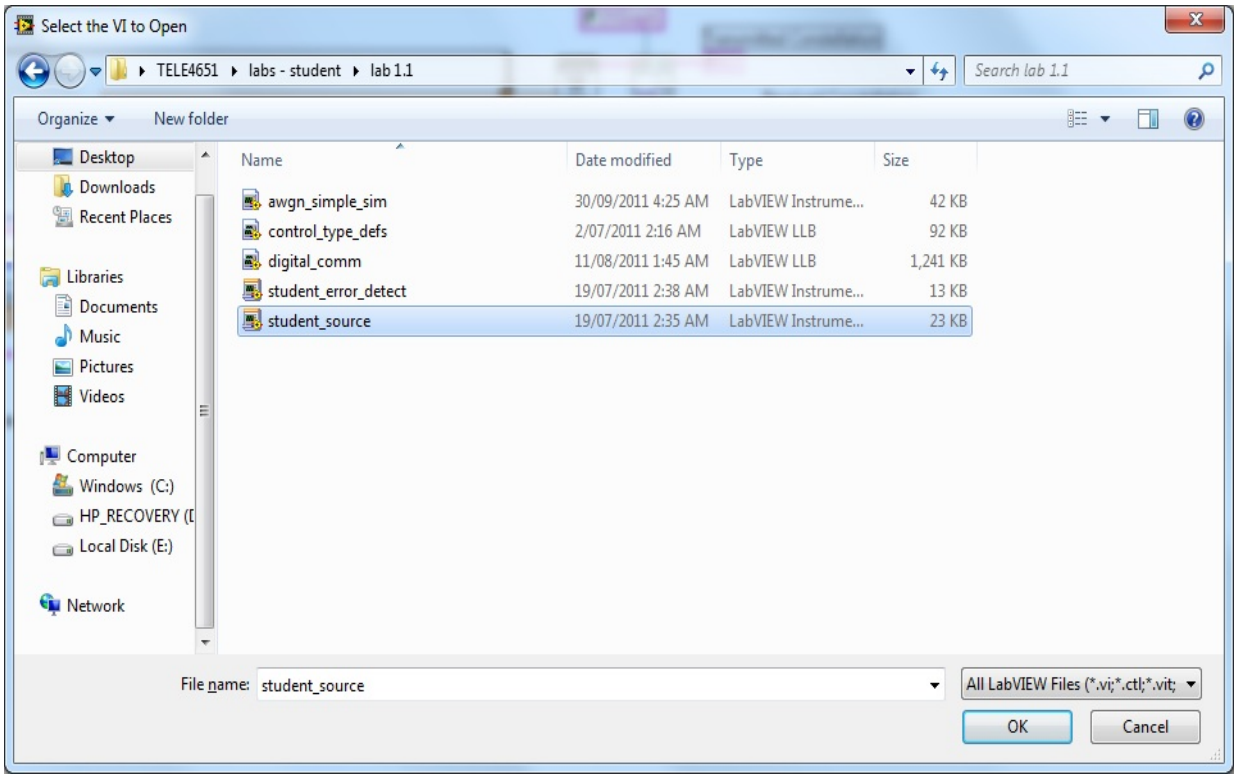

Fig. 7. Select your VIs.

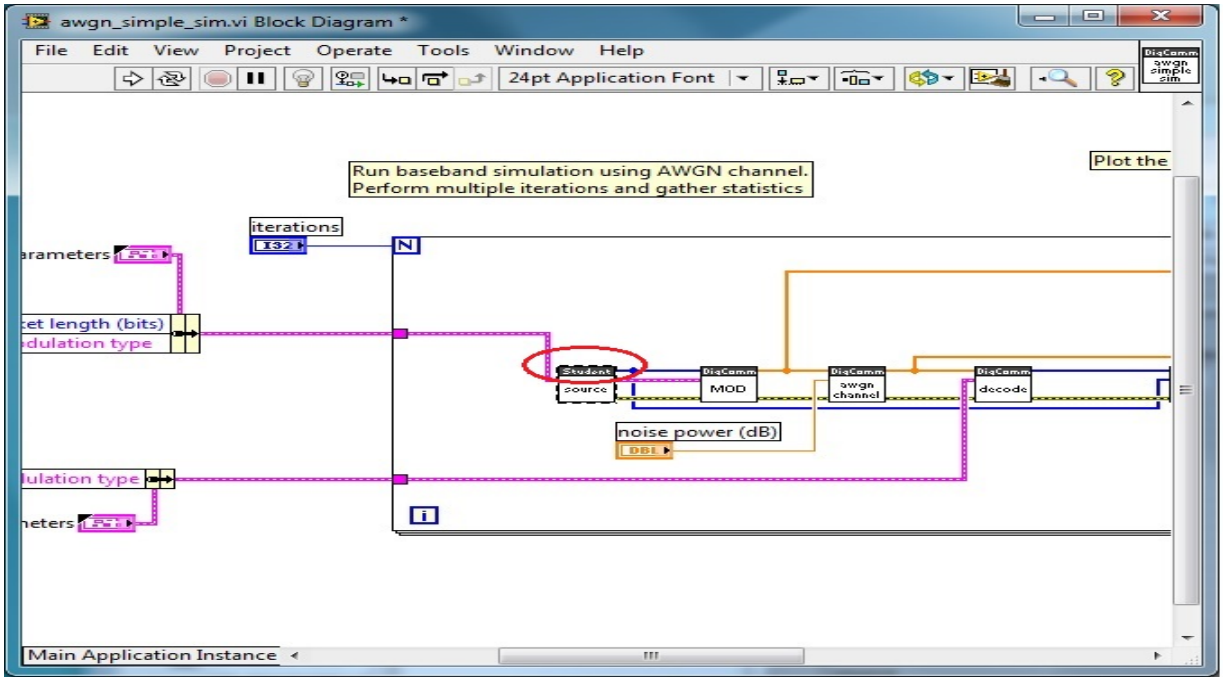

Fig. 8. Confirm the replacement.

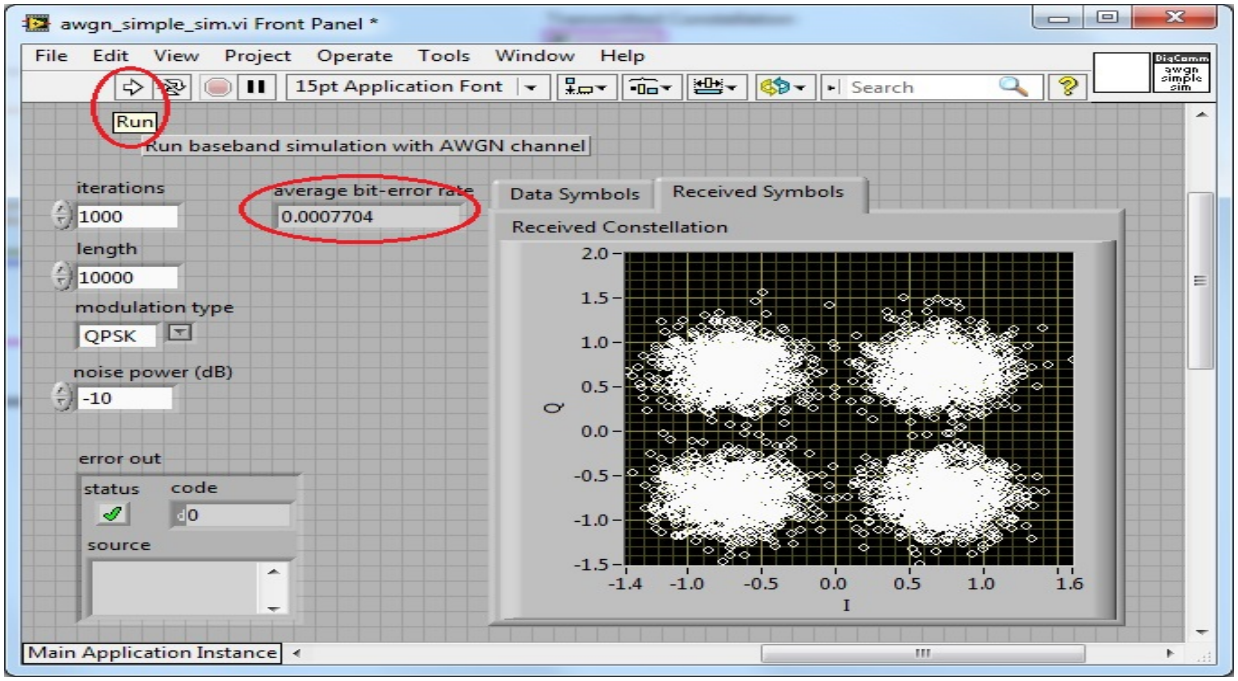

Fig. 9. Run the simulator.

IMPORTANT: The above flow does not verify the correctness of your VI file. In other words, the logics that goes into each VI file highly depends on your understanding of theory.

III. Lab hints

A. Lab 1 - Part 1

1) source.vi: To generate a random bit sequence, you may want to use the **random number generator** provided in the **Numeric palette**, or use the **Generate Bits. vi** module located in the **Digital palette**. If you are using the sub-vi **Generate Bits.vi**, in order to add some randomness when generating a bit sequence, you might want to use the **Tick count** to generate random seed number.

2) error_detection.vi: The components you might need for this vi are: **Array size, Index Array, Not equal, Max&Min, Add array element.** Remember to covert boolean results into numerical results by using **Boolean to (0,1)**. Moreover, convert any integer number into double precision number using **To double precision float** before calculate the bit error rate (BER).

NOTE: you do not need to plot the SNR vs BER graphically, instead, you only need to show the BER

value for each different signal to noise ratio (SNR).

B. Lab 1 - Part 2

Complete the questions asked. No VI file require to complete.

C. Lab 2 - Part 1

1) modulate.vi: This VI perform two tasks, the first task is to map the input binary sequence into symbols by following required constellation. The second task is to normalize the symbol energy such that the constellation has energy equal to the user input. Note that, as required by the lab manual, your *modulate.vi* need to contain both binary phase shift keying (BPSK) and quadrature phase shift keying (QPSK) modulation types, and do not use existing sub-vi file from the modulation toolkit. In addition, it is highly recommend that you separate the two tasks by creating another sub-vi file to perform the energy normalisation.

For the mapping task, you may first use a **DBL** type **Array Constant** to store your constellation symbols. Then, you can use the input bit sequence to index the array using **Index Array** component. As the input bit sequence is **U8** type, for QPSK modulation, you may need the **Decimate 1D Array** component to generate the index for the **Index Array** component.

For the normalisation task, you need to account the average power of the input symbol sequence for different modulation types using **Complex To Polar, mean.vi** and some other numeric components. Then you may need the **Compound Arithmetic** component to generate the final output symbols. The input constant "Symbol energy" in the *student_modulate.vi* file is one of the input of the normalisation sub-vi.

2) decode.vi: This VI maps the input **DBL** type symbols to bit sequence using ML detection. You may need a **DBL** type **Array Constant** to store your constellation symbols. You need to compare the input symbol with each element in the **Array Constant** to find out the one with minimum distance. You may use the **Array Max** & **Min, Interleave 1D Arrays** and **Complex To Polar** components to perform inverse mapping from symbol to binary bit sequence.

D. Lab 2 - Part 2

You will use a number of existing sub-vi files to build these two files.

1) pulse_shaping.vi: The components you might need for this vi are: **MT Generate Filter Coefficients.vi, Upsample.vi** and **Convolution.vi**.

2) matched_filtering.vi: The components you might need for this vi are: **MT Generate Filter Coefficients.vi** and **Convolution.vi**. You may also need to use **Bundle By Name** and **Unbundle By Name** to preserve the information in the output waveform besides the amplitude information in the input waveform.

E. Lab 3

1) *align_MaxEnergy.vi* : You need to perform two tasks. The first task is to find the offset which maximizes energy of receiver input waveform. The second task is to align the waveform according to this offset. To do this, you need to **Unbundle by Name** the input complex waveform, and find the offset base on the 1-D input array denoted by Y. You might want to loop over the RX oversample factor and record the maximum offset position use shift registers. Once you have obtained the maximum offset position, align the input waveform by changing the 1-D array Y according to the maximum offset. Lastly, you need to create a proper aligned complex waveform by bundle the modified array Y using **Bundle by Name**. The useful components for this VI are: **Array Subset**, **Decimate(single shot).vi**, **Complex to Polar**, **Add Array Elements**, **Select**.

2) *align_ELgate.vi*: In this VI, you need to perform an additional task compare to max energy algorithm mentions above; make initial guess for alignment offset. To perform, Early-late gate algorithm, you need to perform similar operation as Max energy algorithm, however, with additional two operations: a) estimated 'early' offset of Y array, b) estimated 'late' offset of Y array. Both operations require the use of the initial guess of alignment offset. Lastly, compute the error metric by averaging the difference between early and late estimates, and the on time Y array. Keep a history of offsets and stop when offset is repeated (use a while loop, not a for loop). Lastly, align waveform according to offset and don't forget to use **Unbundle by Name** and **Bundle by Name**.

You might find the following components are helpful: **Compound Arithmetic**, **Array Subset**, **Decimate(single shot).vi**, **Complex to Polar**, **Add Array Elements**, **Select**, **Search 1D array**.

IMPORTANT: In order to verify your VI files, you need to follow the steps below.

1) Download the "lab 3.zip" attachment from the Email and extracted on to your hard disc.
2) Copy your VI files for pre-lab 3 into the folder.
3) Open the "*digital_comm.llb*" and double click the sub-VI file *"symbol_timing.vi"*.

4) By choosing different case structure, you can replace each sub-VI files with your own file.

5) To verify it, you can always run the *"simulator.vi"*.

F. Lab 4

1) *LLSE.vi*: This is also called the *mmse_linear.vi* in the simulator. The task for this VI is to compute linear minimum mean square error (MMSE) estimate of $\mathbf{x}$ for $\mathbf{Ax} = \mathbf{b}$. You will be use a number of existing VI files that perform matrix operations. Useful components are: **NI_AALPro.lvlib:Transpose Matrix.vi**, **NI_AALPro.lvlib:A x B.vi**, **NI_AALPro.lvlib:Inverse Matrix.vi**, **Complex to Polar**.

2) *toeplitz.vi*: You need to create a 2-D toeplitz matrix from the given input rows and columns. You might find the following array operators are helpful: **Array Size**, **Array Subset**, **Reverse 1D Array**, **Build Array**, **Transpose 2D array**.

3) *direct_equalizer.vi*: Note that both *LLSE.vi* and *toeplitz.vi* are sub-VIs of *direct_equalizer.vi*. To implement *direct_equalizer.vi*, you need to create rows and columns from the sampled signal as the input of your *toeplitz.vi* sub-VI using array operators. Then the output of *toeplitz.vi* together with the training sequence goes into the *LLSE.vi* to compute the mean square error. You might find the following components are helpful: **Array Size**, **Array Subset**, **Reverse 1D Array**, **Build Array**, **Increment**, **Substract**.

IMPORTANT: In order to verify your VI files, you need to follow the steps below.

1) Download the "lab 4.zip" attachment from the Email and extracted on to your hard disc.
2) Copy your VI files for pre-lab 4 into the folder.
3) Open the "*digital_comm.llb*" and double click the sub-VI file *"equalizer.vi"*.
4) You can now replace direct equalizer with your own VI file.
5) To verify it, you can always run the *"simulator.vi"*.

G. Lab 5

1) *Moose.vi*: The task for this VI is to estimate the frequency offset using Moose algorithm and then correct the frequency offset of the received signal.

It is straight forward to estimate the frequency offset using Moose algorithm. First, you may need to find out the length of the training sequence $\mathbf{N_t}$ from the "modulation parameter", then you may get the data from the input signal from two windows with $\mathbf{N_t}$ symbols offset. The remaining work is implementing the equation (9) directly in the lab manual. The symbol period $\mathbf{T}$ in equation (9) can be calculated from the parameters "Rx Sample Rate" and "Rx oversample factor". You may need these components: **Array Size**, **Array Subset**, **Complex conjugate** and some other simple functions.

To compensate for the frequency offset, you may first implement the $-\mathbf{j2}\pi\epsilon\mathbf{n}$ of equation (6) in the lab manual, and then using **Polar To Complex** to generate the exponential term of equation (6). As the factor $\mathbf{n}$ is a sequence from 0 to N-1, where N is the length of the input data, you may need to use **NI_AALBase.lvlib:Ramp Pattern.vi** to generate the sequence.

2) *sliding_correlator.vi*: The task for this VI is to do the frame synchronization, you should first estimate the frame offset and correct the offset, and then you should apply the Moose algorithm to correct the frequency offset.

To estimate the frame offset using the equation (2) in lab manual, you need to extract the training sequence from the "modulation parameter" and then implement the equation directly. To do the sequence correlation, you may need to use **NI_AALBase.lvlib:CrossCorrelation.vi**.

NOTE: To replace the VI file in the simulator, open *frame_detect.vi* from "*digital_comm.llb*", by choosing different option of case structure, you can replace your *sliding_correlator.vi* into the simulator. The *Moose.vi* is a sub-VI for *sliding_correlator.vi*.

H. Lab 6

For this lab, if you read the beginning of section 2 of your lab manual, you are provided with a set of helper VIs. You are welcome to use any of these helpers to build your "*OFDM_modulator.vi*", "*OFDM_demodulator.vi*" and "*OFDM_FEQ.vi*" (worth extra marks).

1) *OFDM_modulator.vi* The tasks for the OFDM modulator contain are: serial to parallel conversion, padding zeros, inverse Fourier transform, add cyclic prefix and lastly parallel to serial conversion. At the output of the last stage, you may want to scale energy to compensate for IFFT.

Helpful components you might need are **Sort 1D array**, **NI_AALPro.lvlib:Inverse FFT.vi**.

2) *OFDM_demodulator.vi* The OFDM demodulator performs exactly the reverse procedures of the OFDM modulator. That is : the serial to parallel conversion, remove the cyclic prefix, do Fourier transform, perform frequency equalisation, remove the zeros that are padded and lastly, the parallel to serial conversion. Before you output the demodulated symbols, you may want to reshape the output of parallel to serial conversion and scale the energy.

Helpful components you might need are **Reshape array**, **Select**, **NI_AALPro.lvlib: FFT.vi**.

3) *OFDM_FEQ.vi* This VI is optional for you to do and worth extra 20 marks. The goal for this VI file is to perform equalization in frequency domain. In other words, it is a simple division of input Y by channel estimate H in frequency domain. Then pad any zeros in frequency domain.

Helpful components you might need are **Reshape array**, **Replace array subset**, **NI_AALPro.lvlib: FFT.vi**.

Note that for all three VIs, you can plot magnitude and phase response of channel using the helper VI "*OFDM_plot_channel.vi*" provided to you. To deal with error event or messages, you may consider use the **Error cluster from error code.vi** to help you.

I. Lab 7

"*OFDM_SchmidlCox.vi*" : You will only need to implement one algorithm for this VI, the Schmidl and Cox algorithm, which perform frequency offset correction and frame synchronization. The tasks this VI file should contain are listed in Section 1.3 of your lab manual. Note that you need the "Moose.vi" from lab 5 to complete the last step of the tasks.

Helpful components you might need for this VI are: **Initialize array**, **Insert into array**, **Array subset**, **complex conjugate**, **Select**, **Array Max & Min**, **Search 1D array** and lastly, your "Moose.vi" file.