

Gnu Radio & Frequency Demodulation - One More Time

Gary Schafer, Blursday, Pandemic Time, Maybe May 2020

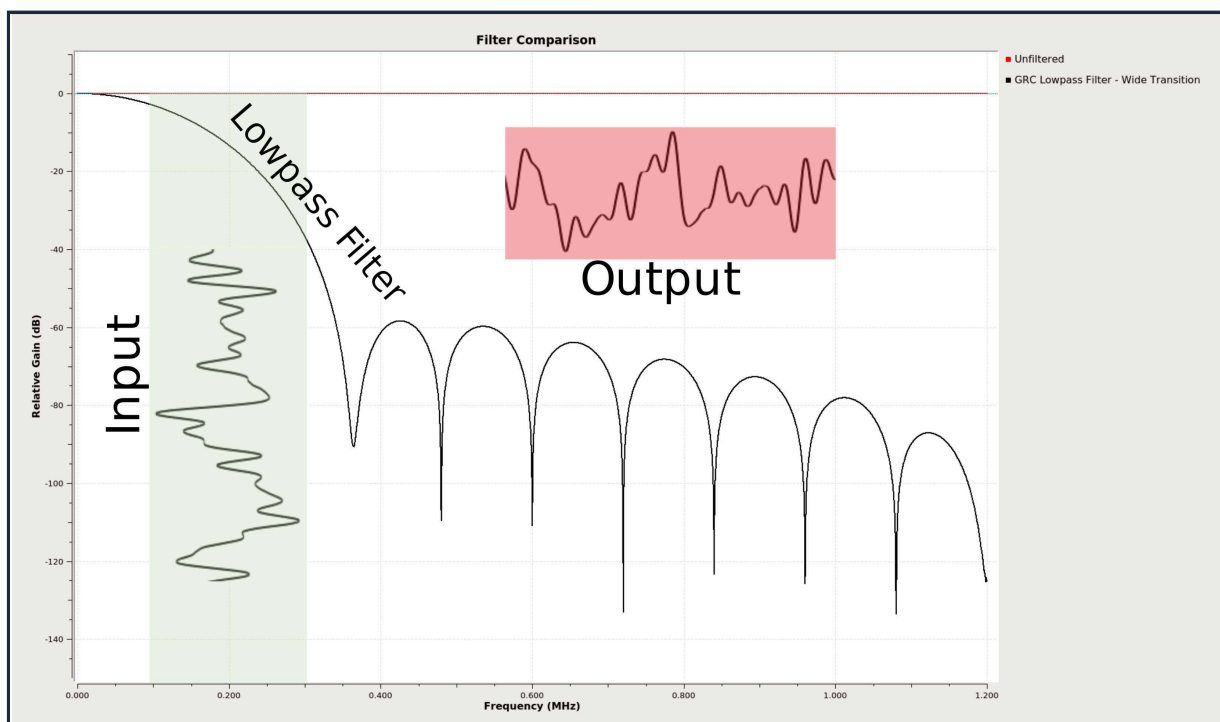
If you're here to find a basic FM receiver because you're just getting started with SDRs and Gnu Radio Companion, I have you covered.

As I said in my last post, I went deep down the rabbit hole in trying to understand how Gnu Radio Companion's frequency demodulator works.

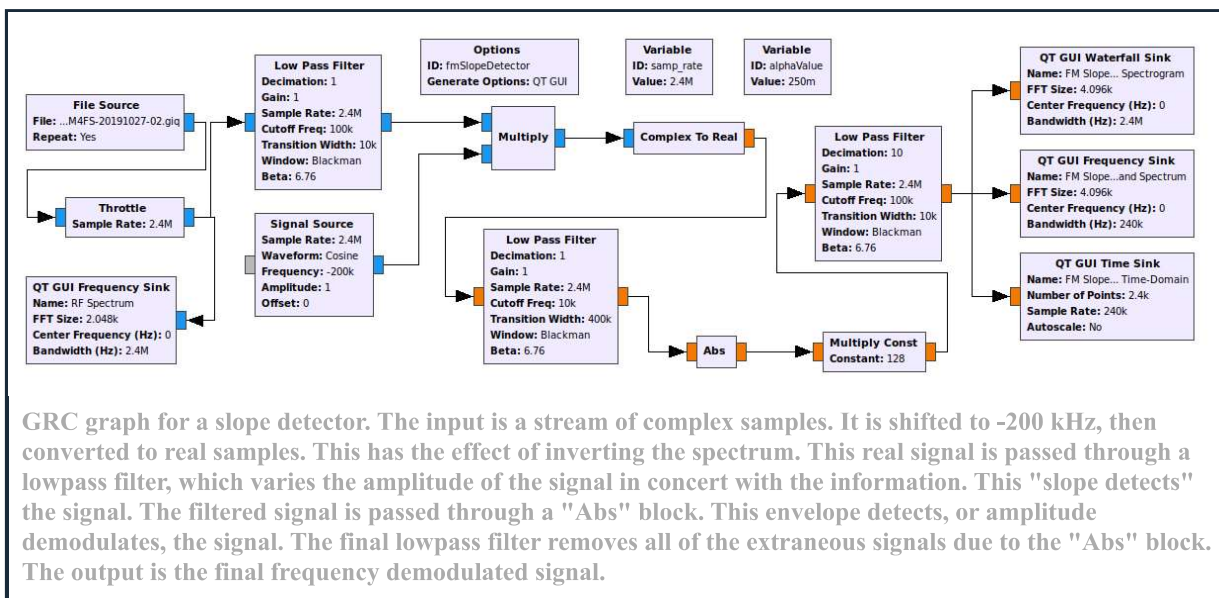
I wound up trying to answer the question, *"How many different ways can you make a frequency demodulator in Gnu Radio Companion?"* Turns out, quite a few. I'm just going to document them here. I'm going to go through these different methods roughly historically.

THE SLOPE DETECTOR

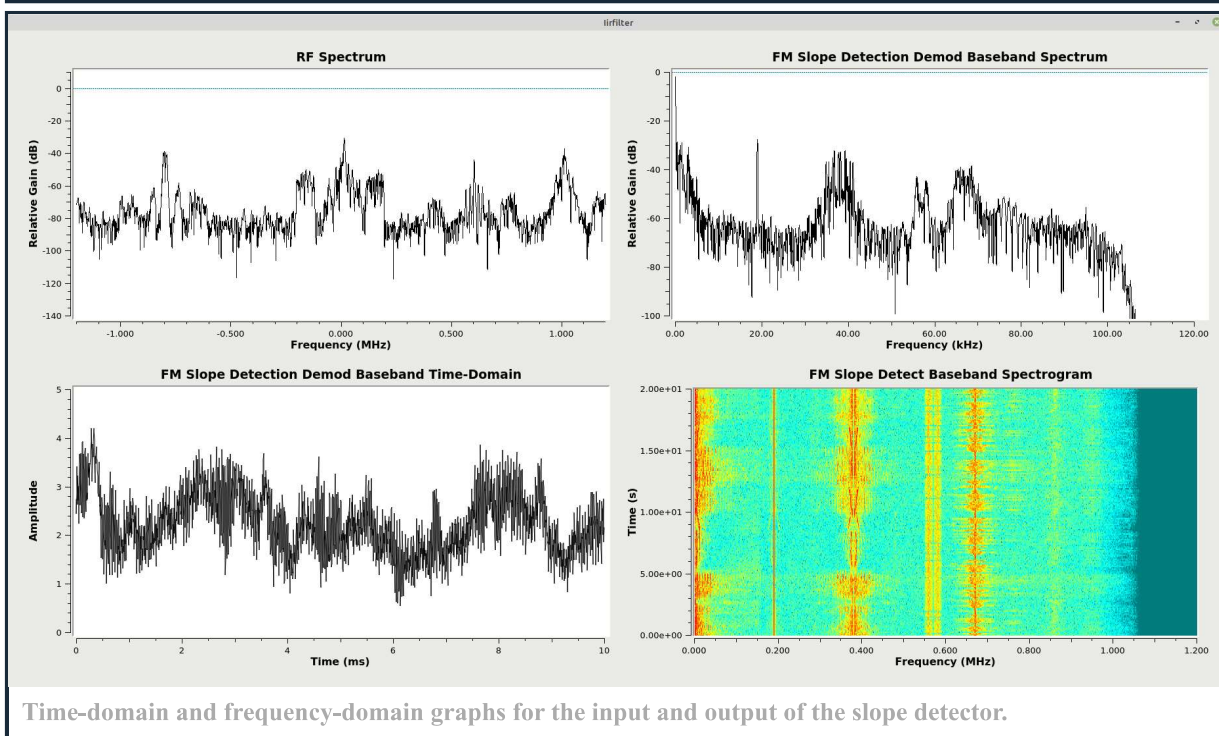
I'm going to start with the old, analog days. One of the first frequency demodulators used^[CITATION NEEDED] was a "slope detector". The idea is this. A frequency modulated (FM) signal will vary its center frequency in concert with the amplitude of the modulating signal. As the modulating signal goes higher in amplitude, the FM signal will increase in frequency. And vice-versa. If you have a filter whose amplitude vs frequency is a slope over the frequency range of the FM signal, then the output of the filter will be the FM signal, but also amplitude modulated with the same information. Now it can be demodulated the same as any AM signal.



Basic concept of the slope detector. As the input FM signal deviates in frequency, the amount of attenuation in the filter will also vary. In this case, as the frequency deviates lower, the signal is attenuated less. As it deviates higher, it will be attenuated more. (NOTE: This is the opposite of how it should be, but the concept is the same.) The output will be the FM signal, but now also amplitude modulated with the same information. This filter was generated using a GRC "Low Pass Filter" block and a relatively high transition width. ($F_s = 2.4$ MHz, cutoff frequency = 10 kHz, transition width = 400 kHz).



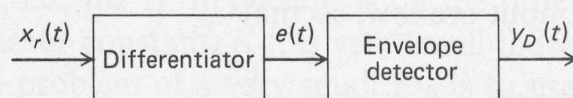
GRC graph for a slope detector. The input is a stream of complex samples. It is shifted to -200 kHz, then converted to real samples. This has the effect of inverting the spectrum. This real signal is passed through a lowpass filter, which varies the amplitude of the signal in concert with the information. This "slope detects" the signal. The filtered signal is passed through a "Abs" block. This envelope detects, or amplitude demodulates, the signal. The final lowpass filter removes all of the extraneous signals due to the "Abs" block. The output is the final frequency demodulated signal.



THE DIFFERENTIATOR AND ENVELOPE DETECTOR

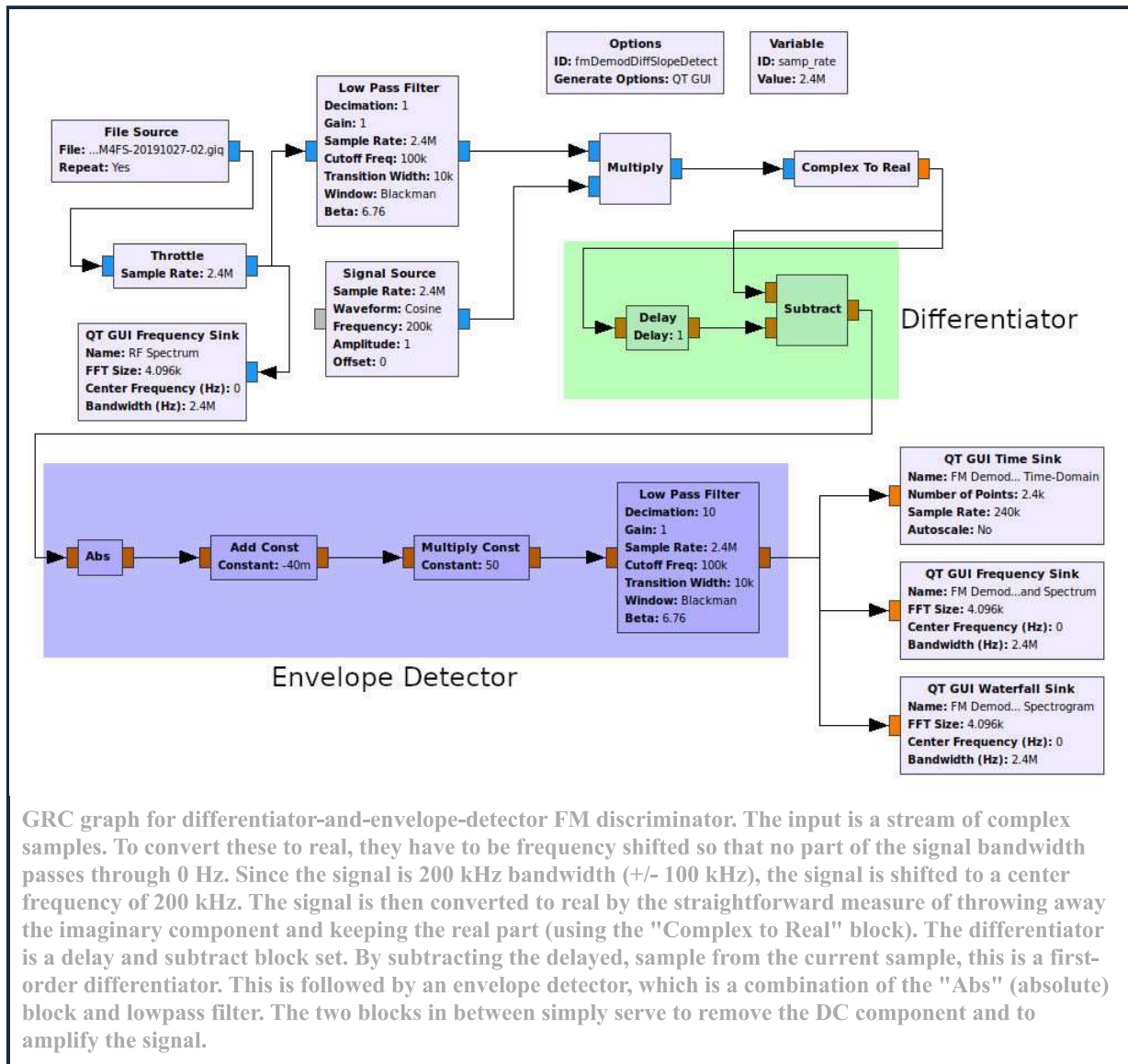
Mathematically, the slope detector is equivalent to a differentiator followed by an envelope (AM) detector, as shown below.

Figure 3.34 FM discriminator

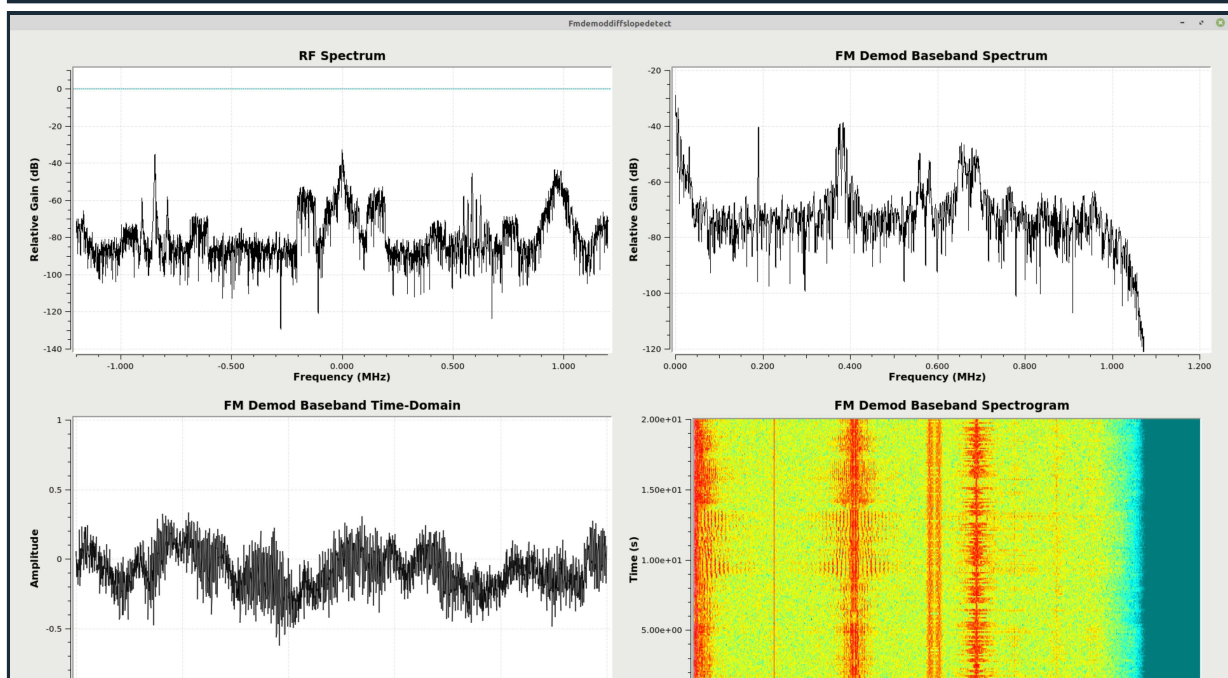


Block diagram for FM discriminator using a differentiator followed by an envelope detector. (From "Principles of Communications: Systems, Modulation, and Noise (Second Edition)", R.E. Ziemer, W.H. Tranter, Houghton Mifflin Company, 1985)

What do computers do really well? Math. GRC provides all of the components necessary to make such a demodulator directly. This is shown below, along with the resulting time- and frequency-domain graphs.



GRC graph for differentiator-and-envelope-detector FM discriminator. The input is a stream of complex samples. To convert these to real, they have to be frequency shifted so that no part of the signal bandwidth passes through 0 Hz. Since the signal is 200 kHz bandwidth (± 100 kHz), the signal is shifted to a center frequency of 200 kHz. The signal is then converted to real by the straightforward measure of throwing away the imaginary component and keeping the real part (using the "Complex to Real" block). The differentiator is a delay and subtract block set. By subtracting the delayed, sample from the current sample, this is a first-order differentiator. This is followed by an envelope detector, which is a combination of the "Abs" (absolute) block and lowpass filter. The two blocks in between simply serve to remove the DC component and to amplify the signal.

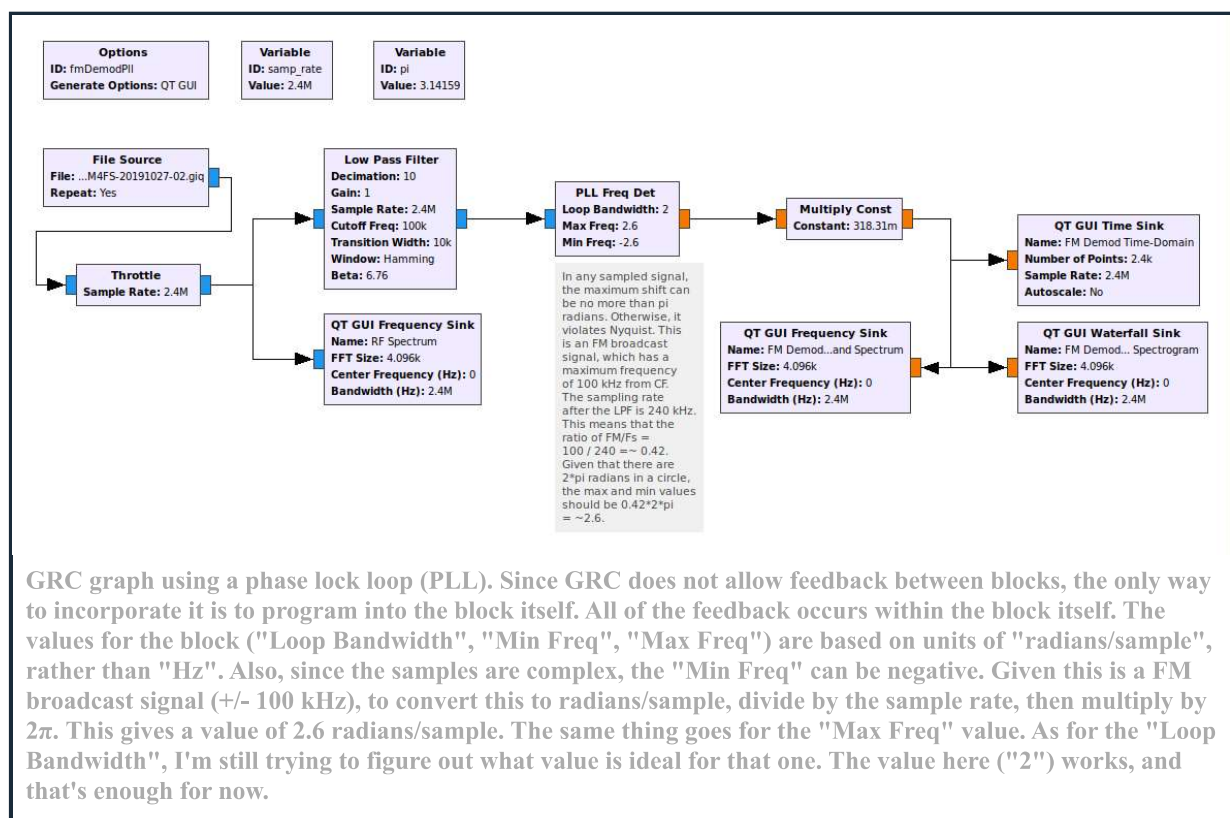


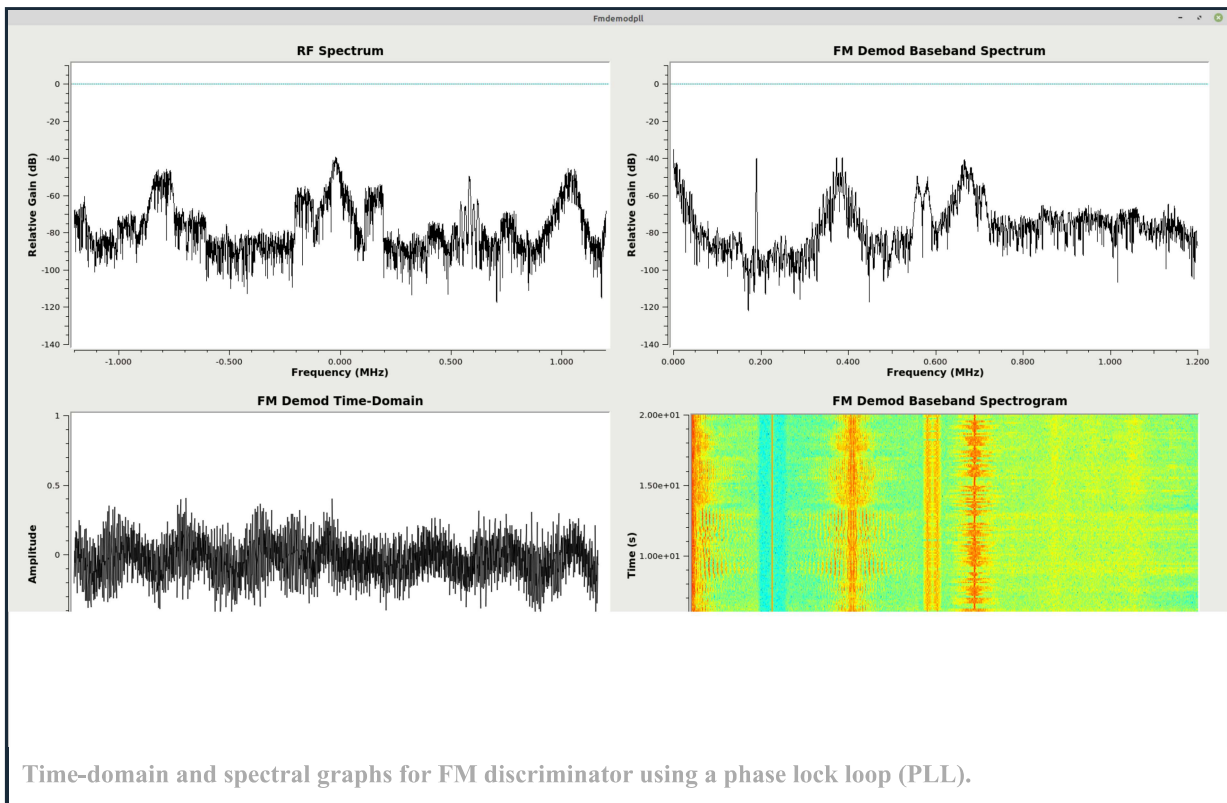
THE FOSTER-SEELY AND RATIO DISCRIMINATORS

I looked at mimicking both a Foster-Seely and ratio discriminators using GRC. They're both based on using transformers. Both of these circuits operate by varying the lead and lag of the current with the voltage. I've not been able to figure out how to mimick this with GRC. On to the next one.

THE PHASE LOCK LOOP (PLL) DEMODULATOR

The phase lock loop (PLL) is the only circuit that requires a feedback loop. GRC does not allow loops between blocks. However, GRC does provide a PLL block, in this case the "PLL Freq Det" block. Here's the problem. The "PLL Freq Det" block does not use normal units for the minimum and maximum frequency. Rather, they use units of "radians/sample". What does *that* mean? We're talking about sampled signals. FM signals will have phases that are changing with each sample. The phase difference per sample will depend on the quotient of the frequency deviation divided by the sample rate. Phase can be measured in one of three units. These are radians, gradians, and degrees. Radians allows for easier math. So that's what is used more often. The phase change per sample, in radians, is nothing more than this quotient multiplied by 2π . For this example, the sample rate at the input of the PLL block is 2.4 MHz (the sample rate) divided by 10 (the lowpass filter decimation value) = 240 kHz. The deviation of an FM broadcast signal is 75 kHz. However, to ensure that there's no overload, I increased this to 100 kHz. The ratio is thus $100 / 240 = 0.4167$. The value in radians/sample is $0.4167 * 2\pi = \sim 2.6$ radians/sample. This corresponds to both plus and minus. These are the values that are in the "PLL Freq Det" block.



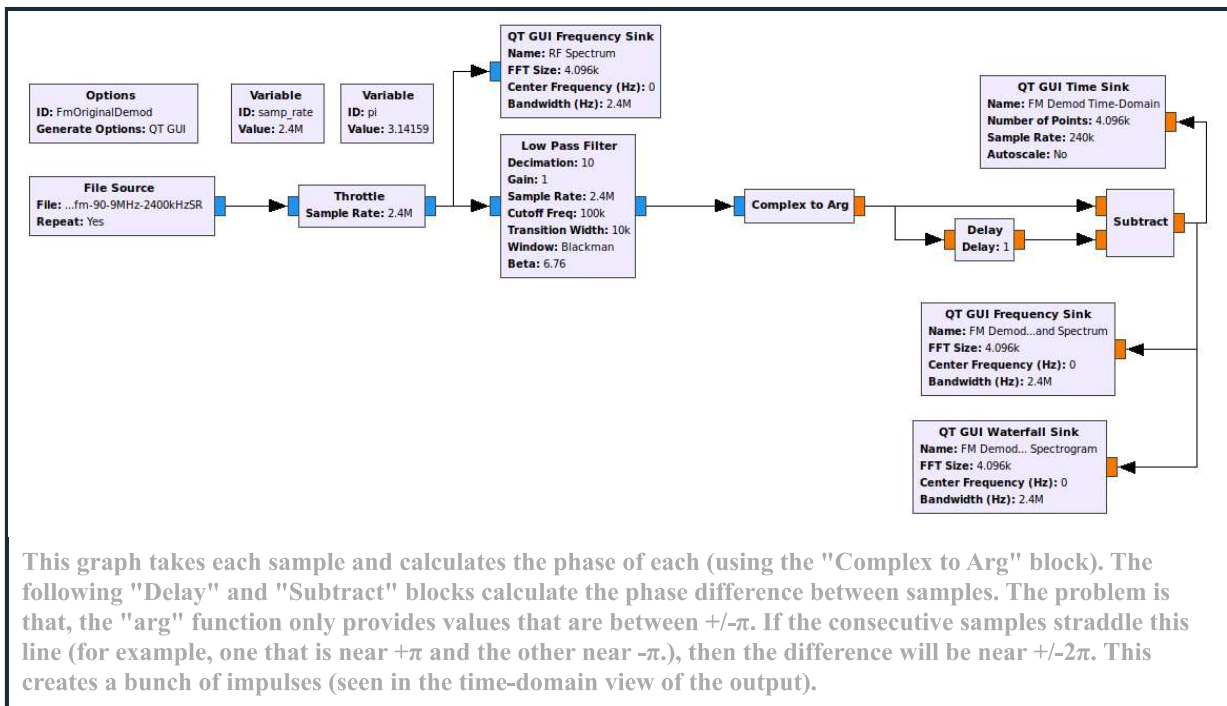


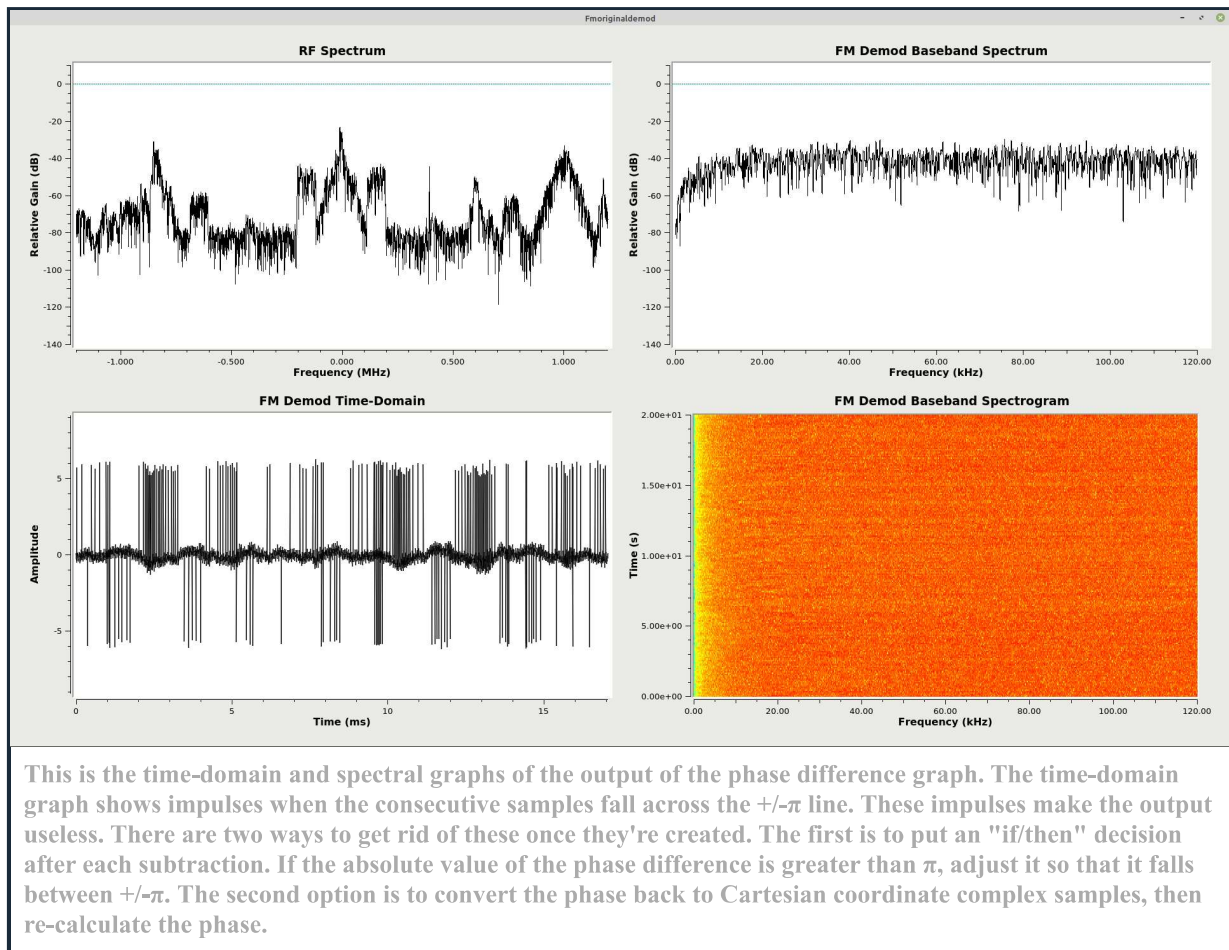
ZERO-CROSSING DISCRIMINATOR

Yet another of the techniques that I've not yet figured out how to mimick in GRC. Nuf said.

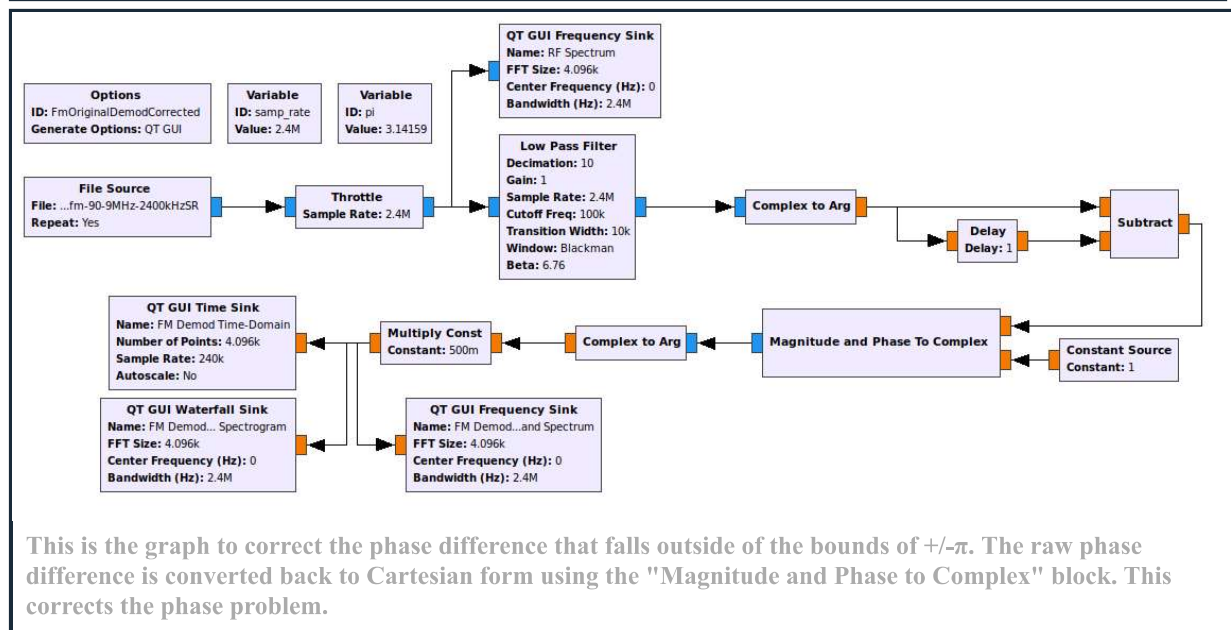
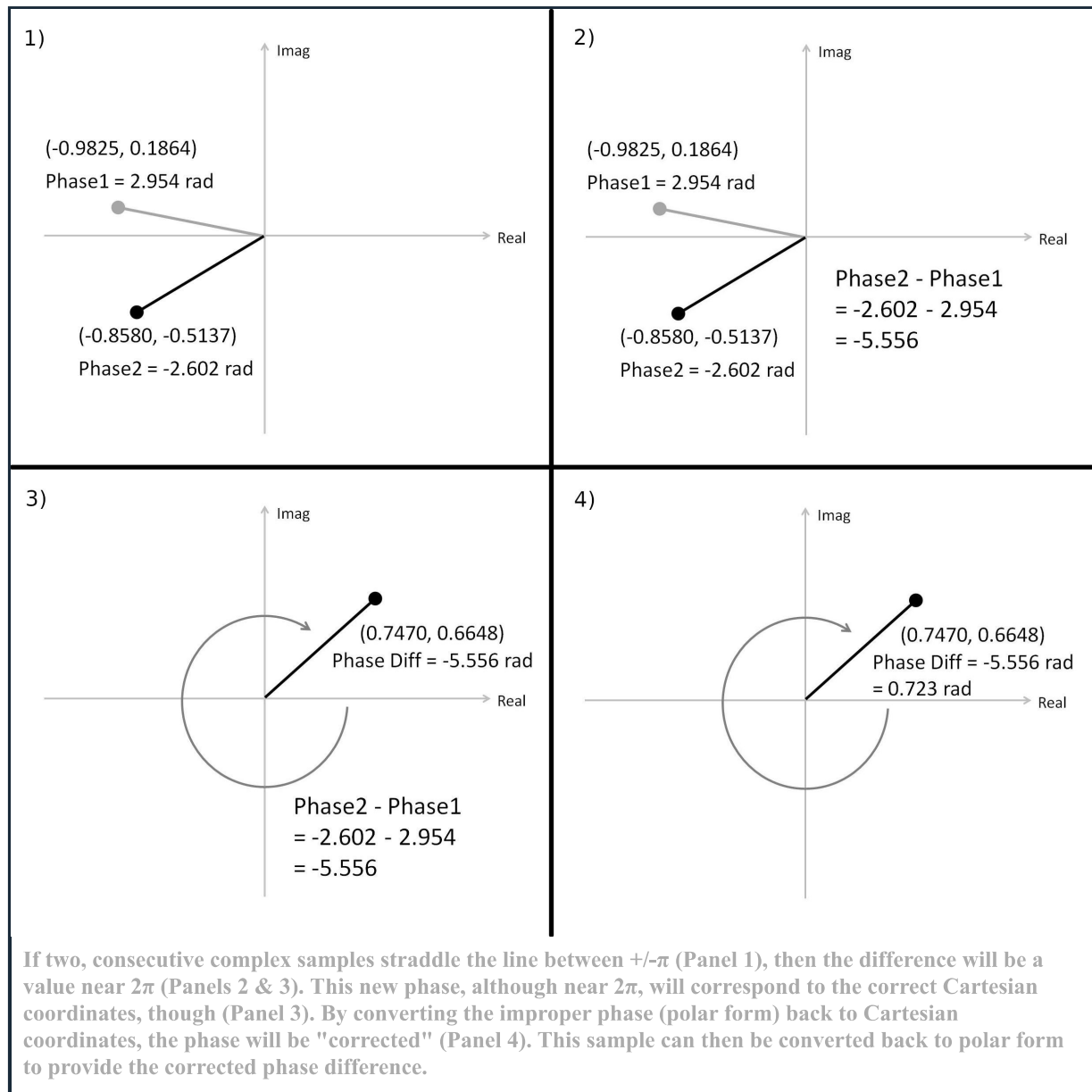
PHASE DIFFERENTIAL

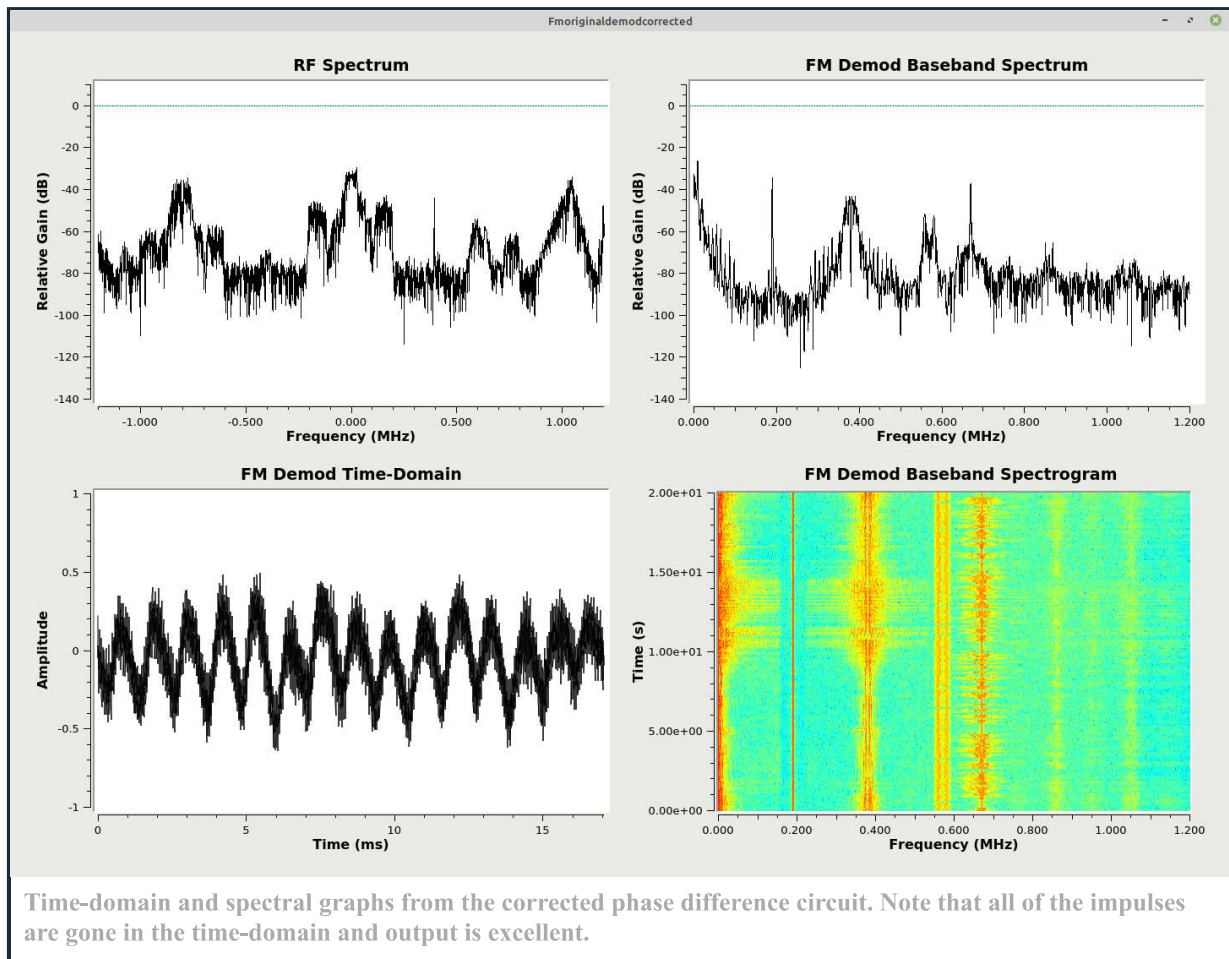
This goes back to the original idea of a frequency demodulator. Frequency is nothing more than the change of phase over time. The basic idea, then, is to measure the phase of the FM signal at each point, then subtract the phase between consecutive samples. Putting this together straight in GRC gives the following:





Calculating the raw phase difference between samples creates a problem, as seen above. To correct this requires one of two methods. The first is to use an "if/then" loop. GRC does not provide this ability (though it could be utilized in Gnu Octave or Matlab). The second method is to convert the phase difference (which is in polar form) back to Cartesian form. This corrects the phase difference. Changing the Cartesian back to polar form (using a second "Complex to Arg" block) provides the corrected phase difference. After scaling, this is the frequency demodulated baseband signal.

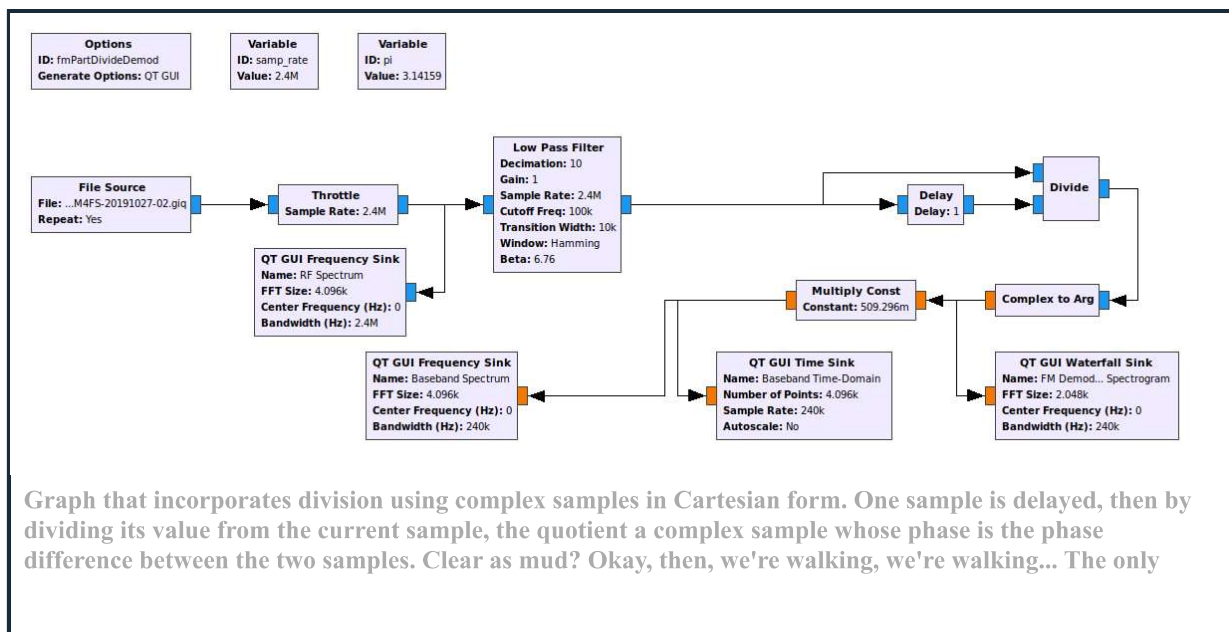




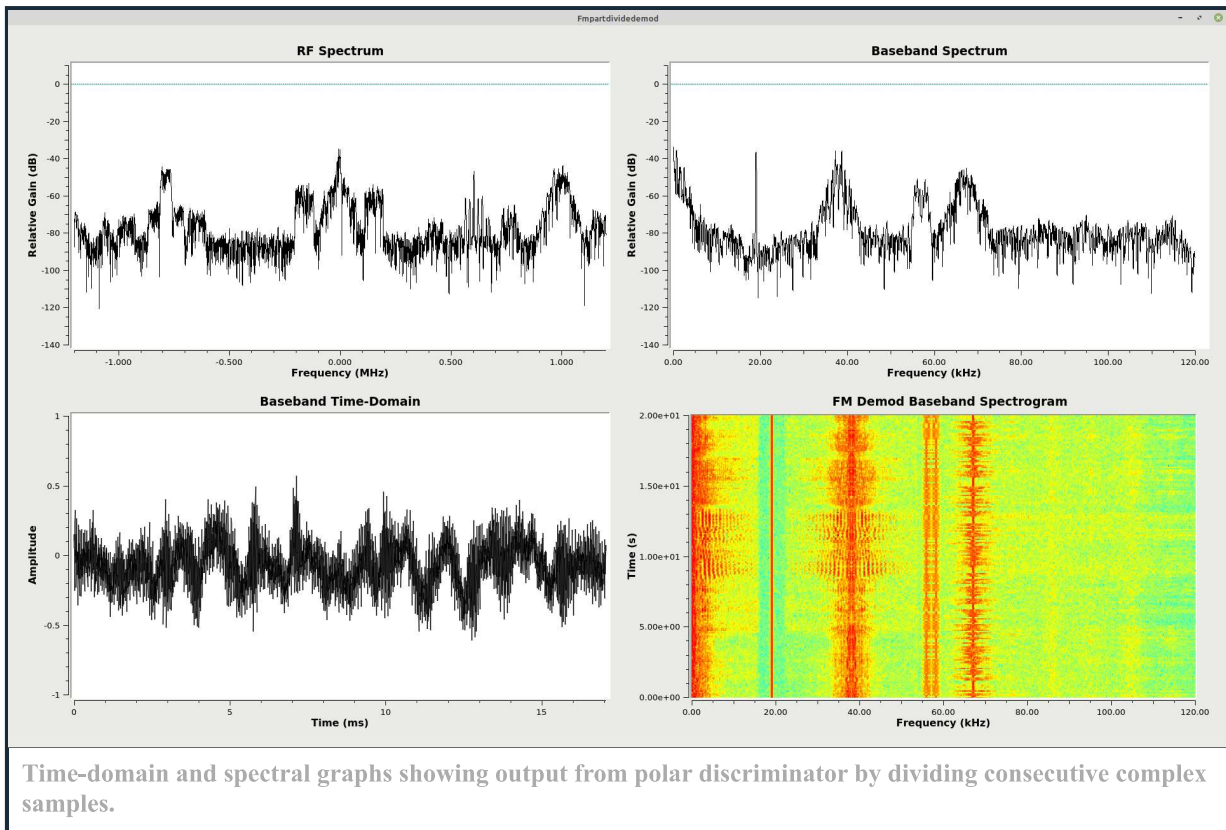
POLAR DISCRIMINATOR

Here's the thing about the phase differential with correction (shown above). It is *complicated*. There's one thing engineers really don't like is *complicated*. Why? Because that means more parts, which means more expensive items and more ways for things to go wrong. It turns out that there is a better way.

The idea here is this. Rather than calculate the phase difference in polar form, calculate the phase difference in Cartesian form. This can be done by dividing a complex sample by the one that came before it. This will be the same as calculating the phase difference. *Except since it is done in Cartesian, rather than polar, form, it will be the "corrected" phase before the actual phase itself is calculated.*



thing left to do is to calculate the actual phase itself ("Complex to Arg" block) and scaling ("Multiply Const" block).



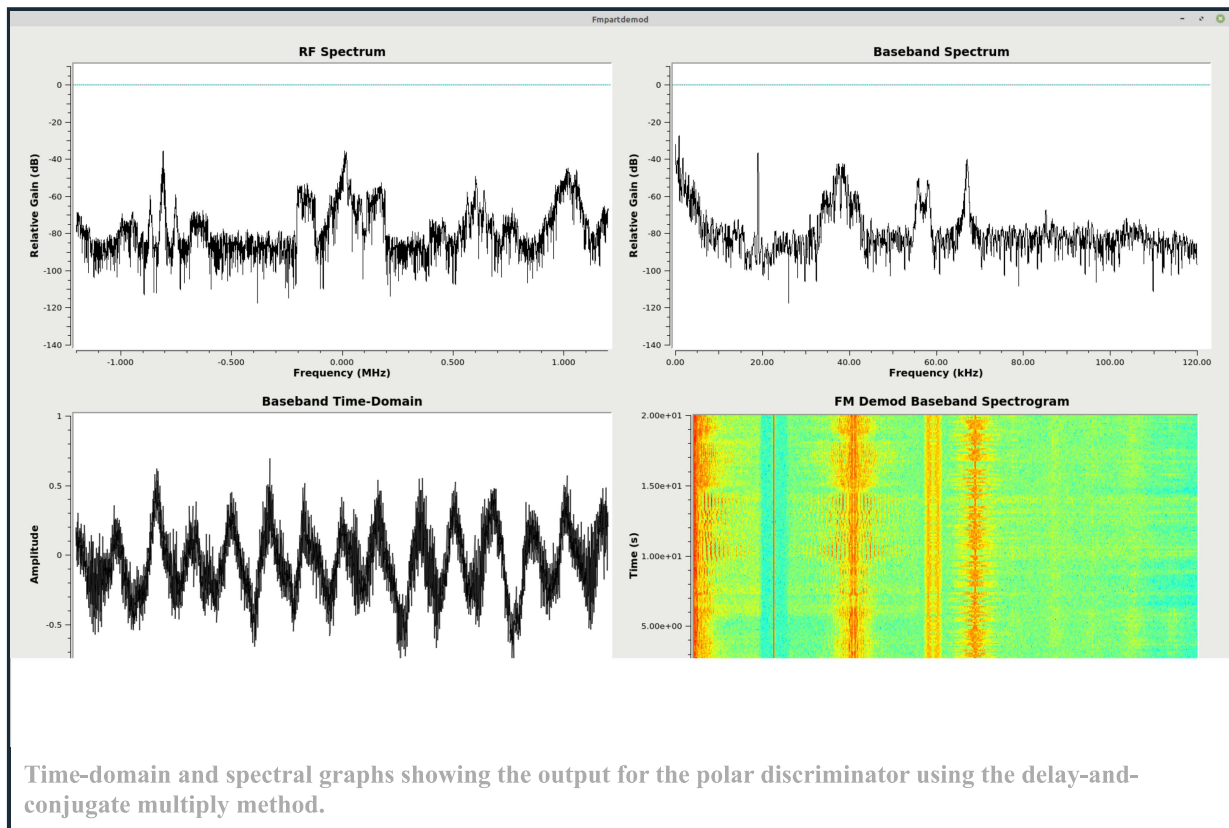
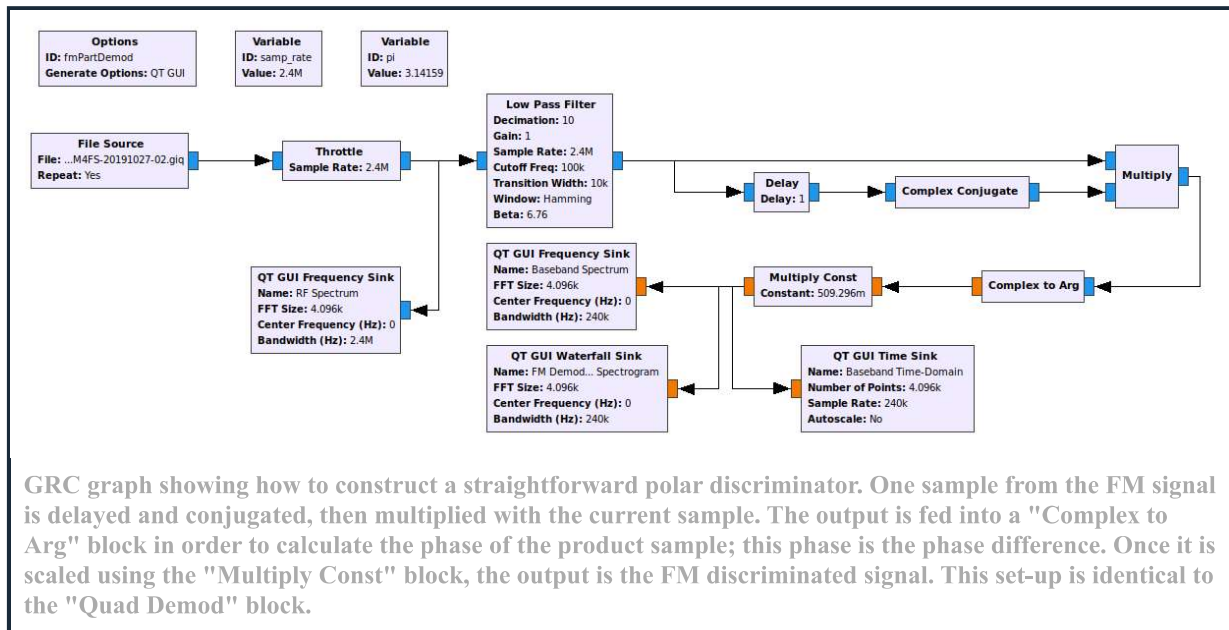
Except computers don't really do *division*. They do multiplication. Which begs the question, "How can you multiply in order to divide?" The most straightforward way is to multiply by the inverse of a number. Actual inverse operations really only work on floating point numbers. And that's what we're talking about here, anyway. GRC complex samples are actually 32-bit floating point numbers, both for the real and imaginary part. That means that each complex sample requires a total of 64 bits (8 bytes). But that's not the important thing. Floating point numbers exist as a mantissa (the decimal part) and exponent. Want to invert a number that has an exponent? Simple. Flip the sign on the exponent.

Except that's not what we're talking about here. We're talking about dividing one complex sample by another complex sample. The math is a bit different here. Skipping over all of the complexity, *if you want to divide one complex sample by another, conjugate the sample in the denominator*. That's it. Flip the sign of the imaginary part of the complex sample in the denominator, and VIOLA! It's essentially "inverted" for purposes of multiplication with another complex sample. **And that's how most systems do this.** They'll take a stream of complex samples, delay-and-conjugate it, then multiply it by the current sample. The delay means it becomes the previous sample (since we're interested in the phase *difference* between samples) and conjugate it (the "invert" part so that we can use multiplication and not division), then we multiply the two samples (delayed-and-conjugated times the current sample). The output is *another* complex sample, but this new sample has as its phase the phase difference between the current and previous sample. The multiplication is followed by the "Complex to Arg" block in order to calculate the actual phase difference, followed by a scaling ("Multiply Const" block) to create the final, frequency demodulated output.

Let me just say this straight. This is *brilliant*. This is what engineers kill for. A workable solution that is elegant and practical. Rather than calculate the phase, *then* calculate the phase difference, first calculate the phase difference, *then* calculate the phase. Again, just, plain brilliant.

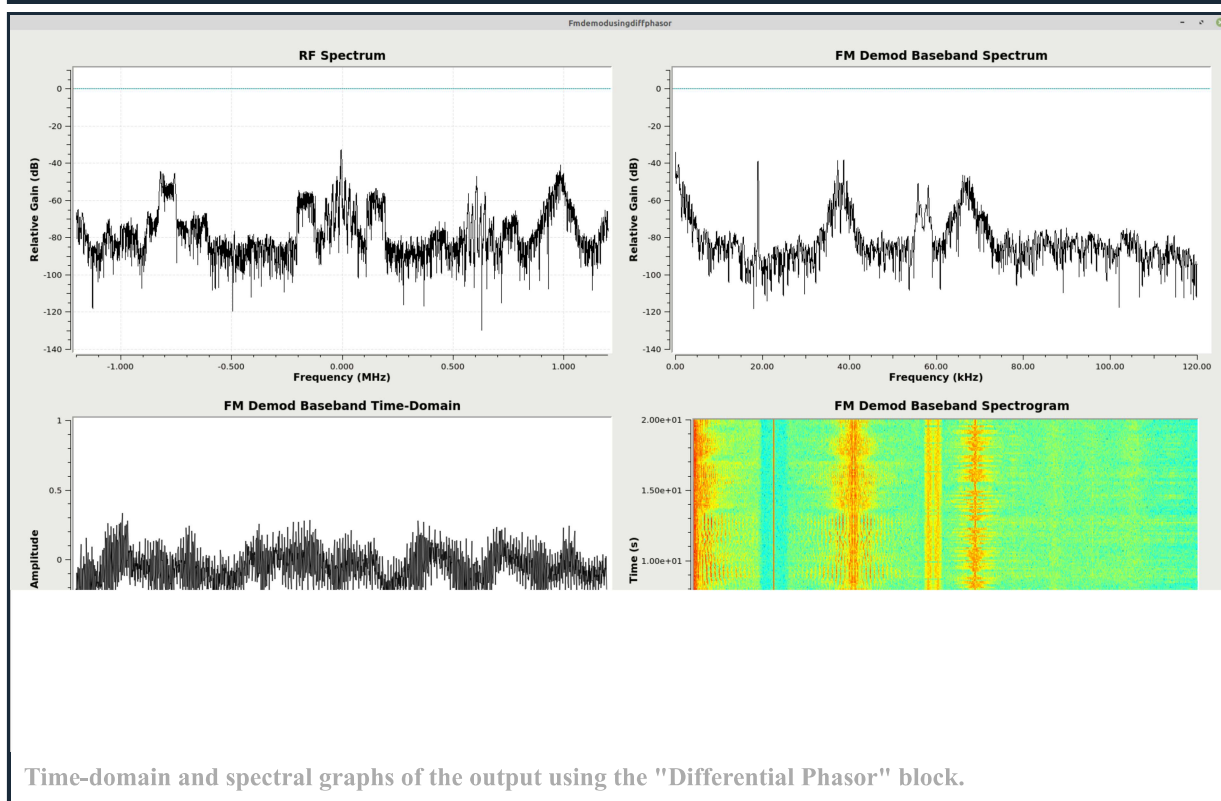
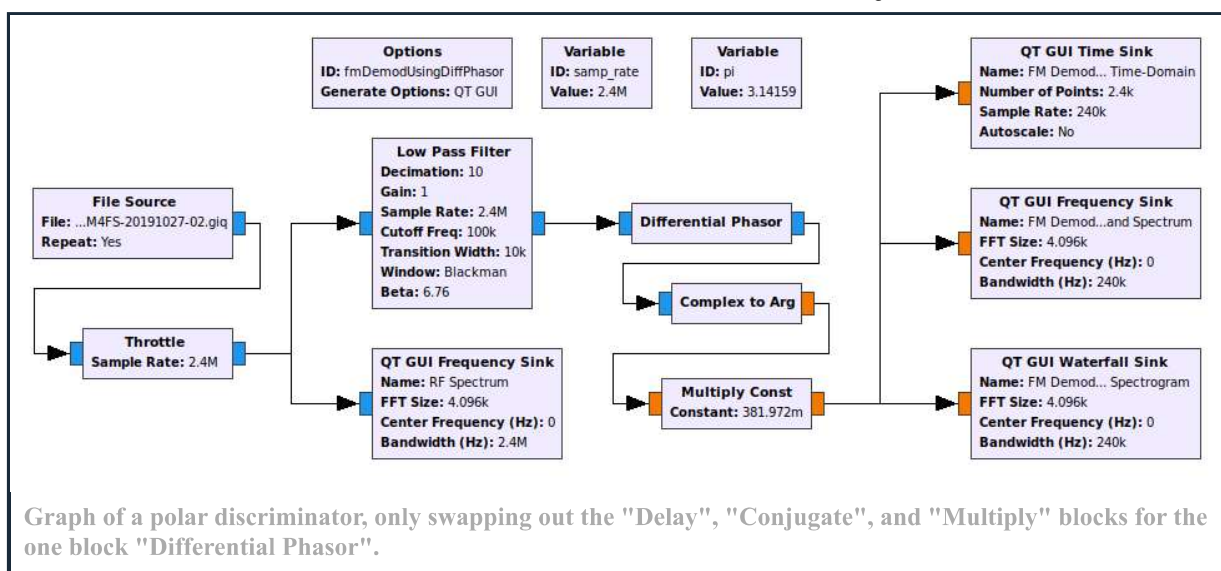
NOTE: This is the math that I got wrong in my previous post. Based on the *text* documentation of the "Quad Demod" block of GRC, I thought that the *current* sample was conjugated, and the *delayed* sample was not. It turns out *that still works*, but technically the output will be

amplitude inverted. The correct one, as explained in [Jim Shima's paper from 1995](#) (and he was not the first to figure this out), is to delay-and-conjugate *the same sample*, then multiply it with the current sample.



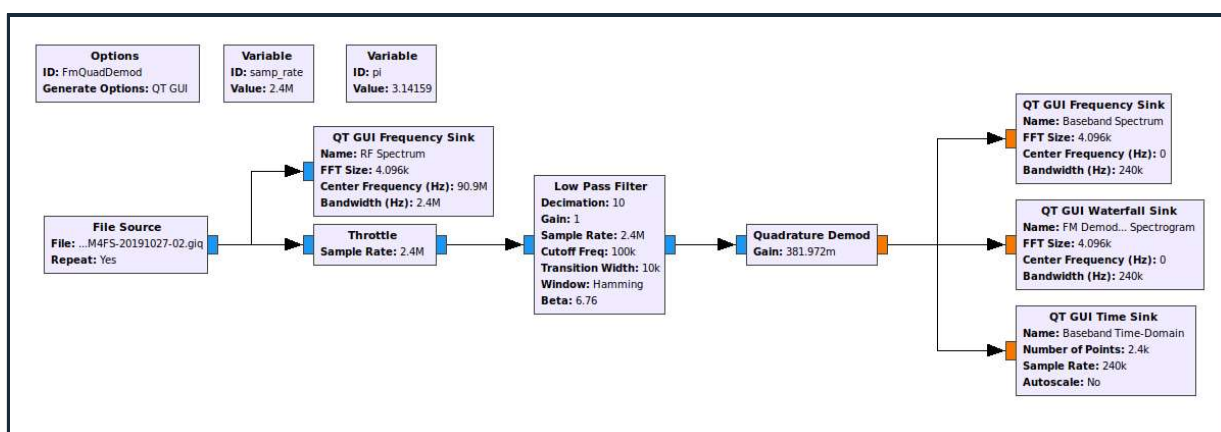
THE POLAR DISCRIMINATOR - SOMEWHAT SIMPLIFIED

It turns out that GRC provides the delay-and-conjugate multiply operation in one block. It's called the "Differential Phasor" block. This gives a graph as shown below.

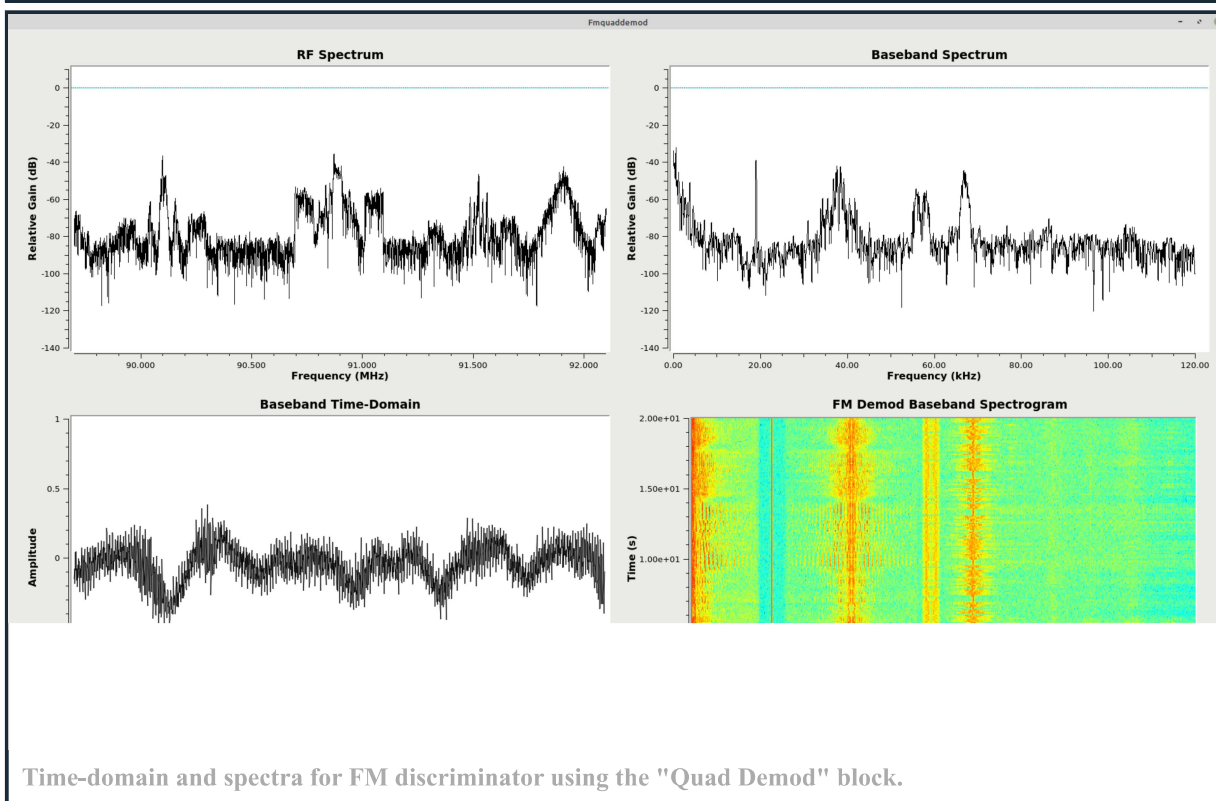


THE POLAR DISCRIMINATOR - COMPLETELY SIMPLIFIED

The graph for the polar discriminator I outlined above is, well, *complicated*. Turns out that GRC puts it altogether in one block, the "Quad Demod" block. That block combines the delay-and-conjugate multiply operation, the arctangent function (the "Complex to Arg" block), and the scaling (which in the "Quad Demod" block is called "Gain"), all in one block.



This is the graph using the "Quad Demod" block. This block combines the delay-and-conjugate operation, the multiply operation, the arctangent (aka "arg") operation and scaling operation (which is called the "Gain" in the "Quad Demod" block) all into one block.



One of the confusing aspects of the "Quad Demod" block when you first use it is, "What is this 'Gain' setting? How do I set it?" *This setting is nothing more than a scaling of the output.* FM, again, is the change of phase over time. The two, important parameters here are the sample rate and the number of radians in a circle (2π). If you want the output of the "Quad Demod" block to be "Hz", then you could calculate the "Gain" setting as follows:

$$Gain = \frac{SampleRate}{2\pi}$$

You may have noticed that I include a "Variable" block in many of my graphs with the value for "pi". That's why.

But what if you want to push the output of the "Quad Demod" block into an "Audio Sink" block? The above equation won't work. The amplitudes will be *waaaaaay* too high. The "Audio Sink" block wants samples that are between +/- 1. To make this possible, you need to divide the value above by the frequency deviation of the FM transmitter. For FM broadcast, this is 75 kHz. I add in a little wiggle room for safety purposes. **If you want to listen to audio with the "Quad Demod" block, the gain should be as follows:**

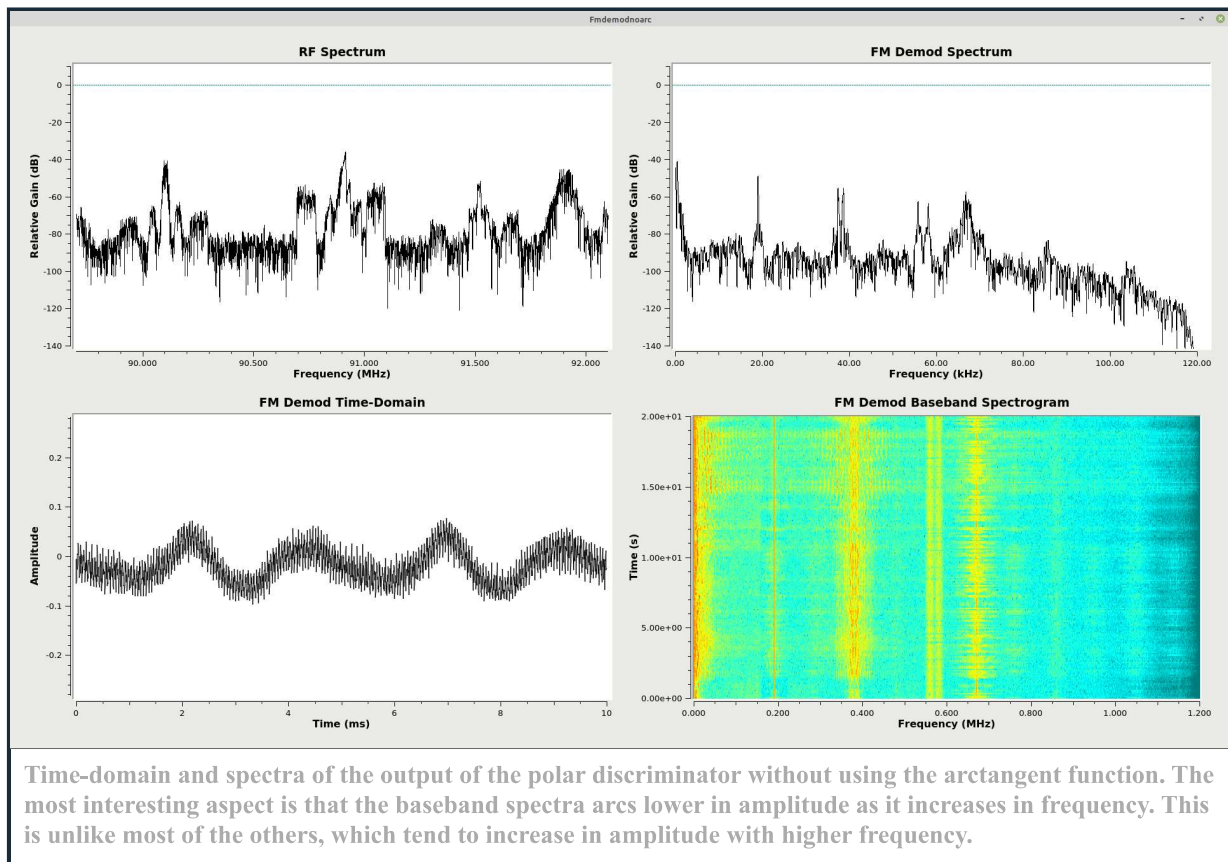
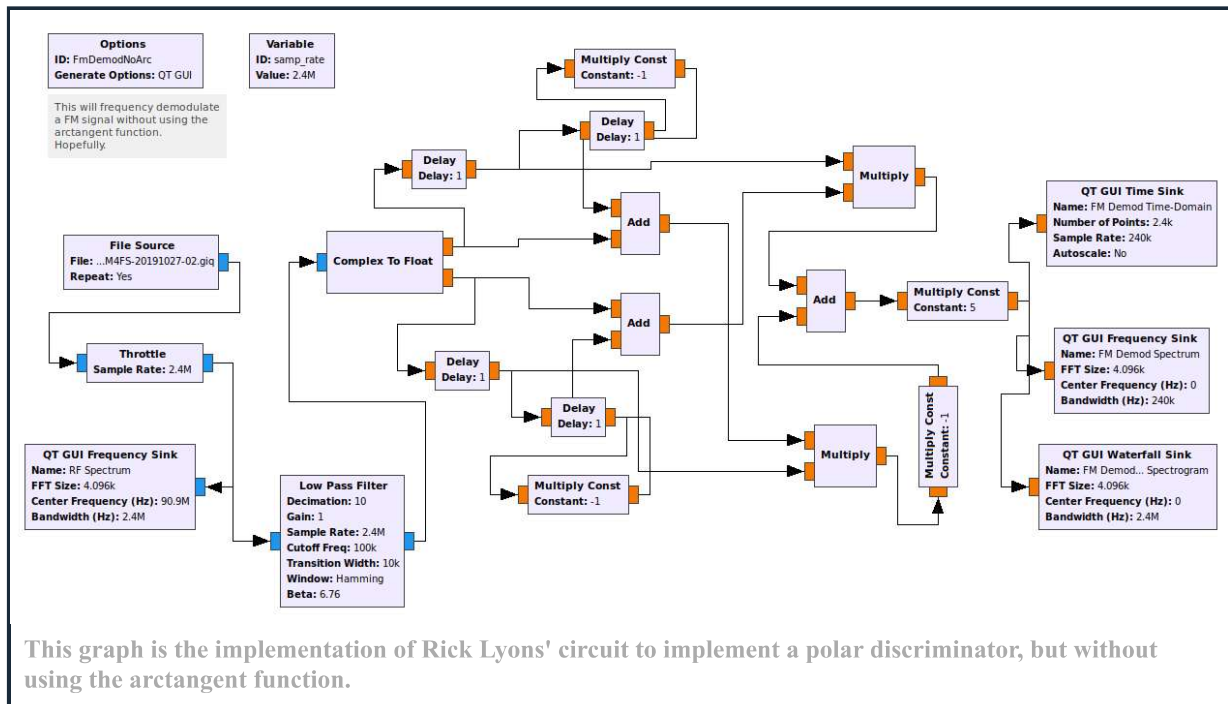
$$Gain = \frac{SampleRate}{2\pi \times 100e3}$$

The code for the "Gain" setting directly would appear as follows. NOTE: This assumes you have a "Variable" block for "pi", where its value is "3.14159".

$$\text{samp_rate}/(2*\text{pi}*100e3)$$

THE POLAR DISCRIMINATOR WITHOUT THE ARCTANGENT FUNCTION - TAKE #1

In my previous post, I went through Rick Lyons' method to create a polar discriminator without using the arctangent ("arg") function. It turns out that that function requires a bit of processing. Here it is, once again, in all of its glory.



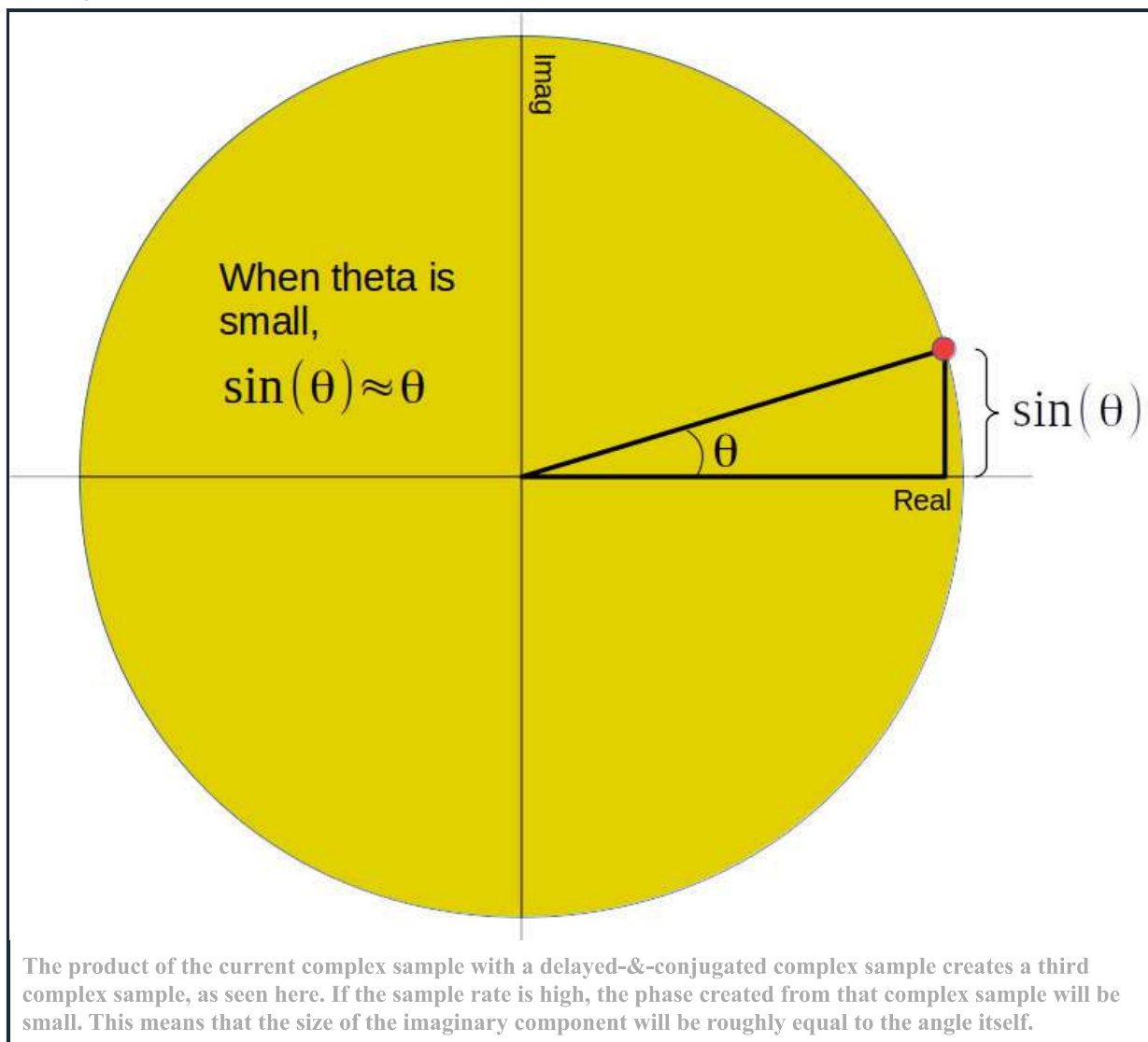
THE POLAR DISCRIMINATOR WITHOUT THE ARCTANGENT FUNCTION - TAKE #2

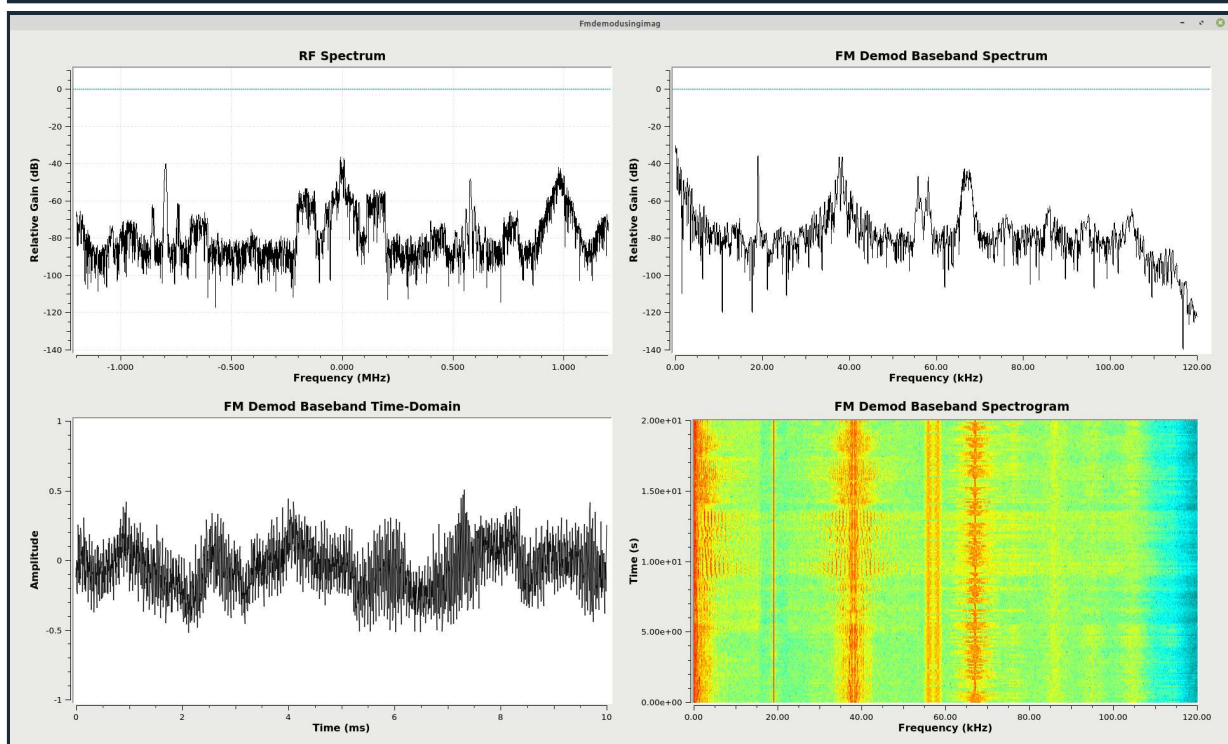
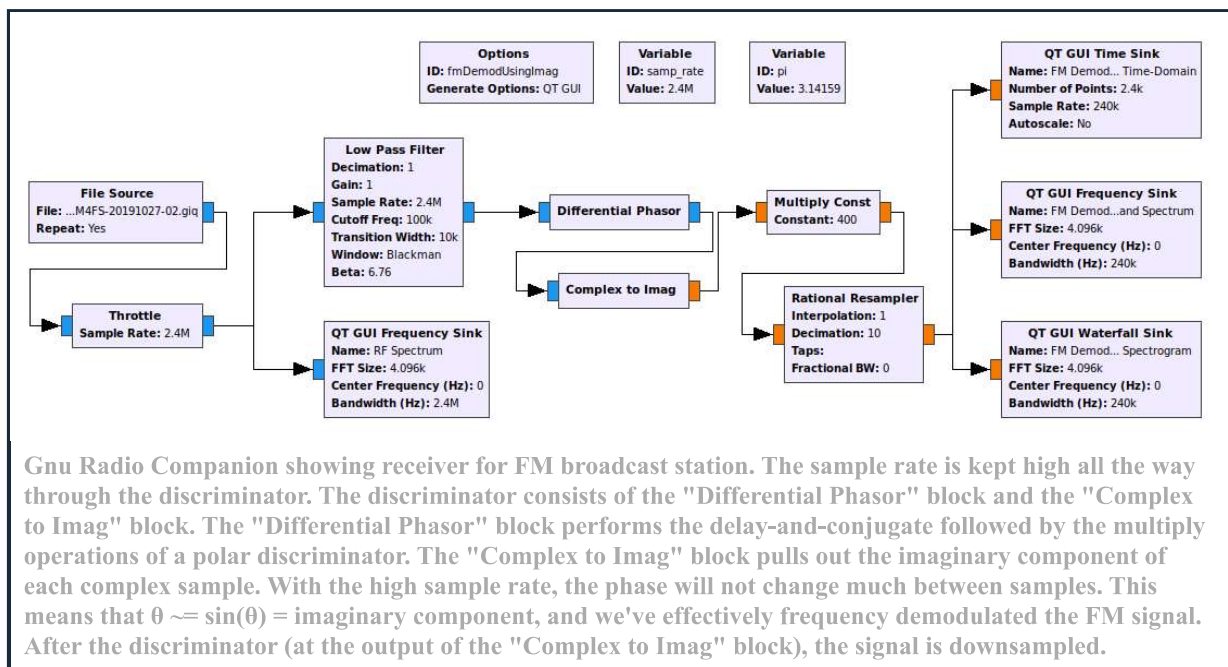
As I was pondering Rick Lyons' frequency demodulator without using the "arctangent" function, I had an epiphany. *There's yet one more way to frequency demodulate a signal*

without the arctangent function. The short answer is this: Run the polar discriminator at a much higher sample rate than Nyquist (typically 6 - 10 times), then simply use the imaginary component at the output of the multiplication of the delayed-&-conjugated sample with the current sample.

I'm not the first one to think of this, either. While looking for information on who developed the delay-&-conjugate method, I found [an entry in Google groups that mentions this oversampling method](#). That entry was posted in October of 1995. I'm at least 24 years late to the party. Typical. Of interest is that one of the people in [that Google groups discussion](#) was "Jim Shima", possibly the same one as the thesis discussed above.

With a high sample rate, the phase of the FM signal will not change much in between samples. The polar discriminator multiplies two complex samples to calculate the phase difference. The product is another complex sample. That final complex sample will have a phase, θ . It turns out that when θ is small, the value of the $\sin(\theta)$ will be roughly equivalent to θ . $\sin(\theta)$ is the same as the imaginary component in the polar plot. So with a relatively high sample rate, we can create a frequency discriminator by multiplying using the delay-&-conjugate method, then taking the imaginary component of the resulting complex sample. No need to perform the arctangent function.





The question becomes, "How high of a sample rate do you need for this to work?" Would you accept, "It depends"? The main thing it depends on is *how much error you're willing to tolerate in the output signal*. At angles up to roughly 30 degrees, the difference between θ and $\sin(\theta)$ is less than 5%. That's my limit. The Google groups discussion mentioned angles up to 45 degrees ($\pi/4$ radians). That much of an angle provides an error of 10%. If the sample rate is at the Nyquist limit, the maximum phase change between samples will be 180 degrees. To ensure that the maximum phase change is 30 degrees or less (my preference), I would have to sample at 6x over the Nyquist rate. The example I used was a FM broadcast station. It's a 200 kHz bandwidth and requires a 200 kHz complex sample rate. This means I would have to sample at 1.2 MHz minimum to ensure that I stay within these bounds.

Frankly, I do not consider this to be a viable technique. It's more of an interesting phenomenon (said in the way of Leonard Nimoy's Spock). However, the Google groups entry suggests at the end:

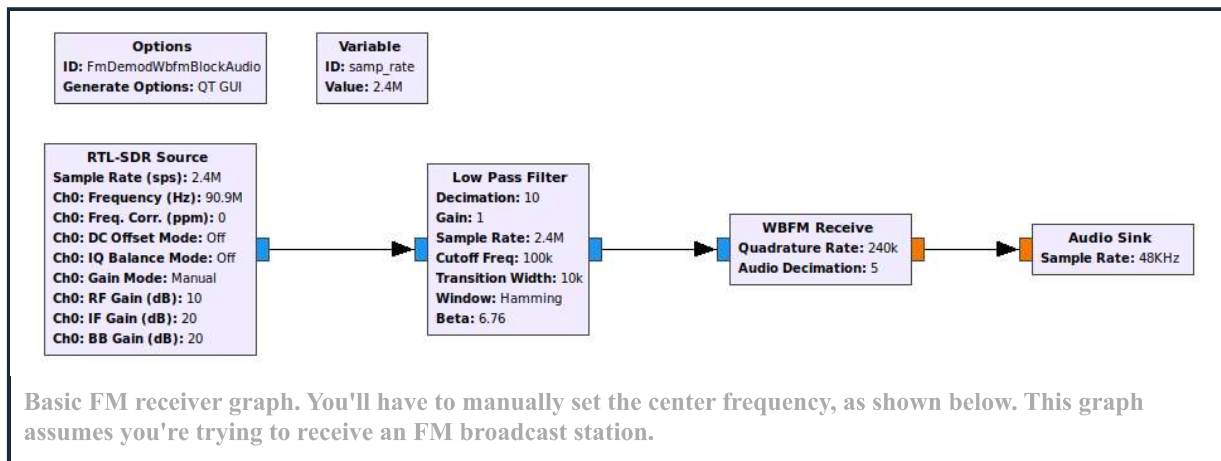
“ Now, not to muddy the water, but for audio FM I have heard of some subjective tests that found users preferring [sic] the compressed audio that results from using the imag approximation for big angles. Is this related to the solid state vs. tube amp debate?

What is the right answer? I'll let you decide.

"GIVE ME THE EASIEST FM RECEIVER!"

If you wound up on this page because some search engine said, "Hey, check out this web site if you want to make a FM receiver using Gnu Radio Companion!", well, **welcome!** Without further ado, here's the most straightforward FM receiver for an FM broadcast signal you can create. In my opinion. And here's [the GRC file, if you just want to download it.](#) You should be able to right-click on the link and select "Save link as..."

NOTE: This is based on using a RTL-SDR as the source. If you have a different front end (HackRF One, LimeSDR, USRP, whatever), you'll have to substitute the appropriate block for the "RTL-SDR Source" block in the graph below.



Properties: Options

General

Advanced

Documentation

ID	FmDemodWbfmBlockAudio
Title	
Author	
Description	
Canvas Size	
Generate Options	QT GUI
<u>Run</u>	Autostart
Max Number of Output	0
Realtime Scheduling	Off
QSS Theme	

OK

Cancel

Apply

"Options" block

Properties: Variable

General

Advanced

Documentation

ID	samp_rate
Value	2.4e6

OK

Cancel

Apply

"Variable" block used to set the overall sample rate. This is the sample rate at the output of the RTL-SDR (or whatever SDR you will use). Once it passes through the "Low Pass Filter" block, the sample rate will be 240 kHz. This is due to the decimation by 10 in the LPF ($2.4 \text{ MHz} / 10 = 240 \text{ kHz}$). Then, at the output of the "WBFM Receive" block, it will be 48 kHz due to the decimation by 5 ($240 \text{ kHz} / 5 = 48 \text{ kHz}$).

Properties: RTL-SDR Source

General

Advanced

Documentation

ID	rtlsdr_source_0		
Output Type	Complex float32 ▾		
Device Arguments			
Sync	don't sync ▾		
Num Mboards	1	▾	
Mb0: Clock Source	Default	▾	
Mb0: Time Source	Default	▾	
Num Channels	1	▾	
Sample Rate (sps)	samp_rate		
Ch0: Frequency (Hz)	90.9e6		
Ch0: Freq. Corr. (ppm)	0		
Ch0: DC Offset Mode	Off	▾	
Ch0: IQ Balance Mode	Off	▾	
Ch0: Gain Mode	Manual	▾	
Ch0: RF Gain (dB)	10		
Ch0: IF Gain (dB)	20		
Ch0: BB Gain (dB)	20		
Ch0: Antenna			
Ch0: Bandwidth (Hz)	0		

OK

Cancel

Apply

"RTL-SDR Source" block settings. If you're not using a RTL-SDR, you'll need to change this block and substitute whatever block is appropriate (HackRF, USRP, LimeSDR, whatever). The center frequency, labeled as "Ch0: Frequency (Hz)", is set to 90.9 MHz ("90.9e6"). Set this to whatever you want the center frequency to be.

Properties: Low Pass Filter

General

Advanced

Documentation

<u>I</u> D	low_pass_filter_0
<u>F</u> IR Type	Complex->Complex (Decimating) ▾
<u>D</u> ecimation	10
<u>G</u> ain	1
<u>S</u> ample Rate	samp_rate
<u>C</u> utoff Freq	100e3
<u>T</u> ransition Width	10e3
<u>W</u> indow	Hamming ▾
<u>B</u> eta	6.76

OK

Cancel

Apply

"Low Pass Filter" block. Note that the "Transition Width" setting drives how much work your processor has to do. The smaller this setting, the more work required. Therefore, if you're running this graph on a low-end processor and you have skipping, one thing you might try is to increase this setting. For example, you could try increasing it to 20 kHz ("20e3").

Properties: WBFM Receive

General

Advanced

Documentation

ID	analog_wfm_rcv_0
Quadrature Rate	samp_rate/10
Audio Decimation	5

OK

Cancel

Apply

"WBFM Receive" block. It asks for a confusing setting, "Quadrature Rate". That's just the sample rate at the input of the block. Because the input for this graph is 2.4 MHz, which is then decimated by 10 (in the "Low Pass Filter" block), the sample rate (aka "Quadrature Rate") is $2.4 \text{ MHz} / 10 = 240 \text{ kHz}$. Audio cards want specific sample rates. My audio card will take 48 kHz. That's why I selected 2.4 MHz. When I decimate it by 10, the output sample rate of 240 kHz is more than enough to handle the FM signal (bandwidth of 200 kHz). When I then decimate *that* by a factor of 5, $240 \text{ kHz} / 5 = 48 \text{ kHz}$.

Properties: Audio Sink
✕

General
Advanced
Documentation

ID

Sample Rate

Device Name

OK to Block

Num Inputs

audio_sink_0

48KHz ▼

Yes ▼

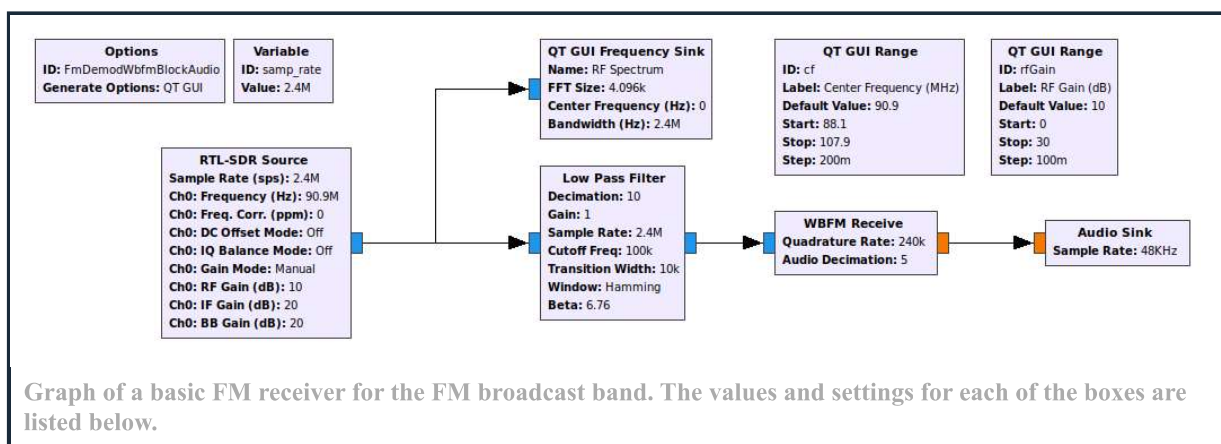
1

OK
Cancel
Apply

"Audio Sink" block. All you should have to do here is select "48 kHz" from the dropdown menu.

THE BASIC RECEIVER WITH A FEW EXTRAS

Here's the same receiver, but with an added frequency and gain adjustment widgets. This makes it easier to set the center frequency and gain while the graph is running. Again, [the GRC file if you just want to download it.](#)



Properties: Options

General

Advanced

Documentation

ID	FmDemodWbfmBlockAudio
Title	
Author	
Description	
Canvas Size	
Generate Options	QT GUI
<u>Run</u>	Autostart
Max Number of Output	0
Realtime Scheduling	Off
QSS Theme	

OK

Cancel

Apply

"Options" block

Properties: Variable

General

Advanced

Documentation

<u>ID</u>	samp_rate
<u>Value</u>	2.4e6

OK

Cancel

Apply

"Variable" block used to set the overall sample rate. This is the sample rate at the output of the RTL-SDR (or whatever SDR you will use). Once it passes through the "Low Pass Filter" block, the sample rate will be 240 kHz. This is due to the decimation by 10 in the LPF ($2.4 \text{ MHz} / 10 = 240 \text{ kHz}$). Then, at the output of the "WBFM Receive" block, it will be 48 kHz due to the decimation by 5 ($240 \text{ kHz} / 5 = 48 \text{ kHz}$).

Properties: RTL-SDR Source

General

Advanced

Documentation

ID	rtlsdr_source_0		
Output Type	Complex float32 ▾		
Device Arguments			
Sync	don't sync ▾		
Num Mboards	1	▾	
Mb0: Clock Source	Default	▾	
Mb0: Time Source	Default	▾	
Num Channels	1	▾	
Sample Rate (sps)	samp_rate		
Ch0: Frequency (Hz)	cf*1e6		
Ch0: Freq. Corr. (ppm)	0		
Ch0: DC Offset Mode	Off	▾	
Ch0: IQ Balance Mode	Off	▾	
Ch0: Gain Mode	Manual	▾	
Ch0: RF Gain (dB)	rfGain		
Ch0: IF Gain (dB)	20		
Ch0: BB Gain (dB)	20		
Ch0: Antenna			
Ch0: Bandwidth (Hz)	0		

OK

Cancel

Apply

"RTL-SDR Source" block settings. If you're not using a RTL-SDR, you'll need to change this block and substitute whatever block is appropriate (HackRF, USRP, LimeSDR, whatever).

Properties: QT GUI Frequency Sink

General

Trigger

Config

Advanced

Documentation

ID	qtgui_freq_sink_x_0
Type	Complex
Name	RF Spectrum
FFT Size	4096
Window Type	Blackman-harris
Center Frequency (Hz)	0
Bandwidth (Hz)	samp_rate
Grid	No
Autoscale	No
Average	None
Y min	-140
Y max	10
Y label	Relative Gain
Y units	dB
Number of Inputs	1
Update Period	0.10
GUI Hint	
Show Msg Ports	No

OK

Cancel

Apply

"QT GUI Frequency Sink" block settings. NOTE: This block is optional. If you want to make your graph even simpler (and just want to listen to the audio), leave it out. I like to have it as a "sanity check" that my graph is working properly.

Properties: Low Pass Filter

General

Advanced

Documentation

ID	low_pass_filter_0
FIR Type	Complex->Complex (Decimating) ▾
Decimation	10
Gain	1
Sample Rate	samp_rate
Cutoff Freq	100e3
Transition Width	10e3
Window	Hamming ▾
Beta	6.76

OK

Cancel

Apply

"Low Pass Filter" block. Note that the "Transition Width" setting drives how much work your processor has to do. The smaller this setting, the more work required. Therefore, if you're running this graph on a low-end processor and you have skipping, one thing you might try is to increase this setting. For example, you could try increasing it to 20 kHz ("20e3").

Properties: QT GUI Range

General

Advanced

Documentation

ID	cf
Label	Center Frequency (MHz)
Type	Float ▾
Default Value	90.9
Start	88.1
Stop	107.9
Step	0.2
Widget	Counter + Slider ▾
Minimum Length	200
GUI Hint	

OK

Cancel

Apply

"QT GUI Range" block used to set the center frequency of the receiver. This block is also optional. If you don't want to use it, put the frequency directly into the "Ch0: Frequency (Hz)" setting in the "RTL-SDR Source" block. For example, if your favorite station is 94.7, then directly enter that into the "Ch0: Frequency (Hz)" setting as "94.7e6". Also, these frequencies are for US stations (88 - 108 MHz, in 200 kHz increments, with the center at odd values starting at 0.1). If you're in a foreign country, you'll have to set this range and step size to whatever values work for you.

Properties: QT GUI Range

General

Advanced

Documentation

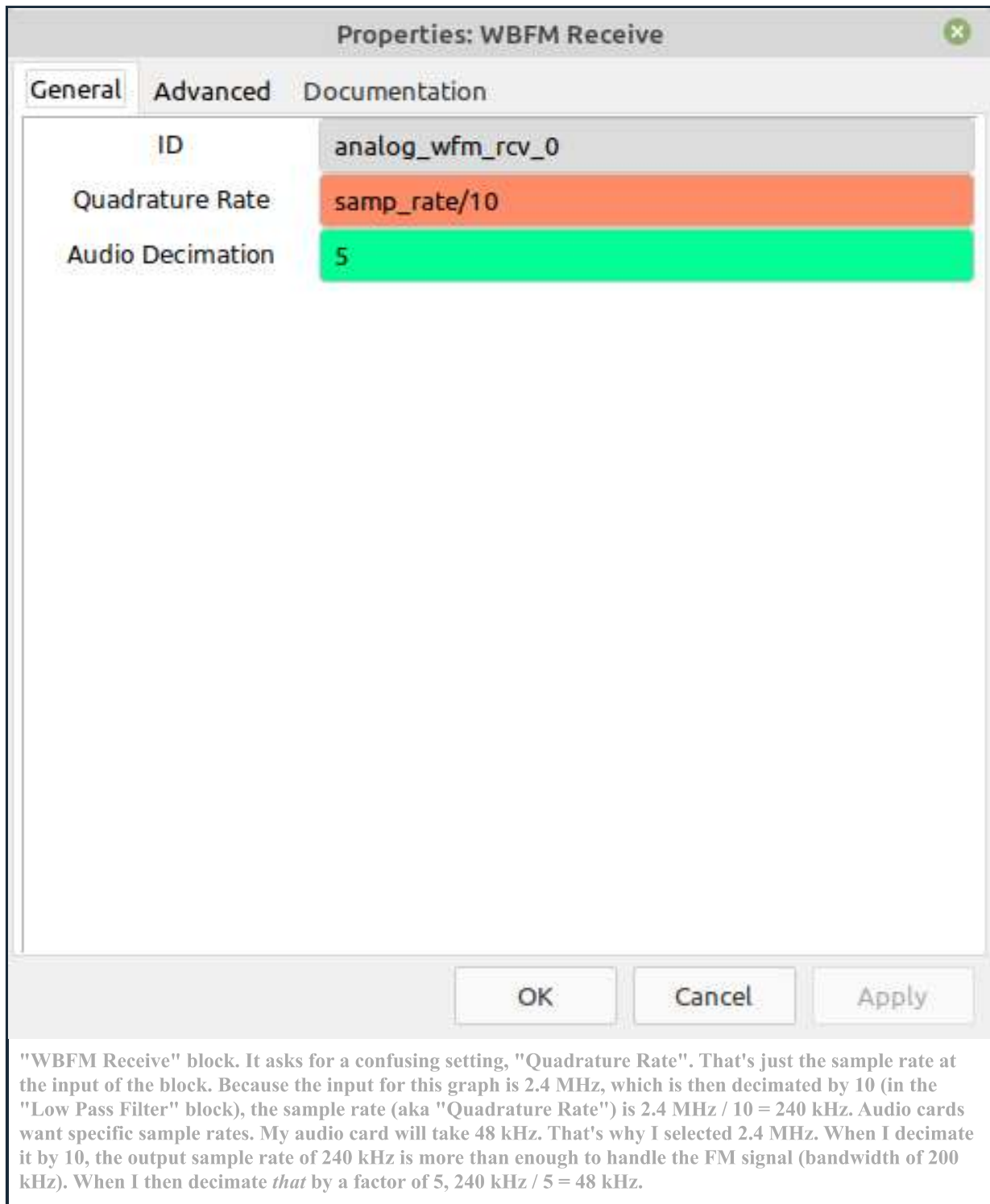
ID	rfGain
Label	RF Gain (dB)
Type	Float ▾
Default Value	10
Start	0
Stop	30
Step	0.1
Widget	Counter + Slider ▾
Minimum Length	200
GUI Hint	

OK

Cancel

Apply

"QT GUI Range" block used to set the RF gain of the RTL-SDR. The RTL-SDR only has one gain option, and that is the "RF Gain". The other, two options listed in the "RTL-SDR Source" block ("IF Gain" and "BB Gain") do not apply to the RTL-SDR, and changing those values has no affect on the RTL-SDR operation. This is another optional block. If you leave this out and do not change the default setting, most likely your RTL-SDR will work just fine. At my house, however, I'm roughly 1.5 km from a FM broadcast tower, with another about 3 km away. Between these two, they can overdrive my RTL-SDR if I do not lower the RF gain from its default setting. YMMV.



That's pretty much it. The graph above takes the input RF spectrum, filters out all but the desired signal (the one centered in the spectrum if you're using the "QT GUI Frequency Sink" block), FM discriminates it (aka "frequency demodulates it"), filters it some more, then sends these demodulated and filtered samples to the audio sink. If everything is working correctly, you'll hear the audio from the broadcast FM station.

Properties: Audio Sink

General

Advanced

Documentation

ID	audio_sink_0
Sample Rate	48KHz
Device Name	
OK to Block	Yes
Num Inputs	1

OK

Cancel

Apply

"Audio Sink" block. All you should have to do here is select "48 kHz" from the dropdown menu.