SDR-Based Spectrum Analyzer Based in Open-Source GNU Radio

Marcelo B. Perotoni[1], Kenedy M. G. dos Santos [2]
[1]*UFABC, Av. dos Estados 5001, Santo Andre, SP, Brazil, 09210-580, marcelo.perotoni@ufabc.edu.br*
[2]*IFBA Av. Amazonas 3150, Vitoria da Conquista, BA, Brazil, 45000-970 kenedymarconi@ifba.edu.br*

***Abstract*— A low-cost spectrum analyzer is presented, based on a commercial software defined radio and an open-source application package. Fundamentals regarding the receiving operation and its sensitivity are presented, along with measurements, of two 8 and 12-bits software defined radio models. An application, run within GNU Radio, is presented, deployed to overcome the hardware analog-to-digital converter limitation to monitor wide bandwidths. Results are shown for two different frequency ranges, 200 MHz bandwidth.**

Index Terms—GNU Radio, Signal Processing, Software-Defined Radio, Spectrum monitoring.

I. Introduction

The concept of software defined radio (SDR) was first proposed in 1991, according to the lines of a “radio whose channel modulation waveforms are defined in software” [1], though the exact definition is blurred, since no exact boundary delimits the degree of hardware reconfigurability needed to label a platform as a true-SDR [2]. The SDR concept paved the way for Cognitive Radios, where the electromagnetic environment is continuously sensed, in order to find empty slots and operate accordingly, dynamically varying the modulation and occupied frequencies. That enables a more efficient and lean use of the spectrum [3], [4]. Spectrum monitoring is, therefore, a preliminary core function for enabling a Cognitive Radio.

Besides Cognitive Radio, the electromagnetic spectrum is a very important and decisive asset not only for communications but also in homeland security and defense, in the area of electronic warfare [5], where adversary communication channels should be identified, monitored and, if possible, demodulated. In [6], a 2.3 GHz to 2.7 GHz spectrum sensing system, intended to be used in Cognitive Radio, was built using a 12-bits USRP SDR, whose samples were saved and later analyzed within Matlab. Another setup, this time with a low-cost dongle 8-bits RTL-SDR, connected to a Raspberry Pi captured, demodulated and displayed broadcast AM and FM signals, with the managing algorithm written in Python [7]. Also, using a Raspberry Pi, a system was designed to monitor unused (so-called “white”) portions of the UHF spectrum, with the RF front end performed by a low-cost RF Explorer spectrum analyzer [8]. SDRs do not necessarily need a computer to operate, an RTL unit was connected to a Raspberry Pi2, running on Raspbian operational system, which sampled and demodulated broadcast AM and FM signals [9]. A channelized receiver, for incoming radar carriers, was implemented in GNU Radio, using the RTL SDR for early warning receivers, with band pass filters selecting different sub-bands covering a 2.4 MHz bandwidth around 940 MHz [5]. An AM-band spectrum analyzer, based on a Si570 VCO (voltage-controlled oscillator) and managed by Labview, had the digital samples made available online by a Webserver, to be used in educational purposes [10]. Another RTL-SDR was used as a spectrum analyzer, connected this time to an Android mobile phone running a Java application, both units connected by a USB cable [11]. Another application, using USRP and GNU Radio, monitors the band around 2.5 GHz to find empty slots for Cognitive Radio, using an energy detection algorithm [12]. Broadcast FM and digital TV were monitored and demodulated using both RTL and USRP SDRs, whose data are sent by TCP/IP by a Raspberry Pi to be remotely analyzed [13]. A system called SwepSense was devised to monitor a large electromagnetic spectrum bandwidth divided in the sub-bands 50 MHz-2.2 GHz; 400 MHz- 4.4 GHz and 1.2 GHz-6GHz, with high temporal resolution, based on an USRP whose local-oscillator chip was connected to a saw-tooth generator circuit [14]. In order to cover two distant sub-bands, two RTL units were used to operate as a low-cost spectrum monitoring system, each one covering a specific band, controlled within the GNU Radio environment [15].

This article describes an 8-bit HackRF One SDR used as a spectrum analyzer, controlled by the open-source GNU Radio environment. In contrast to [15], a Python block inside GNU Radio enables the automatic monitoring of a wide band frequency range with only one SDR, larger than its 20 MHz bandwidth limit. Another approach to cover wide frequency ranges employed parallel-processing, with several host computers connected to an USRP [16], whereas this study is based on a single host PC, running either Windows or Linux Operational Systems. A minimum sensitivity measurement of two SDRs is shown, as to investigate their performance with faint signals. A discussion regarding the power measurement are presented in the appendix, as to show the trade-off between precision and elapsed time in regard to the used filters.

II. SDR Fundamentals

After the first commercially available SDR unit, Ettus USRP, was introduced around 2004, many other different radios were made available, with varying capabilities and prices. One of the most popular option due to its low profile and cost, the 8-bit RTL, was initially based on the direct frequency conversion topology, whereas others (like USRP and HackRF One) have an IF (Intermediate Frequency) which is converted to the digital domain. The RTL family is a receiving-only SDR, whereas HackRF One and USRP are capable of either transmitting and receiving, though not at the same time for the former. Radio amateurs, in particular, used variations of SDRs whose IF's are within the audio band, therefore the PC audio input can be used to input the down-converted signal to a computer, being further discretized and processed, enabling the operation as full-fledged transceivers [17].

In regard to the HackRF One SDR, Fig. 1 shows a block diagram of its receiving branch. The RF input signal from the antenna (SMA connector) can be amplified by a broadband LNA (Low Noise Amplifier, 14 dB gain, MGA-81563), switchable by the user, where the signal can be bypassed through the active device. It can be filtered, by an HPF (High-Pass Filter) or LPF (Low-Pass Filter), depending upon the frequency range chosen by the user. Its quadrature mixer delivers two components, the so-called in-phase (I) and quadrature (Q). The Local Oscillator (LO) signal, upon acting on a non-linear component, brings the high frequency input band to the IF range.

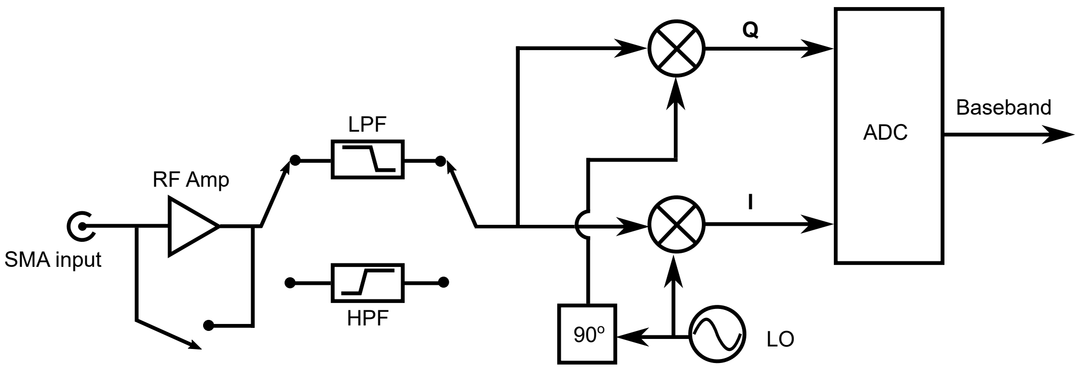


Fig. 1. HackRF One receiving side block diagram.

The PLL (phase locked loop)-stabilized mixer with the VCO (voltage controlled oscillator), RFFC5072 chipset, translates the incoming frequency energy to an IF between 2.3 GHz and 2.7 GHz, later digitized by the 8-bits ADC (analog-to-digital converter). HackRF One, in particular, has an ADC whose band covers up to 22 MHz at once, MAX 5864. The maximum SDR is set though to a 20 MHz bandwidth, whose stream is then sent to a 32-bit ARM Cortex processor, LPC43XX, transferred later to the USB channel. RTL units, on the other hand, have their IFs in the range of 3.57 MHz or 4.57 MHz (case of R802 tuner) or even zero-IF (the defunct E4000 tuner). The choice of a low IF provides better selectivity whereas higher IFs result in lower mixer image responses, therefore there is a trade-off between selectivity and image response [18].

It is important to stress that time-domain IQ samples are transferred to the USB channel, and in cases of USRP, Ethernet input. FFT (Fast Fourier Transform) has to be performed in the PC by the respective application in order to visualize frequency domain information. Nyquist criterion states that in order to recover the original signal without aliasing losses, the sampling frequency has to be at least twice the bandwidth of the original signal. Since the samples are complex and not real, the sampling frequency can be equal to the signal maximum bandwidth, which compensates for the extra complexity of two mixers instead of a single one, for the case of a purely real frequency conversion [19]. Quadrature mixing also provides phase information to the recovered signals, enabling a coherent processing, which is important for radar and imaging applications, to name a few. This phase information is not usually available for typical spectrum analyzers, since they do not perform the quadrature frequency conversion. Complex sampling produces twice as many samples as the purely real case, more demanding in terms of data transfer throughput. Fig. 2 illustrates the process of recovering the original positive frequency block by summing up the real part of the spectrum (I) with the quadrature (Q) part multiplied by $-j$, according to:

$$x_{IQ}[n] = x_I[n] - jx_Q[n]. \tag{1}$$

where x_I and x_Q are the nth in-phase and quadrature samples respectively and x_{IQ} is the final (real) result. Fig. 2, for simplicity, the zero-IF case is shown.

III. SDR Sensitivity

The SDR RF front-end should be able to detect small signals as well as avoid intermodulation products caused by high-amplitude carriers, present within its reception band. The amplitude difference

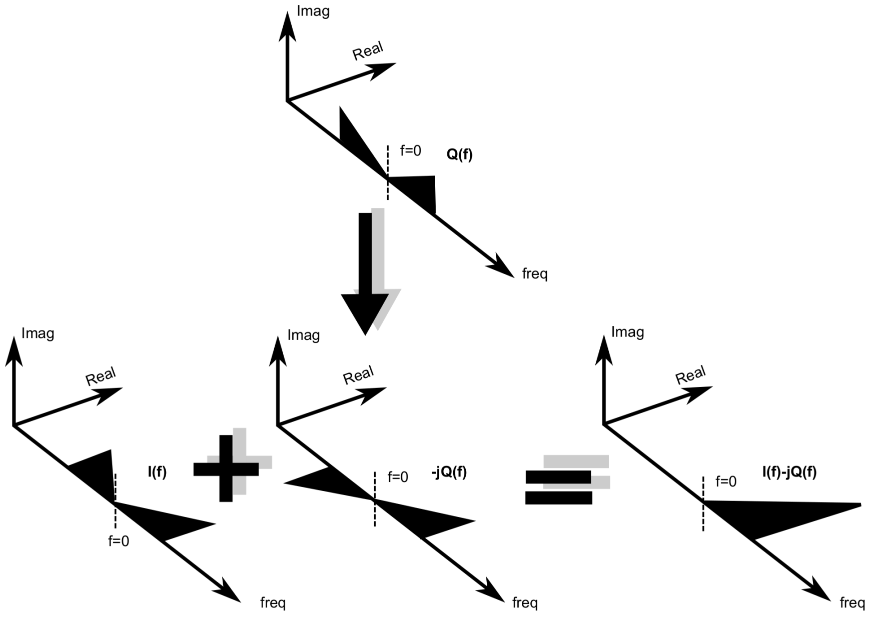


Fig. 2. Process of IQ sampling, diagram adapted from [20]. In the figure, the zero-frequency or DC is shown in detail as the f=0 label.

between the faintest and largest signal amplitudes configures the SDR dynamic range, which may be the principal challenge of a receiver design [2]. The sensitivity can be defined as the weakest signal that a receiver can still detect and it is determined by the noise figure parameters of the different circuits along the receiving chain. Much of the SDR RF performance depends on the ADC [2]. For the HackRF One and RTL units, they have 8-bits converters, whereas USRP and MSI-SDR have 12-bits. The choice of a converter with a larger number of bits N has to take into account the trade-off of its bandwidth and power consumption [3]. The signal-to-noise ratio SNR_{ADC} in decibels, of an N-bits analog-to-digital converter can be written as [3]:

$$SNR_{ADC} = [6.02N + 1.76]\,\mathrm{dB}. \tag{2}$$

for the case where the ADC is excited by a full-scale sinusoidal signal with a given frequency (i.e. the parameter is frequency-senstive), the quantization noise uniformly distributed between $\pm\frac{1}{2}$ LSB (least-significant bit). For an 8-bit ADC results in approximately 50 dB. The effective number of bits ($ENOB$) parameter, which accounts for the noise and intermodulation products generated by the ADC itself, is smaller than the actual number of bits N, though it can be artificially improved with processing techniques and oversampling. ADC own noise is a critical parameter in the converter design and its source can be ascribed to two main causes, thermal and quantization. The noise-floor (in dBm) NFl can be written as [2]:

$$NFl = 10\log(kT_eBW) + NF_{total}. \tag{3}$$

where k is the Boltzamnn constant (-228 dBW/K.Hz); T_e is the temperature in Kelvin and BW the bandwidth. NF_{total} is the total Noise Figure (units in dB) of the receiving chain, which in accordance with Friis Law depends mostly on the first element of the RF front-end chain, usually the LNA. The overall sensitivity of a SDR is, however, not univocally defined, depending on several parameters such as the frequency band, presence of amplifiers, filtering and averaging acting on the samples, etc.

The MDS ,Minimum Discernible Signal, can be found after:

$$MDS = 10\log\frac{kT}{1\text{mW}} + SNR + \log BW. \tag{4}$$

The first term can be written as -174 dBm for ambient temperature. The MDS can measured when $\text{signal}/(\text{signal} + \text{noise}) = 3\,\text{dB}$, i.e. the carrier peak in consideration is 3 dB above the noise floor. Once the MDS is known the overall system SNR can be estimated. Averaging can indeed increase the overall SNR due to the incoming noise uncorrelated nature. Oversampling and averaging increase the SNR as long as there is a white noise model characteristic, i.e. uniform power spectral density over the frequency band of interest. For the case where there are n realizations of a vector, in the best case if the signal is kept unmodified for all realizations, the SNR_{avg} can reach the maximum value of:

$$SNR_{avg} = \frac{SNR_i}{n}. \tag{5}$$

where the SNR_i is the signal-to-noise ratio of a single-realization. The inclusion of an average routine along the way implies a delay, which depending on the case turn the data processing unfeasible. Oversampling, where the sampling frequency is larger than the Nyquist limit, is a technique also used to improve on the SNR. Since the total power due to quantization error remains the same for a larger bandwidth associated to higher sampling rate, its density decreases, which can be cut out by means of a low pass filter [2]. That means smaller bandwidths do not necessarily imply lower noise amplitudes, like in resistors. There is an optimum BW that minimizes the noise in case of ADCs, which can be found after measurements involving several influencing parameters. The OSR parameter (oversampling rate), in particular, is related to the Nyquist limit by:

$$OSR = \frac{F_s}{2BW}. \tag{6}$$

where F_s is the sampling frequency. This particular equation applies to real values, in case of complex series the 2 factor is eliminated. The SNR due to the quantization noise improves approximately 3 dB every time F_s doubles. Decimation can be used to further reduce the quantization noise, where samples are periodically discarded [21].

For the sake of comparison, HackRF One (8 bits) is compared with a 12 bits SDR (MSI.SDR, an RSP1 clone). Theoretically, the 8-bits results in 50 dB of SNR whereas the 12-bit provides 74 dB, a 24 dB difference. Measured data for both SDRs were acquired according to Fig. 3 where a 3 GHz maximum frequency RF generator was directly connected to each one of the SDRs. A GNU Radio application displays the received power, in a PC connected to the radios. The P_o peak power is then read, considered to be 10 dB above the noise floor, without any form of filtering, decimation or averaging, configuring the worst-case scenario, taking into account only the hardware-limit. Internal RF and IF amplifiers are also switched off or kept at the minimum possible gain amplitudes. The goal is to find the specific noise floor of each SDR, parameterized with different settings of their bandwidths. After P_o is read on the GNU Radio application, the power level is read on the generator output, since GNU Radio amplitude readout is not calibrated.

Fig. 4 shows the results parameterized for different bandwidth settings. Since the MSI SDR has a

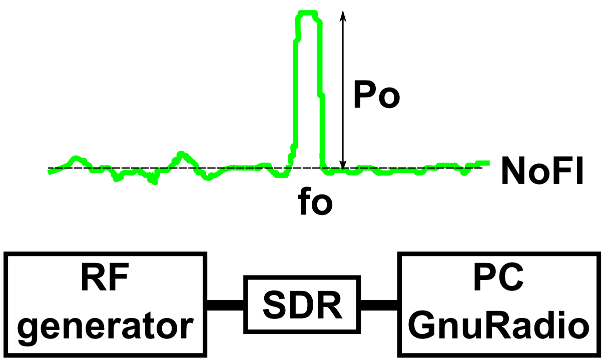


Fig. 3. Setup of the SDR Noise Floor ($NoFl$) measurement.

lower frequency limit of 10 kHz, whereas HackRF One starts at 10 MHz, the smallest allowed MSI bandwidth of 500 kHz was used. The displayed red curves contain both SDRs set to 10 MHz bandwidth, and the 4 bits of difference results in around 30 dB increase of sensitivity for the MSI, taken at 900 MHz - near the theoretical 24 dB difference. The HackRF One has its 10 MHz bandwidth resulting in the highest sensitivity, though with only 3 dB difference from its maximum 20 MHz sampling rate. The cheaper 12-bit MSI SDR also had a large sensitivity variation across the frequency band, with best results around 1 GHz.

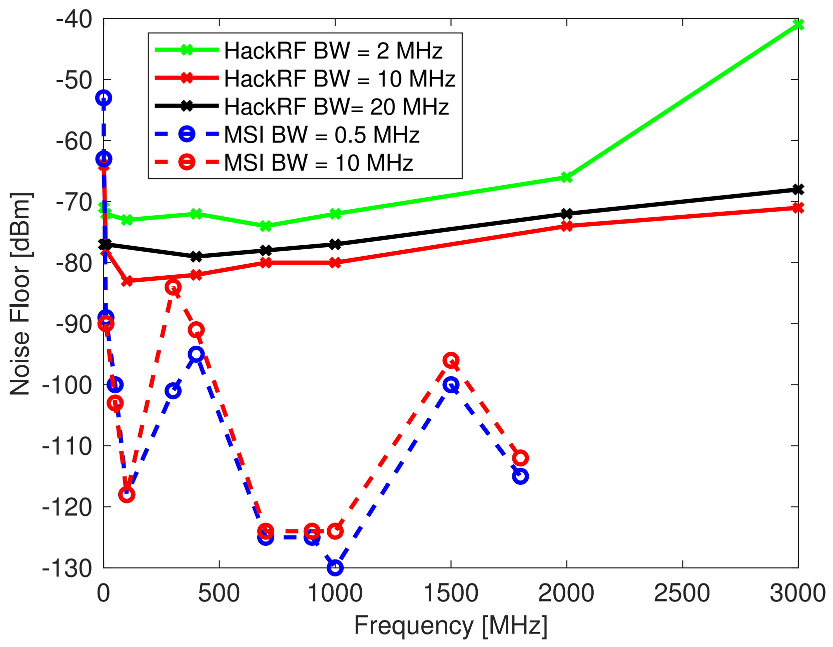


Fig. 4. Measured $NoFl$ for the SDRs with different bandwidths.

Commercial spectrum analyzers, considering bands up to 6 GHz, have on average sensitivities around -120 dBm. An SDR-based spectrum analyzer compensates its slightly poorer sensitivity with lighter weights, sensible lower costs and versatility in the application layer, taking advantage of the USB connection that receives the stream of raw down-converted RF samples, opening the possibility of a large number of digital data processing and visualization.

IV. SOFTWARE

Afer the IQ samples are acquired by the PC, they can be analyzed by several different packages. The RTL family [22] or the 12-bit ADALM PLUTO SDR [3] can be interfaced directly by a Matlab script. The different available USRP versions, on the other hand, are interfaced by Labview. However, a very common package used to interface and work with generic SDR data is the GNU Radio. It is open-source and therefore under constant development, though its support is not as reliable and consistent as similar commercial tools. The RF front end and digitization tasks are performed by the SDRs, whereas signal processing and visualization are executed by the GNU Radio application, with the physical connection between either components established by a USB channel.

GNU Radio has support to many different existing SDRs, and its design follows a graphical block-oriented application called GNU Radio Companion, similar to Labview from National Instruments and Simulink, from Matlab. It is written in C++, for critical processing tasks like drives and low level activities, and Python, for the high-level interface. The final project is compiled into a Python file, which can be latter run without the need for GNU Radio, requiring only a Python interpreter and the respective libraries, native in most Linux distributions. The application can make use of existing Python blocks and even implement its own, written in this language for specific cases. Though GNU Radio runs in either Linux and Windows, the latter is known to present some problems [23], being not so intensely developed and patched. Another issue with GNU Radio is the poor handling of large data chunks by the host computer, such as the cases involving spectrum analysis. Directing the incoming samples to a binary file, by means of a block named File Sink, rapidly becomes too cumbersome due to its size. For instance, a 20 MHz bandwidth generates a throughput 320 MB/s, 20 MHz times 8 times 2, the last term accounting for two complex numbers. Few seconds of a raw file with these settings rapidly result in large files, presenting complications to the storage and further handling.

Alternatively, it is possible to use a kind of server-client TPC/IP socket to transfer the data. Within GNU Radio, a ZMQ PUB block opens a socket that transfers the raw data from a source to a sink, the latter placed elsewhere, where further processing can be performed. Another alternative to deal with the incoming samples is a shell command, available in open libraries, which interface with the hardware directly from a command line. Due to the easiness of deploying an intuitive interface, and given the Python language use, the option was to use GNU Radio to perform all the interface and application development, using a single computer.

V. SPECTRUM ANALYSIS

In contrast to commercial spectrum analyzers, which can perform one-shot, full-bandwidth sweeps, the maximum bandwidth of SDRs is limited by their ADCs, so there are two basic situations for spectrum monitoring:

- Breaking the set maximum bandwidth BW into a smaller band (zooming in in the frequency domain), Fig. 5, top, shown in the red rectangle, or
- covering a frequency range larger than the SDR BW Fig. 5, bottom.

Each situation will be analyzed in detail in the following section.

A. Breaking the bandwidth

One possibility for breaking a large bandwidth into smaller chunks, in cases such as when over-sampling is used and result in large sampling frequencies, is by the use of the polyphase filter [24],

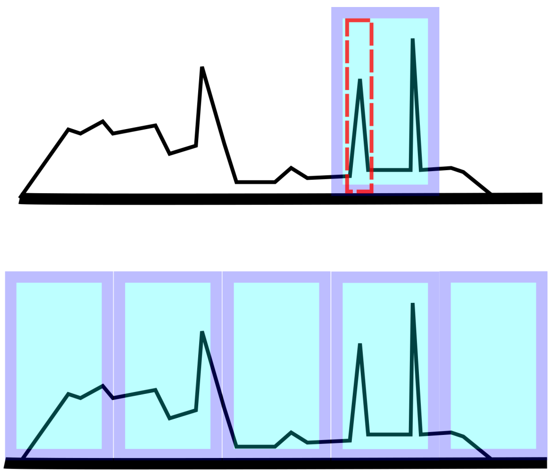


Fig. 5. Spectrum analysis, breaking (top) and assembling (bottom) the band of interest. The light blue area represents the SDR set bandwidth.

available as a default block within GNU Radio. Its T output taps divide the entire bandwidth, defined by the sample rate variable f_s, in T blocks, each one with width f_s/T. Fig. 6 illustrates the order by which the blocks are created. The whole band (width f_s) is shown divided in two (L from lower and U from upper) or three generic sub-bands (L, U and M from middle). After the sub-bands are selected, further processing can be applied to the specific desired range. As seen in Fig. 4, operating the SDR with smaller bandwidths as to narrow down the frequency range can decrease the overall RF sensitivity. So it is advisable to operate with a good enough BW and perform the bandwidth-breaking operation later on.

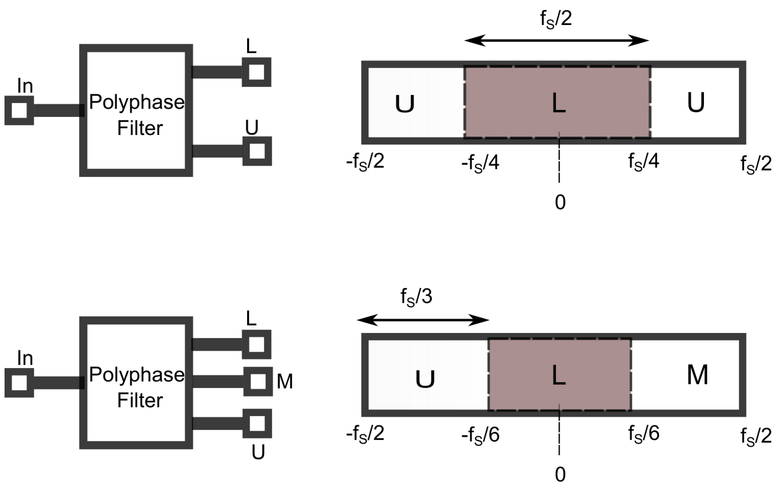


Fig. 6. Polyphase filter operation example within GNU Radio, with T=2 (top) and T=3 (bottom).

Fig. 7 shows an application of the polyphase filter with 3 taps, dividing a 20 MHz bandwidth where a pure 9 MHz CW (continuous wave) signal is used, internally generated by a sinusoidal source. The whole frequency band BW is divided in three sub-bands, isolating the carrier in the last (named "U") frequency block. Breaking the larger bandwidth is done in real-time by the filter. Small artifacts are generated in the other sub-bands, non-existent in the original frequency band, a by-product of the polyphase operation.

The same procedure is applied to a signal captured by the Hack RF One, with a telescopic antenna, a

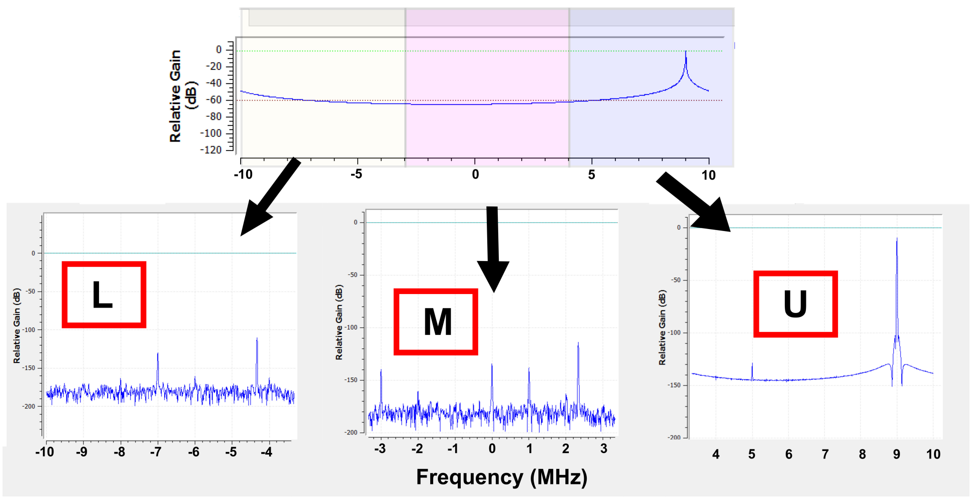


Fig. 7. Polyphase filter operation example, with 3 output taps, input signal generated by an analytical source (sinusoidal).

20 MHz bandwidth around the central frequency of 90 MHz, within the FM broadcast range, shown in Fig. 8. Again, some by-products non-existent on the original sequence are shown up after the filtering.

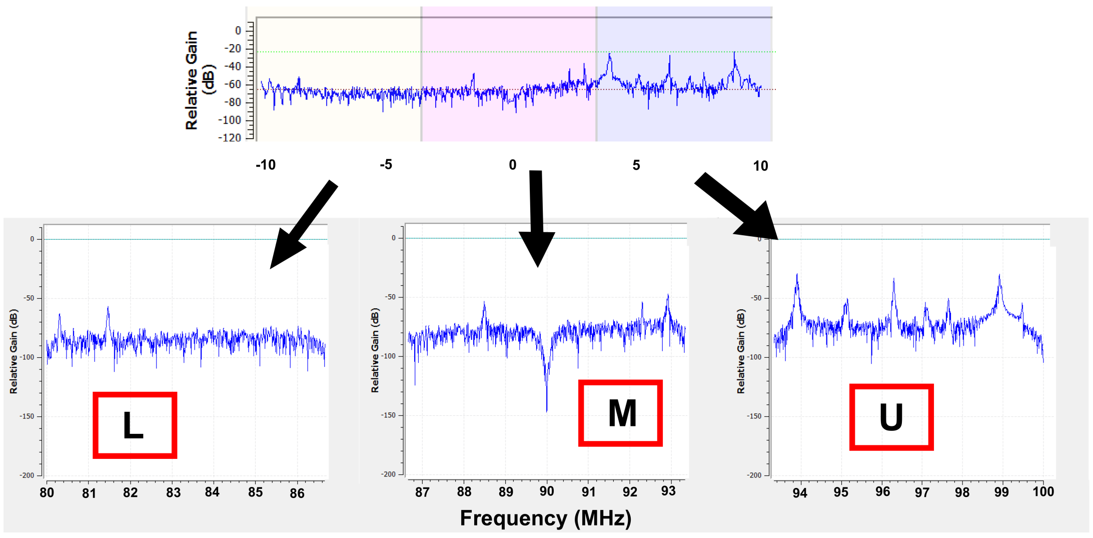


Fig. 8. Polyphase filter operation example, with 3 output taps, input from the Hack RF One.

As an alternative to the polyphase filter, a set of bandpass filters can be used in parallel, to break the large BW, topology known as sequential spectral analysis [25], at expenses of a larger processing time.

B. Assembling the bandwidth

The need for a large bandwidth can be related, for instance, to transient signals, whose spectrum energy distribution is not known a priori. Besides it, the 20 MHz maximum range of HackRF One is not enough to cover some services in the GHz range, such as WiFi IEEE 802.11, since it occupies a 100 MHz slot around 2400 MHz and approximately 700 MHz in the 5800 MHz range.

The logical link between GNU Radio and the SDR Hardware is done by means of a block named OSMOCOM Source (for receiving), or OSMOCOM Sink (for transmitting). It basically sets the central frequency F_c, its sampling rate F_s, alongside eventual gains (RF, IF and Baseband, for the HackRF One). So, in order to cover a bandwidth larger than the F_s one needs to periodically sweep the central frequency F_c as to cover the larger range. An alternative is depicted in the block diagram shown in fig. 9. A square wave oscillator is set to operate at the frequency F_{sync}, routed to a probe signal block. A Function Probe block does the polling, checking for its desired operation, in this case its amplitude level. As it reaches the threshold level a flag is generated. A Python module takes advantage of this flag to run a piece of Python code that updates a frequency list, sweeping its start and stop values with a certain step between consecutive items, set by the user. Though this approach runs as expected when the OSMOCOM source is replaced by an internal sinusoidal signal, the actual SDR connection results in a not pure synchronous operation with the F_{sync} timing, so a Python delay routine was added. Another observation is that the default QT GUI of GNU Radio produced erratic operation, i.e. the system closed down with a warning. A more stable version of the system runs using the WX GUI, performing equally well when tested in either Linux Ubuntu and under Windows 10.

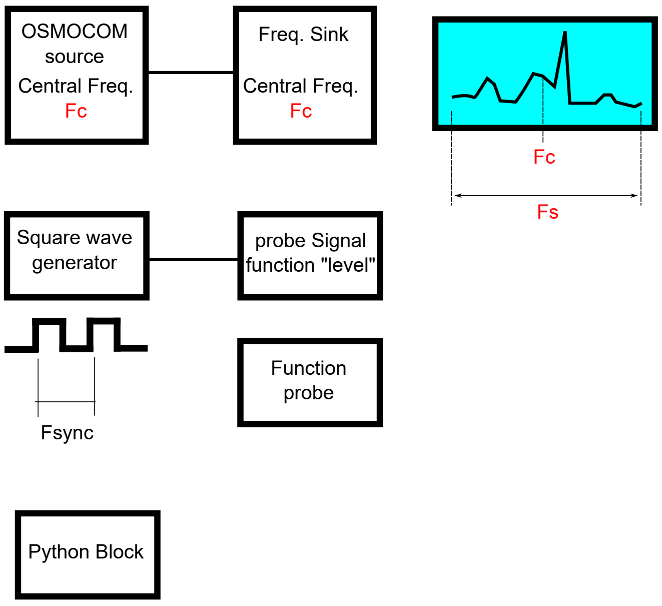
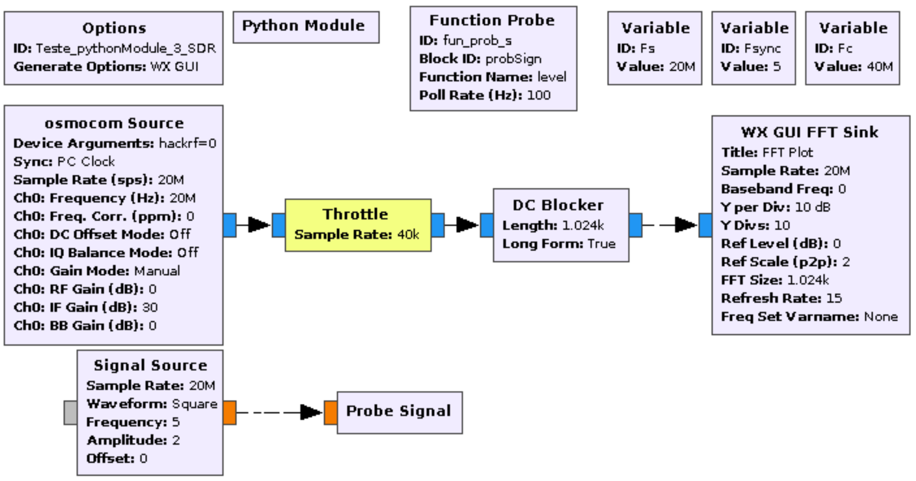


Fig. 9. Block diagram of the Assembly function using GNU Radio. F_c is periodically swept among two limit values, synchronously with the clock signal F_{sync}.

Fig. 10 shows the GNU Radio block diagram. A throttle block was left bypassed (in yellow), so in case a PC hardware overload is detected it can be switched back on.

Fig. 11 shows the results of the SDR set to its maximum bandwidth (20 MHz), sweeping from 10 MHz to 210 MHz and 1.8 GHz and 2 GHz. For the 1.8 to 2 GHz case an external LNA amplifier was added (NF=4.2 dB and 20 dB gain), and it was employed a printed Log Periodic antenna, shown in detail in Fig. 11. The internal LNA was switched off for the sake of protection against overload, and also educing the current drawn from the USB port. The screenshots were manually collected and assembled outside GNU Radio. It is possible to cover an arbitrary large bandwidth with this approach, limited by the SDR hardware only.



Options
ID: Teste_pythonModule_3_SDR
Generate Options: WX GUI

Python Module

Function Probe
ID: fun_prob_s
Block ID: probSign
Function Name: level
Poll Rate (Hz): 100

Variable
ID: Fs
Value: 20M

Variable
ID: Fsync
Value: 5

Variable
ID: Fc
Value: 40M

osmocom Source
Device Arguments: hackrf=0
Sync: PC Clock
Sample Rate (sps): 20M
Ch0: Frequency (Hz): 20M
Ch0: Freq. Corr. (ppm): 0
Ch0: DC Offset Mode: Off
Ch0: IQ Balance Mode: Off
Ch0: Gain Mode: Manual
Ch0: RF Gain (dB): 0
Ch0: IF Gain (dB): 30
Ch0: BB Gain (dB): 0

Throttle
Sample Rate: 40k

DC Blocker
Length: 1.024k
Long Form: True

WX GUI FFT Sink
Title: FFT Plot
Sample Rate: 20M
Baseband Freq: 0
Y per Div: 10 dB
Y Divs: 10
Ref Level (dB): 0
Ref Scale (p2p): 2
FFT Size: 1.024k
Refresh Rate: 15
Freq Set Varname: None

Signal Source
Sample Rate: 20M
Waveform: Square
Frequency: 5
Amplitude: 2
Offset: 0

Probe Signal

Fig. 10. GNU Radio diagram of the assembly operation.

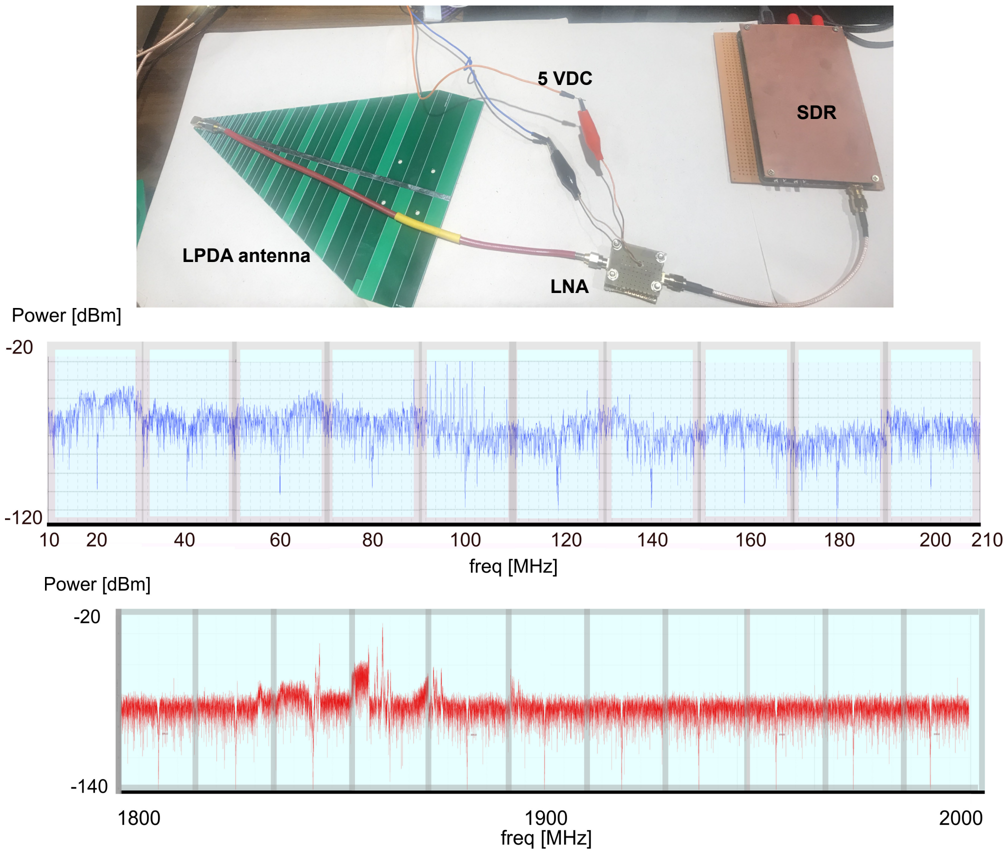


Fig. 11. Top, actual components of the measurement, both SDR and LNA are in shielded cases. Result of the bandwidth swept from 10 MHz to 210 MHz (center) and 1800 MHz and 2000 MHz (bottom). Each light blue rectangular block represents one separate 20 MHz individual slot.

VI. Conclusion

An SDR-based spectrum monitor is presented, with a pure-software solution to achieve wide bandwidth coverage, an innovation to circumvent the hardware limit that negatively contrasts to spectrum analyzers. The system is low-cost and low-profile, with the application software deployed using an open-source package, running in either Linux or Windows Operational Systems. In addition, a SDR-based spectrum analyzer is more protected against the obsolescence, since its software application is made independent of the hardware, so it can be upgraded independently of the SDR, unlike bulky commercial spectrum analyzers.

Appendix - Power Measurement

A core question regarding SDRs and the respective application regards the accuracy and unit of the displayed received power. Though several different applications show the spectrum power results in dBm, a proper calibration procedure is needed to ensure the correct readout.

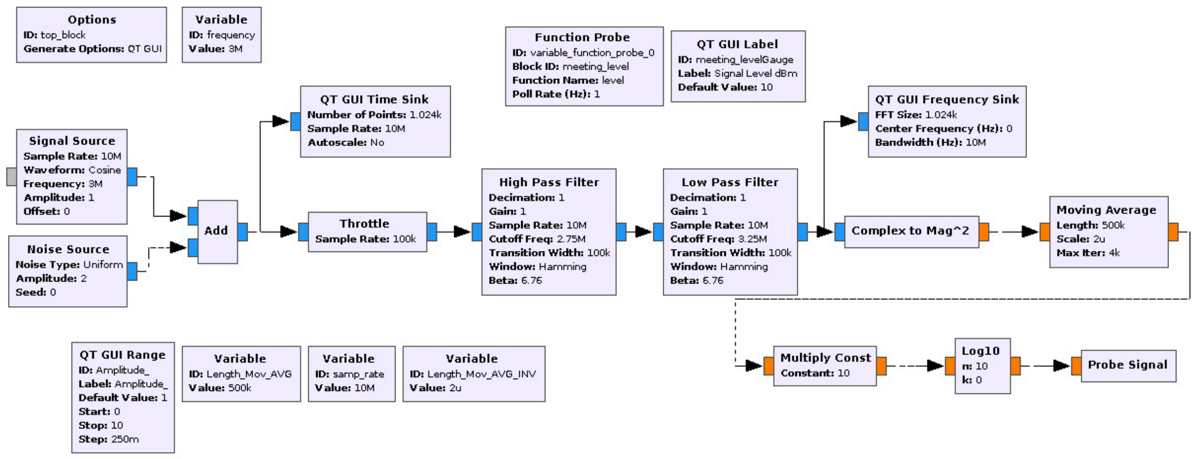
IQ samples delivered to GNU Radio or other application are described by their peak voltages v_I and v_Q respectively. Under a 50 Ω system, the signal power can be written as:

$$P_{RMS} = \frac{V_{RMS}^2}{50} = \frac{v_I^2 + v_Q^2}{2} \cdot \frac{1}{50}. \tag{7}$$

with V_{RMS} representing the RMS voltage of the IQ sample. Converting the linear into dBm results in:

$$P_{dBm} = 10 \log(10 \cdot (v_I^2 + v_Q^2)). \tag{8}$$

A GNU Radio program that computes the power in dBm, following Eq. 8, is shown in Fig. 12. A sinusoidal signal is added to a white noise source. The composed data stream goes through a cascade of low and high-pass filters, and is submitted to a moving average filter, which has the effect of further reducing the noise peak-to-peak amplitude, leaving the signal energy unchanged - as long as the moving average block parameters "scale" and "length" are kept reciprocal to each other. The 10-multiplying factor and logarithm are further included in the flow. GNU Radio has the possibility of using probes to operate on the data, in the diagram it captures the data stream amplitude level. A Function Probe block polls the probe signal and displays it in a GUI label block. Both Time and Frequency Sinks spread along the chain sample the stream and plot the data in frequency and time domains.


Options
ID: top_block
Generate Options: QT GUI

Variable
ID: frequency
Value: 3M

Function Probe
ID: variable_function_probe_0
Block ID: meeting_level
Function Name: level
Poll Rate (Hz): 1

QT GUI Label
ID: meeting_levelGauge
Label: Signal Level dBm
Default Value: 10

QT GUI Time Sink
Number of Points: 1.024k
Sample Rate: 10M
Autoscale: No

QT GUI Frequency Sink
FFT Size: 1.024k
Center Frequency (Hz): 0
Bandwidth (Hz): 10M

Signal Source
Sample Rate: 10M
Waveform: Cosine
Frequency: 3M
Amplitude: 1
Offset: 0

Noise Source
Noise Type: Uniform
Amplitude: 2
Seed: 0

Add

Throttle
Sample Rate: 100k

High Pass Filter
Decimation: 1
Gain: 1
Sample Rate: 10M
Cutoff Freq: 2.75M
Transition Width: 100k
Window: Hamming
Beta: 6.76

Low Pass Filter
Decimation: 1
Gain: 1
Sample Rate: 10M
Cutoff Freq: 3.25M
Transition Width: 100k
Window: Hamming
Beta: 6.76

Complex to Mag^2

Moving Average
Length: 500k
Scale: 2u
Max Iter: 4k

QT GUI Range
ID: Amplitude_
Label: Amplitude_
Default Value: 1
Start: 0
Stop: 10
Step: 250m

Variable
ID: Length_Mov_AVG
Value: 500k

Variable
ID: samp_rate
Value: 10M

Variable
ID: Length_Mov_AVG_INV
Value: 2u

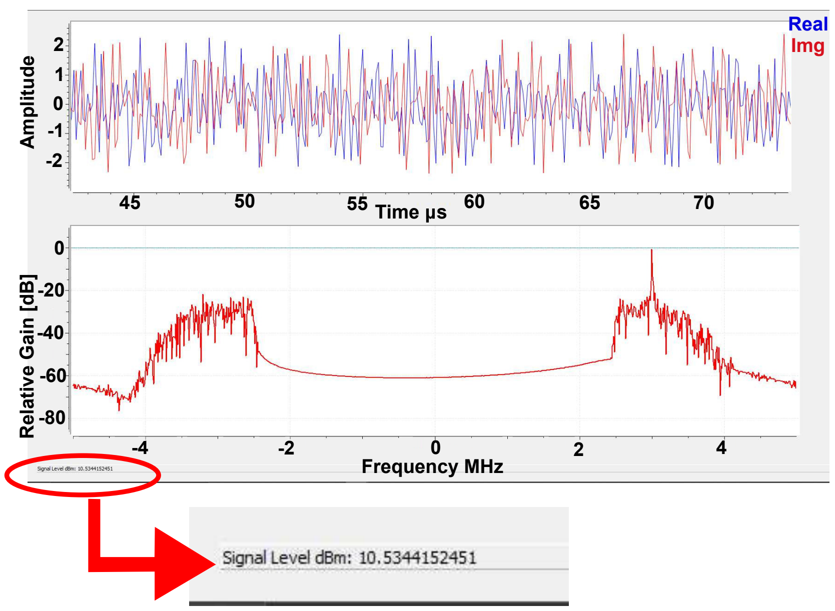
Multiply Const
Constant: 10

Log10
n: 10
k: 0

Probe Signal

Fig. 12. Example of GNURadio program to compute the power of a signal, in dBm.

The result is shown in Fig. 13. The sinusoidal source polluted by the noise is seen, on the top time domain plot. The bottom plot shows the combined effect of the LPF and HPF, leaving the signal peak visible. The probe label is shown down (10.53 dBm). The amplitude of the sinusoidal signal is unitary, which results a theoretical 10 dBm according to the eq. 8. The noise amplitude, set to 2, accounts for the observed computed difference with the analytical expected value.



Signal Level dBm: 10.5344152451

Fig. 13. Output relative to the program shown in Fig 12.

In order to quantify the improvements caused by the filtering, a sampling rate of 10 MHz was used in the program shown in Fig. 12, where a 3 MHz sinusoidal source was added to a uniform white-noise source; both delivering complex numbers. Table I shows the filter bandwidth effect on the computed power. Transition frequencies of both filters were set to 100 kHz and their bandwidths are limited between F_{min} and F_{max}. It can be seen that as the filter bandwidths decrease the noise effects are weighed out and the power result converges to the theoretical value of the signal alone, 10 dBm. Naturally, as the bandwidth and transition width parameters become narrower, the time it takes for the power readout increases, so it is a factor to be considered especially when dealing with time-varying carriers. In addition to this procedure involving filtering and averaging, for the case of an actual connection to a SDR and antenna, a calibration with a known signal generator is needed, in order to create a correction equation between the actual and displayed power levels.

TABLE I. INFLUENCE ON THE MEASURED POWER OF THE FILTER BANDWIDTH - SIGNAL WAS SINUSOIDAL, 3 MHZ.

Fmin (set by the HPF) [MHz]	Fmax (set by the LPF) [MHz]	Measured Power [dBm]
1.00	5.00	13.14
2.00	4.00	11.82
2.50	3.50	11.00
2.75	3.25	10.50

REFERENCES

[1] J. Mitola, *Software Radio Architecture: Object-Oriented Approaches to Wireless Systems Engineering*. John Wiley & Sons, New York, 2000.

[2] J. Reed, *Software Radio: A modern approach to radio engineering*. Prentice Hall, Upper Saddle River, 2002.

[3] T. F. Collins, R. Getz, D. Pu, and A. M. Wyglinksi, *Software-Defined Radio for Engineers*. Artech House, Boston, 2018.

[4] G. J. Minden, J. B. Evans, L. S. Searl, D. DePardo, R. Rajbanshi, J. Guffey, Q. Chen, T. R. Newman, V. R. Petty, F. Weidling, M. Peck, B. Cordill, D. Datla, B. Barker, and A. Agah, "Cognitive radios for dynamic spectrum access: An agile radio for wireless innovation," *IEEE Communications Magazine*, vol. 45, pp. 113–121, 2007.

[5] W. R. du Plessis, "Electronic-warfare training using low-cost software-defined radio platforms," in *XV SIGE, Electronic Warfare Symposium*, pp. 119–123, 2013.

[6] A. J. Costa, L. Antunes, and N. B. de Carvalho, "Spectrum analyzer with USRP, GNU Radio and Matlab," in *7th Conference on Telecommunications*, pp. 1–5, 2009.

[7] E. Santos-Luna, A. Prieto-Guerrero, R. A. Gonzalez, V. Ramos, M. Lopez-Benitez, and M. Cardenas-Juarez, "A spectrum analyzer based on a low-cost hardware-software integration," in *IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, pp. 607–612, 2019.

[8] A. Arcias-Moret, E. Pietrosemoli, and M. Zennaro, "Whisppi:white space monitoring with raspberry pi," in *Global Information Infrastructure Symposium - GIIS 2013*, pp. 1–6, 2013.

[9] E. G. Sierra and G. A. R. Arroyave, "Low cost SDR spectrum analyzer and analog radio receiver using GNU radio, raspberry pi2 and SDR-RTL dongle," in *7th IEEE Latin-American Conference on Communications (LATINCOM)*, pp. 1–6, 2015.

[10] M. A. Taha, M. T. Abdallah, H. A. Qasem, and M. A. Sada, "Dynamic spectrum analyzer using software defined radio," in *Proceedings of 2012 International Conference on Interactive Mobile and Computer Aided Learning (IMCL)*, pp. 167–172, 2012.

[11] J. Jose, R. Edison, and S. Sherin, "Low cost android radio spectrum analyzer using RTL-SDR dongle," *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 6, pp. 98–101, 2017.

[12] M. A. Sarijari, A. Marwanto, N. Fisal, S. K. S. Yusof, R. A. Rashid, and M. H. Satria, "Energy detection sensing based on gnu radio and usrp: An analysis study," in *Proceedings of IEEE 9th Malaysia International Conference on Communications*, pp. 338–342, 2009.

[13] W.-T. Chen, K.-T. Chang, and C.-P. Ko, "Spectrum monitoring for wireless TV and FM broadcast using software-defined radio," *Multimedia Theory and Applications*, vol. 75, pp. 9819–9836, 2016.

[14] Y. Guddeti, R. Subbaraman, M. Khazraee, A. Schulman, and D. Bharadia, "Sweepsense: Sensing 5 GHz in 5 milliseconds with low-cost radios," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pp. 317–330, 2019.

[15] M. Mishra and A. Potnis, "Wireless transmission detection and monitoring system using GNU Radio and multiple RTL–SDR receivers," *International Journal of Engineering Research and Application*, vol. 7, pp. 70–76, 2017.

[16] W. Liu, D. Pareit, and I. Moerman, "Advanced spectrum sensing with parallel processing based on software-defined radio," *EURASIP Journal on Wireless Communications and Networking*, vol. 228, pp. 1–15, 2013.

[17] M. Ewing, *The ABC of Software Defined Radios*. The American Radio Relay League, Newington, 2012.

[18] A. Valkanas, D. Pandey, and H. Leib, "Surfing the radio spectrum using RTL-SDR," *IETE Journal of Education*, vol. 60, pp. 65–73, 2019.

[19] B. E. Dunne, "The what, how and why of complex sampling for SDR transceivers," in *2019 ASEE North Central Section Conference*, pp. 1–9, 2019.

[20] R. Lyons, *Quadrature signals: Complex but not complicated.* available in www.dsprelated.com/showarticle/192.php, 2008.

[21] P. B. Kenington, *RF and Baseband techniques for software defined radio*. Artech House, Norwood, 2005.

[22] B. Stewart, K. Barlee, D. Atkinson, and L. Crockett, *Software Defined radio using Matlab & Simulink and the RTL-SDR*. University of Strathclyde, Glasgow, 2015.

[23] K. V. Ehr, W. Neuson, and B. E. Dunne, "Software defined radio: Choosing the right system for your communications courses," in *2016 ASEE Annual Conference and Exposition*, pp. 1–18, 2016.

[24] I. Alyafawi, A. Durand, and T. Braun, "High-performance wideband SDR channelizers," in *4th International Conference on Wired/Wireless Internet Communication (WWIC)*, pp. 3–14, 2016.

[25] A. Papoulis, *The Fourier integral and its applications*. Mc Graw Hill, New York, 1962.