EECS 307: Lab Handout 1
(FALL 2012)

I- Introduction: The Software Radio System

In this lab you will program a software-defined radio to generate and demodulate an amplitude-modulated signal. We first give a brief introduction to software radio and the hardware/software platform you will be using.

Software radio (See also http://en.wikipedia.org/wiki/Software-defined_radio)

A software-defined radio system, or SDR, is a radio communication system where components that have been typically implemented in hardware (e.g. mixers, filters, amplifiers, modulators/demodulators, detectors, etc.) are instead implemented by means of software on a personal computer or embedded system. While the concept of SDR is not new, the rapidly evolving capabilities of digital electronics have greatly expanded the types of radios and signal processing that can be implemented with SDR.

A basic SDR system may consist of a personal computer equipped with a sound card, or other analog-to-digital converter, preceded by a Radio Frequency (RF) front end. A significant part of the signal processing is handed over to the general-purpose processor, rather than being done in special-purpose hardware. Such a design produces a flexible radio, which can receive and transmit according to a wide range of different radio protocols (including modulation formats, transmitted waveforms, and demodulation and detection procedures) based on the software provided. Software-defined radios are expected by proponents such as the SDRForum (now The Wireless Innovation Forum) to become the dominant technology in radio communications.

GNU-Radio (Source http://gnuradio.org/redmine/projects/gnuradio/wiki)

GNU Radio is a free & open-source software development toolkit that provides signal processing blocks to implement software radios. It can be used with readily-available low-cost RF hardware to create a physical software-defined radio, or in isolation (without hardware) to simulate an actual radio. It is widely used in academic and commercial environments, and by hobbyists, to prototype wireless communications systems.

GNU Radio applications are primarily written using the Python programming language, while performance-critical signal processing is implemented in C++ using processor floating-point extensions, where available. This enables rapid development of real-time, high-data-rate radio systems in a simple-to-use, application-development environment. GNU Radio can also support the development of signal processing algorithms using pre-recorded data (say, from another system), avoiding the need for actual RF hardware.

GNU-Radio Companion (GRC)

(Source http://gnuradio.org/redmine/projects/gnuradio/wiki/GNURadioCompanion)

GNU-Radio Companion (GRC) refers to the Graphical User Interface (GUI) in the Integrated Development Environment (IDE) for developing GNU Radio applications. GRC is a graphical tool for creating signal flow graphs and generating flow-graph source code (like Simulink for Matlab).

Universal Software Radio Peripheral (USRP)

(Source http://gnuradio.org/redmine/projects/gnuradio/wiki/Hardware
https://www.ettus.com/product/details/USRP-PKG)

GNU Radio is a software library only, but does support several radio (hardware) front-ends. The most commonly used equipment with GNU Radio have been developed by Ettus Research, LLC, and are called *Universal Software Radio Peripherals (USRPs)*. In this lab, we will be using **USRP1.** The USRP1 is the original Universal Software Radio Peripheral™ hardware that provides entry-level RF processing capability. The USRP1 platform can support two complete RF daughterboards. We will be using the **BasicTx/BasicRx** daughterboards, which provide transmission/reception capability from/to the USRP from 1 to 250 MHz.

USRP1 specifications:
The architecture includes an Altera Cyclone FPGA, 64 MS/s dual Analog to Digital Converter, 128 Mega-samples/sec (MS/s) dual Digital to Analog Converter and USB 2.0 connectivity to provide data to host processors. A modular design allows the USRP1 to operate from DC to 6 GHz. The two daughterboards makes the USRP ideal for applications requiring high isolation between transmit and receive chains, or dual-band dual transmit/receive operation. The USRP1 can stream up to 8 MS/s to and from host applications, and users can implement custom functions in the FPGA fabric.

The family of hardware now includes different motherboards with USB or Gigabit Ethernet interfaces, possible sampling rates up to 100 MS/s, a range of front-ends for reception and transmission from zero Hz (DC) up to over 5.8 GHz. The devices can be operated with a PC or as standalone embedded devices. USRPs can be used without GNU Radio as well without any restrictions.

Universal Hardware Driver (UHD)

(Source http://code.ettus.com/redmine/ettus/projects/uhd/wiki)

All USRPs use the same *Universal Hardware Driver (UHD)*, which is natively supported by GNU Radio (officially distributed since release 3.4.0). UHD is a separate project from GNU Radio, but all UHD devices are compatible with GNU Radio. It works on all major platforms (Linux, Windows, and Mac); and can be built with GCC, Clang, and MSVC compilers.

After you have identified the USRP1, its daughterboards, and the USB interface, we are ready to start using the GNU Radio Companion.

II- Create an AM Signal with GNU Radio Companion

1. Open a new terminal and type: **gnuradio-companion**. This will launch GNU radio companion.

2. Start a new file in GNU Radio Companion (GRC). There should already be two blocks in the workspace, labeled "Options" and "Variable". The variable's ID is **samp_rate.** Since we are dealing with digital (sampled) signals, we must specify the rate at which we sample our "analog" signals.

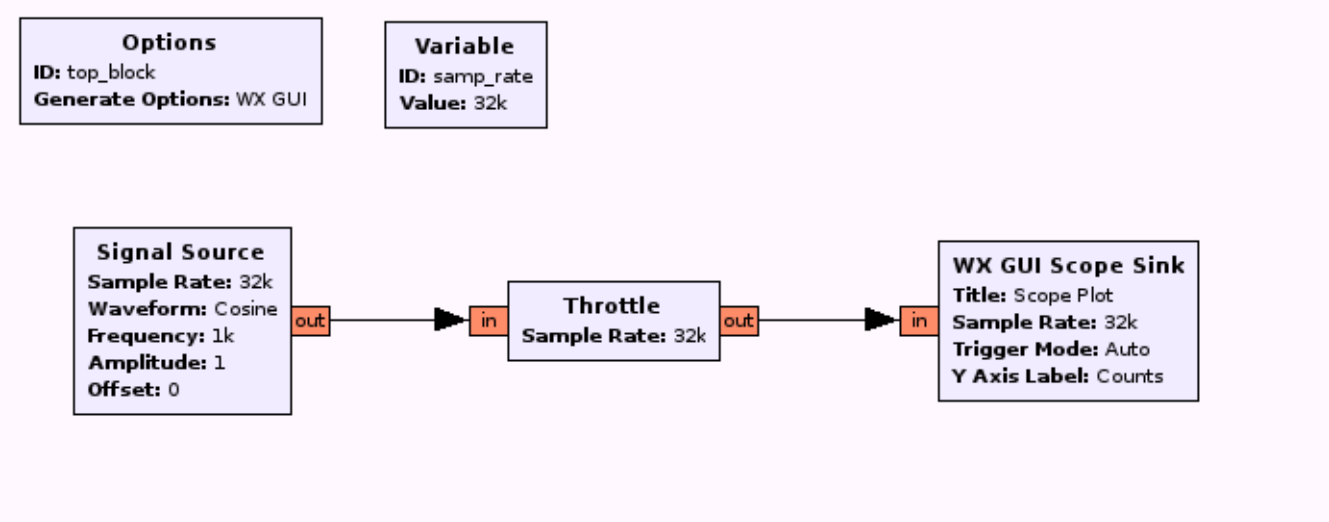
3. Double click on the **Variable** block to set the sampling rate. The value is set to 32 kHz and variable name is samp_rate. To use the variable, simply enter its symbolic name as a parameter of another block.

4. Create a **source signal block** by selecting "Signal Source" listed under [Sources] in the Blocks sidebar on the right and dragging it onto the workspace. Double click the block to edit its properties. This block generates a signal based on the properties listed here. It should already be set to generate a cosine of frequency 1 kHz and amplitude 1. Change the output type from "Complex" to "Float" and hit OK.

5. To view the signal, you can attach the output of the source block to a graphical sink. Create two blocks: a **throttle (in the Misc category)** and a **WX GUI scope sink (under WX GUI Widgets).** In the properties of both of these blocks, change the type to **Float** to match the source block. Also notice that the sampling rate is set to **samp_rate.**

   Note without a clock to determine the rate at which the samples are processed, such as in an audio device or a USRP, the graph would simply run as fast as possible and would consume 100% of the processing resources when executed, causing the GUI elements to be unresponsive. These flow graphs must therefore include a **Misc --> Throttle block** to throttle the rate of the streaming data. *If you omit the throttle block, the CPU may attempt to run the flow graph at full speed, consuming all of the computer's processing power.*

6. Connect the signal to the throttle by clicking the "out" (orange) block from the source block, then the "in" box of the throttle. Do likewise for the throttle and scope sink. The setup should look *roughly* something like this (the values may be different):



7. You should be able to execute the file. Do so by hitting the gear icon in the GRC menu (taskbar) or by hitting the F6 key. If you have not saved the file already, it will tell you to do so. Adjust the time/div option (controlling the horizontal axis) until the signal becomes clear. Verify that the frequency is correct and then close the window showing the signal.

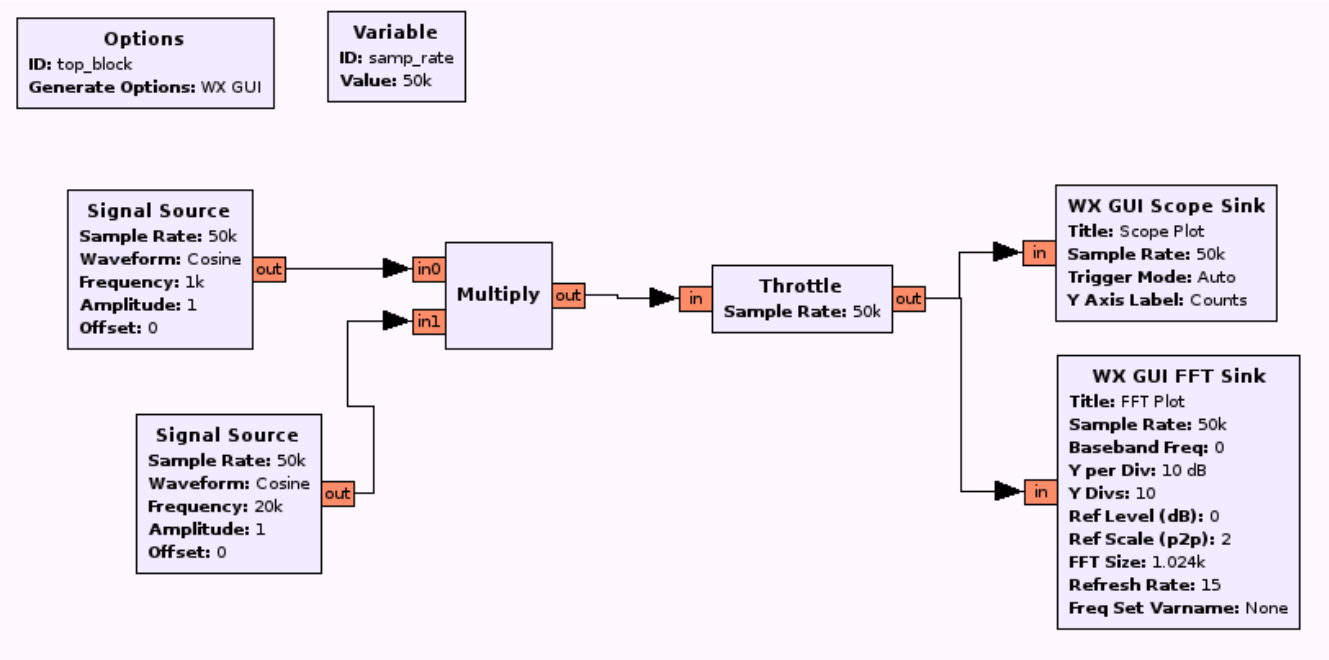
8. To view the signal in the frequency domain, create a WX GUI FFT Sink block, change the type to Float, and connect it to the output of the throttle. Do not disconnect the scope sink. Execute the file again to display the signal in both time and frequency. There should be a spike in the FFT plot corresponding to the frequency of the generated signal. Close the plots.

9. To use those plots in your report, click ALT+PrtScn to take a snapshot of the graphs (After you transfer your files, delete them from the system).

10. Now you will modulate the signal. First, disconnect the signal source from the throttle by selecting the connecting arrow and hitting delete. Create another signal source block, then create a multiply block (under Operators) and set both blocks to use the Float type.

11. Modify the new signal source to have a frequency of 20 kHz. This will be the carrier frequency. Notice that the current sampling rate as set by the samp_rate variable block is 32 kHz, which is below the Nyquist rate (minimum sampling rate needed to reproduce the original analog signal). *To fix this, set the samp_rate block value property to 100 kHz.*

12. Connect the two signal sources to the multiply block and the multiply block to the throttle. The final block diagram should look like this:



13. Execute the file to see the modulated signal in time and frequency.

14. Take a snapshot of the plots.

15. **What type of AM signal is this? (Hint: Can you see the carrier frequency?)**

16. Experiment with the settings in the signal source block. Specifically, look at the waveforms in time and frequency both before and after modulation. To view the signal before modulation copy the throttle, FFT, and scope sink blocks and connect the new throttle to the original signal source block. (Do not disconnect the source from the multiply block.) *If too many graphs are displayed, then disable one or two sinks by selecting them and hitting 'd'. (To re-enable select them and hit 'e'.)*

III- Coherent Demodulation

1. In this part, you will recreate the original signal from the modulated one using coherent demodulation. **What is meant by coherent demodulation?**

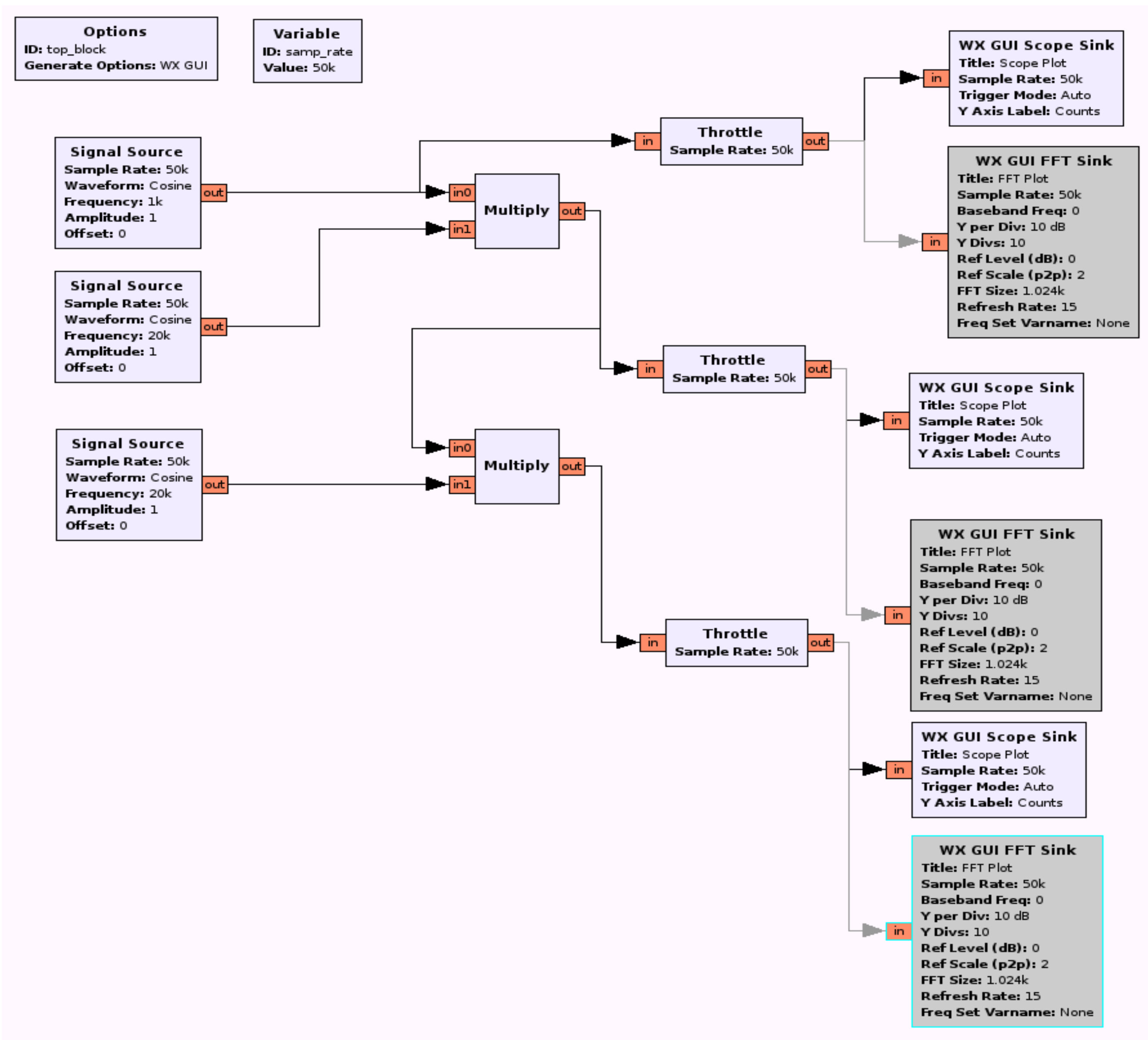
2. Start with the final diagram from the previous part. You should have two sets of outputs: an FFT and scope sink attached to the original signal, and another set for viewing the modulated signal.

3. Copy and paste the carrier signal block and the multiply block to build the coherent

demodulator. Connect the output of the original multiply block (i.e., the modulated signal) to the input of the newly added multiply block.

4. To view the final output, copy a throttle, scope sink, and FFT sink and connect them to the output of the second multiply block. At this point, the diagram should look like the figure below.

5. Disable for now the throttle, Scope sink, and FFT sink of *the modulated signal* (blocks in the middle) by selecting them and hitting 'd'.



6. Plot the original signal and the demodulated signal in the time domain. To do this disable the FFT sinks connected to both the original signal and demodulated signal. Run the file. There should be two plots in the time domain: one displaying the original signal and a second displaying the demodulated signal.

7. Take a snapshot of the plots and save it.

8. Plot the original signal and the demodulated signal in the frequency domain. Enable the FFT sinks for both the original signal and demodulated signal and disable the scope sinks. Run the file. There should be two plots in the frequency domain: one displaying the original signal and a second displaying the demodulated signal.

9. Take a snapshot of the plots and save it.

10. **Explain why the demodulated signal and original signals are not the same. Note that you modulated a 1 kHz cosine with a carrier of 20kHz. At the receiver the signal is multiplied by a 20 kHz carrier. What frequencies should the demodulated signal contain?**

11. To get rid of the high frequencies, add a Low Pass Filter (under Filters). Change the FIR type to Float --> Float (Decimating). Set the **Cutoff Freq** to 5kHz and **transition width** to 1kHz.

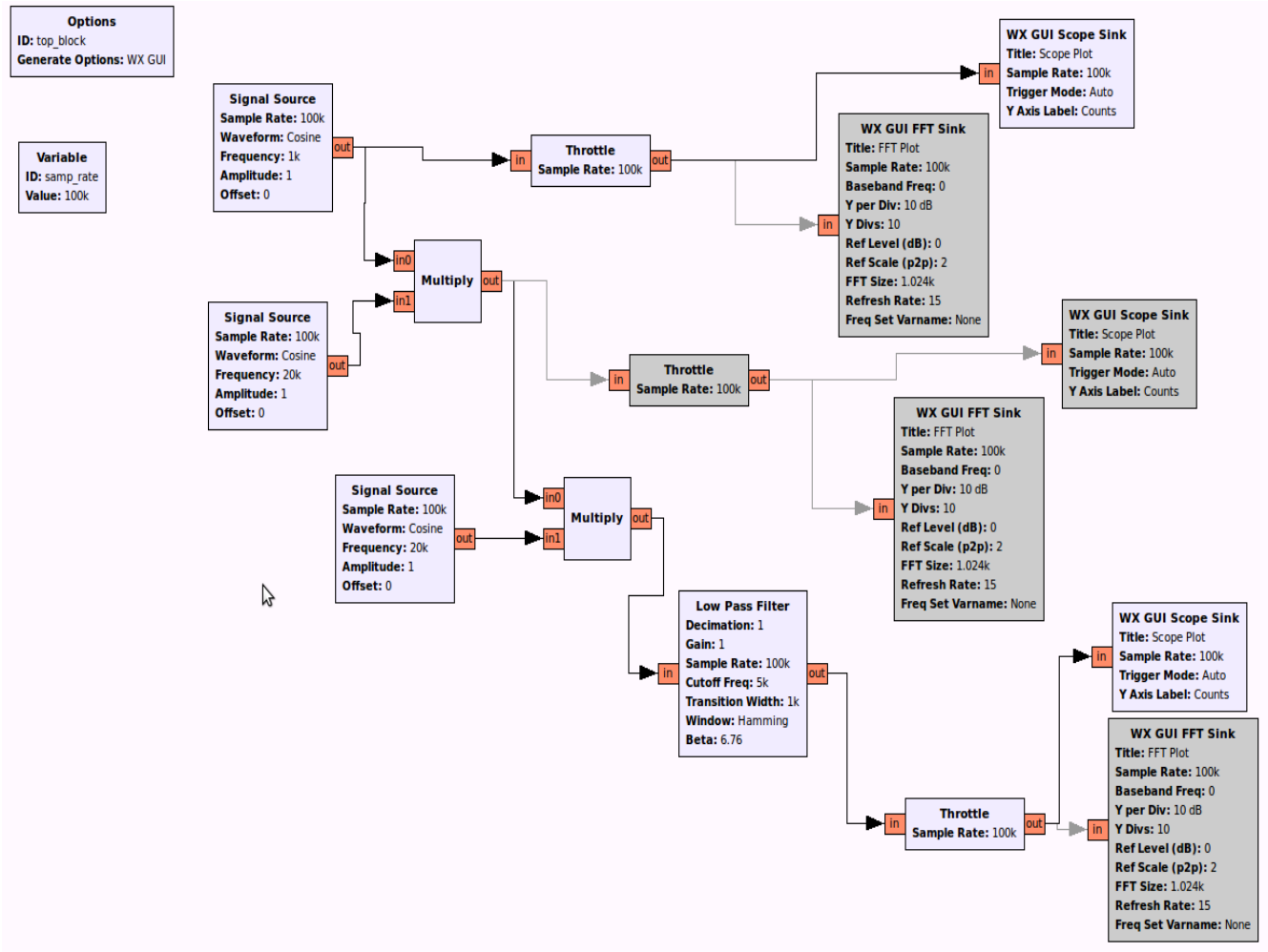
12. Disconnect the output of the demodulation throttle from the demodulation multiplier. Connect the output of the demodulation multiplier to the input of the Low Pass filter. Connect the output of the filter to the input of the demodulation throttle. The block diagram should look like the figure below.

13. Repeat steps 6-9 to see the final demodulated signal in both time and frequency. Save the corresponding two plots for your report.

IV- Envelope Detector

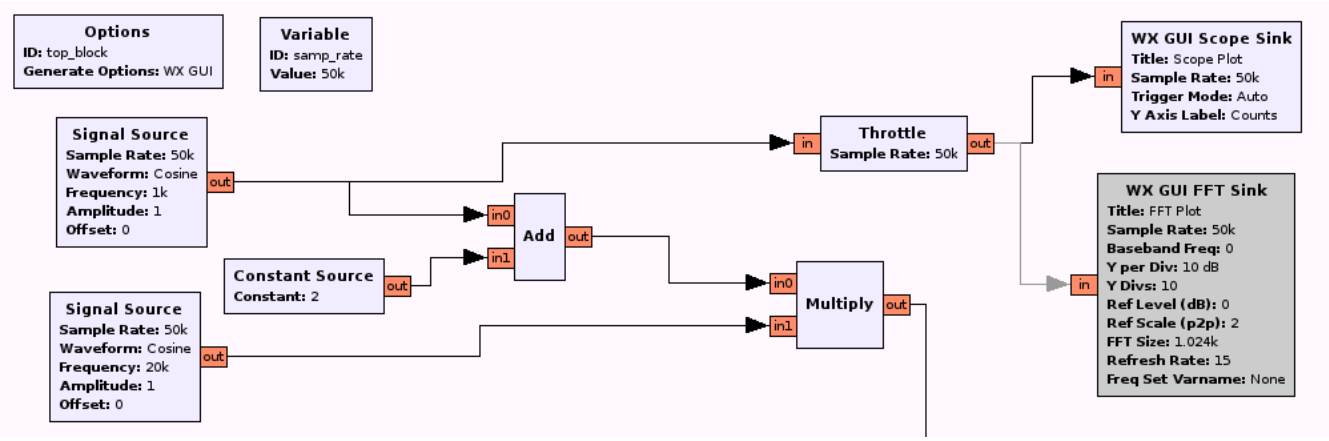
1. Now you will demodulate the signal using an envelope detector. **What are the advantages of using an envelope detector as opposed to a coherent demodulator? Can we use an envelope detector with suppressed carrier modulation?**

2. Save your current GRC file. Open a new file and copy the blocks of your previous file into the new one. That way it is easier to modify the block diagram. Save your new GRC file. *Make sure to set samp_rate to 100kHz in the new file.*



3. Add a DC component to the original signal. To do this disconnect the original signal source from the modulation multiply block and create two blocks: an add block (under Operators) and a constant source block (under Sources).

4. Change the type of both blocks to Float, and set the constant for the constant source to 2. Connect the constant and original sources to the add block inputs, and the add output to the modulation multiply block input. That part of the diagram should now look like this:



5. Delete the coherent demodulation multiply and carrier signal source blocks you created in the previous section.

6. To build the envelope detector, add two more blocks to your receiver: a “Max” (maximum value) block (under Operators), and a Null Source block (under Sources). Change the type of the null source and subtract to Float. *The null source just creates a stream of zeros.*

7. Connect the null source to one of the inputs of the max block, and the output of the modulation multiply block to the other. This will serve as a diode, since if the input is positive, it remains unchanged at the output, and otherwise, the output is zero.

8. Open the low pass filter's properties. Change the FIR Type of the filter to “Float --> Float (Decimating)”, then set the cutoff frequency to 5k and the transition width to 1k.

9. Your block diagram should look like the figure below.

10. For now disable the throttle, Scope sink, and FFT sink of *the original signal* by selecting them and hitting 'd'. Let us focus on the modulated and demodulated signals. (You need to enable the demodulation throttle, scope, and FFT.)

11. Plot the modulated and demodulated signals in the time domain. To do this, disable the FFT sinks of both the modulated signal and demodulated signal. Run the file. There should be two plots in the time domain: one displaying the modulated signal and a second displaying the demodulated signal.

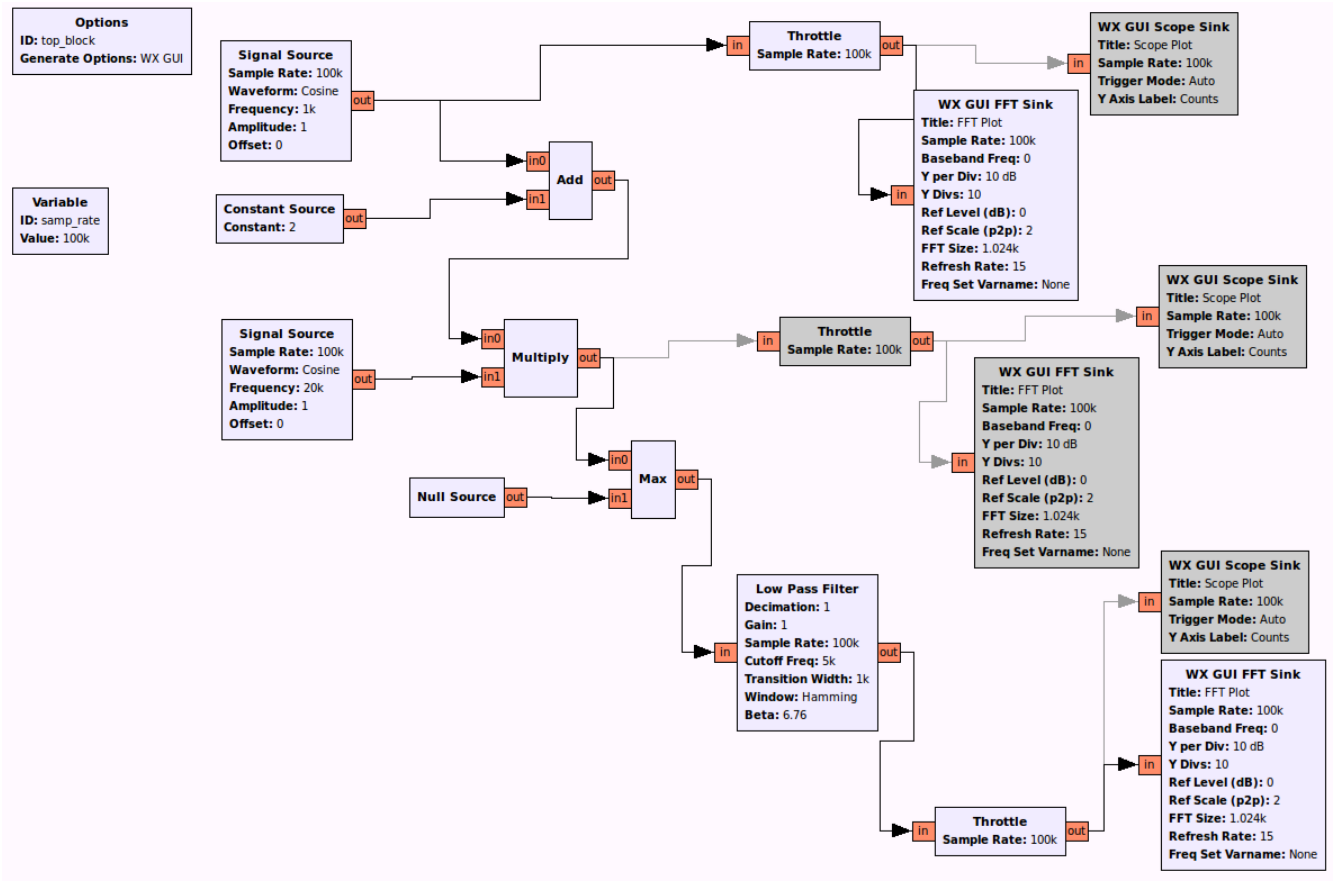
12. Take a snapshot of the plots and save it.

13. Plot the modulated and demodulated signals in the frequency domain. To do this, disable the scope sinks of both the modulated signal and demodulated signal. Run the file. There should be two plots in the frequency domain: one displaying the modulated signal and a second displaying the demodulated signal.

14. Take a snapshot of the plots and save it.

15. **What is the modulation index? Looking at the time domain plots, is the demodulated signal a distorted version of the message? Increase the amplitude of the original signal so that the modulation index exceeds 100%, and explain the plots.**

16. Looking at the time and frequency domain plots of the demodulated signal, there should still be a DC component. This can be filtered out by subtracting the average of the signal or by using a bandpass filter. The original signal is then recovered.



V- Lab Report

For the lab report, please answer all questions in this handout. Attach all plots and explain the results.