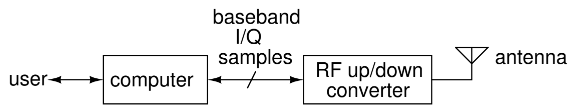
Introduction to SDR and GNU Radio

Introduction

Modern communication systems process signals as digitized (sampled and quantized) signals rather than as analog signals. This "Digital Signal Processing" (DSP) is less expensive and consumes less power than analog electronics.

Exceptions include the Analog Front End (AFE) that converts between digital baseband signals and the passband RF signal, RF power amplifiers, and power supplies, all of which are necessarily analog.

Most DSP is implemented with application-specific integrated circuits (ASICs) to minimize costs and power consumption. However, a sufficiently-fast CPU can also implement the signal processing. This is known as "Software Defined Radio" (SDR):



GNU Radio is an open-source project to develop software for SDR. GNU Radio Companion (GRC) is a graphical design tool that allows you to create GNU Radio software without writing code. Blocks representing signal processing functions are added to a diagram, configured, and interconnected using a graphical user interface. This "flowgraph" is then turned into an executable script that executes the processing blocks in the required order to process the samples. In this lab we will use GRC.

The advantages of GNU Radio and similar tools such as MathWorks's Simulink and National Instrument's LabVIEW are that they don't require programming and that the flowgraphs can be self-documenting. However, many prefer the flexibility of programming languages such as Matlab (e.g. using the Communications Toolbox).

GNU Radio and similar tools are also used to simulate communication systems. The simulations can be used to optimize designs and to generate test vectors for verification of DSP hardware.

A GNU Radio flowgraph can send/receive samples to/from radio hardware, files, network connections and other GNU Radio flowgraphs[1].

An example of a flowgraph created by GRC is shown in Figure 1.

Dataflow Flowgraphs

It may appear from the flowgraph that the blocks operate in parallel (simultaneously). However, since the blocks are implemented in software they must execute sequentially. This "dataflow" architecture means that blocks execute only when they have data to process. The order in which blocks are executed is determined by a scheduler that examines the amount of data available on the input ports of each block.

For efficiency reasons there are buffers between blocks that allow blocks to process multiple samples each time the block is executed. However, this buffering increases latency and can prevent proper operation of certain delay-sensitive applications such as those using feedback control (e.g. PLLs). Many blocks allow the maximum buffer sizes to be adjusted to minimize latency.

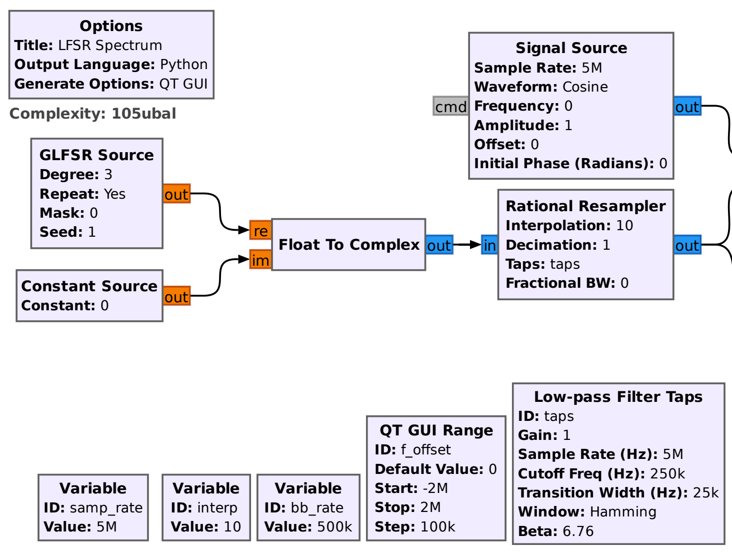
Complex Baseband Representation

It is possible to down-convert a real bandpass signal centered around a carrier frequency into a complex baseband signal centered about zero (DC). The complex baseband signal is centered on zero and, unlike real signals, the frequency components are not constrained to have even (or odd) symmetry.

The sampling rate for complex signals must be greater than the bandwidth of the signal to avoid aliasing. This is unlike the minimum sampling rate for real signals, which must be greater than *twice* the bandwidth.

The complex baseband architecture reduces implementation costs because receivers and transmitters can be implemented using only low-pass filters that can be integrated into ICs. This reduces costs by reducing parts count and board space.

[1]Including those written in programming languages such as Python, C++ and Matlab.


Options
Title: LFSR Spectrum
Output Language: Python
Generate Options: QT GUI

Complexity: 105ubal

GLFSR Source
Degree: 3
Repeat: Yes
Mask: 0
Seed: 1

Constant Source
Constant: 0

Float To Complex

Signal Source
Sample Rate: 5M
Waveform: Cosine
Frequency: 0
Amplitude: 1
Offset: 0
Initial Phase (Radians): 0

Rational Resampler
Interpolation: 10
Decimation: 1
Taps: taps
Fractional BW: 0

Multiply

QT GUI Frequency Sink
FFT Size: 1024
Center Frequency (Hz): 0
Bandwidth (Hz): 5M

Soapy PLUTO Sink
Sample Rate: 5M
Center Freq (Hz): 222.5M

QT GUI Time Sink
Number of Points: 1.024k
Sample Rate: 5M
Autoscale: No

Variable
ID: samp_rate
Value: 5M

Variable
ID: interp
Value: 10

Variable
ID: bb_rate
Value: 500k

QT GUI Range
ID: f_offset
Default Value: 0
Start: -2M
Stop: 2M
Step: 100k

Low-pass Filter Taps
ID: taps
Gain: 1
Sample Rate (Hz): 5M
Cutoff Freq (Hz): 250k
Transition Width (Hz): 25k
Window: Hamming
Beta: 6.76

Figure 1: GNU Radio Companion flowgraph for PRBS-modulated BPSK (2-QAM).

The baseband interface of the SDR hardware uses sampled complex baseband signals in I/Q (real/imaginary) format and most of the signal processing operates on these complex signals.

As an example, in this lab we will translate the frequency of signal by multiplying it by a complex sinusoid. GNU Radio also includes blocks that convert between signal of different types (e.g. byte, float and complex).

ML PRBS

In evaluating, testing, and implementing communication systems we often need a signal that has statistics similar to that of a random binary source but which is deterministic. This is called a pseudo-random bit sequence (PRBS).

The most common type of PRBS is the Maximal-Length PRBS (ML-PRBS) which has certain desirable properties, including having the maximum possible period for a given generator complexity.

An ML-PRBS can be generated using flip-flops and exclusive-or gates to compute feedback in a circuit called a linear-feedback shift register (LFSR).

Filters

The radio spectrum is divided into bands which are allocated to different uses (broadcasting, cellular phones, radio-navigation, etc.). Each band is typically divided into smaller ranges, called channels with each channel allocated to a different set of users.

Filtering is a important part of most communication systems. Transmitters use them to limit out-of-band power that might interfere with users of other channels and receivers use them to reject signals present on other channels.

Low-pass filters are also used when changing the sampling rate of digitized signals. Increasing ("interpolation") or decreasing ("decimation") the sampling rate requires post- or pre-low-pass filtering the signal to avoid aliasing.

There are many digital filter types (IIR, FIR), architectures (parallel, cascade, FFT) and design methods (window, mini-max). In this lab we will use a simple window-based design of an FIR filter to interpolate a low-bandwidth baseband signal's sampling rate to the higher rate required by the SDR transmitter.

Variables and GUI Widgets

A GNU Radio flowgraph is converted into a Python script. This allows us to embed variables (using the "Variable" block) and arbitrary Python code into a flowgraph.

It's also possible to include blocks that create user interface components ("widgets") that are active while the block is running. These can be used to display waveforms or frequency spectra and to change values of variables [2]. These variables can be used in expressions that configure the operation of blocks. You saw examples of this in the UI presented by the previous lab which was implemented using GNU Radio.

Procedure

In this lab you will practice using GNU radio by building a transmitter that incorporates some typical blocks. The flowgraph generates a ML-PRBS, multiplies it by a complex exponential to shift the frequency and then interpolates to increase the sampling rate.

The diagram in Figure 1 shows the flowgraph.

Using GRC

GRC can be installed with the radioconda distribution or run from AppsAnywhere. Start GRC. Under the View menu enable the Block Tree Panel, Console Panel and Variable Editor. Make sure 'Hide Variables' is not checked.

The easiest way to add a block is to search for it by name. Click on the magnifying glass icon (🔍) on the menu bar or type Control-F. Type part of the block name and double-click on the desired block name. This adds it to the flowgraph. You can then move the block by dragging it (any connections will follow).

To connect block outputs to inputs click on the output port and then click on the input port you want to connect it to.

As in any diagram, the blocks should be arranged so that the logical flow is left-to-right and top-to-bottom whenever possible.

Samples flowing between blocks can be of different types. Bits are typically transferred in the least-significant bit of a Byte type and signal samples are typically Float or Complex types. In GRC the color of the input and output connectors reflects the signal type (e.g. orange for floats, blue for complex).

If there is a mismatch between the signal types the connector will be drawn in red.

You can double-click on a block to open up its properties dialog box. This allows you to configure the input and output types and other aspects of the block's operation. The block's title and some of the block's properties are displayed inside the block in the flowgraph.

Some blocks define data structures instead of processing samples. Examples include the **Variable** blocks that set the value of a variable. This variable can be used when configuring other blocks. By default a variable, `samp_rate`, is included in a new flowgraph. Another example is the **Low-pass Filter Taps** block that calculates the tap weights for a low-pass filter.

[2]Parameters whose names are underlined in a block's properties dialog box can be updated during flowgraph execution.

Configure the Flowgraph

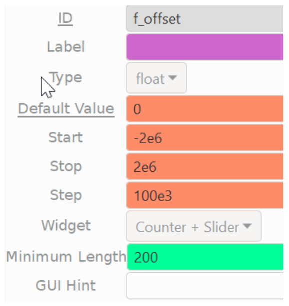
Use the File ▶ Open menu item to start a new QT GUI flowgraph. Then add, configure and connect the following blocks:

Variable Add a second variable with ID `interp`. This will be used to set the interpolation ratio between baseband samples and RF samples.

Add a third variable with ID `bb_rate` that has the value `samp_rate/interp`. Note that the value of a variable can be set to an arbitrary Python expression. The code generated for each block is shown in the "Generated Code" tab.

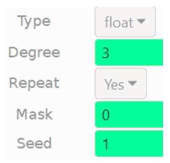
Values should be entered as Python numeric literals (e.g. `5e6`) although the GUI will show an SI prefix instead (`5M`).

QT GUI Range This block generates a run-time control that allows adjusting the carrier frequency offset between −2 and +2 MHz in steps of 100 kHz.



ID	f_offset
Label	
Type	float
Default Value	0
Start	-2e6
Stop	2e6
Step	100e3
Widget	Counter + Slider
Minimum Length	200
GUI Hint	

GLFSR Source This block ("Galois" LFSR) generates a ML-PRBS. Configure this block as follows:

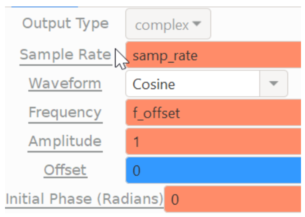


Type	float
Degree	3
Repeat	Yes
Mask	0
Seed	1

Degree, N, is the number of bits in the shift register. The PRBS period is $2^N - 1$ bits. Configure $N = 3$ which will generate a PRBS with a period of 7 bits. Mask defines the LFSR taps. If specified as 0 a suitable generator polynomial for the specified degree will be chosen. Note that float outputs have values ± 1 while byte outputs have values 0 or 1.

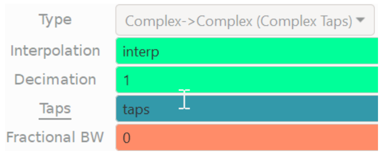
Type Conversions Type conversions are sometimes required. For example, the GLFSR Source above is configured to output floats but these need to be converted to Complex values. Add a **Float To Complex** block that sets the imaginary component to zero by connecting a **Constant Source** with value 0.

Signal Source Add a [Co-]Sinusoidal signal source with a Complex output whose amplitude is 1 and whose frequency is set by the variable `f_offset`. It is configured as follows:



Output Type	complex
Sample Rate	samp_rate
Waveform	Cosine
Frequency	f_offset
Amplitude	1
Offset	0
Initial Phase (Radians)	0

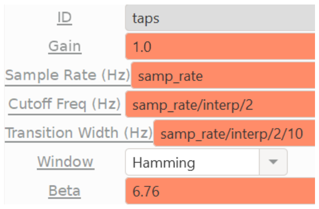
Rational Resampler Add an interpolator block configured as follows:



Type	Complex->Complex (Complex Taps)
Interpolation	interp
Decimation	1
Taps	taps
Fractional BW	0

This interpolates by a factor `interp` and the low-pass anti-aliasing filter uses the taps in the variable `taps` defined in the block:

Low-pass Filter Taps This block computes the FIR fiter weights for a low-pass filter whose (one-sided) bandwidth is half of the sampling rate and whose transition band is 10% of this:



ID	taps
Gain	1.0
Sample Rate (Hz)	samp_rate
Cutoff Freq (Hz)	samp_rate/interp/2
Transition Width (Hz)	samp_rate/interp/2/10
Window	Hamming
Beta	6.76

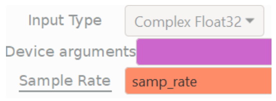
Multiplier This block does a complex multiplication of two inputs.

QT GUI Frequency Sink This block is similar to a spectrum analyzer and shows the power spectrum when the flowgraph is running.

QT GUI Time Sink This block is similar to an oscilloscope and shows the real and imaginary components versus time when the flowgraph is running.

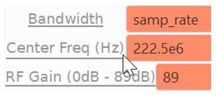
QT GUI Range This block defines a GUI interface item that allows run-time changes to variables when the flowgraph is run. Set ID to `f_offset` to allow run-time control of the frequency of the complex [co]sinusoidal signal source block added above.

Soapy Pluto Sink This block represents the interface to the transmit portion of the SDR hardware. It is configured with the following General options:



Input Type	Complex Float32
Device arguments	
Sample Rate	samp_rate

and the following RF Options:



Bandwidth	samp_rate
Center Freq (Hz)	222.5e6
RF Gain (0dB - 89dB)	89

Sample Rate sets the DAC sampling rate and Bandwidth selects the (two-sided) baseband bandwidth which is the same as the sampling rate for complex signals. Center Frequency sets the RF center frequency[3] and the RF Gain value sets the gain of the digital and analog portions of the transmitter.

Measurements

Connect the ADALM-Pluto SDR module to one of the PC's USB ports using the supplied USB extension cable. Connect the SDR's **Tx** output to the spectrum analyzer's **RF In** input using the supplied SMA-to-BNC cable.

[3]This falls in an amateur radio band although we will not be transmitting "over the air" in this lab.

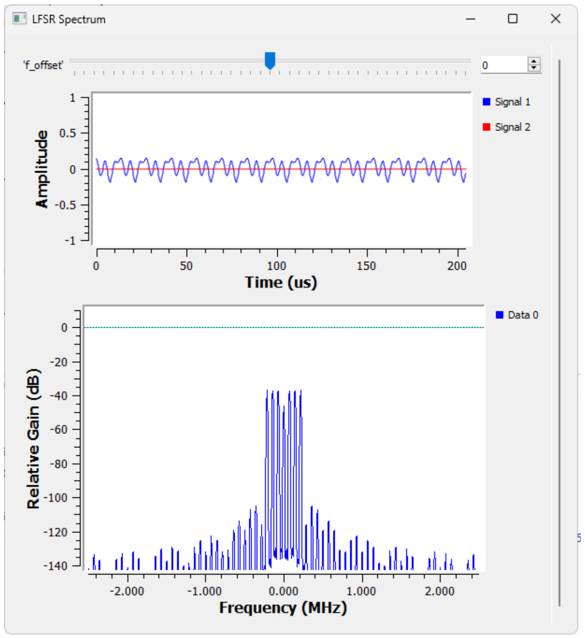
Select Run ▶ Execute, press F6, or press on the run (▷) icon. This will convert the flowgraph to a Python script and run it.

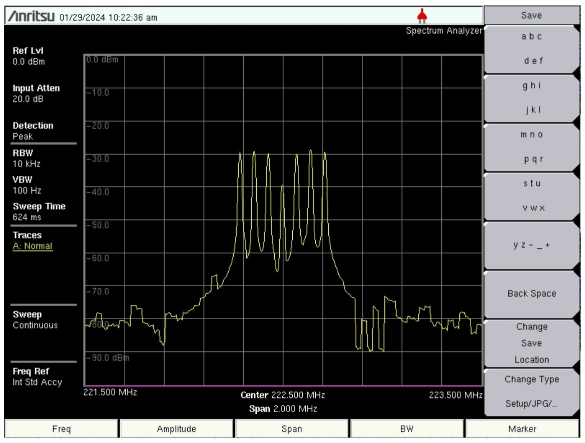
Error messages will show up in the console panel on the bottom left.

View the RF Spectrum

Configure the spectrum analyzer for a center frequency of 222.5 MHz, a span of 2 MHz, reference level of 0 dBm, enable auto resolution and video bandwidths, and set both the Span/RBW and RBW/VBW ratios to 100.

Capture the baseband spectrum shown on the GNU Radio Companion run-time window ("Top Block") and the RF spectrum as displayed on the spectrum analyzer:





Ref Lvl 0.0 dBm
Input Atten 20.0 dB
Detection Peak
RBW 10 kHz
VBW 100 Hz
Sweep Time 624 ms
Traces A: Normal
Sweep Continuous
Freq Ref Int Std Accy
221.500 MHz
Center 222.500 MHz
Span 2.000 MHz
223.500 MHz

To capture the GNU Radio Companion GUI you can use the Windows Snipping Tool (Windows-Shift-S).

To capture the spectrum analyzer display, plug a USB flash drive into the connector on the analyzer's front panel. Select File (Shift-7) ▶ Save. Select Change Save Location and select USB1. Select Change Type and select JPEG. Enter a file name and press Enter to save an image file showing the spectrum analyzer display.

Additional Test Cases

Make the following changes and capture the resulting baseband signal and RF spectrum:

- shift the signal up in frequency by 500 kHz using the GUI slider
- change the PRBS period from 7 to 255 bits
- change the filter Taps value to `[1]` to disable the interpolator's anti-aliasing filter

Lab Report

Submit a report in PDF format to the appropriate dropbox on the course web site.

Your report should include the four GUI and spectrum analyzer captures for each of the test conditions above as well as answers to the following questions:

- What is the result of multiplying $e^{-j\omega_1 t}$ by $e^{-j\omega_2 t}$? What is/are the resulting frequency(ies)? How does this differ from multiplying two real sinusoids?
- What is the period of the PRBS sequence for $N = 3$? What is the spacing of the frequency components? What are the period and frequency spacing for $N = 8$?
- What happens when you disable the interpolator's low-pass filtering? What is the name of this effect?