Synthesizing Robust Training Data for Machine Learning

by

Kyle W. McClintick

A Thesis
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Master of Science
in
Electrical and Computer Engineering
by

December 2018

APPROVED:

Professor Alexander Wyglinski, Major Advisor

x

x

Abstract

Developing machine learning-based signal classifiers that generalize well requires training data that capture the underlying probability distribution of real signals. To synthesize a set of training data that can capture the large variance in signal characteristics, a robust framework that can support arbi- trary baseband signals and channel conditions is required. Furthermore, a classifier trained on a probability distribution that reflects the expected range of channel conditions is better able to detect anomalies (e.g., poisoning at- tacks).

Acknowledgements

I would like to express my deepest gratitude to my advisor Professor Alexander Wyglinski for his continuous guidance and support towards my degree. I am very thankful for the opportunity to work with him in the Wireless Innovation Laboratory at Worcester Polytechnic Institute.

I want to thank Professor Kaveh Pahlavan and Dr. Travis Collins for serving on my committee and providing valuable suggestions and comments with regards to my thesis.

I would also like to thank my WILab team members Dr. Srikanth Pagadarai, Kuldeep, Renato, and Nivetha for their immense support during my graduate studies. And I would also like to thank my friends abroad and in the states who have stayed in contact and given me the support I need. Finally, I'm thankful to my family: Colin, Dawn, and George. Without their constant support I wouldn't be here.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

motive

1.2 State of the Art

sota

1.3 Thesis Contributions

contributions

1.4 Thesis Organization

org

1.5 List of Related Publications

The following publications resulted from the activities of this thesis research:

- pub

Chapter 2

Data Synthesis and Neural Networks

intro

2.1 The Radio Frequency Front End

tx rx chain, complications, ways to correct

2.2 Classical Channel Models

list the modeling algorithms

2.3 Machine Learning Algorithms

loss functions, etc

2.4 Neural Network Architectures

layers, roles, types, etc

2.5 Summary

review and conclude

Chapter 3

Physical Layer Neural Network Framework for Training Data Formation

We present a novel open-source dataset modification framework designed to study the effect of wireless channels on the training and testing of modulation classifiers employing Neural Networks (NN). Communication systems optimize for capacity by packing information bits very closely together. Consequently, RF datasets contain less redundancy and context than other NN domains such as image and speech classification. As a result, NN performance is poor when brought to implementation if training is not done with datasets properly describing gathered test sets. Our framework pushes datasets through a sequential, parallelized, modular, block-style set of wireless channels, where blocks can be written by operators or pulled from a core library. Utilizing our datasets, we perform analysis of a NN. We determine the NN requires datasets collected by a receiver capable of phase correction to below 4 degrees offset, and Carrier Frequency Offset (CFO) correction to below **1**% normalized to sample rate to maintain near-peak modulation classification accuracy.

3.0.1 Introduction

There exist numerous approaches published in open literature to model wireless channels [1]. Receive chains apply an equally numerous variation of corrections to signals to remove or limit channel effects on data through frame synchronization, CFO correction, timing corrections, and other methods [2]. Contributions to the field are met with scrutiny due to the maturity of these models and correction systems. However, NNs can adjust for nonlinearities unaccounted for by classical communication and information theory [3]. Significant gains can be achieved by breaking the traditionally sequential nature of activities performed by transmit and receive chains [4]. Additionally, recent NN publications have shown high-capacity learning algorithms that have demonstrated high energy efficiency and computational throughput [5]. Consequentially, interest has been expressed [6, 7] for the use of NNs in physical layer communications. NN applications in the physical layer can be classified into two categories: end-to-end autoencoders that determine unknown hidden layers to recreate an input data layer at an output data layer, and NNs that determine specific design choices of the physical layer.

For any physical layer NN, an often used framework for dataset generation is GNU Radio Companion's (GRC) channel model blocks. Each channel block offers a modular, sequential, parallelized method of dataset manipulation. While a powerful tool, GRCs gpl3 license makes privacy of work difficult for companies with export control and Non-Disclosure Agreements (NDA). Most all blocks must be written by the user which require documentation, upkeep, and time. Additionally, GRC is better at some use cases than others, requiring creativity to generate some datasets. Most importantly, GRC cannot easily and properly construct massive arrays of varied wireless channels.

In [8], a dataset is generated for use in a NN modulation classifier, and interest is expressed for "iteratively better, more complex, and more challenging datasets", which "will be required to help compare, measure, and evaluate these (communication) tasks". It is the fundamental assumption made in supervised learning of a NN that the training data is representative of practical data. That assumption cannot be broken in data partitioning or through a lack of resolution in the distribution of training data. For many NNs, that

assumption is broken by the highly unique nature of wireless transmissions, which are varied by a virtually endless number of channel effects.

In this paper, we propose a framework that creates datasets satisfying this assumption through their size and diversity in a computationally efficient, parallelized method. Channel effects are sequentially applied to complex valued datasets in a modular way, where channel blocks are chosen by a configuration file from a library or user created blocks. Channels described by design parameters can be iterated through, achieving thorough file multiplication labeled in a clear and concise way.

This paper presents our framework in section 3.0.2, and discusses implementation details of training and testing in section 3.0.3. Section 3.0.4 displays simulation results, and concluding thoughts are presented in section 3.0.5.

3.0.2 Proposed Framework

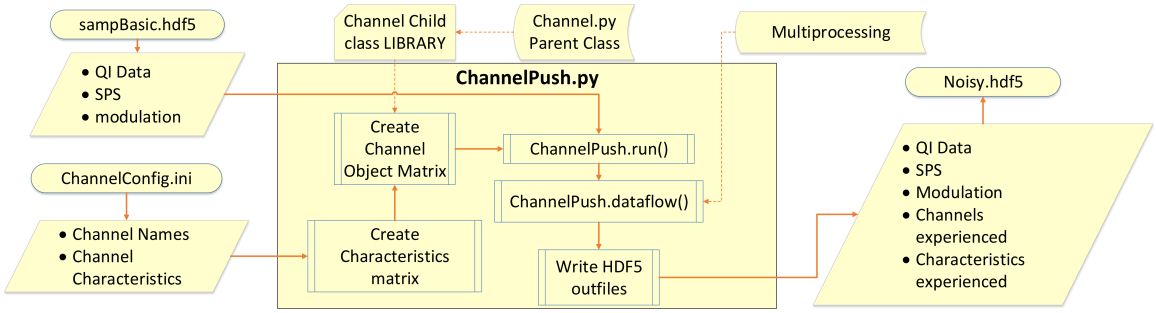

Figure 3.1: ChannelPush.py takes in sampBasic.hdf5 and ChannelConfig.ini as inputs, and writes one or many Noisey.hdf5 outfiles. In the create channel object matrix sub-process, the channel library is used to initialize channel objects, which inherit from the channel parent class. The dataflow() function sequentially pushes data through each list of channels in parallel, making use of the Multiprocessing package.

Section 3.0.4 of this paper and [9] display a clear need to consider the radio front end and wireless channel effects in any NN experiment. NNs must either be trained with datasets effected by channel effects or tested with datasets gathered by receivers that correct for channel effects to maintain near-peak accuracy. Either condition must be met to maintain the fundamental training assumption by expanding training datasets to represent test datasets,

or to bound test datasets to represent training datasets. The latter is often preferable, as more complex training datasets always yield lower peak accuracy. Our framework generates supersets of data that can be used to help make this training decision, and to what extent training complexity should be taken for different wireless channel phenomenon. By testing NNs with supersets from our framework, certain physical layer corrections may be revealed to be unnecessary, inadequate, or overly-complex. As an example, a NN experiment makes use of a frame synchronization block created by the user to create a superset from their Dataset Under Test (DUT). A resulting conclusion on their NNs performance might be that a longer Barker Code must be used, reducing data throughput, or training must be done with a greater range of frame synchronization errors, reducing peak NN performance. This might not be an easy decision, requiring multiple uses of our framework to hypothesize, train, and evaluate NN performance and data throughput.

The inputs and outputs of the framework are described in Figure 3.1. The framework requires a DUT and instructions for which channel blocks to push the DUT through. Framework outputs are instances of the DUT that have been modified by a unique combination of channel block realizations. Any type or amount of channel blocks may be used from a library by editing the instructions. The DUT is pushed through channel blocks one sample at a time. Channel objects may be added, modified, or removed from the framework by minimal editing of the instructions. Each output is computed and written in a separate Central Processing Unit (CPU) process.

To experiment with supersets created with our framework, training and testing is done in [10] with Keras2, an Application programming interface (API). TensorFlow, an open-source software library that makes use of data flow graphs to perform numerical computations, is used as the backend. In our training and testing, the DUT is divided into training and testing sets, making sure there is no overlap of values. We utilize a sequential NN, which is composed of layers which only interact with their neighboring layers. Common sequential NN layers include reshape, zeropadding2D, convolutional2D, dropout, flatten, activation, and dense layers. Reshape layers adjust the height and width of layer nodes structures and batch size (which determine the number of training parameters). Zeropadding2D adds zero values to node's sides, tops, and bottoms. Convolutional2D inputs are convolved over 2

dimensions to produce a tensor of outputs functioning as hidden layers between input and output layers. Dropout layers randomly set a fraction of the inputs to 0 at each update during training to prevent overfitting. Flatten layers reduce the dimension of the input, often to reduce the last hidden layer's output shape to the smaller output layer's input shape. The activation and dense layers apply a non-linear function to inputs, most popularly a Mean Square Error (MSE) function.

Perhaps the most notable layer here is the activation layer, which applies a non-linear function to the layers inputs. During the most popular method of training, Stochastic Gradient Descent (SGD), training parameters are determined by an algorithm involving a loss function approximation. That approximation is described as a batch-normalized sum of non-linear functions. For an overview of Machine Learning (ML), the reader is directed towards section II of [6]

3.0.3 Implementation

In development, sampBasic.hdf5 has served as our DUT, a dataset of complex values representing constellation points of equiprobable random binary bits. In the hdf5 format, data can be assigned to subsets, and each subset, or the whole hdf5 file, assigned labels in the form of ['Key'] = value. Labels utilized are the Samples Per Symbol (SPS) or oversampling factor of the data, the source file of the dataset, the version of hdf5 the file is source encoded with, and the modulation type of the dataset. ChannelConfig.ini represents the configuration for the framework used in development. The .ini format follows a key = value format, organized under headers [HEADER], where all values are strings. Channels are described by keys 'channelx' under the [CHANNELS] header and 'characteristicsx' under the [CHARACTERISTICS] header. Each channel's characteristic parameters are '—' separated, while a sweep over that parameter's values are comma separated. Each channel realization writes a uniquely modified version of the DUT as noiseyx.hdf5. This file multiplication can extend over any number of sequential channel objects, as seen in Figure 3.2. Each output file utilizes hdf5's labeling capabilities, detailing the types of channels and parameter values of those channels that each file was pushed through in the form ['channel:parameter'] = value.

After the configuration file is imported, delimiters convert the configuration into a 3D

matrix. Within this structure, each primary index signals a channel block type, the second a parameter, and the tertiary index a value of that parameter.

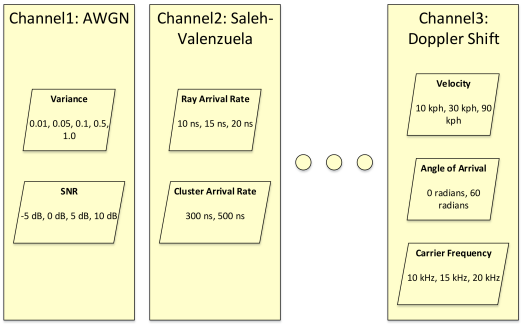

Figure 3.2: Example 2D channel matrix built from ChannelConfig.ini. There exist 20 AWGN, 6 Saleh-Valenzuela, and 18 Doppler Shift channel objects in this example, which will result in 2160 hdf5 files written. Each dataset features a unique combination of the 3 wireless channels modifying the same input dataset, with labels describing which realization was experienced

From that 3D matrix, a 2D matrix of channel objects is initialized, where the primary list signals grouping of channel types, and the secondary list is each realization of that channel type. Channel classes share similarities in the way they handle states, inputs, and outputs, so they inherit from a parent Channel class. A template exists to add a child channel class to the core library. in the run() function, the 2D matrix is fragmented into 1D lists of channel objects, where the total set of lists represents every possible sequential combination of unique channel realizations such that there is at least one of each channel type in each list. The next function call, dataflow(), sequentially modifies the input data by each channel of each list. This task, as well as file writing and attaching labels, is performed in parallel using the Multiprocessing import.

We modify [10] to run with a python3.5 interpreter instead of 2.7, using TensorFlow as a backend instead of Theano, as in the journal. The NN structure used is identical to theirs besides the dimensions of our datasets (see Table 3.1). The dimension size change is due to a difference in Samples Per Symbol (SPS). The input layer has an output shape of (None, 1, 2, 4), having 4 channels as opposed to 128. This represents 4 samples of each

Table 3.1: The structure of the NN used in section 3.0.4. Each row represents a layer, where output shape is displayed as (batch, height, width, channels) and (batch_size, input_dim). Each layer is connected to the previous layer, and number of params is proportional to SPS and the size of convolutional layers.

Layer (type)	Output Shape	Param	Connected to
reshape_1 (Reshape)	(None, 1, 2, 4)	0	reshape_input_1[0][0]
zeropadding2d_1 (ZeroPadding2D)	(None, 1, 2, 8)	0	reshape_1[0][0]
conv1 (Convolution2D)	(None, 256, 2, 6)	1024	zeropadding2d_1[0][0]
dropout_1 (Dropout)	(None, 256, 2, 6)	0	conv1[0][0]
zeropadding2d_2 (ZeroPadding2D)	(None, 256, 2, 10)	0	dropout_1[0][0]
conv2 (Convolution2D)	(None, 80, 1, 8)	122960	zeropadding2d_2[0][0]
dropout_2 (Dropout)	(None, 80, 1, 8)	0	conv2[0][0]
flatten_1 (Flatten)	(None, 640)	0	dropout_2[0][0]
dense1 (Dense)	(None, 256)	164096	flatten_1[0][0]
dropout_3 (Dropout)	(None, 256)	0	dense1[0][0]
dense2 (Dense)	(None, 11)	2827	dropout_3[0][0]
activation_1 (Activation)	(None, 11)	0	dense2[0][0]
reshape_2 (Reshape)	(None, 11)	0	activation_1[0][0]
Total params 290,907			
Trainable params 290,907			
Non-trainable params 0			

symbol in our datasets rather than 128, where the data transmitted remains constant but probabilistic channel effects vary.

3.0.4 Simulation and Results

In this paper, we focus on the wireless channel effect of CFO and phase ambiguity due to their computational simplicity. In any communication system, the transmitter and receiver typically are not using the same Local Oscillator (LO). This leads to issues in the baseband

when a received signal is demodulated from its carrier frequency, proportional to the max possible frequency offset:

$$f_{o,max} = \frac{f_c \times PPM}{10^6} \tag{3.1}$$

where PPM is the maximum possible Parts Per Million offset of the LO, and f_c is the received signal carrier frequency. We investigate the performance of [11] when the 2016.10a dataset has the additional channel effect of CFO, uncorrected by classical Coarse Frequency Correction (CFC) and Fine Frequency Correction (FFC) methods. As shown in [9], testing done against channel effects not trained with causes classification accuracy to significantly decrease. We have observed a relationship between the complexity of the environment a NN is trained for and peak accuracy. Consequently, it should be the goal of any additional complexity in NN physical layer datasets to represent expected channel effects realistically. To obtain a realistic PDF of CFOs for training, ideally an infinite number of random values would maximize accuracy. However, training is very computationally expensive, so such a goal is unrealizable, particularly when the proposed framework is considered (as each additional channel type can represent one, or several, orders magnitude additional samples). Considering this, the datasets in Table 3.2 are presented to learn more about practical performance of the classifier [11].

Table 3.2: Four datasets generated with our framework. Datasets are trained, partitioned, and tested against a modulation classifier.

Dataset	Description
RML_4sps	RML2016.10a dataset at 4 SPS and 0 CFO
CFO_grc	RML_4sps trained with GRC's dynamic channel
CFO_super	RML_4sps file multiplied over 100 discrete CFOs channels evenly spaced from 0 to 20% normalized to sample rate
Phase_super	RML_4sps file multiplied over 100 phase ambiguities evenly spaced from 0 to 2π radians

The RML2016.10a dataset contains 1000 symbols, oversampled by a factor of 128, passed

through GRC's dynamic channel model object, which has an 8-tap multipath profile of K-factor 4, fixed Doppler shift 1 Hz, random STO of variance 0.01 max value 50 samples, and random CFO of variance 0.01 max value 0.25% normalized to 200 kHz sample rate. The dataset contains 11 modulation schemes (8 digital, 3 analog). SNR is swept from -20 dB to 20 dB in 2 dB steps. The CFO alter each sample of each dataset with the following phase rotation per sample:

$$\Theta[i] = e^{j\frac{0.2\times i}{N}\frac{2\pi}{sps}} \tag{3.2}$$

where N is the number of CFO values, $i = 0, 1...N-1$, and sps is the oversampling factor. Since frequency is known as phase over time, the phase rotation at sample k in a dataset for constant SNR, modulation type, and phase rotation per sample can be written as:

$$\phi[k,i] = \Theta[i]^k,\ \ k = 1, 2...M \tag{3.3}$$

where M is the number of samples in a transmission with the above mentioned constants, and ϕ is only a function of i for the phenomenon of initial phase offset. The CFO_super test set aims to provide complete coverage of all possible CFO outcomes, making testing more reproducible and consistent. In Figure 3.3, we present the accuracy of [11] in classifying the modulation scheme of data trained by RML_4sps and tested by CFO_super, data trained by CFO_grc in Figure 3.4, and data trained by RML_4sps and tested by Phase_super in Figure 3.6.

Since the classifier is shown in confusion matrices to guess AM-SSB when SNR is low, resulting classifier accuracy nears 9.09%. The accuracy values peak when there is no CFO and sharply decreases as CFO grows in Figure 3.3. The small improvement in accuracy at high SNR values and CFO values of 12.53% and 17.78% in Figure 3.4 are due to the classifier correctly predicting the modulation scheme is Continuous Phase Frequency Shift Keying (CPFSK). Although accuracy improvement is small, if the classifier is only used to decide between CPFSK and a few other modulation schemes rather than 11, the accuracy increase would be much more significant, perhaps enough for the CFO correction to be relaxed or removed. Figure 3.3 reveals that the classifier does not perform with more than 0.2% CFO

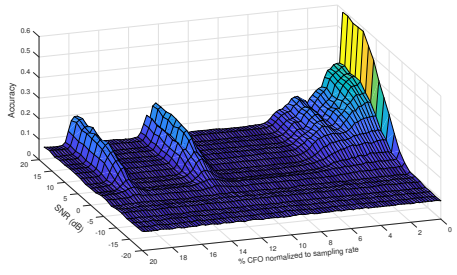

Figure 3.3: CFO_super test results when trained with RML_4sps. Accuracy is averaged over 11 modulation schemes, and significantly decreases when tested against a CFO channel of just 0.2% normalized to sample rate. Accuracy peaks at 60%.

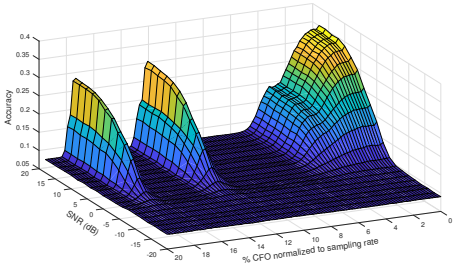

Figure 3.4: CFO_super test results when trained with CFO_grc, as in the original 2016 dataset. Accuracy is averaged over 11 modulation schemes, and significantly decreases when tested against CFO channels of more than 1% normalized to sample rate. Accuracy peaks at 38%.

when trained with RML_4sps, which is problematic. Figure 3.4 displays, as shown in [9], the peak accuracy dropping due to the added complexity of the data from about 60% to 40%, but that accuracy now stretches to 1% CFO normalized to sample rate. A training decision here is to identify the CFO (3.1) of the system used in experimentation, and train the classifier for CFOs no greater than that to maintain peak accuracy.

Given the uniform distribution of initial phase offset in radio communications, Figure 3.6 reveals significant drops in accuracy at many phase offset values. Consequently, this classifier would likely require phase correction through either codewords, differential encoding, or equalization to see consistent near-peak performance.

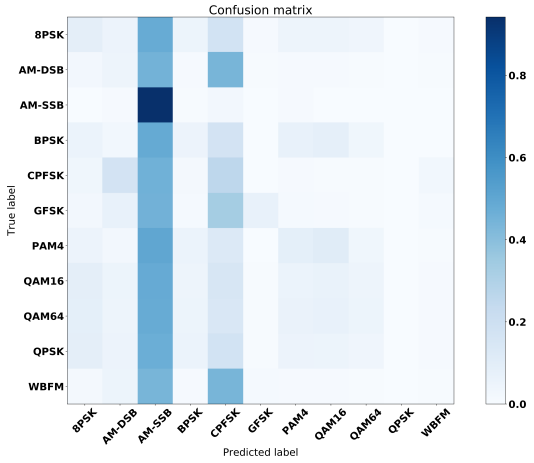

Figure 3.5: CFO_super confidence matrix for 12.53% CFO averaged over SNR, see Figure 3.4. AM-SSB is the predicted label at low SNR values as when there is no CFO, and often correctly predicts CPFSK at higher SNR values. This seems to imply certain periodic CFO values do not affect the appearance of CPFSK constellation plots.

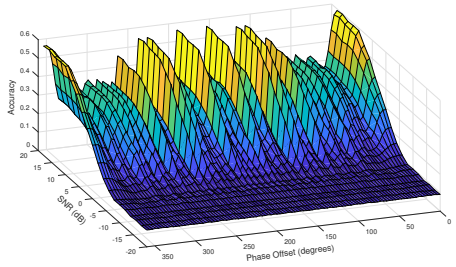

Figure 3.6: Phase_super test results when trained with RML_4sps. Accuracy is averaged over 11 modulation schemes, and remains close to peak performance for phase offsets of less than 4 degrees and certain discretized values. 60% peak accuracy can drop by as much as half for many phase offset values.

3.0.5 Summary

In section 3.0.1, we described the usefulness of NNs in the communications community, and referenced an overview of recent publications. In section 3.0.2 the frameworks's structure was discussed, and the implementation details of RadioML and our framework were visited in section 3.0.3. Finally, section 3.0.4 displayed simulation results using NN datasets

modified by our framework.

Our proposed framework showcases an ability to identify NN testing and training decisions, and hypothesize and test for answers to what extent training complexity should be taken for different wireless channel phenomenon. Future work aims to generate multi-user supersets from lists of DUTs, where channel blocks applied to different transmissions are related by correlation factors.

Chapter 4

Proposed LTE-R Channel Model and Framework

abstract

4.1 test

test

4.2 Summary

summary

Chapter 5

Proposed Heterogeneous CSS Prototype

abstract

5.0.1 test

test

5.1 Summary

summary

Chapter 6

Conclusion

test

6.1 Research Outcomes

test

6.2 Future Work

test

Bibliography

[1] TIA, *Wireless Communications Systems Performance in Noise and Interference Limited Situations Part 2: Propagation and Noise.* Telecommuncations Industry Association, 2016, vol. 2, revision E.

[2] T. S. Rappaport *et al.*, *Wireless communications: principles and practice.* Prentice Hall PTR New Jersey, 1996, vol. 2.

[3] T. Schenk, *RF imperfections in high-rate wireless systems: impact and digital compensation.* Springer Science & Business Media, 2008.

[4] H. Wymeersch, *Iterative receiver design.* Cambridge University Press Cambridge, 2007, vol. 234.

[5] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *Proceedings of the 26th Annual International Conference on Machine Learning.* ACM, 2009, pp. 873–880.

[6] T. O'Shea and J. Hoydis, "An introduction to deep learning for the physical layer," *IEEE Transactions on Cognitive Communications and Networking*, vol. 3, no. 4, pp. 563–575, Dec 2017.

[7] T. Wang, C.-K. Wen, H. Wang, F. Gao, T. Jiang, and S. Jin, "Deep learning for wireless physical layer: Opportunities and challenges," *China Communications*, vol. 14, no. 11, pp. 92–111, 2017.

[8] T. J. O'Shea and N. West, "Radio machine learning dataset generation with gnu radio," in *Proceedings of the GNU Radio Conference*, vol. 1, no. 1, 2016.

[9] S. C. Hauser, W. C. Headley, and A. J. Michaels, "Signal detection effects on deep neural networks utilizing raw iq for modulation classification," in *MILCOM 2017 - 2017 IEEE Military Communications Conference (MILCOM)*, Oct 2017, pp. 121–127.

[10] T. J. O'Shea, J. Corgan, and T. C. Clancy, "Convolutional radio modulation recognition networks," *arXiv preprint arXiv:1602.04105*, 2016.

[11] T. J. O'Shea, J. Corgan, and T. C. Clancy, "Convolutional radio modulation recognition networks," in *International Conference on Engineering Applications of Neural Networks*. Springer, 2016, pp. 213–226.

Appendix A

Heterogeneous Cooperative Spectrum Sensing Code

A.1 harddecisionpdroc.m

```
% Hard-Decision Combining Results For Sensor Nodes
clc;
close all;
clear all;
%% Parameter Initialization
N = 32;
k=4;%sensor nodes..
variance = 24.32e-9;
pfa = 0.05;
threshold = (qfuncinv(pfa)+sqrt(N)).*sqrt(N)*2*variance;
snrthreotical = -18:0.5:20;
snrlinear = 10.^(snrthreotical/10);
%% SNR values from USRP and RTL-SDR
snrpracticalavg = [-18.23,-14.22,-13.1,-12.22,-10.35,-8.25,-5.3,-2.1,0.13,...
    1.34,2.58,3.76,8.98,11.33,12.35,13.45,15.77];
snrlinearprac = 10.^(snrpracticalavg/10);
%% Computing Detection Probability and ROC Characteristics
```

```
pdprac = qfunc((threshold-2*N*variance.*(1+snrlinearprac))./...
    (sqrt(N.*(1+2*snrlinearprac))*(2*variance)));
pdpracor = 1-(1-pdprac).^4;
pdpracand = pdprac.^4;
tmp1 =  (1-pdprac).^2;
tmp2 = (1-pdprac);
pdpracmjr = (6*pdprac.^2).*tmp1+(4*pdprac.^3).*tmp2+pdprac.^4;

pfapracor = 1-(1-pfa).^4;
pfapracand = pfa.^4;
tmp1 =  (1-pfa).^2;
tmp2 = (1-pfa);
pfapracmjr = (6*pfa.^2).*tmp1+(4*pfa.^3).*tmp2+pfa.^4;

%% ROC Characteristics...
figure(1)
hold on;
grid on;
plot(pfapracor,pdpracor(:,5),'-->','LineWidth',2,'MarkerFaceColor','auto');
plot(pfapracor,pdpracor(:,7),'-->','LineWidth',2,'MarkerFaceColor','auto');
plot(pfapracand,pdpracand(:,5),'-d','LineWidth',2,'MarkerFaceColor','auto');
plot(pfapracand,pdpracand(:,7),'-d','LineWidth',2,'MarkerFaceColor','auto');
plot(pfapracmjr,pdpracmjr(:,5),'--','LineWidth',2,'MarkerFaceColor','auto');
plot(pfapracmjr,pdpracmjr(:,7),'--','LineWidth',2,'MarkerFaceColor','auto');
xlabel('Average Probability Of False Alarm');
ylabel('Average Probability of Detection');
title('ROC Characterisitcs of Hard Decision Combining');
hold off;
set(gca,'fontsize',30,'box','on','LineWidth',2,'GridLineStyle','--','GridAlpha'
    ,0.7);
lgd = legend('OR SNR=-10.35dB','OR SNR=-5.3dB','AND SNR=-10.35dB',...
    'AND SNR=-5.3dB','Majority SNR=-10.35dB','Majority SNR=-5.3dB');
lgd.FontSize=20;

%% Probability of detection
figure(2)
hold on;
```

```
grid on;
plot(snrpracticalavg,pdpracor,'->','LineWidth',2);
plot(snrpracticalavg,pdpracand,'-<','LineWidth',2);
plot(snrpracticalavg,pdpracmjr,'-d','LineWidth',2);
xlabel('SNR_{avg} in (dB)');
ylabel('Average Probability of Detection');
title('P_{davg} vs SNR_{avg} for Hard Decision Combining');
hold off;
set(gca,'fontsize',30,'box','on','LineWidth',2,'GridLineStyle','--','GridAlpha'
    ,0.7);
lgd = legend('OR Decision','AND Decision','Majority Rule Decision');
lgd.FontSize=20;
axis([-18.23 15.77 0 1])
```

A.2 softharddecisionpd.m

```
% Soft Decision Combining for sensor nodes...
%% Initializing parameters..
close all;
clear all;
N = [100,200,300,400];% Different sum factor
k=4;% Number of Sensor Nodes
variance = [24.025e-9,23.695e-9,25.678e-9,0.0323e-9];
pfa = 0.05;%Probability of false alarm
for i=1:4
threshold(i) = (qfuncinv(pfa)+sqrt(N(i)))*sqrt(N(i))*2*variance(i);
end
% SNR Values from four sensor nodes
snrpractical = [-21.45,-18.23,-15.45,-13.3,-12.67,-9.35,-2.23,-4.32,2.98,6.95,13
    .57,21.78;...
                -22.23,-20.22,-17.34,-15.32,-13.45,-11.27,-7.75,-5.67,2.53,4.78
                    ,12.67,20.32;...
                -25.34,-23.34,-21.67,-20.33,-16.76,-13.38,-8.56,-6.53,0.38,1.34
                    ,7.89,16.54;...
                -27.32,-24.97,-22.34,-22.23,-17.34,-14.32,-11.35,-7.85,-2.35,-0
```

```
                    .98,2.38,5.98];
for i=1:4
    snrlinearprac(i,:) = 10.^(snrpractical(i,:)/10);
end
for i=1:4
pdprac(i,:) = qfunc((threshold(i)-2*N(i)*variance(i).*(1+snrlinearprac(i,:))./
    ...
    (sqrt(N(i)*(1+2*snrlinearprac(i,:)))*(2*variance(i))));
end
for i=1:12
snravg(i) = mean(snrpractical(:,i));
end
%% MNE based CS..
pdpracmne = 1-(1-pdprac(1,:)).*(1-pdprac(2,:)).*(1-pdprac(3,:)).*(1-pdprac(4,:))
    ;
pdpracand = mean(pdprac).^4;
pdpracm = mean(pdprac);
pdpracor = 1-(1-pdpracm).^4;
tmp1 =  (1-pdpracm).^(k-2);
tmp2 = (1-pdpracm);
pdpracmjr = (6*pdpracm.^(k-2)).*tmp1+(4*pdpracm.^(k-1)).*tmp2+pdpracm.^k;
figure(1)
hold on;
grid on;
plot(snravg,pdpracmne,'-<','LineWidth',2,'MarkerFaceColor','auto');
axis([-20 10 0 1])
%% EGC based CS..
snrlinearmean = 10.^(snravg/10);
snrlinear = 10.^(snrpractical/10);
pfa = 0.01;
M= mean(N);
threshold = mean(threshold);
for i=1:length(snravg)
    num(i) = threshold-snrlinearmean(i);
    den(i) = (1/16)*((1+2*snrlinear(1,i))/N(1)+variance(1)+(1+2*snrlinear(2,i))/
        N(2)+variance(2)+(1+2*snrlinear(3,i))/N(3)+variance(3)+(1+2*snrlinear(4,
        i))/N(4)+variance(4));
```

```
    pdegc(i) = qfunc(num(i)/sqrt(den(i)));
end

%% Plotting the data..
plot(snravg,pdegc,'-d','LineWidth',2,'MarkerFaceColor','auto');
plot(snravg,pdpracand,'-s','LineWidth',2,'MarkerFaceColor','auto');
plot(snravg,pdpracor,'-s','LineWidth',2,'MarkerFaceColor','auto');
plot(snravg,pdpracmjr,'->','LineWidth',2,'MarkerFaceColor','auto');
title('P_{davg} vs SNR_{avg} for Soft and Hard Decision Combining');
xlabel('SNR_{avg} in (dB)');
ylabel('Average Probability of Detection');
set(gca,'fontsize',30,'box','on','LineWidth',2,'GridLineStyle','--','GridAlpha'
    ,0.7);
legend('MNE Combining','EGC Combining','AND Rule','OR Rule','Majority Rule');
```

A.3 spectrumsenseusrp.py

```
#!/usr/bin/env python
#
# Copyright 2005,2007,2011 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# GNU Radio is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 3, or (at your option)
# any later version.
#
# GNU Radio is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with GNU Radio; see the file COPYING.  If not, write to
```

```
# the Free Software Foundation, Inc., 51 Franklin Street,
# Boston, MA 02110-1301, USA.
#

from gnuradio import gr, eng_notation
from gnuradio import blocks
from gnuradio import audio
from gnuradio import filter
from gnuradio import fft
from gnuradio import uhd
from gnuradio.eng_option import eng_option
from optparse import OptionParser
import sys
import math
import struct
import threading
from datetime import datetime
import time
from gnuradio.wxgui import stdgui2, fftsink2, form
import wx

sys.stderr.write("Warning: this may have issues on some machines+Python version
    combinations to seg fault due to the callback in bin_statitics.\n\n")

class ThreadClass(threading.Thread):
    def run(self):
        return

class tune(gr.feval_dd):
    """
    This class allows C++ code to callback into python.
    """
    def __init__(self, tb):
        gr.feval_dd.__init__(self)
        self.tb = tb

    def eval(self, ignore):
```

```
        """
        This method is called from blocks.bin_statistics_f when it wants
        to change the center frequency.  This method tunes the front
        end to the new center frequency, and returns the new frequency
        as its result.
        """

        try:
            new_freq = self.tb.set_next_freq()
            while(self.tb.msgq.full_p()):
                time.sleep(0.1)
            return new_freq

        except Exception, e:
            print "tune: Exception: ", e


class parse_msg(object):
    def __init__(self, msg):
        self.center_freq = msg.arg1()
        self.vlen = int(msg.arg2())
        assert(msg.length() == self.vlen * gr.sizeof_float)
        t = msg.to_string()
        self.raw_data = t
        self.data = struct.unpack('%df' % (self.vlen,), t)


class my_top_block(gr.top_block):

    def __init__(self):
        gr.top_block.__init__(self)

        usage = "usage: %prog [options] min_freq max_freq"
        parser = OptionParser(option_class=eng_option, usage=usage)
        parser.add_option("-a", "--args", type="string", default="",
                          help="UHD device device address args [default=%default
                              ]")
```

```
parser.add_option("", "--spec", type="string", default=None,
                  help="Subdevice of UHD device where appropriate")
parser.add_option("-A", "--antenna", type="string", default=None,
                  help="select Rx Antenna where appropriate")
parser.add_option("-s", "--samp-rate", type="eng_float", default=1e6,
                  help="set sample rate [default=%default]")
parser.add_option("-g", "--gain", type="eng_float", default=None,
                  help="set gain in dB (default is midpoint)")
parser.add_option("", "--tune-delay", type="eng_float",
                  default=0.25, metavar="SECS",
                  help="time to delay (in seconds) after changing
                      frequency [default=%default]")
parser.add_option("", "--dwell-delay", type="eng_float",
                  default=0.25, metavar="SECS",
                  help="time to dwell (in seconds) at a given frequency
                      [default=%default]")
parser.add_option("-b", "--channel-bandwidth", type="eng_float",
                  default=6.25e3, metavar="Hz",
                  help="channel bandwidth of fft bins in Hz [default=%
                      default]")
parser.add_option("-l", "--lo-offset", type="eng_float",
                  default=0, metavar="Hz",
                  help="lo_offset in Hz [default=%default]")
parser.add_option("-q", "--squelch-threshold", type="eng_float",
                  default=None, metavar="dB",
                  help="squelch threshold in dB [default=%default]")
parser.add_option("-F", "--fft-size", type="int", default=None,
                  help="specify number of FFT bins [default=samp_rate/
                      channel_bw]")
parser.add_option("", "--real-time", action="store_true", default=False,
                  help="Attempt to enable real-time scheduling")

(options, args) = parser.parse_args()
if len(args) != 2:
    parser.print_help()
    sys.exit(1)
```

```
self.channel_bandwidth = options.channel_bandwidth

self.min_freq = eng_notation.str_to_num(args[0])
self.max_freq = eng_notation.str_to_num(args[1])

if self.min_freq > self.max_freq:
    # swap them
    self.min_freq, self.max_freq = self.max_freq, self.min_freq

if not options.real_time:
    realtime = False
else:
    # Attempt to enable realtime scheduling
    r = gr.enable_realtime_scheduling()
    if r == gr.RT_OK:
        realtime = True
    else:
        realtime = False
        print "Note: failed to enable realtime scheduling"

# build graph
self.u = uhd.usrp_source(device_addr=options.args,
                         stream_args=uhd.stream_args('fc32'))

# Set the subdevice spec
if(options.spec):
    self.u.set_subdev_spec(options.spec, 0)

# Set the antenna
if(options.antenna):
    self.u.set_antenna(options.antenna, 0)

self.u.set_samp_rate(options.samp_rate)
self.usrp_rate = usrp_rate = self.u.get_samp_rate()

self.lo_offset = options.lo_offset
```

```
    if options.fft_size is None:
        self.fft_size = int(self.usrp_rate/self.channel_bandwidth)
    else:
        self.fft_size = options.fft_size

    self.squelch_threshold = options.squelch_threshold

    s2v = blocks.stream_to_vector(gr.sizeof_gr_complex, self.fft_size)

    mywindow = filter.window.blackmanharris(self.fft_size)
    ffter = fft.fft_vcc(self.fft_size, True, mywindow, True)
    power = 0
    for tap in mywindow:
        power += tap*tap
    c2mag = blocks.complex_to_mag_squared(self.fft_size)
    self.freq_step = self.nearest_freq((0.75 * self.usrp_rate),
        self.channel_bandwidth)
    self.min_center_freq = self.min_freq + (self.freq_step/2)
    nsteps = math.ceil((self.max_freq - self.min_freq) / self.freq_step)
    self.max_center_freq = self.min_center_freq + (nsteps * self.freq_step)
    self.next_freq = self.min_center_freq
    tune_delay  = max(0, int(round(options.tune_delay * usrp_rate /
        self.fft_size)))  # in fft_frames
    dwell_delay = max(1, int(round(options.dwell_delay * usrp_rate /
        self.fft_size))) # in fft_frames
    self.msgq = gr.msg_queue(1)
    self._tune_callback = tune(self)        # hang on to this to keep it
        from being GC'd
    stats = blocks.bin_statistics_f(self.fft_size, self.msgq,
                                    self._tune_callback, tune_delay,
                                    dwell_delay)
self.connect(self.u, s2v, ffter, c2mag, stats)

    if options.gain is None:

        g = self.u.get_gain_range()
        options.gain = float(g.start()+g.stop())/2.0
```

```
    self.set_gain(options.gain)
    print "gain =", options.gain

def set_next_freq(self):
    target_freq = self.next_freq
    self.next_freq = self.next_freq + self.freq_step
    if self.next_freq >= self.max_center_freq:
        self.next_freq = self.min_center_freq

    if not self.set_freq(target_freq):
        print "Failed to set frequency to", target_freq
        sys.exit(1)

    return target_freq


def set_freq(self, target_freq):
    """
    Set the center frequency we're interested in.

    Args:
        target_freq: frequency in Hz
    @rypte: bool
    """

    r = self.u.set_center_freq(uhd.tune_request(target_freq, rf_freq=(
        target_freq + self.lo_offset),rf_freq_policy=
        uhd.tune_request.POLICY_MANUAL))
    if r:
        return True

    return False

def set_gain(self, gain):
    self.u.set_gain(gain)
```

```
    def nearest_freq(self, freq, channel_bandwidth):
        freq = round(freq / channel_bandwidth, 0) * channel_bandwidth
        return freq

def main_loop(tb):

    def bin_freq(i_bin, center_freq):
        freq = center_freq - (tb.usrp_rate / 2) + (tb.channel_bandwidth * i_bin)
        return freq

    bin_start = int(tb.fft_size * ((1 - 0.25) / 2))
    bin_stop = int(tb.fft_size - bin_start)
    fid = open("./usrp.dat","wb")
    while 1:
    m = parse_msg(tb.msgq.delete_head())
        for i_bin in range(bin_start, bin_stop):
            center_freq = m.center_freq
            freq = bin_freq(i_bin, center_freq)
        power_db = 10*math.log10(m.data[i_bin]/tb.usrp_rate)
        signal = m.data[i_bin]/(tb.usrp_rate)

            if (power_db > tb.squelch_threshold) and (freq >= tb.min_freq) and (
                freq <= tb.max_freq):
        print freq,signal,power_db
        fid.write(struct.pack('<f',signal))
    fid.close()#closing the file

if __name__ == '__main__':
    t = ThreadClass()
    t.start()

    tb = my_top_block()
    try:
        tb.start()
        main_loop(tb)

    except KeyboardInterrupt:
```

```
        pass
```

A.4 gnuradiortlsdrsense.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
##################################################
# GNU Radio Python Flow Graph
# Title: DTv Spectrum Sensing
# Author: Gill
# Description: Frequency Sweep for UHF White Spaces
# Generated: Fri Mar 10 14:30:20 2017
##################################################

if __name__ == '__main__':
    import ctypes
    import sys
    if sys.platform.startswith('linux'):
        try:
            x11 = ctypes.cdll.LoadLibrary('libX11.so')
            x11.XInitThreads()
        except:
            print "Warning: failed to XInitThreads()"

from PyQt4 import Qt
from gnuradio import blocks
from gnuradio import eng_notation
from gnuradio import fft
from gnuradio import gr
from gnuradio import qtgui
from gnuradio.eng_option import eng_option
from gnuradio.fft import window
from gnuradio.filter import firdes
from optparse import OptionParser
import numpy as np
```

```
import osmosdr
import sip
import sys
import time


class spectrum_sensing(gr.top_block, Qt.QWidget):

    def __init__(self):
        gr.top_block.__init__(self, "DTv Spectrum Sensing")
        Qt.QWidget.__init__(self)
        self.setWindowTitle("DTv Spectrum Sensing")
        try:
            self.setWindowIcon(Qt.QIcon.fromTheme('gnuradio-grc'))
        except:
            pass
        self.top_scroll_layout = Qt.QVBoxLayout()
        self.setLayout(self.top_scroll_layout)
        self.top_scroll = Qt.QScrollArea()
        self.top_scroll.setFrameStyle(Qt.QFrame.NoFrame)
        self.top_scroll_layout.addWidget(self.top_scroll)
        self.top_scroll.setWidgetResizable(True)
        self.top_widget = Qt.QWidget()
        self.top_scroll.setWidget(self.top_widget)
        self.top_layout = Qt.QVBoxLayout(self.top_widget)
        self.top_grid_layout = Qt.QGridLayout()
        self.top_layout.addLayout(self.top_grid_layout)

        self.settings = Qt.QSettings("GNU Radio", "spectrum_sensing")
        self.restoreGeometry(self.settings.value("geometry").toByteArray())

        ##################################################
        # Variables
        ##################################################
        self.samp_rate = samp_rate = int(2e6)
        self.freq = freq = 450e6
        self.N = N = 1000
```

```
##################################################
# Blocks
##################################################
self.rtlsdr_source_0 = osmosdr.source( args="numchan=" + str(1) + " " +
    '' )
self.rtlsdr_source_0.set_time_source('external', 0)
self.rtlsdr_source_0.set_sample_rate(samp_rate)
self.rtlsdr_source_0.set_center_freq(freq, 0)
self.rtlsdr_source_0.set_freq_corr(0, 0)
self.rtlsdr_source_0.set_dc_offset_mode(2, 0)
self.rtlsdr_source_0.set_iq_balance_mode(0, 0)
self.rtlsdr_source_0.set_gain_mode(True, 0)
self.rtlsdr_source_0.set_gain(15, 0)
self.rtlsdr_source_0.set_if_gain(15, 0)
self.rtlsdr_source_0.set_bb_gain(15, 0)
self.rtlsdr_source_0.set_antenna('', 0)
self.rtlsdr_source_0.set_bandwidth(0, 0)

self.qtgui_freq_sink_x_0 = qtgui.freq_sink_c(
    1024, #size
    firdes.WIN_BLACKMAN_hARRIS, #wintype
    0, #fc
    samp_rate, #bw
    "Recieved Signal", #name
    1 #number of inputs
)
self.qtgui_freq_sink_x_0.set_update_time(0.10)
self.qtgui_freq_sink_x_0.set_y_axis(-120, 0)
self.qtgui_freq_sink_x_0.set_y_label('Relative Gain', 'dB')
self.qtgui_freq_sink_x_0.set_trigger_mode(qtgui.TRIG_MODE_FREE, 0.0, 0,
    "")
self.qtgui_freq_sink_x_0.enable_autoscale(True)
self.qtgui_freq_sink_x_0.enable_grid(True)
self.qtgui_freq_sink_x_0.set_fft_average(1.0)
self.qtgui_freq_sink_x_0.enable_axis_labels(True)
self.qtgui_freq_sink_x_0.enable_control_panel(False)
```

```
if not True:
  self.qtgui_freq_sink_x_0.disable_legend()

if "complex" == "float" or "complex" == "msg_float":
  self.qtgui_freq_sink_x_0.set_plot_pos_half(not True)

labels = ['', '', '', '', '',
          '', '', '', '', '']
widths = [2, 1, 1, 1, 1,
          1, 1, 1, 1, 1]
colors = ["blue", "red", "green", "black", "cyan",
          "magenta", "yellow", "dark red", "dark green", "dark blue"]
alphas = [1.0, 1.0, 1.0, 1.0, 1.0,
          1.0, 1.0, 1.0, 1.0, 1.0]
for i in xrange(1):
    if len(labels[i]) == 0:
        self.qtgui_freq_sink_x_0.set_line_label(i, "Data {0}".format(i))
    else:
        self.qtgui_freq_sink_x_0.set_line_label(i, labels[i])
    self.qtgui_freq_sink_x_0.set_line_width(i, widths[i])
    self.qtgui_freq_sink_x_0.set_line_color(i, colors[i])
    self.qtgui_freq_sink_x_0.set_line_alpha(i, alphas[i])

self._qtgui_freq_sink_x_0_win = sip.wrapinstance(
    self.qtgui_freq_sink_x_0.pyqwidget(), Qt.QWidget)
self.top_layout.addWidget(self._qtgui_freq_sink_x_0_win)
self.fft_vxx_0 = fft.fft_vcc(1024, True, (window.blackmanharris(1024)),
    True, 1)
self.blocks_vector_to_stream_0 = blocks.vector_to_stream(gr.sizeof_float
    *1, 1024)
self.blocks_stream_to_vector_0 = blocks.stream_to_vector(
    gr.sizeof_gr_complex*1, 1024)
self.blocks_moving_average_xx_0 = blocks.moving_average_ff(N, 1, 4000)
self.blocks_file_sink_2_0 = blocks.file_sink(gr.sizeof_float*1, '/home/
    gill/Desktop/ms-thesis/gr-spectrumsensing/grc/rtl-sdr_sensing/
    Results/snr_check.dat', False)
```

```
        self.blocks_file_sink_2_0.set_unbuffered(False)
        self.blocks_complex_to_mag_squared_0 = blocks.complex_to_mag_squared
            (1024)

        ##################################################
        # Connections
        ##################################################
        self.connect((self.blocks_complex_to_mag_squared_0, 0), (
            self.blocks_vector_to_stream_0, 0))
        self.connect((self.blocks_moving_average_xx_0, 0), (
            self.blocks_file_sink_2_0, 0))
        self.connect((self.blocks_stream_to_vector_0, 0), (self.fft_vxx_0, 0))
        self.connect((self.blocks_vector_to_stream_0, 0), (
            self.blocks_moving_average_xx_0, 0))
        self.connect((self.fft_vxx_0, 0), (self.blocks_complex_to_mag_squared_0,
             0))
        self.connect((self.rtlsdr_source_0, 0), (self.blocks_stream_to_vector_0,
             0))
        self.connect((self.rtlsdr_source_0, 0), (self.qtgui_freq_sink_x_0, 0))

    def closeEvent(self, event):
        self.settings = Qt.QSettings("GNU Radio", "spectrum_sensing")
        self.settings.setValue("geometry", self.saveGeometry())
        event.accept()

    def get_samp_rate(self):
        return self.samp_rate

    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate
        self.rtlsdr_source_0.set_sample_rate(self.samp_rate)
        self.qtgui_freq_sink_x_0.set_frequency_range(0, self.samp_rate)

    def get_freq(self):
        return self.freq

    def set_freq(self, freq):
```

```
        self.freq = freq
        self.rtlsdr_source_0.set_center_freq(self.freq, 0)

    def get_N(self):
        return self.N

    def set_N(self, N):
        self.N = N
        self.blocks_moving_average_xx_0.set_length_and_scale(self.N, 1)


def main(top_block_cls=spectrum_sensing, options=None):

    from distutils.version import StrictVersion
    if StrictVersion(Qt.qVersion()) >= StrictVersion("4.5.0"):
        style = gr.prefs().get_string('qtgui', 'style', 'raster')
        Qt.QApplication.setGraphicsSystem(style)
    qapp = Qt.QApplication(sys.argv)

    tb = top_block_cls()
    tb.start()
    tb.show()

    def quitting():
        tb.stop()
        tb.wait()
    qapp.connect(qapp, Qt.SIGNAL("aboutToQuit()"), quitting)
    qapp.exec_()


if __name__ == '__main__':
    main()
```

Appendix B

LTE-R Analysis Code

B.1 kfactordist.m

```
% Calculating K-factor for the tunnel environment for HST
clear all;
close all;
clc;
%% Creating a doppler profile for high speed raiway scenario..
Ds = 30;%Initial Distance between tx and rx times 2..
Dmin = 2;% Distance between raiway tracks and leaky feeder cables...
Kf = [];
fc = 3e9;%center frequency..
c = 3e8;
v = 138.9;%300;
t = linspace(0,(2*Ds)/v(1),100);
fd = (v*fc)/3e8;%maximum doppler frequency...
costheta = zeros(size(t));%angle between BS and MS
d1 = [];
for i=1:length(t)
    d1(i) = sqrt(2^2+(Ds/2-v(1)*t(i))^2);%distance between tx and rx..
    if t(i) >=0 && t(i)<= (Ds/v)
        costheta(i) = ((Ds/2)-v*t(i))./sqrt(Dmin^2+(Ds/2-v*t(i))^2);
```

```
    elseif t(i) > (Ds/v) && t(i)<=(2*Ds)/v
        costheta(i) = (-1.5*Ds+v*t(i))./sqrt(Dmin^2+(-1.5*Ds+v*t(i))^2);
    end
end
fs  = fd*costheta;
thetadeg = acosd(costheta);
fc_wds = fc-fs;
lambda = c./fc_wds;
Cin = 5-((0.1*1.8e10)./fc_wds)*1j;
C = Cin;
gammanum = C.*sind(thetadeg)-sqrt(Cin-(cosd(thetadeg)).^2);
gammaden = C.*sind(thetadeg)+sqrt(Cin-(cosd(thetadeg)).^2);
gamma = gammanum./gammaden;
ht = 6.1;%height of feeder cable
hr = 4.2;%height of the train
var1 = sqrt(d1.^2+(ht+hr)^2);
var2 = sqrt(d1.^2+(ht-hr)^2);
phase = ((((2*pi)./lambda).*(var1-var2))*180)/pi;
gammad = atan2d(imag(gamma),real(gamma));
phasegamma = abs(cosd(gammad-phase));
K = abs(gamma).^2+2*abs(gamma).*phasegamma;
Kf = 10*log10(1./K);
```

B.2 bercalculation.m

```
% Demonstration of Eb/N0 Vs SER for M-QAM modulation scheme
clc;
load Kf;
load t;
%---------Input Fields------------------------
%% QPSk
bitsperframe=1e3; %Number of input symbols
EbN0dB = [linspace(0,20,50) fliplr(linspace(0,20,50))]; %Define EbN0dB range for
    simulation
M=4; %for QPSk modulation.
```

```
hMod = comm.RectangularQAMModulator('ModulationOrder',M);
const = step(hMod,(0:3)');
%-------------------------------------------
refArray =1/sqrt(2)*const';
k=log2(M);
totPower=15; %Total power of LOS path & scattered paths

EsN0dB = EbN0dB + 10*log10(k);
biterrsim = zeros(size(EsN0dB));
%---Generating a uniformly distributed random numbers in the set [0,1,2,..,M-1]
data=ceil(M.*rand(bitsperframe,1))-1;
s=refArray(data+1); %QPSK Constellation mapping with Gray coding
%--- Reference Constellation for demodulation and Error rate computation--
refI = real(refArray);
refQ = imag(refArray);
%---Place holder for Symbol Error values for each Es/N0 for particular M value--
index=1;
u=1;
% Kf = 4.9;
K = 10.^(Kf/10);
for x=EsN0dB
    sn=sqrt(K(u)/(K(u)+1)*totPower); %Non-Centrality Parameter
    sigma=totPower/sqrt(2*(K(u)+1));
    h=((sigma*randn(1,bitsperframe)+sn)+1i*(randn(1,bitsperframe)*sigma+0));
    numerr = 0;
    numBits = 0;
    while numerr < 100 && numBits < 1e7
        %-----------------------------------------
        %Channel Noise for various Es/N0
        %-----------------------------------------
        %Adding noise with variance according to the required Es/N0
        noiseVariance = 1/(10.^(x/10));%Standard deviation for AWGN Noise
        noiseSigma = sqrt(noiseVariance/2);
        %Creating a complex noise for adding with M-QAM modulated signal
        %Noise is complex since M-QAM is in complex representation
        noise = noiseSigma*(randn(size(s))+1i*randn(size(s)));
        received = s.*h + noise;
```

```
        %-------------I-Q Branching---------------
        received = received./h;
        r_i = real(received);
        r_q = imag(received);
        %---Decision Maker-Compute (r_i-s_i)^2+(r_q-s_q)^2 and choose the
            smallest
        r_i_repmat = repmat(r_i,M,1);
        r_q_repmat = repmat(r_q,M,1);
        distance = zeros(M,bitsperframe); %place holder for distance metric
        minDistIndex=zeros(bitsperframe,1);
            for j=1:bitsperframe
            %---Distance computation - (r_i-s_i)^2+(r_q-s_q)^2 --------------
            distance(:,j) = (r_i_repmat(:,j)-refI').^2+(r_q_repmat(:,j)-refQ').
                ^2;
            %---capture the index in the array where the minimum distance occurs
            [dummy,minDistIndex(j)]=min(distance(:,j));
            end
        y = minDistIndex - 1;
        %--------------Symbol Error Rate Calculation
            -------------------------------
        dataCap = y;
        numerr = sum(dataCap~=data)+numerr;
        numBits = numBits+bitsperframe;
        disp(numerr);
    end
symErrSimulatedqpsk(1,index) = numerr/numBits;
biterrsim(1,index) = symErrSimulatedqpsk(1,index)/k;
index=index+1;
% u=u+1;
end

%% 16 QAM
bitsperframe=1e3; %Number of input symbols
EbN0dB = [linspace(0,10,50) fliplr(linspace(0,10,50))]; %Define EbN0dB range for
     simulation
M=16; %for QPSk modulation.
hMod = comm.RectangularQAMModulator('ModulationOrder',M);
```

```
const = step(hMod,(0:M-1)');
%-------------------------------------------
refArray =1/sqrt(10)*const';
k=log2(M);
totPower=10; %Total power of LOS path & scattered paths


EsN0dB = EbN0dB + 10*log10(k);
biterrsim = zeros(size(EsN0dB));
%---Generating a uniformly distributed random numbers in the set [0,1,2,..,M-1]
data=ceil(M.*rand(bitsperframe,1))-1;
s=refArray(data+1); %QPSK Constellation mapping with Gray coding
%--- Reference Constellation for demodulation and Error rate computation--
refI = real(refArray);
refQ = imag(refArray);
%---Place holder for Symbol Error values for each Es/N0 for particular M value--
index=1;
u=1;
K = 10.^(Kf/10);
for x=EsN0dB
    numerr = 0;
    numBits = 0;
    while numerr < 100 && numBits < 1e7
        sn=sqrt(K(u)/(K(u)+1)*totPower); %Non-Centrality Parameter
        sigma=totPower/sqrt(2*(K(u)+1));
        h=((sigma*randn(1,bitsperframe)+sn)+1i*(randn(1,bitsperframe)*sigma+0));
        %-------------------------------------------
        %Channel Noise for various Es/N0
        %-------------------------------------------
        %Adding noise with variance according to the required Es/N0
        noiseVariance = 1/(10.^(x/10));%Standard deviation for AWGN Noise
        noiseSigma = sqrt(noiseVariance/2);
        %Creating a complex noise for adding with M-QAM modulated signal
        %Noise is complex since M-QAM is in complex representation
        noise = noiseSigma*(randn(size(s))+1i*randn(size(s)));
        received = s.*h + noise;
        %-------------I-Q Branching---------------
```

```
        received = received./h;
        r_i = real(received);
        r_q = imag(received);
        %---Decision Maker-Compute (r_i-s_i)^2+(r_q-s_q)^2 and choose the
            smallest
        r_i_repmat = repmat(r_i,M,1);
        r_q_repmat = repmat(r_q,M,1);
        distance = zeros(M,bitsperframe); %place holder for distance metric
        minDistIndex=zeros(bitsperframe,1);
            for j=1:bitsperframe
            %---Distance computation - (r_i-s_i)^2+(r_q-s_q)^2 --------------
            distance(:,j) = (r_i_repmat(:,j)-refI').^2+(r_q_repmat(:,j)-refQ').
                ^2;
            %---capture the index in the array where the minimum distance occurs
            [dummy,minDistIndex(j)]=min(distance(:,j));
            end
        y = minDistIndex - 1;
        %--------------Symbol Error Rate Calculation
            --------------------------------
        dataCap = y;
        numerr = sum(dataCap~=data)+numerr;
        numBits = numBits+bitsperframe;
        disp(numerr);
    end
symErrSimulatedqam(1,index) = numerr/numBits;
biterrsim(1,index) = symErrSimulatedqam(1,index)/k;
index=index+1;
u=u+1;
end

%% 64 QAM Modulation...
bitsperframe=1e3; %Number of input symbols
EbN0dB = [linspace(0,10,50) fliplr(linspace(0,10,50))]; %Define EbN0dB range for
     simulation
M=64; %for QPSk modulation.
hMod = comm.RectangularQAMModulator('ModulationOrder',M);
const = step(hMod,(0:M-1)');
```

```
%-------------------------------------------
refArray =1/sqrt(42)*const';
k=log2(M);
totPower=10; %Total power of LOS path & scattered paths
EsN0dB = EbN0dB + 10*log10(k);
biterrsim = zeros(size(EsN0dB));
%---Generating a uniformly distributed random numbers in the set [0,1,2,..,M-1]
data=ceil(M.*rand(bitsperframe,1))-1;
s=refArray(data+1); %QPSK Constellation mapping with Gray coding
%--- Reference Constellation for demodulation and Error rate computation--
refI = real(refArray);
refQ = imag(refArray);
%---Place holder for Symbol Error values for each Es/N0 for particular M value--
index=1;
u=1;
K = 10.^(Kf/10);
for x=EsN0dB
     sn=sqrt(K(u)/(K(u)+1)*totPower); %Non-Centrality Parameter
     sigma=totPower/sqrt(2*(K(u)+1));
     h=((sigma*randn(1,bitsperframe)+sn)+1i*(randn(1,bitsperframe)*sigma+0));
     numerr = 0;
     numBits = 0;
    while numerr < 100 && numBits < 1e7
        %-----------------------------------------
        %Channel Noise for various Es/N0
        %-----------------------------------------
        %Adding noise with variance according to the required Es/N0
        noiseVariance = 1/(10.^(x/10));%Standard deviation for AWGN Noise
        noiseSigma = sqrt(noiseVariance/2);
        %Creating a complex noise for adding with M-QAM modulated signal
        %Noise is complex since M-QAM is in complex representation
        noise = noiseSigma*(randn(size(s))+1i*randn(size(s)));
        received = s.*h + noise;
        %-------------I-Q Branching---------------
        received = received./h;
        r_i = real(received);
        r_q = imag(received);
```

```
        %---Decision Maker-Compute (r_i-s_i)^2+(r_q-s_q)^2 and choose the
            smallest
        r_i_repmat = repmat(r_i,M,1);
        r_q_repmat = repmat(r_q,M,1);
        distance = zeros(M,bitsperframe); %place holder for distance metric
        minDistIndex=zeros(bitsperframe,1);
            for j=1:bitsperframe
            %---Distance computation - (r_i-s_i)^2+(r_q-s_q)^2 --------------
            distance(:,j) = (r_i_repmat(:,j)-refI').^2+(r_q_repmat(:,j)-refQ').
                ^2;
            %---capture the index in the array where the minimum distance occurs
            [dummy,minDistIndex(j)]=min(distance(:,j));
            end
        y = minDistIndex - 1;
        %--------------Symbol Error Rate Calculation
            --------------------------------
        dataCap = y;
        numerr = sum(dataCap~=data)+numerr;
        numBits = numBits+bitsperframe;
        disp(numerr);
    end
symErrSimulatedqam64(1,index) = numerr/numBits;
biterrsim(1,index) = symErrSimulatedqam64(1,index)/k;
index=index+1;
u=u+1;
end

%%
fig = figure;
semilogy(t*1e3,symErrSimulatedqpsk(1,:),'-d','LineWidth',2);
hold on;
grid on;
semilogy(t*1e3,symErrSimulatedqam(1,:),'-d','LineWidth',2);
semilogy(t*1e3,symErrSimulatedqam64(1,:),'-d','LineWidth',2);
xlabel('Time (ms)');
ylabel('Bit Error Rate (Pb)');
title(['BER For OFDM Under Rician Fading Environment Inside Tunnel']);
```

```
set(gca,'fontsize',30,'box','on','LineWidth',2,'GridLineStyle','--','GridAlpha'
    ,0.7);
axis([0 max(t)*1e3 10e-7 0])
lgd = legend('QPSK','16QAM','64QAM');
lgd.FontSize=20;
```