

Heterogeneous sensing for high-fidelity spectrum characterization

By

Arunkumar Odedara (001265810)

A Report submitted in partial fulfillment for the
masters degree of Computer Science



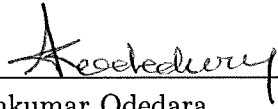
COMPUTER SCIENCE
UNIVERSITY AT ALBANY, SUNY
ALBANY, NY, USA

Advisor: Dr. Mariya Zheleva

Signature

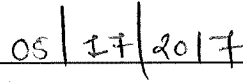
AUTHORIZATION FOR REPRODUCTION OF PROJECT

I grant permission for the reproduction of this project in its entirety, without further authorization from me, on the condition that the person or agency requesting reproduction, absorb the cost and provide proper acknowledgment of authorship.



Arunkumar Odedara

To: Thoyana, Taluka: Ranavav
Porbandar, Gujarat, 360575



Date

Acknowledgement

I would like to acknowledge many people who helped me during my journey through my masters degree at University at Albany. First of all I would thank my research/project advisor and mentor Dr. Mariya Zheleva for all the guidance, knowledge, patience and support. It was an honor and pleasure working with her. It was a great opportunity to be research assistant under her. I learned many new things and developed myself in various ways. I look forward to her guidance and friendship in the future.

I would thank Stuti Misra for being a great research/project partner and Paul Schmitt for providing access to the various materials of project.

I would like to thank all my friends and staff of UAlbany computer science department for all the help during Master's Degree.

Finally, I would thank my parents Keshavbhai and Miniben, my brother Hitesh for their love and support. Thanks dad for always putting me first and believing in me throughout my life. I cannot ask for better family in the world.

Abstract

Next generation spectrum measurement infrastructures have to provide ubiquitous spectrum sensing and characterization with fine granularity over time, frequency and space. This goal cannot be met in a scalable and economically-feasible fashion by traditional, stationary-sensor measurement infrastructures. Recent efforts have considered spectrum measurement with a variety of sensors in isolation, ranging from low-cost (e.g. RTL-SDR) to mid- and high-cost (e.g. USRP, HackRF, bladeRF, CRFS-RfEye); and from stationary to mobile. While each approach in isolation is promising, it is clear that a single-sensor infrastructure will not meet the needs of ubiquitous spectrum measurements. Thus, we envision a hybrid cost/mobility measurement infrastructure and, in this paper, we aim to evaluate the potential of such infrastructure to minimize cost while maximizing spectrum characterization outcomes. We aim to understand *How can we utilize a composition of variable cost and mobility sensors in a unified spectrum measurement infrastructure to achieve ubiquitous spectrum sensing and characterization?* and *What is the impact of sensor properties on the outcomes of spectrum measurements?* Our study leverages a mix of low-end RTL-SDR sensors and mid-end USRP sensors for spectrum sensing. It also makes use of our previously-developed methodology for wavelet-based transmitter characterization. We demonstrate that with careful positioning and scan configuration, the characterization outcomes of a low-cost sensor can be as good as those of its more expensive counterpart. Our insights inform efficient design of spectrum measurement infrastructures.

1 Introduction

Next generation mobile wireless networks operation will hinge on networks' ability to dynamically and opportunistically select their operating radio frequencies. Such opportunistic spectrum access is a significant departure from the current practice of exclusive frequency allocation. Currently, in order to communicate wirelessly, two devices are assigned an operation frequency and are explicitly configured to operate on that frequency. Future dynamic spectrum access paradigms will democratize spectrum access by allowing communication links to be established on any radio frequency that is not actively used. While such an operation will lead to significant increase in the efficiency of spectrum utilization, it also creates a plethora of technological, regulatory and enforcement challenges that stem from the need of dynamic spectrum characterization to identify available bands, and device reconfigurability to allow agile access and operation. *A key enabler of dynamic spectrum access in next generation wireless networks is deep understanding of spectrum utilization in time, frequency and space.*

Past research on spectrum sensing and characterization takes one of two approaches: (i) *direct* or (ii) *indirect*. In the *direct* approach, communicating devices (i.e. clients and base stations) are also tasked with identifying a common operation frequency. The *indirect* approach, in turn, makes use of a dedicated spectrum sensing and characterization infrastructure to gain information of spectrum occupancy [8]. Communicating devices query the spectrum infrastructures in order to obtain an operation frequency for opportunistic access. The benefit and drawbacks of these approaches can be discussed across several key criteria including (i) scalability, (ii) economic feasibility, (iii) sensing and characterization accuracy, (iv) communication overhead and (iv) applicability. Table 1 provides a summarized comparison of the two sensing modalities across these criteria. The direct spectrum sensing approach naturally scales well with the size and geographical distribution of the communication network, as each device participates in sensing. This reduces the monetary cost for spectrum sensing and characterization, making the direct sensing approach economically-feasible. On the flip-side, direct sensing is plagued with low accuracy, high overhead and low applicability. In terms of accuracy, the differences in heterogeneous sensing devices lead to variable quality of the spectrum scans [14, 13, 5, 4], which has an adverse impact on the accuracy of spectrum characterization. As a result, communicating devices may be unable to find a common operating frequency. Furthermore, direct sensing results in additional time, protocol and communications overhead, which effectively reduces the time devices are

	Direct	Indirect
Scalability	✓	✗
Economic	✓	✗
Accuracy	✗	✓
Overhead	✗	✓
Applicability	✗	✓

Table 1: Pros and cons of *direct* and *indirect* spectrum sensing and characterization.

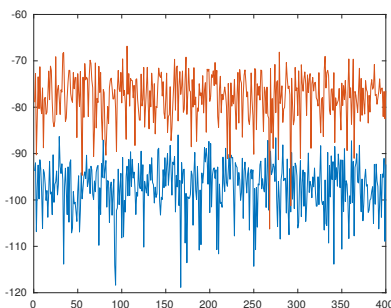


Figure 1: Signal variation of both sensors

left with to perform useful communication. The latter may have a detrimental impact on network performance and user experience. Last but not least, direct sensing has limited applicability. While it is geared to serve DSA technology, it is not well-suited for DSA policy and spectrum enforcement that require longitudinal, detailed and preemptive spectrum characterization. indirect spectrum measurements solve the limitations of direct sensing by ensuring high accuracy and applicability, and low overhead of spectrum measurements. Key drawbacks of indirect spectrum sensing, however, are related to scalability and economic feasibility, as dedicated infrastructures require careful design to be able to inform ubiquitous opportunistic access and are associated with a large additional cost for deployment and maintenance. Researchers and practitioners in next generation spectrum access agree that the benefits of indirect spectrum sensing outweigh its drawbacks, and thus, opportunistic spectrum access should be supported by dedicated spectrum sensing and characterization infrastructures [15].

2 Related Work

The goal of spectrum measurement infrastructures is to provide fine-grained spectrum characterization in time, frequency and space. The ability to characterize spectrum is directly dependent on the quality of collected scans, and thus, on the employed sensors and their cooperation within a unified sensing infrastructure. Early work on spectrum measurement infrastructures [11, 3, 16] employs high-end stationary sensors such as the CRFS Rf-Eye [11, 3, 16] or USRP [3, 16]. While they can provide high- and persistent-quality scans they do not scale well for ubiquitous spectrum sensing. Recent work has considered a crowd-sourced [14, 7, 19, 17, 10, 6] or vehicular-based [18] approach. While crowd-sourced sensing can provide high spatial resolution, such scans suffer from poor quality, which is further aggravated by the heterogeneity of consumer electronics and the measurement artifacts they introduce. Furthermore, consumer electronics' chipsets only cover ISM and cellular bands, which makes them infeasible for wideband sensing. Previous work suggests external augmentation of consumer electronics' RF front-end by RTL-SDR [14] or down-converters [19], however, this makes devices bulky and hard to operate in a crowd-sourced setup. Vehicular spectrum measurements [18] combine the benefits of stationary and crowd-sourced spectrum sensing by providing high spatial coverage and persistent scan quality. A key takeaway from prior work is that *a homogeneous-sensor infrastructures will not meet the needs of future spectrum management* for high scan resolution and learning. Expensive sensors can produce fine-granularity scans, however, as the spatial coverage requirements increase, so do the volume of data and infrastructure cost. Low-cost sensors allow economically-efficient spectrum sensing but have low resolution and sensitivity, resulting in poor data quality. We draw from the strengths of these two extremes and envision a hybrid measurement infrastructure that features a mix of mobile, static, low- and high-cost sensors. Such infrastructure can minimize cost while maximizing the quality and granularity of spectrum scans to achieve highly-efficient spectrum characterization. Towards this vision, we need a deep understanding of the benefits and limitation of mixed sensor infrastructures on spectrum characterization.

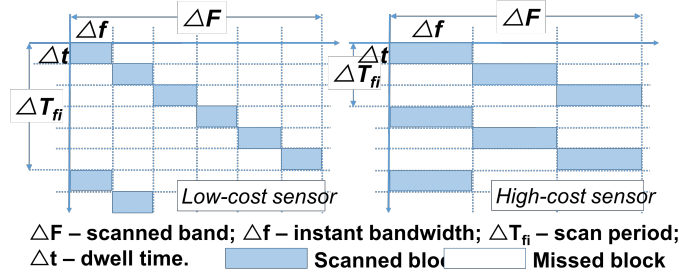
3 Background

3.1 Spectrum measurement objectives

Different spectrum measurement objectives pose different requirements on measurement algorithms and infrastructures. For example, if the goal of measurements is to simply capture the idle and occupied time-frequency blocks, then lightweight characterization can be performed at the sensors, and raw spectrum traces can be discarded. On the contrary, if the goal is detailed analysis of number of transmitters and their time-frequency usage patterns, then sensor-side characterization might not be feasible, requiring migration of spectrum scans to a centralized server for processing. Finally, if the goal is validation of analytical methods and Dynamic Spectrum Access (DSA) protocols, then there is a need for longitudinal sensing, centralization and storage of spectrum traces. A recent survey on spectrum measurement objectives [12] identified a wide range of priorities. Spectrum measurements (i) should help incumbents and secondary users to make real-time decisions for spectrum use, (ii) should support validation of analytical methods and protocols, (iii) should assist in spectrum enforcement and (iv) should be able to serve multiple objectives. Thus, there is a need for a spectrum measurement infrastructure that can provide continuous spatial coverage of spectrum measurements, store spectrum scans longitudinally and characterize the spectrum occupancy including number of transmitters, their temporal and frequency characteristics and the opportunity they grant for secondary spectrum access.

4 Methodology

Effects of sensor cost. The goal of spectrum measurements is to collect representative scans from a target frequency band ΔF . ΔF is typically in the order of GHz, however, a spectrum sensor is only able to scan a few MHz at a time. In order to cover the target wideband spectrum (30MHz-6GHz [15]), spectrum measurement infrastructures such as Microsoft’s Spectrum Observatory [3], utilize sequential scanning of consecutive bands as illustrated in Fig. 2 (top). The key limitation of sequential scanning is that any given frequency chunk f_i is scanned in discrete times t_j , as opposed to being scanned continuously. This causes some time-frequency blocks to be missed, which may lead to omission of important transmitter characteristics. Two key spectrum scan properties affect the impact of sensors on the measurement



Sensor cost	Instant Bandwidth	Sampling Rate	Noise Figure
Low-cost (RTL-SDR)	3.2MHz*	2MSps	8-13dB**
Mid-cost (USRP N210, WBX)	40MHz	25MSps***	5dB

*Stable at 2MHz; ** Dependent on host configuration; *** Limited by host bandwidth

Figure 2: Sensor capabilities vs. cost.

outcomes: (i) *scan periodicity*, or how often do we get data from a given spectrum chunk and (ii) *scan quality*.

The periodicity and quality of spectrum data collection are determined by the scan configuration and the sensor capabilities. In terms of configuration, the desired dwell time and FFT size determine the utility and quality of scan data. In terms of sensor capabilities, fundamental limiting factors are the instantaneous bandwidth, sampling rate and sensitivity (as determined by the noise figure). These capabilities vary drastically with the cost of the sensor, as illustrated in Fig. 2. Beyond the software defined radio (SDR), the processing power of the sensor also plays a critical role in how fast data can be processed (i.e. PSD estimation) and stored. We demonstrate that these tasks take a negligible amount of time on a general purpose PC but may take a substantial amount of time if the SDR host device is a phone or an embedded computer.

To quantify the *scan periodicity* of a spectrum chunk f_i , we define the *scan period* ΔT_{f_i} as the time between recurring scans of f_i . ΔT_{f_i} depends on sensor capabilities and scan configuration as follows: $\Delta T_{f_i} = (\Delta t * \Delta F / \Delta f) + t_{proc}$, where Δt is dwell time, Δf is the sensor's instantaneous bandwidth and t_{proc} is a delay that factors in the processing overhead (e.g. time to calculate the FFT). Our preliminary evaluation demonstrates that the overhead in wideband sequential scanning by a low-cost sensor can be substantial as the computation load increases. Fig. 2(a) (up-left) presents the average tune-to-record time, defined as the delay from when the radio is tuned to a new frequency until the sensor records the first PSD value on that frequency.

The processing requirements of FFT calculation is proportional to the FFT size N . As the processing demand increases, the tune-to-record time grows rapidly from $200ms$ at $N = 256$ to $6s$ at $N = 4096$.

The *quality* of spectrum data in a scanned block can be quantified by its (i) time-frequency *granularity* and (ii) *dynamic range*. The frequency granularity, of a scanned band f_i can be expressed as $f_i^n = \Delta f / N$, where N is the desired FFT size, $f_i^n, n \in \{1, \dots, N\}$ is the size of a frequency bin and Δf is the sensor's instantaneous bandwidth. The temporal granularity of spectrum data is determined by the number of sweeps J that can be completed in a frequency chunk f_i for a given dwell time Δt . The time to complete a single sweep $t_j, j \in \{1, \dots, J\}$ is dependent on the sensor capabilities as follows: $t_k = (N/f_s) + t_{proc}$, where f_s is the sensor's sample rate. As Fig. 2(a) (up-right) shows, the processing overhead of a sensor can significantly impact the temporal granularity of spectrum data. Along with granularity, the dynamic range of the sensor plays a key role in spectrum data quality. It depends on the sensitivity of the sensor, which is influenced by the quality of the receiver RF chain. Previous work [14] observed that the noise figure of low-cost RTL-SDR is 8-13dBm higher than that of mid-cost USRPs. This limits the capabilities of RTL-SDRs to sense low-power transmissions.

Effects of sensor mobility. Sensor mobility increases the spatial coverage of spectrum scans, however, it also increases the scan period and reduces the data quality. Let us consider a mobile sensor that traverses a fixed route L of length d meters with speed s meters per second and scans the full target band ΔF at discrete locations $l_i, i \in \{1, \dots, |L|\}$ along the route. The scan period of a frequency chunk f_i at location l_i will then be $\Delta T_{f_i}^{l_i} = d/s + \Delta T_{f_i}$. Intuitively, the scan intermittency of a band f_i increases due to sensor mobility, which further increases the chance of missed transmitter characteristics. Mobility also impacts the spectrum scan quality, as it introduces variability of the signal level at which transmitters are sensed as the sensor moves towards and away from these transmitters.

We perform a preliminary study of the effects of sensor cost on spectrum learning outcomes in a controlled scenario. We are particularly interested in (i) the impact of distance between transmitter and sensor and (ii) sensing duration on spectrum learning outcomes. We use one USRP B210 setup as a transmitter that sends a periodic OFDM pattern. We simultaneously collect spectrum traces with an RTL-SDR and a USRP N210 sensor for a fixed duration of 17 seconds while increasing the distance between the sensors and the transmitter. To further understand the benefits and limitation of mixed cost/mobility sensors, we will carry out controlled and real-world

spectrum measurement studies, described in more detail in Sec. 5.3 and 5.5

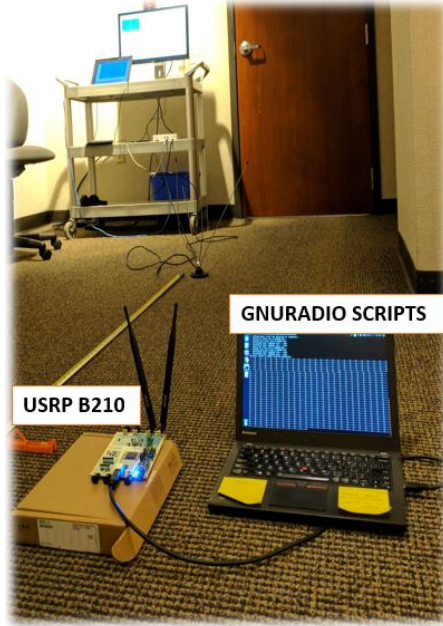
5 Spectrum measurement results

In this section we present results of spectrum measurement characterization with mixed cost/mobility sensors.

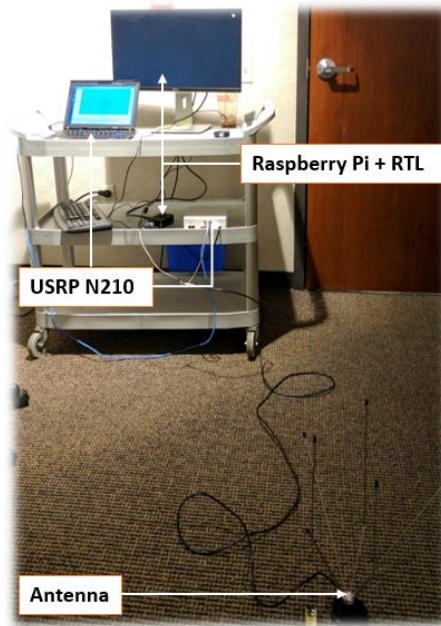
5.1 Experiment Setup

1. *Hardware Setup* One of our main objectives is to compare and analyze the performance of low cost sensor in comparison with high-cost sensor. Our low cost sensor solution is the RTL-SDR device, which is easily available at under a 30USD. The RTL-SDR has a maximum sampling rate of 2 MHz. For our higher cost sensor for comparison with the RTL-SDR, we are using USRP N210 devices provided by Ettus Research. The USRP has a maximum sampling rate of 22 MHz. We are also using a USRP N210 for controlled transmissions at the specified frequencies with provided sample rate and transmission pattern. We also use a USRP N210 as our high-cost sensor for comparison with the low-cost sensor. RTL-SDR and USRP N210 both sensors share one common antenna during measurements to reduce experimental variability. To collect our traces, we are connecting our RTL-SDR to a Raspberry Pi device. Pi provides a Linux platform called Raspbian and it receives the traces from the RTL-SDR. Raspberry pi is an ideal candidate for a low-cost sensor since it is readily available, has a Linux platform, is mobile, and is extremely low-cost (35USD). Raspberry Pi 3(one of the devices we use for our experiment) includes a quad-core Cortex-A7 CPU running at 900 MHz and 1 GB RAM.

2. *Software Setup* We are using the GNURadio Open Source library for the software components of our sensors. We use blocks provided by GNURadio [1] to configure our transmitter and receiver. By using and modifying various available blocks in GNURadio, we set up various environments for our experiments. For example, configuring the GNURadio library allowed us to get various types of transmitter pattern we need to modify the code of transmission file. After getting raw data, it needs to be converted to Power Spectral Density measures according to FFT, sample rate, bandwidth. After collecting raw samples through air using GNURadio blocks, we generate binary file from raw data. Based on the FFT of raw samples we process binary files to get PSD values in spreadsheet. Using Matplotlib [9] we also plot heat maps to see ongoing transmission traces in



(a) Transmitter



(b) Receiver with both Sensors

the form of image.

3. *Parameter Selection:* Our experiment require setting various parameters at the controlled transmitter and sensors. We describe these parameters below:

- **Sample Rate:** Sampling rate, f_s , is the average number of samples obtained in one second (samples per second), thus $f_s = 1/T$. Sample rate define size of a frequency window. In other words based on the sample rate sensor sense and record data for that range. As said above for most of our experiments we chose 2MHz to be used for fair comparison of both sensors, as the maximum supported frequency by RTL-SDR is 2MHz.
- **Dwell-Delay:** Dwell Delay refers to the time period ΔT in which sensor be collecting the data. By providing the dwell delay both sensor sense the given frequency for given amount of time. For our experiment purpose we have dwell delay as 20 Seconds.
- **Gain:** Gain is a key performance number which combines the antenna's

directivity and electrical efficiency. As a transmitting antenna, the gain describes how well the antenna converts input power into radio waves headed in a specified direction. As a receiving antenna, the gain describes how well the antenna converts radio waves arriving from a specified direction into electrical power.

- **FFT Size:** The selected FFT size directly affects the resolution of the resulting spectra. For instance, if the FFT size is 1024 and the Sampling Rate is 8192, the resolution of each spectral line be: $8192 / 1024 = 8$ Hz. Larger FFT sizes provide higher spectral resolution but take longer to compute.
- **Frequency:** The number of complete oscillations per second of energy (such as sound or electromagnetic radiation) in the form of waves is frequency. As we talked in the background frequencies are used for various purposes. We transmit a particular frequency with no overlapping to any real world transmitters, with 2MHz of sample rate and sensing the same frequency through both sensors.
- **Transmitter Pattern:** To understand the complete behaviour of sensor under various real world scenarios we made our transmitter transmit various kinds of signals. There are main three type of transmission we used continuous, periodic, aperiodic.

5.2 Challenges

1. *Android Platform:* In the beginning of this project we considered an Android application RF-Analyzer. As a low-cost sensor we plug RTL-SDR sensor with android tablet NEXUS 7. We modify the application to get all needed PSD values into spreadsheet. From collected data with PSD values we can generate a heat-map for visual representation of those values. But, because of internal CPU use for various background processes the setup seemed to be dropping tremendous amount of samples comparing to the USRP. We did experiments regarding distance as well as granularity with both sensors simultaneously for better comparison between sensors. As we use GNURadio with USRP-laptop pair and application for RTL we could not provide same parameters as needed to both sensors. That is why we switched from android to Raspberry Pi which provide better

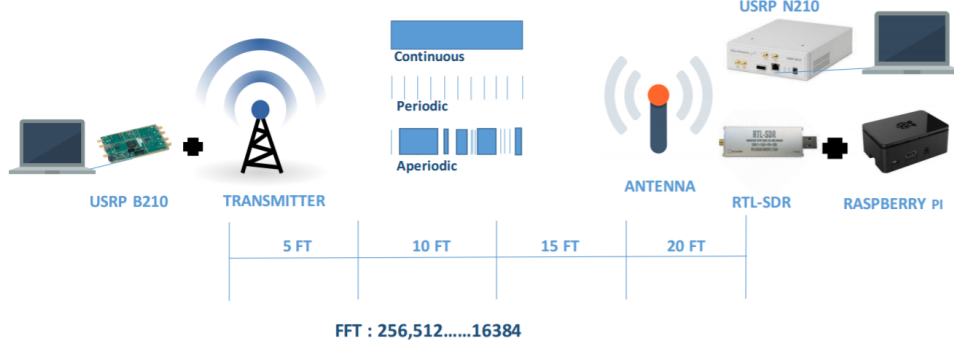
and cheaper option for research purpose. Now, we can run almost the same scripts with same parameters for both sensors using GNURadio blocks.

2. Gain Power Value: While doing experiments we figured out that setting gain power on each side transmitter as well as receiver is one of the most important parameter. In theory, a high gain leads to better sensitivity of the sensor. A too high gain, however may lead to receiver saturation, which leads to undesirable signal artifacts. Thus, we had to select the gain carefully, to make sure we are not saturating the receiver. The same holds for the transmitter as well. So, setting the appropriate value of gain power is one of the main challenge we came across during the project. Even for each sensor, range of the gain is different and, stable gain for both the sensors be different according to experiments. Therefore, we conducted some experiments to find and decide stable gain power for each of the sensors. At the end we conclude that gain for USRP works better around 20dB and for RTL-SDR gain around 25dB. After selecting these values there were no signal artifacts captured while collecting raw data.

5.3 Controlled spectrum measurements

We begin our study on mixed sensor infrastructures in a controlled setup, to be able to tune both transmitter properties as well as sensors' configuration and mobility and evaluate their effects on spectrum characterization. Specifically, we utilize real transmitters and sensors based on a range of SDRs (e.g. RTL-SDR, USRP N210 and USRP B210) and host platforms (e.g. laptops and embedded Raspberry Pi platforms). We program the transmitters to emit a diverse set of benchmark patterns in order to evaluate the *effects of transmitter count and dynamics on learning outcomes*. We sense these transmitters with a mix of cost/mobility sensors and while varying their sensing configurations in order to evaluate the *effects of sensor type and configuration on learning outcomes*. Specifically, we evaluate the effects of the following factors on spectrum characterization outcomes: We connected a USRP B210 device with a laptop running GNURadio scripts for transmitting given frequency. We had another USRP N210 to collect the traces as well as an RTL-SDR connected to a Raspberry Pi collecting the traces simultaneously.

Effects of spectrum data granularity (f_i^n and t_k): We vary the sample rate f_s and FFT size N of two stationary mixed-cost sensors, while sensing a certain transmitter pattern. We, thus, measure the effects of t_{proc}



(c) Device Setup

and data granularity on spectrum characterization.

For this experiment we set up devices as depicted in figure 3(c) for setup. USRP with GNURadio scripts transmitting a continuous signal and the other USRP and RTL-SDR be collecting these traces with the different FFT size. Based on the FFT size we know how sparse data can each of the sensor senses. We set the FFT size as 256, 512, 1024, 2048, 4096, 8192, 16382.

Effects of sensor sensitivity: As detailed earlier, the noise figure of different sensors determines their sensitivity to transmitters at various SNR. We compare the learning outcomes of a low-cost (less sensitive) and high-cost (more sensitive) sensor for a given transmitter pattern with varying SNR. We ensure the same scan period and granularity, in order to focus only on effects due to sensitivity.

As it can be seen by figure 3(c), The setup is similar to that figure but, here we changed the distance manually between transmitter and receiving antenna. This will give us the maximum distance for both sensors, till what distance they can sense the signal. We have same pair of sensors defined in hardware setup, but we change transmitter's location periodically from the sensors.

Effects of transmitter pattern: For this study we program the transmitters to emit a variety of patterns: broadcast, periodic and aperiodic TDM. We sense these patterns with a low- and high-cost sensor simultaneously. Both sensors have the same configuration to allow for fair comparison.

As we can see in the figure 3(c), we setup one USRP as transmitter and another USRP and RTL pair as receivers collecting traces of transmitted signal. And through various scripts we generated the three transmission patterns discussed above. This experiment give us better understanding of

the sensors in the sense of transmit signal behaviour. In real world signal can vary according to factors like, transmitting speed, frequency, distance from transmitter. Signal from transmitter can broadcast continuously, aperiodically or aperiodically. To understand how both sensors perform under these behaviours we transmit signal in these three patterns. On the receiver side we collect raw data samples from both sensors simultaneously.

Effects of sensor mobility: As detailed earlier, sensor mobility increases the scan period $\Delta T_{f_i}^{l_i}$, which impacts characterization outcomes. To study the effects of mobility, we emulate sensor mobility by periodically halting the sensing activity for a predefined time window (effectively emulating the sensor moving away from location l_i). We configure two mixed-cost sensors as mobile and two other mixed-cost sensors as stationary. We measure a given transmitter pattern with all four sensors to benchmark the effects of mobility on spectrum characterization.

As described in the figure 3(c), One USRP device transmitted the signal in random and discontinuous pattern. In this experiment we collect data samples with these transmitted patterns for specific time period ΔT from both sensors simultaneously. In real world it is not necessary that our sensors will be at any stationary position i.e. fixed. They can be mobile for an example, travelling on the top of vehicle. To compute the performance of the sensor while losing contact with transmitter and coming close again we have a special kind of setup.

We first collect samples in traditional manner for ΔT period of time, this file will have full data as it is stationary and sense the transmitter throughout the time. Then as sensor moves away from transmitter it will not be able to sense the transmitter in that period until it comes back to near the transmitter. Therefore we introduce two parameters 1)sleep time (time when sensor is away from transmitter and will not sense transmitter) and, 2)awake time (time when sensor is close to transmitter and able to sense it's signal). We will delete the data recorded in sleeping time and keep the data for awake time.

For an example, we collect data for 20s, for 3 seconds sensor is close to transmitter and, for 4s it is away from transmitter and it comes back for next 3s and same thing for 20s. So, 4s is our sleeping time and 3s is our awake time from ΔT total time. More Sleeping time represents the longer distance from the transmitter of sensor. When it more frequently wake up that represent the behaviour where sensor is coming in the contact of transmitter sooner.

Equalizing learning outcome: We aim to answer if there is a sensing configuration of a low-cost sensor that bring its learning outcomes on par

with those of a high-cost sensor. We first evaluate the learning outcomes of the two sensors using the same sensing configuration. We then vary the sensing configuration of the low-cost sensor to determine if/when it characterization outcomes become equivalent to these of high-cost sensors. This study quantifies the overhead of low-cost sensors to produce high-fidelity characterization.

5.4 Sensor artifacts

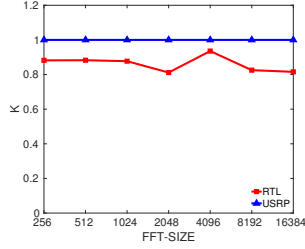
Fewer lines in RTL than in USRP due to dropped samples.

Less capable sensors such as the RTL tend to collect fewer lines than their more capable counterparts. We postulate that the reason is due to dropped samples. To explore this better, we evaluate the sweep loss rate as $k = \frac{nlines_{ACT}}{nlines_{EXP}}$, where $nlines_{EXP}$ is the expected number of sweeps for a given scan configuration and $nlines_{ACT}$ is the actual number of collected sweeps. For a sensor that collect spectrum scans as PSD values over a frequency range, the sensor's scan configuration can be expressed in terms of the sensor's sample rate and FFT size. Given a sample rate s and FFT size N the time it takes to collect a single spectrum sweep Δt is $\Delta t = N * \frac{1}{s}$

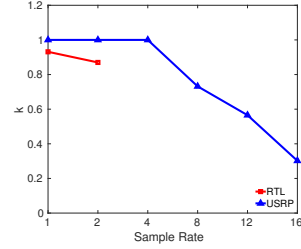
We do the following experiments to understand the behaviour of sample drop by RTL-SDR comparing to USRP :

1) k with increasing sensor sample rate: We collect data from each sensor at different sample rate. As maximum stable rate for RTL-SDR is 2MHz, we collect data samples with 1M and 2M from both sensors. As 1024 is highest stable FFT for RTL we keep same FFT size with increasing sample rate. In the figure, x-axis represents increasing sample rate and y-axis represents ratio of collected lines and expected number of lines. This gives us the ratio of lines dropped by each sensors. As it can be seen in the figure 3(e) (below) RTL-SDR is doing almost same as USRP. USRP collects 100% of data captured at 1M, 2M and 4M. USRP drops samples after 4M and at 16M it collects only 30% of data. RTL-SDR seems to be collecting 93% of data at 1M and 88% at 2M.

2) k with increasing data granularity: We collect data from each sensor with increasing FFT. As maximum stable rate for RTL-SDR is 2MHz, we collect raw data samples with 2M from both sensors. After keeping same sample rate 2M we increase FFT size for each runs. We collect 4 runs for each FFT and represent average over those 4 runs. In the figure, x-axis represents increasing sample rate and y-axis represents ratio



(d) k with Increasing FFT Size



(e) k with Increasing Sample Rate

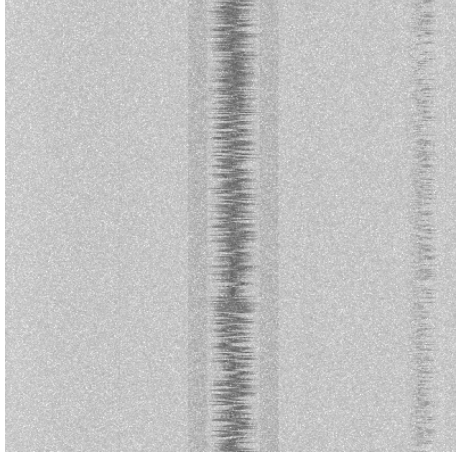
of collected lines and expected number of lines. This gives us the ratio of lines dropped by each sensors. As it can be seen in the figure 3(d) (below) RTL-SDR is doing almost same as USRP. USRP collects 100% of data captured at all granularity values. RTL drops samples at FFT 2046 and surprisingly it collects more than 90% data at FFT 4096 and then drops again. Overall RTL-SDR collects 80% of data with all granularity values.

5.5 Real-world spectrum measurements

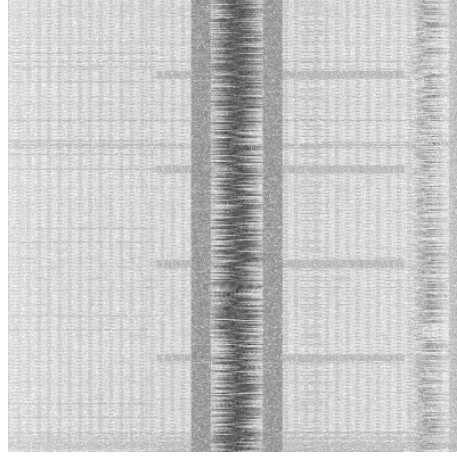
Here we collect data from real world transmitter to understand the behaviour of both sensors in a real-world spectrum measurement task. We have the stable sample rate of 2MHz for both sensors and they sense one frequency from real world transmitter. We represent data in spreadsheet and visualize the data by script or MATLAB [2] in the form of heatmap. Examples of the real world transmitters are FM radio stations, Television channels from various broadcasters, Military communication frequencies, Wireless communication frequencies, etc. We sense one of the transmitter and see how both sensor behave instead of manually generated frequency by user-defined transmitter. We set sample rate of 2M, gain of 20dB, frequency 107.7M, ΔT record time and 1024 FFT for both sensors. They also share common antenna as described earlier to reduce signal variability. Below in the figure 3(f) represents the RTL-SDR behaviour in real world radio station WGNA-FM with 107.7M frequency. And, figure 3(g) represents heat map for USRP sensor of PSD values received from raw sample.

6 Design implications

As discussed above, we collect raw samples through GNURadio blocks and get PSD values in the csv or binary format. RTL-SDR and USRP sensors



(f) RTL real world



(g) USRP real world

will collect data simultaneously in controlled transmission manner which are either periodic or aperiodic (depending on the experiment). Characterization of collected data is done by one of the other member of research. To characterize these transmissions collected from the transmitter, we evaluate our transmission into the following metrics:

Gap: The gap is the idle time period between two transmissions. We calculate this as the time between the beginning of the transmission and the end of the previous transmission.

Cycle: The cycle is the time from the beginning of the current transmission to the beginning of the next transmission.

Duration: The duration is the time a transmission is transmitted for. This is measured from the beginning of the transmission to the end of the transmission.

Bandwidth: Bandwidth indicated the occupied frequency. It is the frequency that the transmission is transmitted over.

We use this gap, cycle, duration, and bandwidth metrics and compare them for the low-cost and high-cost sensor. This indicates if the characterizations from the scans of the sensors could be comparable.

7 Discussion and conclusion

Our preliminary evaluation shows that low-cost sensors are disadvantaged compared to their more expensive counterparts due to fundamental limitations of their hardware capabilities. At the same time, spectrum character-

ization outcomes by low- and high-cost sensors are not drastically different, which brings hope for the feasibility of low-cost spectrum measurements. Our goal is to further investigate optimal mixed sensor configurations that will result in high-quality spectrum characterization. Our findings informs the design of a framework for automatic configuration of heterogeneous sensors in a unified sensing infrastructure. Furthermore, we will also evaluate the effects of sensor configuration on storage requirements to inform efficient data management in a mixed sensor infrastructure.

Overall, low-cost sensor is capable for analyzing the spectrum as well as spectrum characterization in it's own limits. Our findings gives a better and calculated platform that will work as a tool to analyze given spectrum based on factors like transmitter behaviour, geographical location, sensor mobility.

**Code snippets of the files
used in this project**

```

1 #!/usr/bin/python2
2 #
3 # Copyright 2005,2007,2011 Free Software Foundation, Inc.
4 #
5 # This file is part of GNU Radio
6 #
7 # GNU Radio is free software; you can redistribute it and/or modify
8 # it under the terms of the GNU General Public License as published by
9 # the Free Software Foundation; either version 3, or (at your option)
10 # any later version.
11 #
12 # GNU Radio is distributed in the hope that it will be useful,
13 # but WITHOUT ANY WARRANTY; without even the implied warranty of
14 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
15 # GNU General Public License for more details.
16 #
17 # You should have received a copy of the GNU General Public License
18 # along with GNU Radio; see the file COPYING. If not, write to
19 # the Free Software Foundation, Inc., 51 Franklin Street,
20 # Boston, MA 02110-1301, USA.
21 #
22
23 from gnuradio import gr, eng_notation
24 from gnuradio import blocks
25 from gnuradio import audio
26 from gnuradio import filter
27 from gnuradio import fft
28 from gnuradio.fft import window
29 from gnuradio import uhd
30 from gnuradio.eng_option import eng_option
31 from optparse import OptionParser
32 import sys
33 import math
34 import struct
35 import threading
36 from datetime import datetime
37 import time
38
39 sys.stderr.write("Warning: this may have issues on some machines+Python version combinations to seg fault due to the callback in bin_statistics.\n\n")
40
41 class ThreadClass(threading.Thread):
42     def run(self):
43         return
44
45 class tune(gr.feval_dd):
46     """
47     This class allows C++ code to callback into python.
48     """
49     def __init__(self, tb):
50         gr.feval_dd.__init__(self)
51         self.tb = tb
52
53     def eval(self, ignore):
54         """
55         This method is called from blocks.bin_statistics.f when it wants
56         to change the center frequency. This method tunes the front
57         end to the new center frequency, and returns the new frequency
58         as its result.
59         """
60
61     try:
62         # We use this try block so that if something goes wrong
63         # from here down, at least we'll have a prayer of knowing
64         # what went wrong. Without this, you get a very
65         # mysterious:

```

```

65         # mysterious:
66         #
67         # terminate called after throwing an instance of
68         # 'Swig::DirectorMethodException' Aborted
69         #
70         # message on stderr. Not exactly helpful ;)
71
72         new_freq = self.tb.set_next_freq()
73
74         # wait until msgq is empty before continuing
75         while(self.tb.msgq.full_p()):
76             #print "msgq full, holding.."
77             time.sleep(0.1)
78
79         return new_freq
80
81     except Exception, e:
82         print "tune: Exception: ", e
83
84
85 class parse_msg(object):
86     def __init__(self, msg):
87         self.center_freq = msg.arg1()
88         self.vlen = int(msg.arg2())
89         assert(msg.length() == self.vlen * gr.sizeof_complex)
90
91         # FIXME consider using NumPy array
92         t = msg.to_string()
93         self.raw_data = t
94         self.data = struct.unpack('%df' % (self.vlen,), t)
95
96 class my_top_block(gr.top_block):
97
98     def __init__(self):
99         gr.top_block.__init__(self)
100
101         usage = "usage: %prog [options] min_freq max_freq"
102         parser = OptionParser(option_class=eng_option, usage=usage)
103         parser.add_option("-s", "--args", type="string", default="",
104             help="UHD device device address args (default=%default)")
105         parser.add_option("--spec", type="string", default=None,
106             help="Subdevice of UHD device where appropriate")
107         parser.add_option("-A", "--antenna", type="string", default=None,
108             help="select fx Antenna where appropriate")
109         parser.add_option("-r", "--samp-rate", type="eng_float", default=32e6,
110             help="set sample rate [default=%default]")
111         parser.add_option("-g", "--gain", type="eng_float", default=None,
112             help="set gain in dB (default is midpoint)")
113         parser.add_option("-t", "--tune-delay", type="eng_float",
114             default=1.00, metavar="SECS",
115             help="time to delay (in seconds) after changing frequency [default=%default]")
116         parser.add_option("--dwell-delay", type="eng_float",
117             default=0.25, metavar="SECS",
118             help="time to dwell (in seconds) at a given frequency [default=%default]")
119         parser.add_option("-B", "--channel-bandwidth", type="eng_float",
120             default=6.25e3, metavar="Hz",
121             help="channel bandwidth of fft bins in Hz [default=%default]")
122         parser.add_option("-l", "--lo-offset", type="eng_float",
123             default=0, metavar="Hz",
124             help="lo offset in Hz [default=%default]")
125         parser.add_option("-s", "--squell-threshold", type="eng_float",
126             default=None, metavar="dB",
127             help="squell threshold in dB [default=%default]")
128         parser.add_option("-r", "--rounds", type="int", default=1,
129             help="specify number capture rounds [default=1]")
130

```

```

122         default=0.2563, metavar='Hz',
123         help='channel bandwidth of fft bins in Hz [default=%default]')
124     parser.add_option("--lo-offset", type="eng_float",
125                       default=0, metavar="Hz",
126                       help="lo offset in Hz [default=%default]")
127     parser.add_option("-g", "--squellch-threshold", type="eng_float",
128                       default=None, metavar="dB",
129                       help="squellch threshold in dB [default=%default]")
130     parser.add_option("-r", "--rounds", type="int", default=1,
131                       help="specify number capture rounds [default=1]")
132     parser.add_option("-s", "--fft-size", type="int", default=1024,
133                       help="specify number of FFT bins [default=1024]")
134     parser.add_option("--real-time", action="store_true", default=False,
135                       help="Attempt to enable real-time scheduling")
136     parser.add_option("-k", "--keep", type="int", default=1,
137                       help="keep one in N [default=1]")
138     (options, args) = parser.parse_args()
139     if len(args) != 2:
140         parser.print_help()
141         sys.exit(1)
142
143     self.channel_bandwidth = options.channel_bandwidth
144     self.rounds = options.rounds
145     self.roundcount = 0
146     self.keep = options.keep
147
148     self.min_freq = eng_notation.str_to_num(args[1])
149     self.max_freq = eng_notation.str_to_num(args[2])
150
151     if options.fft_size is None:
152         self.fft_size = int(self.usrp_rate/self.channel_bandwidth)
153     else:
154         self.fft_size = options.fft_size
155
156
157     if self.min_freq > self.max_freq:
158         # swap them
159         self.min_freq, self.max_freq = self.max_freq, self.min_freq
160
161     if not options.real_time:
162         realtime = False
163     else:
164         # Attempt to enable realtime scheduling
165         r = gr.enable_realtime_scheduling()
166         if r == gr.RT_OK:
167             realtime = True
168         else:
169             realtime = False
170             print "Note: failed to enable realtime scheduling"
171
172     # build graph
173     self.rtl = osmosdr.source(args="numchans" + str(1) + " " + "" + "")
174     self.blocks_keep_one_in_n_0 = blocks.keep_one_in_n(gr.sizeof_gr_complex, self.keep)
175
176     # Set the subdevice spec
177     if(options.spec):
178         self.rtl.set_subdev_spec(options.spec, 0)
179
180     # Set the antenna
181     if(options.antenna):
182         self.rtl.set_antenna(options.antenna, 0)
183
184     self.rtl.set_sample_rate(options.samp_rate)
185     self.usrp_rate = usrp_rate = self.rtl.get_sample_rate()

```



```

183
184 self.rtl.set_sample_rate(options.samp_rate)
185 self.usrp_rate = usrp_rate = self.rtl.get_sample_rate()
186 self.center_freq = 0
187
188 self.lo_offset = options.lo_offset
189
190
191 self.squelch_threshold = options.squelch_threshold
192
193 self.freq_step = self.nearest_freq((0.75 * self.usrp_rate), self.channel_bandwidth)
194 if (self.min_freq == self.max_freq):
195     self.min_center_freq = self.min_freq
196 else:
197     self.min_center_freq = self.min_freq + (self.freq_step/2)
198     nsteps = math.ceil((self.max_freq - self.min_freq) / self.freq_step)
199     self.max_center_freq = self.min_center_freq + (nsteps * self.freq_step)
200
201 self.next_freq = self.min_center_freq
202 self.dwell_delay = options.dwell_delay
203
204 tune_delay = max(0, int(round(options.tune_delay * usrp_rate / self.fft_size))) # in fft_frames
205 dwell_delay = max(1, int(round(options.dwell_delay * usrp_rate / self.fft_size))) # in fft_frames
206
207 self.msgq = gr.msg_queue(1)
208
209 center = self.set_next_freq()
210
211 print "Tuned to: ", center, options.dwell_delay
212 self.blocks_file_sink_0 = blocks.file_sink(gr.sizeof_gr_complex*1, "/home/pi/paul/code/5/RTL-5-1.cfile", False)
213 self.blocks_file_sink_0.set_unbuffered(False)
214 self.connect(self.rtl, self.blocks_keep_one_in_n_0)
215 self.connect(self.blocks_keep_one_in_n_0, self.blocks_file_sink_0)
216
217 if options.gain is None:
218     # if no gain was specified, use the mid-point in dB
219     g = self.rtl.get_gain_range()
220     options.gain = float(g.start()+g.stop())/2.0
221
222 self.set_gain(options.gain)
223 print "gain =", options.gain
224
225 def retune(self):
226     self.disconnect_all()
227     center = self.set_next_freq()
228     print "Tuned to: ", center, self.dwell_delay
229     self.blocks_file_sink_0 = blocks.file_sink(gr.sizeof_gr_complex*1, "/home/pi/paul/code/Mobility/RTL-30-2M.cfile", True)
230     self.blocks_file_sink_0.set_unbuffered(False)
231     self.connect(self.rtl, self.blocks_keep_one_in_n_0)
232     self.connect(self.blocks_keep_one_in_n_0, self.blocks_file_sink_0)
233
234 def set_next_freq(self):
235     target_freq = self.next_freq
236     self.next_freq = self.next_freq + self.freq_step
237     if self.next_freq >= self.max_center_freq:
238         self.roundcount += 1
239         self.next_freq = self.min_center_freq
240
241 if not self.set_freq(target_freq):
242     print "Failed to set frequency to", target_freq
243     sys.exit(1)
244
245 return target_freq

```

```

234 def set_next_freq(self):
235     target_freq = self.next_freq
236     self.next_freq = self.next_freq + self.freq_step
237     if self.next_freq >= self.max_center_freq:
238         self.roundcount += 1
239         self.next_freq = self.min_center_freq
240
241     if not self.set_freq(target_freq):
242         print "Failed to set frequency to", target_freq
243         sys.exit(1)
244
245     return target_freq
246
247
248 def set_freq(self, target_freq):
249     """
250     Set the center frequency we're interested in.
251
252     Args:
253         target_freq: frequency in Hz
254         @rtype: bool
255     """
256     r = self.rtl.set_center_freq(self.next_freq, 0)
257     if r:
258         return True
259
260     return False
261
262 def set_gain(self, gain):
263     self.rtl.set_gain(gain)
264
265 def nearest_freq(self, freq, channel_bandwidth):
266     freq = round(freq / channel_bandwidth, 0) * channel_bandwidth
267     return freq
268
269 def main_loop(tb):
270
271     timestamp = 0
272     centerfreq = 0
273     while 1:
274         if timestamp == 0:
275             timestamp = time.time()
276             centerfreq = tb.center_freq
277             centerfreq = tb.center_freq
278             time.sleep(tb.dwell_delay)
279             tb.stop()
280             tb.wait()
281             if tb.roundcount < tb.rounds:
282                 tb.retune()
283             else:
284                 break
285             tb.start()
286
287 if __name__ == '__main__':
288     t = ThreadClass()
289     t.start()
290
291     tb = my_top_block()
292     try:
293         tb.start()
294         main_loop(tb)
295     except KeyboardInterrupt:
296         pass
297

```

```

Open ▾ Save
1 #!/usr/bin/env python
2 #####
3 # Gnuradio Python Flow Graph
4 # Title: Top Block
5 # Generated: Fri May 20 14:57:36 2016
6 #####
7
8 from gnuradio import blocks
9 from gnuradio import eng_option
10 from gnuradio import fft
11 from gnuradio import filter
12 from gnuradio import gr
13 from gnuradio.eng_option import eng_option
14 from gnuradio.fft import window
15 from gnuradio.filter import firdes
16 from optparse import OptionParser
17 import sys, math
18
19 inputfilename = ''
20 outputfilename = ''
21 keep = 0
22
23 class top_block(gr.top_block):
24
25     def __init__(self):
26         gr.top_block.__init__(self, "Top Block")
27
28         #####
29         # Variables
30         #####
31         self.fft_size = fft_size = options.fft_size
32         self.fft_rate = fft_rate = 15
33
34         # blocks
35         #####
36         self.single_pole_iir_filter_xx_0 = filter.single_pole_iir_filter_ff(0, fft_size)
37         self.fft_vxx_0 = fft.fft_vcc(fft_size, True, (window.blackmanharris(fft_size)), True, 1)
38         self.blocks_stream_to_vector_0 = blocks.stream_to_vector(gr.sizeof_gr_complex*1, fft_size)
39         self.blocks_nlog10_ff_0 = blocks.nlog10_ff(1, fft_size, "math.log10fft_size")
40         self.blocks_keep_one_in_n_0 = blocks.keep_one_in_n(gr.sizeof_gr_complex*fft_size, keep)
41         self.blocks_file_source_0 = blocks.file_source(gr.sizeof_gr_complex*1, inputfilename, false)
42         self.blocks_file_sink_0 = blocks.file_sink(gr.sizeof_float*fft_size, outputfilename, false)
43         self.blocks_file_sink_0.set_unbuffered(True)
44         self.blocks_complex_to_mag_0 = blocks.complex_to_mag(fft_size)
45         self.dc_blocker_xx_0 = filter.dc_blocker_cc(256, False)
46
47         # connections
48         #####
49         self.connect((self.blocks_file_source_0, 0), (self.dc_blocker_xx_0, 0))
50         self.connect((self.dc_blocker_xx_0, 0), (self.blocks_stream_to_vector_0, 0))
51         self.connect((self.blocks_stream_to_vector_0, 0), (self.blocks_nlog10_ff_0, 0))
52         self.connect((self.blocks_nlog10_ff_0, 0), (self.blocks_keep_one_in_n_0, 0))
53         self.connect((self.blocks_keep_one_in_n_0, 0), (self.fft_vxx_0, 0))
54         self.connect((self.fft_vxx_0, 0), (self.blocks_complex_to_mag_0, 0))
55         self.connect((self.blocks_complex_to_mag_0, 0), (self.single_pole_iir_filter_xx_0, 0))
56         self.connect((self.single_pole_iir_filter_xx_0, 0), (self.blocks_file_sink_0, 0))
57         self.connect((self.blocks_file_sink_0, 0), (self.blocks_file_sink_0, 0))
58
59
60
61
62
63
64     def get_fft_size(self):
65         return self.fft_size
66
67     def set_fft_size(self, fft_size):
68         self.fft_size = fft_size
69
70     def get_fft_rate(self):
71         return self.fft_rate
72
73     def set_fft_rate(self, fft_rate):
74         self.fft_rate = fft_rate
75
76 if __name__ == '__main__':
77     parser = OptionParser(option_class=eng_option, usage="%prog inputfile outputfile")
78     parser.add_option("-k", "--keep", type="int", default=1,
79                       help="keep one in N [default=1]")
80     parser.add_option("-f", "--fft_size", type="int", default=1024,
81                       help="fft size")
82     (options, args) = parser.parse_args()
83     if len(args) != 2:
84         sys.exit("Incorrect number of arguments")
85
86     keep = options.keep
87     inputfilename = args[0]
88     outputfilename = args[1]
89     tb = top_block()
90     tb.start()
91     tb.wait()

```

```

63
64     def get_fft_size(self):
65         return self.fft_size
66
67     def set_fft_size(self, fft_size):
68         self.fft_size = fft_size
69
70     def get_fft_rate(self):
71         return self.fft_rate
72
73     def set_fft_rate(self, fft_rate):
74         self.fft_rate = fft_rate
75
76 if __name__ == '__main__':
77     parser = OptionParser(option_class=eng_option, usage="%prog inputfile outputfile")
78     parser.add_option("-k", "--keep", type="int", default=1,
79                       help="keep one in N [default=1]")
80     parser.add_option("-f", "--fft_size", type="int", default=1024,
81                       help="fft size")
82     (options, args) = parser.parse_args()
83     if len(args) != 2:
84         sys.exit("Incorrect number of arguments")
85
86     keep = options.keep
87     inputfilename = args[0]
88     outputfilename = args[1]
89     tb = top_block()
90     tb.start()
91     tb.wait()

```

```

1 import scipy
2 from optparse import OptionParser
3 import sys, math, csv
4 from gnuradio.eng_option import eng_option
5
6
7 parser = OptionParser(option_class=eng_option, usage="%prog [options] input txers_file output")
8 parser.add_option("-k", "--keep", type="int", default=1,
9                  help="keep one in k (default:1)")
10 parser.add_option("-f", "--fft-size", type="int", default=1024,
11                  help="fft size (default=1024)")
12 (options, args) = parser.parse_args()
13 if len(args) == 2:
14     inputfilename = args[1]
15     txersfilename = args[2]
16     outputfilename = args[1]
17 elif len(args) == 3:
18     inputfilename = args[1]
19     outputfilename = args[2]
20 else:
21     print "Incorrect number of arguments."
22 fft_size = options.fft_size
23 oneinn = options.keep
24
25
26
27 f = scipy.fromfile(open(inputfilename), dtype=scipy.float32)
28 f = f.reshape((-1, fft_size))
29 extractedData = f[:,(fft_size/8)::(-1*(fft_size/8+1))]
30
31
32
33 from numpy import random
34 import matplotlib as mpl
35 import matplotlib.pyplot as plt
36 import matplotlib.patches as patches
37
38 fig = plt.figure()
39 plt.set_cmap('magma')
40 fig.set_size_inches(4, 4)
41 ax = plt.Axes(fig, [0., 0., 1., 1.])
42 ax.set_axis_off()
43 fig.add_axes(ax)
44
45 rows, columns = f.shape
46 im = ax.imshow(f, extent=[0, columns, 0, rows], aspect = 'auto', interpolation='nearest')
47
48 im.set_clim(-120, 70)
49 #im.set_clim(-97, -60)
50
51 if len(args) == 3:
52     # Open txers file
53     f = open(txersfilename, "rb")
54     txers = csv.reader(f)
55     # Loop through each row, add a rectangle for each: first tuple is bottom left corner of rectangle, then width and height
56     for row in txers:
57         ax.add_patch(patches.Rectangle((int(row[0]), int(row[1]), int(row[2]), int(row[3])), fc='none', ec='r', lw=2))
58
59 # Save the figure
60 fig.savefig(outputfilename+".png", dpi=100)

```

```

1 import scipy
2 from optparse import OptionParser
3 import sys, math, csv
4 from gnuradio.eng_option import eng_option
5
6
7 parser = OptionParser(option_class=eng_option, usage="%prog [options] input output")
8 parser.add_option("-k", "--keep", type="int", default=1,
9                  help="keep one in k (default:1)")
10 parser.add_option("-f", "--fft-size", type="int", default=1024,
11                  help="fft size (default=1024)")
12 (options, args) = parser.parse_args()
13 if len(args) != 2:
14     sys.exit("Incorrect number of arguments.")
15
16 fft_size = options.fft_size
17 oneinn = options.keep
18 inputfilename = args[1]
19 outputfilename = args[2]
20 outfile = open(outputfilename, 'wb')
21 wr = csv.writer(outfile)
22
23 f = scipy.fromfile(open(inputfilename), dtype=scipy.float32)
24 f = f.reshape((-1, fft_size))
25
26 #Write CSV file - keep one in n using [:::n]
27 for i in f[:oneinn]:
28     #keep middle 3/4 of fft to deal with scaling
29     wr.writerow([i[(fft_size/8)::(-1*(fft_size/8+1))]])

```

```

Open ▾  Save
1 import csv, os, sys
2 from optparse import OptionParser
3 from gnuradio.eng_option import eng_option
4
5 parser = OptionParser(option_class=eng_option, usage="%prog [options] input txers_file output")
6 parser.add_option("-t", "--threshold", type="float", default=100,
7                  help="threshold value for SNR (default=100)")
8
9 (options, args) = parser.parse_args()
10
11 filename = args[0]
12 newFile = args[1] if len(args) > 1 else "Aligned.csv"
13
14 ##### Finding Starting Point #####
15
16 def starting_row(filename):
17     with open(filename, "r") as f:
18         lines = [line.strip().split(',') for line in f]
19         values = [float(i) for i in lines]
20         print "values are: ", values
21         threshold = options.threshold
22         flag = False
23         start_row = 0
24         for i in values:
25             if float(i) >= threshold:
26                 flag = True
27                 start_row = float(values.index(i))
28             if(flag == True):
29                 os.system("echo '###' " + filename + " > " + newFile)
30             if(flag == False):
31                 starting_row(filename) print "No Data collected according to selected threshold value"
32
33 starting_row(filename)
34
35
Python ▾ Tab Width: 8 ▾ Ln 1, Col 1 ▾ INS

```

```

Open ▾  Save
1 from future import division
2 import csv, os, sys
3
4 from optparse import OptionParser
5 from gnuradio.eng_option import eng_option
6
7 parser = OptionParser(option_class=eng_option, usage="%prog [options] input txers_file output")
8 parser.add_option("-f", "--fft-size", type="int", default=1024,
9                  help="FFT size (default=1024)")
10 parser.add_option("-s", "--sample-rate", type="int", default=200,
11                  help="sample rate (default=200)")
12 parser.add_option("-w", "--wake-time", type="int", default=1,
13                  help="wake time of sensor (default=1)")
14 parser.add_option("-l", "--sleep-time", type="int", default=1,
15                  help="sleep time of sensor (default=1)")
16
17 (options, args) = parser.parse_args()
18
19 input_file = args[0]
20 output_file = args[1]
21
22 fft_size = options.fft_size
23 samp_rate = options.samp_rate
24 t_sleep = options.sleep_time
25 t_wake = options.wake_time
26
27 def clip():
28     #CLIPPING OF THE DATA WHERE TRANSMITTER IS SLEEPING AND KEEPING THE DATA WHILE TRANSMISSION IS GOING ON
29     total_time = 30
30     samp_rate_real = samp_rate
31     t_line = fft_size/samp_rate_real #TIME REQUIRED FOR ONE LINE
32     delete = round(t_sleep/t_line) #PART WHERE TRANSMITTER IS SLEEPING
33     keep = round(t_wake/t_line) #PART WHERE TRANSMISSION IS GOING AND AWAKE
34     lines = [line.strip().split(',') for line in open(input_file, "r")]
35     one_round = delete + keep
36     start_row = one_round - 1
37     next = one_round + keep
38     print "Length of lines: ", len(lines), "Inkeep is: ", keep, "Indelete is: ", delete
39     writer = csv.writer(open(output_file, "ab"))
40     count = 0
41     while count < len(lines):
42         if lines.index(row) < keep:
43             os.system("echo '###' " + filename + " > " + newFile)
44             print "CD: " + str(count)
45             writer.writerow(row)
46             print lines.index(row)
47             if lines.index(row) > start_row and lines.index(row) < next:
48                 print
49                 writer.writerow(row)
50                 print lines.index(row)
51                 if lines.index(row) == next:
52                     start_row = one_round
53                     next = one_round + keep
54
55 clip()
56
Python ▾ Tab Width: 8 ▾ Ln 1, Col 1 ▾ INS

```

```

Open  Save
1 package utils;
2 import java.io.IOException;
3 import java.io.RandomAccessFile;
4 import java.nio.ByteBuffer;
5 public class Clip {
6     static int fft_size = 0;
7     static int samp_rate = 0;
8     static int t_sleep = 0;
9     static int t_wake = 0;
10
11     public static void main(String[] args) throws IOException {
12         fft_size = 1024;
13         samp_rate = 1000000;
14         t_wake = 4;
15         t_sleep = 0;
16         String csv_input = "/home/mariya/paul/code/Deadline/Mobile/Ran/Ran/Clipped/USRP-Ran-Mobile-clip-1.csv";
17         String output = "/home/mariya/paul/code/Deadline/Mobile/Ran/4/USRP-Ran-Mobile-clip-1-4.bin";
18         int num_rows = -1;
19         clip(csv_input, output, num_rows);
20         System.out.println("Done.");
21     }
22     private static void clip(String csv_input, String output, int num_rows) throws IOException {
23         double t_line = (float) fft_size / samp_rate; // TIME REQUIRES FOR ONE LINE
24         int delete = (int) Math.round(t_sleep / t_line); // PART WHERE TRANSMITTER IS SLEEPING
25         int keep = (int) Math.round(t_wake / t_line); // PART WHERE TRANSMISSION IS GOING ON i.e. AWAKE
26         int one_round = delete + keep;
27         int start_row = one_round - 1;
28         int next = one_round + keep;
29
30         double[][] in = Utils.fillFromFile(csv_input);
31         if (num_rows < 0) {
32             num_rows = in.length;
33             System.out.println("Number of Rows : " + num_rows);
34         }
35         try {
36             RandomAccessFile raf = new RandomAccessFile(output, "rw"); // loop through each entry
37             System.out.println("Writing Data.");
38             for (int row = 0; row < num_rows; row++) {
39                 for (int col = 0; col < in[row].length; col++) {
40                     float f = (float) in[row][col];
41                     byte[] t = ByteBuffer.allocate(4).putFloat(f).array();
42                     byte[] b = new byte[t.length];
43                     for (int j = t.length - 1; j >= 0; j--)
44                         b[(b.length - 1) - j] = t[j];
45                     if (row < keep) {
46                         raf.write(b);
47                     }
48                     if (row > start_row && row < next) {
49                         raf.write(b);
50                         // System.out.println("Row : "+row);
51                     }
52                     if (row == next) {
53                         start_row = one_round;
54                         next = one_round + keep;
55                     }
56                 }
57             }
58             raf.close();
59         } catch (Exception e) {
60             System.out.println("Caught exception in writeBinary(): "
61                             + e.getMessage());
62             e.printStackTrace();
63         }
64     }
65 }

```

Java Tab Width: 8 Ln 46, Col 62 INS

```

1 indata=importdata('/home/mariya/Documents/MATLAB/FFT_K.csv');
2 rtl=indata.data(:,1);
3 usrp=indata.data(:,2);
4
5 figure(7);
6 clf('reset');
7 hold on; box on;
8 set(gca, 'LineWidth', 2.5, 'FontSize', 32, 'PlotBoxAspectRatio', [3 2.5 1.5]); % 'Position', [-1 -1 3 5]); % 'PlotBoxAspectRatio', [3 1.5 1]); % 'FontWeight', 'bold');
9 title('Sample Drop ratio (k)');
10 xlabel('FFT-SIZE');
11 ylabel('K');
12
13 plot(rtl, 'r-s', 'LineWidth', 6, 'MarkerSize', 12);
14 plot(usrp, 'b-^', 'LineWidth', 6, 'MarkerSize', 12);
15 %plot(perc_err, line_type(3), 'LineWidth', 6, 'MarkerSize', 6)
16 ylim([0 1.2]);
17
18 xticks([1 2 3 4 5 6 7]);
19 xticklabels(['256', '512', '1024', '2048', '4096', '8192', '16384']);
20 %xtickangle(20);
21
22 leg1=['RTL'; % s='num2str(scale)'];
23 leg2=['USRP']; % s='num2str(scale)'];
24 leg3=['Percentage error']; % s='num2str(scale)'];
25 legend(leg1, leg2, 'location', 'SouthEast');
26 legend boxoff
27
28 filename = ['/home/mariya/Documents/MATLAB/2.eps'];
29 saveas(gca, filename, 'eps');
30 hold off;

```

Bibliography

- [1] GNURadio. <https://www.gnuradio.org/>.
- [2] MATLAB. <http://www.mathworks.com/help/matlab/index.html>.
- [3] Microsoft's Spectrum Observatory. <https://observatory.microsoftspectrum.com/>.
- [4] Ayon Chakraborty, Udit Gupta, and Samir R. Das. Benchmarking resource usage for spectrum sensing on commodity mobile devices. In *Proceedings of the 3rd Workshop on Hot Topics in Wireless*, HotWireless '16, pages 7–11, New York, NY, USA, 2016. ACM.
- [5] Ayon Chakraborty, Md Shaifur Rahman, Himanshu Gupta, and Samir R Das. Specsense: Crowdsensing for efficient querying of spectrum occupancy. In *IEEE INFOCOM*, 2017.
- [6] G. Ding, J. Wang, Q. Wu, L. Zhang, Y. Zou, Y. D. Yao, and Y. Chen. Robust spectrum sensing with crowd sensors. *IEEE Transactions on Communications*, 62(9):3129–3143, Sept 2014.
- [7] A. Dutta and M. Chiang. "See Something, Say Something": Crowdsourced Enforcement of Spectrum Policies. *IEEE Transactions on Wireless Communications*, 15(1):67–80, Jan 2016.
- [8] A. Ghasemi and E. S. Sousa. Optimization of spectrum sensing for opportunistic spectrum access in cognitive radio networks. In *2007 4th IEEE Consumer Communications and Networking Conference*, pages 1022–1026, Jan 2007.
- [9] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

- [10] Xiaocong Jin and Yanchao Zhang. Privacy-preserving crowdsourced spectrum sensing. In *IEEE INFOCOM '16*, San Francisco, CA, USA, April 2016.
- [11] M. A. McHenry, P. A. Tenhula, D. McCloskey, D. A. Roberson, and C. S. Hood. Chicago Spectrum Occupancy Measurements and Analysis and a Long-term Studies Proposal. Proc. of TAPAS Conference, Aug 2006.
- [12] Mark McHenry. Spectrum measurements requirements survey. http://www.cs.albany.edu/~mariya/nsf_smsmw/docs/NSF_SMIW_survey_results.pdf, April 6-7 2016.
- [13] Ana Nika, Zhijing Li, Yanzi Zhu, Yibo Zhu, Ben Y. Zhao, Xia Zhou, and Haitao Zheng. Empirical validation of commodity spectrum monitoring. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, SenSys '16, pages 96–108, New York, NY, USA, 2016. ACM.
- [14] Ana Nika, Zengbin Zhang, Xia Zhou, Ben Y. Zhao, and Haitao Zheng. Towards commoditized real-time spectrum monitoring. HotWireless '14, Maui, Hawaii, USA, 2014.
- [15] Illinois Institute of Technology. NSF Workshop on Spectrum Measurement Infrastructures, Chicago, IL, USA. http://www.cs.albany.edu/~mariya/nsf_smsmw/, April 6-7 2016.
- [16] M. Souryal, M. Ranganathan, J. Mink, and N. E. Ouni. Real-time centralized spectrum monitoring: Feasibility, architecture, and latency. In *IEEE DySPAN'15*, Stockholm, Sweden, 29 September - 2 October 2015.
- [17] X. Ying, S. Roy, and R. Poovendran. Incentivizing crowdsourcing for radio environment mapping with statistical interpolation. In *IEEE DySPAN'15*, Stockholm, Sweden, 29 September - 2 October 2015.
- [18] Tan Zhang, Ning Leng, and Suman Banerjee. A vehicle-based measurement framework for enhancing whitespace spectrum databases. MobiCom '14, Maui, Hawaii, USA, 2014.
- [19] Tan Zhang, Ashish Patro, Ning Leng, and Suman Banerjee. A wireless spectrum analyzer in your pocket. HotMobile '15, Santa Fe, New Mexico, USA, 2015.