Implementation of a Total Power Radiometer in Software Defined Radios

by

Matthew Erik Nelson

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering (Embedded Systems)

Program of Study Committee:

Phillip H Jones, Major Professor

John Basart

Mani Mina

Iowa State University

Ames, Iowa

2015

DEDICATION

I would like to dedicate this thesis to my wife Jennifer who has stood by me through this long journey towards my Masters. I would also like to thank my parents without whose support I would not have been able to complete this work.

I would also like to thank my friends and family for their continued support and constant encouragement.

TABLE OF CONTENTS

LIST OF TABLES vii

LIST OF FIGURES viii

ACKNOWLEDGEMENTS x

ABSTRACT xi

CHAPTER 1. INTRODUCTION AND BACKGROUND 1

1.1 Background 1

1.2 Introduction 4

1.2.1 Software Defined Radio Radiometer 4

1.2.2 RF Front End 7

1.3 Related Works 8

CHAPTER 2. SDR RADIOMETER THEORY OF OPERATION 10

2.1 Traditional Radiometer Theory of Operation 11

2.1.1 Measuring RF power 11

2.2 Software Defined Radio Theory of Operation 14

2.2.1 Software Defined Power Detection 15

2.2.2 Software Defined Integrator 15

2.2.3 SDR Radiometer Summary 18

2.2.4 RF Front End 19

2.2.5 Software 19

2.2.6 GNURadio 20

2.3 General Specifications 22

2.3.1 N200 Software Defined Radio 22

2.3.2 DBSRX2 Noise Temperature Consideration 26
2.4 Comparison of a Software Defined Radio Radiometer vs Traditional Radiometer 27
2.5 Comparison of a Software Defined Radio Radiometer and other Digital Radiometers 27
CHAPTER 3. IMPLEMENTATION OF A SOFTWARE DEFINED RADIOMETER 29
3.1 Requirements 29
3.1.1 Hardware Requirements 30
3.1.2 Software Requirements 30
3.2 Power detection 31
3.3 Integrator through a IIR Filter 32
3.4 GNURadio Blocks 32
3.5 Control of the SDR Hardware through GNURadio 32
3.5.1 Impact of the Controls Related to Radiometry 34
3.6 GNURadio Data Handling 34
3.6.1 GNURadio Data Display 35
3.6.2 GNURadio Data Processing 36
CHAPTER 4. EVALUATION AND EXPERIMENTAL SETUP 38
4.1 Square-law Detector 38
4.1.1 Analog Devices ADL5902 39
4.1.2 Data acquisition and storage 40
4.1.3 Tests on the ADL5902 43
4.2 Software Defined Radio tests 44
4.3 Total Power Radiometer Test with Ice Water Bath 44
4.3.1 Laboratory Setup 45
4.3.2 Test Results 46
4.4 Liquid Nitrogen Test 47
4.4.1 Testing Apparatus 47

4.4.2 Calibration and verification experiment 48
4.5 Testing with an injected noise source 49
4.5.1 Test setup 50
4.5.2 Test results 51
4.6 Further testing 52
CHAPTER 5. RESULTS AND ANALYSIS 53
5.1 Performance of a radiometer 53
5.1.1 Sensitivity 53
5.1.2 Accuracy and Stability 54
5.2 Required Performance Requirements 55
5.3 Square-law Detector Performance 55
5.4 Software Defined Radio Performance 56
5.5 Benefits to Software Defined Radio Radiometer 56
5.5.1 Cost Benefits 56
5.5.2 Weight and component size benefits 58
5.5.3 Value added benefits 59
5.6 Disadvantages of a SDR Radiometer 59
5.6.1 Power Consumption 59
5.6.2 Bandwidth constraints 60
CHAPTER 6. CONCLUSION AND FUTURE WORK 61
6.1 Conclusion 61
6.2 Future work 61
6.2.1 Improving on the FPGA firmware 62
6.2.2 Correlation 62
6.2.3 Improving stability 63
APPENDIX A. Source code 65
APPENDIX B. EE 518 Test Results 115
B.0.4 Test setup 116

B.0.5 Lab Experiment 117
B.0.6 Lab Results 117
B.0.7 Conclusion 118
APPENDIX C. Direct-Sampling Digital Correlation Radiometer 119
C.0.8 Implantation in the ISU Radiometer 119

LIST OF TABLES

Table 3.1 ISU Radiometer requirements 30

Table 4.1 ADL5902 Specifications 39

Table 5.1 Required Radiometer performance 55

Table 5.2 Cost Analysis 57

Table 5.3 Weight Analysis 58

LIST OF FIGURES

Figure 1.1 Block diagram of the N200 SDR 5

Figure 1.2 The ADISIMRF program used to verify the design of the RF Front End 7

Figure 2.1 The ideal radiometer block diagram 12

Figure 2.2 A more realistic radiometer model 13

Figure 2.3 A total power radiometer 14

Figure 2.4 A simple RC circuit 16

Figure 2.5 Screenshot of the GNURadio Companion editor and the N200 Radiometer block diagram used in many of the experiments 21

Figure 2.6 The USRP N200 from Ettus Research 23

Figure 2.7 The DBSRX2 daughter board from Ettus Research 26

Figure 3.1 A block diagram showing how the radiometer performs the equivalent square law detector in software. 31

Figure 3.2 A screenshot of the interface made for communication with and controlling the software defined radio 33

Figure 3.3 A screenshot showing the ticker tape display for the total power readings. In addition raw and calibrated noise temperature is shown below. 36

Figure 3.4 A screenshot showing the Python code and related graphs generated for parsing GNURadio data 37

Figure 4.1 An image of the ADL5902 square-law detector used in these experiments 40

Figure 4.2 A screenshot of the Labview GUI interface 41

Figure 4.3 A screenshot of the Labview block diagram 42

Figure 4.4 Graph showing the linearity of the ADL5902 44
Figure 4.5 A screenshot of the GNURadio program using a noise generator block. 45
Figure 4.6 An image of the typical lab setup used to test the software defined radiometer 46
Figure 4.7 Graph showing both the square-law detector and the software defined radio total power readings. 48
Figure 4.8 Screen shot of the Labview program capturing total power information from the radiometer 49
Figure 4.9 Image of the interfering signal appearing on the spectrum display of the SDR Radiometer. 50
Figure 4.10 Image of the offending signal being filtered out by the SDR. It can be seen that the signal is no longer visible. 51
Figure 4.11 Image showing both the SDR and square-law detector recording TPR, but with the SDR using a filter to remove the offending signal. 52

Figure 6.1 The key blocks used for creating a correlating radiometer in software. The key blocks is the USRP source, which allows us to address both daughter-boards and the ADD block which sums the signals. 63

Figure B.1 Students rotating the radiometer for an experiment on Agronomy Hall 116

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Phillip Jones for his guidance, patience and support throughout this research and the writing of this thesis. I would also like to thank Dr. John Basart who patiently spent many hours with me going over both the fundamentals and details with radiometers. I would like to thank Dr. Brian Hornbuckle for his continued support and for allowing me to utilize his radiometer, lab and students towards my research. Finally I would like to thank Dr. Mani Mina who not only was instrumental in my undergraduate career but continued his support into my graduate path as well. If it was not for Dr. Mina hiring me as the Chief Engineer of the Spacecraft Systems and Operations Lab, I would not be here today.

ABSTRACT

A software defined radio is defined as a communication system where components of a communication system that are typically done in hardware are now done in software. The result is a highly flexible communication system that can adapt to changes to the system based on requests by the user or due to conditions in the radio frequency channel. Software Defined Radios (SDRs) have been used for a variety of applications, mostly in the area of communications. However, they have not been widely applied to remote sensing applications such as a radiometer. A SDR based radiometer offers a very flexible and robust system more so than a more traditional radiometer which has fixed hardware and usually little room for flexibility and adaptability based on the needs of the radiometer application or due to changes in the radiometers environment. The price of SDRs have also been steadily dropping making implementing them into radiometers a more attractive option compared to using traditional RF hardware. In this thesis we will look into the feasibility and theory of using an off the shelf SDR hardware platform for a radiometer application. In addition, we will look into how we can use the GNURadio software to create a total power radiometer within software and how this software can be used to make easy changes to the functionality of the radiometer. Finally we will look at the preliminary results obtained from laboratory tests of the SDR radiometer system.

CHAPTER 1. INTRODUCTION AND BACKGROUND

1.1 Background

Software Defined Radios Software Defined Radios (SDRs) have been used for a variety of applications, but their primary application has been in the area of communications. They appeal to applications where being able to change a modulation scheme or filter on the fly is desirable. In these areas, SDRs often outperform a traditional hardware only radio with their ability to rapidly change their operations by simply changing their software. Early SDRs were expensive due to the high costs in the analog to digital converters (ADCs) needed and in the high speed Field Programmable Gate Arrays (FPGA) used. In recent years however, the cost of SDRs have decreased due to the cost of these key components decreasing in cost as well. Even though the cost has gone down, the performance of many SDRs have increased. This has lead to new applications for SDRs being developed and using SDRs in new and different ways.

The basic concept behind a SDR is that it will digitize the RF signal as soon as possible. Once digitized, it can now be evaluated by a computer, FPGA, or a dedicated System on Chip (SoC). A canonical software defined radio architecture is one that consists of a power supply, antenna, multi-band RF converter, and a single chip that contains the needed processing and memory to carry out the radio functions in software [**?**)]. This allows us to extract certain hardware functions, such as filtering, into the digital domain which can then be manipulated by software. Since software is now manipulating the signal, we can rapidly change what functions we execute on the signal by changing the software. This gives SDRs a high amount of flexibility as various components that are normally done in hardware can now be done in software and can be changed by simply uploading new software or firmware to the system. This also gives us a benefit in cost as certain components are no longer needed and changes done in software

do not require additional hardware to be added or to be swapped out.

Radiometers Radiometers are radio receivers that simply listen to and record the amount of power received. However, the power received is not a coherent signal, instead it is the amount of noise the radiometer sees. Radiometers, at the basic level, listen to noise that is generated naturally from a source. These sources can vary and the applications vary as well. Some examples of radiometer applications have been in evaluating soil moisture content, ocean salinity levels and celestial objects[**?**)].

The amount of noise that is generated is due to the thermal agitation of the charge carriers, usually the electrons, which is directly correlated to the physical temperature of the source[**?**)]. This correlation is done as a noise temperature. Various objects can emit this noise and the intensity will vary on various parameters and on what the source is. One source that has current research at Iowa State University is in detecting soil moisture. The brighter or warmer the noise temperature is, the more RF noise that has been received which correlates to a drier soil. The less RF noise power received, the cooler the noise temperature and this indicates wetter soil area. Radiometers such as these are already in service on satellites such as the Soil Moisture and Ocean Salinity (SMOS) satellite launched by the European Space Agency (ESA) and are used by scientists to monitor the Earth's soil moisture and ocean salinity[**?**)][**?**)][**?**)][**?**)][**?**)].

A traditional radiometer uses several Low Noise Amplifiers (LNAs), filters and power is often measured using a square law detector. Even though we are measuring noise, we want to measure the right noise and we want the noise generated by the radiometer hardware itself to be as low as possible. In other words, we are really listening to the noise being generated from an outside source. Additional noise in the system is impossible to eliminate, however we can take steps to reduce it as much as possible. In addition, we can calculate what this noise is and take steps to account for it in our measurements. However, this means that stability is another factor within the radiometer. If the noise generated within the radiometer is constantly changing, this makes it difficult to account for this additional noise. There are steps we can take to work with this though. A traditional method often used is the Dicke radiometer which switches between the measurement of the antenna and a known source[**?**)]. By referencing this known source the Dicke radiometer can calibrate and account for any drift due to variations in

the system. Another method is to use highly stable components and keep them stable during the operation of the radiometer. For LNAs, temperature directly effects the overall gain from the LNA. In some radiometer applications we can control the temperature which allows us to keep the LNAs stable during the operation. Stability will be discussed in greater detail later in this thesis.

Additional improvements to radiometers have also been done by digitizing parts of the radiometer. The most common method for doing this is by digitizing the analog output of the square-law detector and sending that to a computer or processing unit to analyze the data[**?**)]. While this does allow for easier computation and storage of the information, it does not alleviate the needs to maintain stability or reduce possible additional noise of the system since this data is digitized after the RF signal chain.

A more modern Radiometer Iowa State University currently owns a 1.4 GHz, dual polarization, correlating radiometer. This radiometer is currently in use by Dr. Brian Hornbuckle and his research team. However, in recent years the radiometers digital circuitry has suffered from various issues which has made it unusable. Part of the driving force behind this research in this thesis is to determine new methods that can be used to rebuild this radiometer.

The ISU Radiometer was built at the University of Michigan and put into service at ISU in 2006. The ISU radiometer is unique in that it is one of the few direct sampling radiometers in use**?**). This radiometer takes the RF signal, amplifies and filters the signal, and then sends it directly to an analog to digital converter. At the time the ISU radiometer was built, an A/D that could sample accurately at 1.4 GHz were expensive and hard to come by due to the fact that Nyquist's theorem states that we must sample at least two times the frequency. However, the ISU radiometer does not sample at 2.8 GHz or above, instead it samples at 1.4 GHz. The reason the ISU radiometer can do this is due to the fact that a radiometer is generally only interested in the power of the incoming signal. This means that we are not interested in recreating the entire signal and therefore we can under-sample the signal. The A/D data is then sent to a Field Programmable Gate Array (FPGA) which then processes the data. The ISU radiometer is also a correlating radiometer, which means that it looks at both the vertical polarization (V-Pol) and the horizontal polarization (H-Pol) and then correlates

this information[**?**)].

1.2 Introduction

The goal of this thesis is to explore the use of a software defined radio that can function as well as or better than current radiometers. In addition, we aimed to develop a radiometer that is more flexible than most radiometers and still maintain the accuracy and stability of a traditional radiometers if not exceed these specifications. A secondary goal was to use off the shelf components and components that are generally more accessible and often less expensive. This would allow radiometers to be more accessible to a wider scope of researchers in this field. And finally a tertiary goal was to ensure that the system as a whole is fairly easy to use. This ties to our secondary goal of making radiometers more accessible to a wider range of researchers and research topics.

This thesis looks to explore the following questions: (1) Can we use an off the shelf SDR along with GNURadio to recreate a radiometer in software that is easy to use and cost effective? (2) If so, what performance can we get from the system? (3) What benefits do we gain (if any) from using a SDR from a more traditional radiometer? The results of this research and experimentation are the subject of this thesis.

1.2.1 Software Defined Radio Radiometer

Software defined radios consist of both hardware and software that allow it to perform the operations that is required. A software defined radio used for radiometer applications is identical to a software defined radio used for, as an example, a 802.11b radio with one major difference. Since we do need to amplify the signal more than what most communication applications require, we do have more powerful or additional LNAs to boost this signal. In addition, since the first LNA plays a major role in the overall system noise and this system noise does affect performance of the radiometer, the selection of this LNA is important. However, all other components are the same components used in other applications.

We will now introduce the three major components that make up a software defined radio radiometer.

1.2.1.1 N200 Software Defined Radio

The equipment that was selected for researching into this topic was the Ettus Research Group N200 SDR. This SDR uses daughter boards as the RF front end to the SDR and up to two daughter boards may be installed into a N200. This is an important consideration as one of the requirements is to be able to look at both the V-Pol and the H-Pol signal coming from the antenna. This allows us to correlate the signal and other signal analysis can also be done. Another important reason the N200 was selected was due to the fact that it can handle up to 50 MHz of bandwidth to the computer. This means that it is very possible to have two receive cards that can stream up to 25 MHz bandwidth each.

The N200 utilizes a flexible architecture for a variety of RF interface systems based on the frequency range desired and if receive and/or transmission is needed. These daughter boards directly receive the RF signal and then outputs the analog I and Q signals that are then sampled by the N200 A/D converter for reception or receives the I and Q values from the N200 D/A converter for transmission.

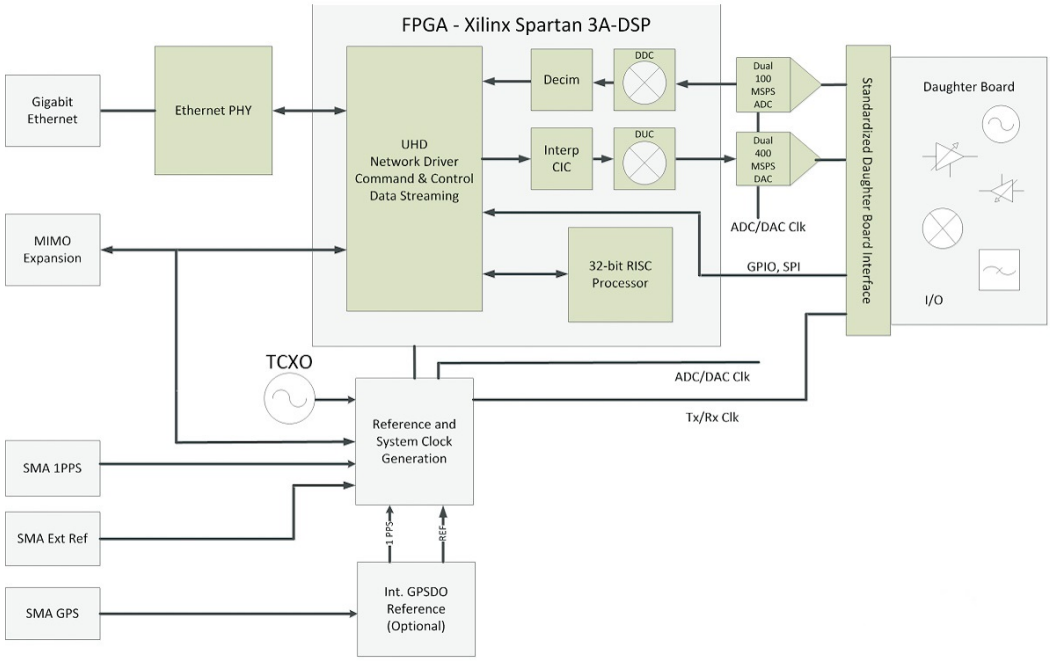

Figure 1.1 Block diagram of the N200 SDR

The daughter board selected was the DBSRX2 card. This card is a receive only card that

operates between 800 MHz and 2.4 GHz and would thus work in the 1.4 GHz band we are interested in. The DBSRX2 also has built in amplification that is adjustable through software.

1.2.1.2 GNURadio Software

For the software, we settled to use GNURadio, an open source software package that is well supported by the community and by the Ettus Research Group and the N200 SDR. GNURadio also comes with what is known as GNURadio Companion or GRC. This program provides us with a GUI interface and allows for the drag and drop of blocks that represent certain functions that can be used with the SDR. GNURadio and GRC use Python as it's main scripting language and GNURadio uses C++ code for directly accessing the hardware. The hardware interface for the N200 is provided by Ettus as well. They provide drivers that allow GNURadio to talk to their hardware. Like GNURadio, these drivers are also available to all platforms. In addition, Ettus has released these drivers to the open source community.

GNURadio has been ported to several platforms including Windows, Mac OS X and Linux. Linux is by far the most popular platform to work on and most of this thesis research was done within the Linux environment. Testing was also done though with a MacBook Pro running OS X 10.9. The OS X implementation is fairly well supported and is installed through MacPorts.

Through GNURadio, we can now write code that will take the data given to us from the SDR and manipulate the signal as we need to mimic a radiometer. The power detection, filtering and recording of this data is all done through GNURadio. This also means that we are shifting more of the computational power done on the signal from the FPGA to a computer running GNURadio. This is different from the current ISU radiometer where the FPGA did almost all of the work and then simply sent the data to a computer to be stored. The benefit of this however is that components of the radiometer can be updated by simply changing things in GNURadio and does not require reprogramming the FPGA. It does however mean that the host computer must be powerful enough to handle the signal and specifically the large bandwidth that we wish to send to it.

1.2.2 RF Front End

The RF front end plays a critical role in the radiometer as the LNAs used in the front end has a large impact on the system noise generated by the radiometer itself. A traditional radiometer utilizes both amplification through the LNAs and also includes filtering to the desired bandwidth. A SDR radiometer does not require the filters as we are able to create these in software, however the amplification stages need to remain.

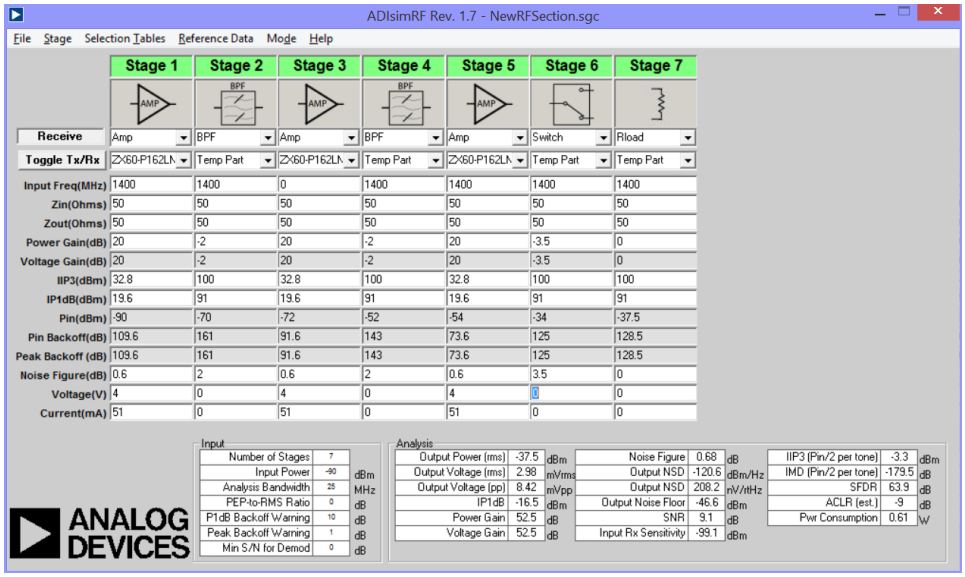

Figure 1.2 The ADISIMRF program used to verify the design of the RF Front End

A typical RF front end uses a 3 stage Low Noise Amplifier (LNA) to amplifier the noise while keeping the noise contributed to the system as low as possible. As with any radiometer, the first LNA is the most critical as it contributes the most to the overall system noise temperature. For this reason a LNA that did not have a large gain but had a very low noise figure was chosen. The second and third LNA has higher gain values at the cost of a higher noise figure, although not by much. However, since they are further down the chain, they do not contribute as much to the total system noise. The reason for this is further explained in chapter 3.

1.3 Related Works

As mentioned before, software defined radios have been used in a number of applications. While as far as the author has determined, this is the first application of using an off the shelf software defined radio for a radiometer in remote sensing for soil moisture measurements, there has been similar applications done. The closest application that has been found is with radio astronomy. Radiometers used in radio astronomy is nothing new and has been used for quite some time[**?**)]. by There are many similarities between radio astronomy and remote sensing of the ground. Both are using a radiometer to listen to a source of interest. In radio astronomy the basic principal is that a "hot" source such as a star will produce more noise than the cooler background of space. In remote sensing we are looking at the overall change of the source to determine it's characteristics. In both cases we are measuring the total power of the noise and based on that information we can determine some properties of the source we are looking at.

The Shirleys Bay Radio Astronomy Consortium (SBRAC) located in Smiths Falls, Ontario is currently using a USRP software defined radio in conjunction with GNURadio. SBRAC has successfully used this configuration to obtain radio astronomy data by looking at the hydrogen line at 1420.4058 MHz [**?**)]. The person in charge of this facility, Marcus Leech, contributed software to the GNURadio specifically for radio astronomy applications. It was this software branch that was used as the base for the GNURadio program that was used in this thesis. Marcus Leech continue to contribute to the GNURadio community and continues to provide support for these functions as well[**?**)].

Another example of a software defined radio used as a radiometer is from students at the University of Illinois and Grand Valley State University that built a software defined radio to listen to emissions from Jupiter[**?**)]. This software defined radio was built using an Analog Devices AD9460 and a Xilinx Spartan-3E-500 FPGA to build the SDR itself. A RF front end was also built to filter and amplify the signal coming into the SDR. Finally, this group also used GNURadio to interface to and talk to the SDR and used both Python and wxGUI to build a working interface. The students reported that the SDR radiometer worked very well and was able to do so at a fairly low price point.

It should be noted that much of the related works found worked with using a radiometer for astronomy or for looking at the sky. For the ISU radiometer however, we are looking at the ground and we are using the radiometer for soil moisture instead of measuring stars and other points of interest in the sky. While the fundamentals is the same for either radiometer, some adjustments need to be made due to the fact that a radiometer looking at the sky often sees a "cool" brightness temperature whereas a radiometer looking at the ground sees a much "warmer" brightness temperature. This is due to the albedo of the Earth and the fact that it has a much warmer noise temperature then what you find with radio astronomy[**?**)].

CHAPTER 2. SDR RADIOMETER THEORY OF OPERATION

The work of this thesis is to use an off the shelf software defined radio (SDR) to perform the same operation or better of a traditional analog radiometer. Using a SDR radio also means that we are able to be more flexible in how the radiometer performs, is capable of frequency agility and adapting to changing conditions such as interference. Using a software defined radio also allows for implantation of different radiometer types such as a polarimetric radiometer, a correlation radiometer and also a polarimetric radiometer that uses Stokes parameters [**?**)]. Normally, these would require changes to hardware, but all of these types of radiometers can be implemented in software increasing the flexibility of the system.

The hardware that was selected for this research was the Ettus Research N200 Software Defined radio. Information and rationale behind the selection of this unit will be covered later. The N200 however gives us the standard building blocks for a typical software defined radio which includes a A/D converter and on-board FPGA. The N200 however also gives a flexible front end by selecting various daughter boards that fit our application. In addition, the N200 supports up to two daughter boards to be installed.

A software defined radio of course is only as good as the software that is written to work with the hardware. As the name implies, the software defines how the radio will act and function. From a computer engineering perspective though, software can take on several roles in a system. There is software that runs on the hardware, in our case the FPGA, and then there is software that may run on a computer to interface with the hardware. In some cases, the software may only reside on the hardware, however, to increase the user usability of the system, we will be using a combination of firmware that is running on the FPGA and software that runs on a computer. GNURadio is an open source software define radio framework that runs on multiple OSes and offers a rich set of features. In addition, GNURadio is well supported by

the Ettus Research group and is the preferred software for interfacing with their hardware. An easy to use interface was another driving requirement for our implementation of a radiometer in a SDR. GNURadio helps us with this through the use of the GNU Radio Companion or GRC. This was important as it was anticipated that many operators of the radiometer would not know much about programming. GNURadio uses a simple to use graphical system that is very similar to things such as LabView, offering a drag and drop system for adding various radio components such as filters. Like LabView, you can also simply wire up the boxes and complete the circuit path for the RF signal as it gets processed.

2.1 Traditional Radiometer Theory of Operation

The primary goal of a radiometer is to measure power. While that statement sounds easy, there are in fact many factors that go in to how well a radiometer can measure the power it sees. A better statement would be that a radiometer's primary goal is to accurately measure the correct power within a certain degree of accuracy. In order to accurately and within a high degree of precision measure power, a radiometer must take into account various factors such as the system noise, the bandwidth of the signal and the stability of the system as a whole[**?**)].

2.1.1 Measuring RF power

To measure power in a radiometer, several factors are taken into consideration. To begin with we have the noise signal coming from the antenna. Our antenna is assumed to be looking at our target of interest and it is assumed that we can relate the antenna noise to the noise from the source. It is often easier to refer to this noise as the brightness temperature. Therefore the brightness temperature of the source can be related to the brightness temperature at the antenna. We will refer to this brightness temperature as T_A.

Figure 2.1 shows us an ideal radiometer. That is a radiometer that has an input from the antenna, T_A, a known bandwidth denoted as B and a known gain denoted as G. At the end of the block is the detector, which measures the power from the radiometer.

Only a certain selection of the radio spectrum is observed by the radiometer. This is referred to as the bandwidth of the radiometer and is denoted as B or as β. This bandwidth is then

Figure 2.1 The ideal radiometer block diagram

centered around a center frequency. In our case, we center around 1.4125 GHz. There is a reason why 1.4125 GHz is selected and that is from 1.4000 to 1.4260 GHz is protected internationally to be as radio frequency interference free as possible. This reduces interference from outside sources such as transmitters that can interfere with the operation of the radiometer.

The power coming from the antenna is amplified so it is easier to determine changes in the brightness temperature. The overall gain of the radiometer system is referred to as G in this case. Finally, we need to apply Boltzmann's constant, referred to as k. With these values, we can now compute the power the radiometer will see for an ideal radiometer. This can be shown in equation 2.1

$$P = k * \beta * G * (T_A) \tag{2.1}$$

However, since we do not have an ideal radiometer, we have another key component that needs to be addressed. This is the noise that is added to the system by the radiometer itself, primarily from the amplifiers used to increase the signal. Most of the additional noise is from the Low Noise Amplifiers (LNA) that are used to increase the signal while attempting to keep the noise added by the LNA to a minimum. However, noise is also added from virtually every component in the RF front end. However, by far the largest contribution usually comes from the LNA, which is why the selection of the LNAs is a critical decision. Figure 2.2 shows the additional noise that is injected into the system.

As it can be seen, this additional noise is added to the noise coming from the antenna source. Therefore T_N is added to T_A and our final equation for the power measured is shown in equation 2.2.

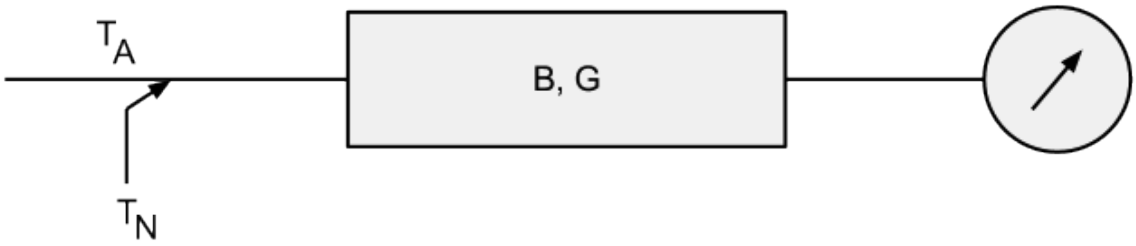

Figure 2.2 A more realistic radiometer model

$$P = k * \beta * G * (T_A + T_N) \tag{2.2}$$

The issue with all radiometers is that it must detect small signal changes in a noisy environment. To understand this, let us look at the example of T_A has a value of 200 K and T_N has a value of 800 K. Since T_N is added to our antenna signal, we have a total noise temperature of 1,000 K. This means that if we want to detect a change as small as 1 K, we must be able to measure the difference between 1,000 K and 1,001 K. [**?**)]

The ability of a radiometer to detect these small changes is the radiometer's sensitivity, or the standard deviation of the output signal from the radiometer. This sensitivity is also referred to as the Noise Equivalent Δ Temperature or NEΔT and is shown in equation 2.3.

$$NE\Delta T = \frac{T_A + T_N}{\sqrt{\beta * \tau}} \tag{2.3}$$

The sensitivity of the radiometer is based on both the bandwidth, β, of the incoming signal and the integration time, τ. As it can be seen in the equation, we would want to have as much bandwidth as possible. In a traditional radiometer, this bandwidth is often fixed and is dependent on the band-pass filters used in the radiometer. We can however control τ and a longer integration time will help improve the sensitivity of the radiometer.[**?**)]

This covers a very simplified radiometer, however most radiometers are more complicated than the one shown in Figure 2.2.

A more typical radiometer can be shown in Figure 2.3. Here we have expanded the blocks

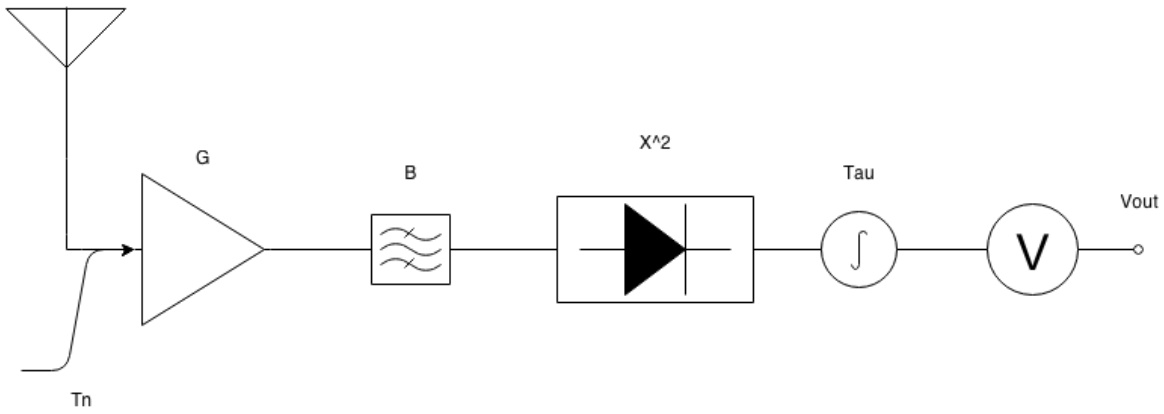

Figure 2.3 A total power radiometer

used and have separated the bandwidth and gain blocks. In addition, we have detection, shown as X^2, and integration. These last two blocks make up the detection and results in the power we can measure, which is a voltage represented as V_{out}. This results in equation 2.4. In most cases the detection is performed by a square-law detector[**?**)].

$$V_{OUT} = c * (T_A + T_N) * G \tag{2.4}$$

Here V_{OUT} is shown by the addition of both the noise from the system T_N and the noise from the antenna, T_A and multiplied by the gain in the system, G.

The voltage output from this radiometer is then either measured or may also be sampled by an analog to digital converter. In both cases the output has no real meaning until it is calibrated.

2.2 Software Defined Radio Theory of Operation

A software defined radio (SDR) mimics radio functions in software instead of relying on dedicated hardware. As stated in the previous section, a radiometer is a radio that can measure and detect changes in power. Therefore the SDR needs to be able to measure power coming from the source that we are looking at and do so with the same or better sensitivity as a traditional radiometer.

The SDR is able to do all of this but in a slightly different way than a traditional radiometer. As discussed in section 2.1, a traditional radiometer will use a square-law detector to measure the incoming RF signal. This device is simply a diode where the input voltage is squared and the output from the diode is proportional to the AC input voltage. Therefore a 3 dB increase in the RF power will result in a two times increase in the voltage. This power measurement will be fluctuating rapidly, therefore we will then run the output from the square-law detector to an integrator and integrate over a set time period. As shown in equation 2.3, this integration time also affects the NEΔT or sensitivity of the radiometer as well.

2.2.1 Software Defined Power Detection

For a SDR, the incoming signal is sampled and converted to I/Q values by the hardware within the SDR. The I/Q values represent the amplitude and phase information of the signal. In GNURadio we are then able to square these values within software. This block in GNURadio mathematically performs what is shown in equation 2.5.

$$I^2 + Q^2 = P_{out} \tag{2.5}$$

Like the analog square-law detector, this signal will fluctuate rapidly and to improve the sensitivity of the radiometer we wish to integrate this signal. We now want to look at how we can replicate a RC filter or integrator in the software defined radio.

2.2.2 Software Defined Integrator

A RC filter is analogous to an integrator where the R and C values determine our time constant and our integration time for the filter[**?**)]. A SDR however operates in the digital domain at discrete intervals. One type of filter that can be used is the Infinite Impulse Response (IIR) filter.

To begin with, we look at what an analog RC filter looks like.

This circuit can be represented by equation 2.6.

$$\frac{V_{in} - V_{out}}{R} = C\frac{dV_{out}}{dt} \tag{2.6}$$

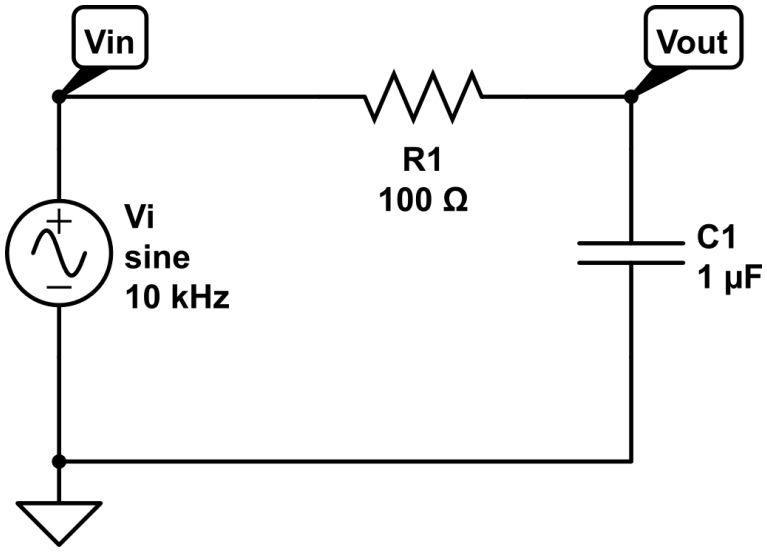

Figure 2.4 A simple RC circuit

A Finite Impulse Response (FIR) filter is a digital filter that can take an impulse signal and decays to zero after a finite number of iterations. This type of digital filter can be represented by equation 2.7 which mathematically expresses the FIR Filter.

$$y_n = \sum_{i=o}^{P-1} c_i x_{n-i} \tag{2.7}$$

This simply says that the nth output is a weighted average of the most recent P inputs.

An Infinite Impulse Response (IIR) filter is the same as the FIR filter, except that we add an additional summation term which feeds back the previous output.

$$y_n = \sum_{i=o}^{P-1} c_i x_{n-i} + \sum_{j=1}^{Q} d_j y_{n-j} \tag{2.8}$$

Equation 2.8 shows that a FIR filter is really a IIR filter, except that $Q = 0$[**?**)].

To get a better understanding on how our digital IIR filter relates to our RC filter analog, we can look at the Fourier Transform and the relationship of the input to the output in the frequency domain.

$$H(f) = \frac{\sum_{j=o}^{P-1} c_j e^{-2\pi i j f T}}{1 - \sum_{k=1}^{Q} d_k e^{-2\pi i k f T}} \tag{2.9}$$

In equation 2.9, f is our frequency in Hz and T is the time between samples in seconds and is related to our sampling frequency.

We now want show the link between our analog RC circuit and the IIR filter. Looking at equation 2.6, which represents the differential equation relating the input voltage V_{in} to the output voltage V_{out}, we can substitute for input and output of our IIR filter. Since we are now in the time domain, we need to define what T is and we can do that using equation 2.10.

$$T = timebetweensamples = \frac{1}{samplingrate} \tag{2.10}$$

We can now relate our input voltage to the input to our IIR filter and the output voltage to the output of our IIR filter.

$$x_n = v_{in}(nT) \tag{2.11}$$

$$y_n = v_{out}(nT) \tag{2.12}$$

We can now rewrite our difference equation with x_n and y_n.

$$\frac{x_n - y_n}{R} = C\frac{y_n - y_{n-1}}{T} \tag{2.13}$$

Finally, we can solve for y_n which results in our final equation for showing how a IIR filter is related to an RC filter.

$$y_n = \frac{T}{T + RC}x_n + \frac{RC}{T + RC}y_{n-1} \tag{2.14}$$

It can be seen that an IIR filter can have the same frequency response as we would expect from an analog RC filter. As our sampling rate approaches infinity, the approximation gets closer to the original response from the analog RC circuit.

For the cutoff frequency of a RC circuit, we know that it has the relationship shown in equation 2.15.

$$f_c = \frac{\sqrt{3}}{2\pi RC} \rightarrow RC = \frac{\sqrt{3}}{2\pi f_c} \tag{2.15}$$

The RC term gives us our time constant of the circuit and can be used to calculate out our coefficients. We are not concerned about the actual values of R and C with our IIR filter, instead we just need the product of R and C.

For GNURadio most of the work is done for us. We can simply enter in our desired cutoff frequency and GNURadio will calculate our IIR filter coefficients. However, this shows that an IIR filter works very much like an analog RC low pass filter.

Like a traditional radiometer, the SDR will use an antenna to look at the target of interest. SDRs still use a RF stage that takes the power from the source and amplifies it. The difference though begins after that. A SDR will then sample and generate I and Q values that represents the amplitude and phase of the signal. From there, this data is sent to a computer to be processed. We can then use this information to calculate the power that is being seen. In addition, we can manipulate the signal in other ways such as applying a filter to filter out an unwanted source.

2.2.3 SDR Radiometer Summary

As we have shown the two of the major components of a traditional radiometer, the power detection and integration of the signal can be replicated in software and therefore can be implemented in a software defined radio. The information can now be stored, displayed or both for further analysis.

There is one component of the software defined radio that we can not implement in software and that is with the signal amplification. This however does play a major role in the performance of the radiometer and is a key element that can not be overlooked. While this is not implemented in software, it still plays a critical role in our software defined radio radiometer.

2.2.4 RF Front End

Hardware with a software defined radio still plays a critical role. A traditional software defined radio will focus on digitizing the signal as soon as possible and the hardware used will be focused on performing that critical digitizing process. For a software defined radio radiometer, we still want to digitize as soon as possible, but we also need to consider our noise factor. Most software defined radios are designed for communication purposes such as implementing 802.11b or other wireless protocols. In these cases a higher noise factor is not as detrimental to the overall system as it would be with a radiometer. This is mainly due to the fact that we have a known signal that is magnitudes greater than the noise floor. In a radiometer however, we do not have this large separation between the information we need and the noise of the system. And so our noise factor is a more critical component.

To counter this, we use Low Noise Amplifiers to amplify the signal but it can also help lower our noise factor. The first LNA in the chain contributes significantly to our noise factor, so by selecting a LNA that has a very low noise factor, it reduces the noise factor for the system. This can be shown by looking at the equation that calculates our noise factor.

$$F = F_1 + \frac{F_2 - 1}{G_1} + \frac{F_3 - 1}{G_1 G_2} + \frac{F_4 - 1}{G_1 G_2 G_3} + \cdots + \frac{F_n - 1}{G_1 G_2 G_3 \cdots G_{n-1}} \tag{2.16}$$

While additional components do add to the noise figure, their contribution is significantly lower than the first LNA put in the RF chain.

This however is the only major change we need to do for doing radiometer work with an off the shelf radiometer. The rest will be done within software which is the principal reason for using a software defined radio.

2.2.5 Software

Software of course plays a critical role in a software defined radio and also in our software defined radio radiometer. There are really two pieces of software that are in play with the software defined radio we are using. The first is the firmware that used in the FPGA of the N200. However, this firmware currently does not implement the actual radiometer functions

but instead provides us a link to both the I/Q data and also the control of the of the SDR itself.

The second is the software that is running on the host computer. It is this software that provides the calculations on the I/Q data to give us the information we need and also creates a GUI for the user to interface with the radio. For this software, we will be using GNURadio, an open source software program that is used in many software defined radios including the N200 SDR that we have.

2.2.6 GNURadio

GNURadio fills in the software side of the software defined radio. Although there is firmware that runs on the FPGA in the N200, this firmware is designed to communicate with a host PC. It is this software that does most of the work in terms of the calculations that are done with the signal. The FPGA simply sends the raw IQ data to the host PC, which then performs the necessary math functions. Again, the reason why software defined radios are desirable is the ability to change the behavior of the radio very quickly. In our case we can change functionality by simply loading a new software program in the host PC.

This functionality is ideal for communication type of radios where different modulation schemes and encoding and decoding methods can easily be changed out. However, in a radiometer we are not interested in this aspect of the SDR. However, one functionality is available that can be very valuable for a radiometer, and that is with filtering. Although we often use frequencies that should be free from interference, this is not always the case. Interference can and often does still occur, even in these protected frequencies. With the SDR, we are able to quickly adapt to changing conditions by moving the frequency, changing our bandwidth and even filter out an offending signal.

GNURadio was selected as it is an open source software platform. GNURadio is licensed under the GPL license and has a strong community that continually updates the software. It is also well supported by third parties such as Ettus Research Group, National Instruments and other SDR developers. In addition GNURadio has a strong set of tools that can be used to develop programs that run under GNURadio. Tools such as the GNURadio Companion (GRC)

allows for an easy to use GUI to develop code for GNURadio. GNURadio is also written in Python, which allows for easy modification and access to additional tools that can be used with GNURadio.

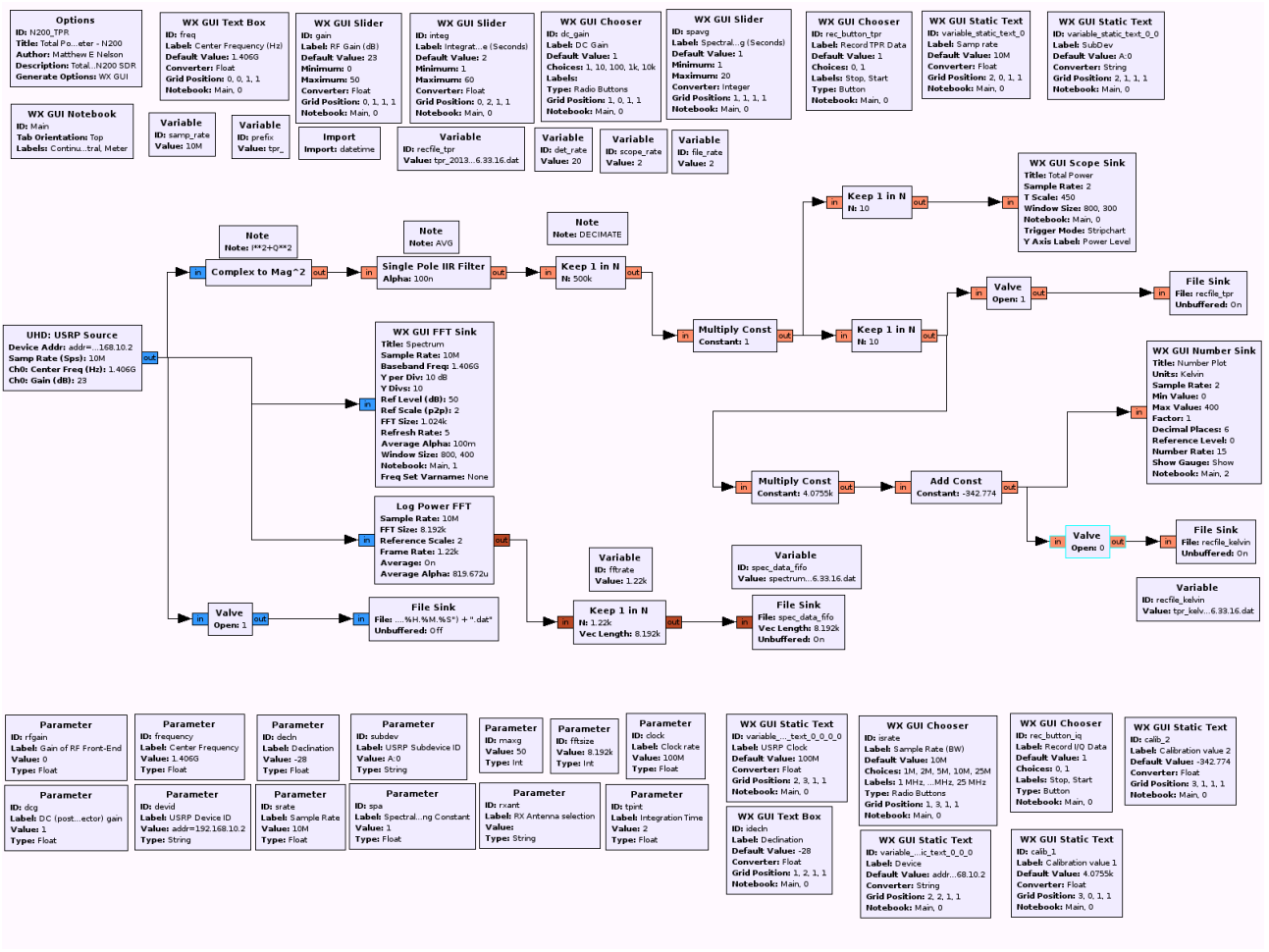

Figure 2.5 Screenshot of the GNURadio Companion editor and the N200 Radiometer block diagram used in many of the experiments

GNUradio Companion works by having many of the common functions as blocks that can picked and placed on the screen. Once placed, the blocks can be wired up, very much like LabView, and the flow of data can be controlled in this fashion. While GNURadio Companion provides most of the essential blocks used in most applications, additional blocks can by added if needed. This is due to the fact that GNURadio is built using Python and the blocks in GNUradio Companion are simply blocks of Python code. To "compile" a GNURadio Com-

panion sheet, you simply run the sheet, which then generates the Python code that is then executed. GNURadio actually uses a combination of Python and C++, where Python handles most of the interface and the C++ is usually most of the drivers and low level interface to the hardware. This allows for a easy to use system but still meets some of the demanding performances needed for handling large amounts of data.

GNURadio Companion also includes blocks that allow it to create a GUI type of interface. The typical method it uses for this is using wxGUI although GNURadio Companion does also include blocks that can use QT for generating widgets as well. However, the wxGUI tends to work better and has better support in GNURadio than QT.

Through these blocks can not only manipulate the data we need to perform a total power radiometer in software but to also create a user interface that allows us to control the radiometer as well. We are also able to display the information in real time so the user can see changes in power and even monitor spectral information during the operation of the radiometer.

2.3 General Specifications

In general, since we are digitizing the signal early on the components do not add much in terms of noise when we look at the performance of the radiometer, at least when comparing to a traditional radiometer. There are however other performance metrics that we need to look at that come in to play with a digital system. These performance metrics deal with how well the system can handle the bandwidth, the frequency and the resolution of the signal as it is digitized. The information below outlines the specification of the devices used and it's impact on the performance of the SDR as a digital radiometer.

2.3.1 N200 Software Defined Radio

The N200 is a SDR developed and built by Ettus Research. It is one of the newer models in the companies USRP line of SDRs. It offers several key features that were desirable for a radiometer type of application while still being an economical option. It also offered a highly flexible architecture which will allow this radio to be up-gradable in the foreseeable future.

The N200 has the following features that made it well suited for our specific application.

Figure 2.6 The USRP N200 from Ettus Research

- Dual 14-bit ADC
- Dual 16-bit DAC
- 50 MS/s Gigabit Ethernet streaming
- Modular daughter-board system for RF front end

These specifications had a large impact on the selection of the N200 for this application. Specifically the 14-bit ADC, the 50 MS/s and the modular daughter-board system were the largest factors in the decision to use the N200 SDR. Further explanation on these specifications are explained in the following sections.

2.3.1.1 14-bit ADC

The analog to digital converters (ADC) allow us to take the analog I and Q values from the daughter boards and digitize this information. Once digitized, we can now work with the

signal both on the on-board FPGA board or stream it to the computer so that our software can manipulate the signal. In radiometry, we are primarily looking at the overall power of the signal and this does not require us to accurately recreate the signal. However, as will discuss later there are times were we do want frequency information and having an accurate frequency representation of the signal will then be an important factor.

2.3.1.2 50 MS/s Bandwidth

For this application, the N200 is required to receive a signal at 1.4 GHz and is at least 20 MHz wide. Bandwidth plays an important role in remote sensing and to the amount of power that we receive. It also plays a key role in the sensitivity of the radiometer. This was discussed in the theory of a traditional radiometer and equation 2.2 shows us the relationship that our bandwidth plays in the overall power received.

Equally important however is the the sensitivity or how small of a change the radiometer can detect. This sensitivity or NEAT function is shown in equation 2.3. Again, the amount of bandwidth the radiometer receives plays a large part in the performance of the radiometer.

Another factor however is the amount of bandwidth the existing RF front end of the ISU radiometer is able to provide to us. The current filters on the ISU Radiometer keep the bandwidth of the signal to 20 MHz. This meant that at the very minimum a 20 MS/s bandwidth from the SDR is required.

The N200 is capable of working with up to 100 MS/s signal and can stream up to 50 MS/s through the Gigabit Ethernet connection. The N200 also has the ability to have up to 2 daughter boards installed. If we assume each will have up to a 20 MHz signal, this means up to 40 MS/s of data will be required. This means that the N200 meets and can exceed the bandwidth requirements that we were looking for. In addition, the FPGA on the N200 is capable of working with up to 100 MS/s, so there is room handling additional bandwidth by using the on board FPGA to process the signal.

2.3.1.3 The DBSRX2 Receiver

The Ettus Research N200 SDR uses daughter boards which allow for easy replacement of the RF front end to the software defined radio. Ettus Research makes a number of daughter boards that range from a a wide range of frequencies and is available with transmitters, receivers and transceivers. Because these boards are modular and do not touch the analog to digital converters or the on-board FPGA, very little change is required in the software. The daughter boards however do have the required RF hardware for the signal to be processed. In this application it was required that the signal was detected at 1.4 GHz with a bandwidth of 20 MHz. The DBSRX2 receiver met this requirement and was selected to be used with the N200. In this radiometer application transmission is not needed and is in fact illegal in the 1.4 GHz band, which is reserved for radiometer applications.

The DBSRX2 receiver board is capable of receiving signals between 800 MHz and 2.3 GHz. The board is able to plug into one of expansion slots available on the N200 that we are using. This board then down-coverts the signal into analog I and Q values that are then fed into the N200 analog to digital converter.

The DBSRX2 has several key components on it that is used to take the analog RF signal and prepare it for digitization by the analog to digital converter. First the signal is amplified through a Programmable Gain Amplifier (PGA). This PGA is accessible from the software and can be configured by the software. Next the signal goes into a direct-conversion integrated circuit that directly converts the RF signal to analog I and Q values. The integrated circuit, a Maxim MAX2112 device, also includes a Low Noise Amplifier (LNA), mixer and Low Pass Filter (LPF). This essentially amplifies the signal, mixes into baseband and then applies a low pass filter.

These analog I and Q values are then passed to the N200 to be sampled by the analog to digital converter. The IQ values are differential signals to minimize noise possible interference.

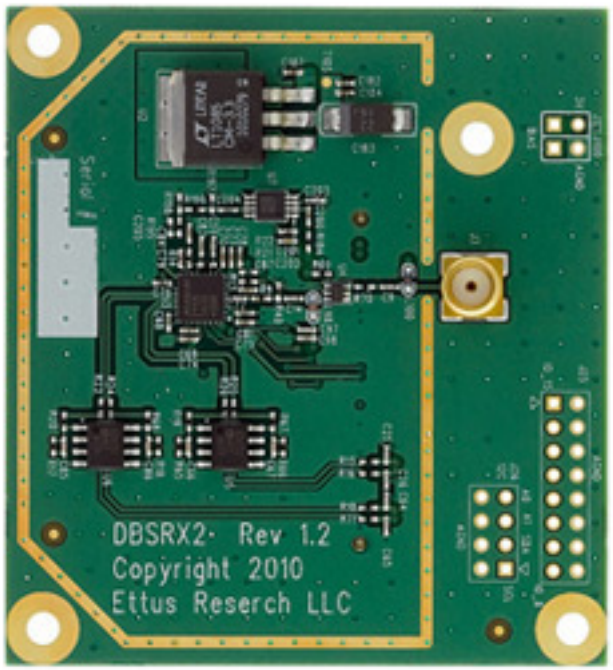

Figure 2.7 The DBSRX2 daughter board from Ettus Research

2.3.2 DBSRX2 Noise Temperature Consideration

The N200 uses daughter board cards that allow for us to easily swap out different RF front ends. The DBSRX2 was selected due to the fact that it is receive only and that it will work in the frequency spectrum that we are interested in, primarily in the 1.4 GHz range. Since we are using the DBSRX2 after the LNAs that are already on the, the noise temperature added by the DBSRX2 will be quite small. The DBSRX2 adds approximately 5 dB to the noise factor of the system. Again though, since this is at the end of the RF chain, the total contribution of the DBSRX2 to the overall system noise temperature is small, and has been calculated to be 1.05 dB to the overall noise factor of the system.

While the DBSRX2 does have additional gain, this gain is not as critical since we have the additional gain from the ISU radiometer front end. While the gain the DBSRX2 is significant, the noise figure on the DBSRX2 by itself is quite high. Therefore, the DBSRX2 would not be a good candidate if the N200 would be used solely by itself. Ideally, there would be at least

one if not multiple LNA's that have much better noise figure numbers than what the DBSRX2 and would be placed in front of the DBSRX2 daughter-board.

2.4 Comparison of a Software Defined Radio Radiometer vs Traditional Radiometer

As outlined above, a software defined radio implements a traditional radiometer but in the digital domain. For power detection, we simply sum the squares of the I and Q values that have been sampled by the software defined radios analog to digital converters. As with a traditional radiometer, we also want to filter this information to remove much of the jitters that comes from the rapid fluctuations in the power readings. This is done using an IIR low pass filter, which mimics a traditional RC low pass filter. Once completed, we now have the total power reading from the radiometer and we can now store, display or do both with this information.

A traditional radiometer may also use an analog to digital converter in order to digitize the analog voltage from the square-law detector. Because the sample rate of this voltage is very low, almost any analog to digital can be used for this. At this point however, there is no frequency information, only the magnitude information is being retained and recorded.

2.5 Comparison of a Software Defined Radio Radiometer and other Digital Radiometers

A digital radiometer is not a new concept. Early radiometers would often digitize the analog voltage information from the square-law detector and then send that information to a computer for storage or analysis.

A pre-cursor to a software defined radio, some radiometers would also digitize the incoming RF signal, but under-sample this information. Since only power is the information desired this was acceptable. However, these radiometers did often use the same components you might find in a software defined radio such as an A/D converter and FPGA. These components however were used in different ways.

One reason why these devices were used differently from a SDR was due to cost. Most

radiometer operations happen at 1.4 GHz or above. A/D converters at these higher frequencies become more expensive and harder to obtain. In recent years however, these costs have come down.

The major difference between other digital radiometers and what is discussed in this thesis is that we retain both phase and magnitude information and instead mimics a traditional radiometer in software by summing and squaring the I and Q values and then running this information through a low-pass IIR filter. By retaining this information, we can perform a more in-dept analysis of the signal coming into the radiometer which allows for greater agility in the system.

CHAPTER 3. IMPLEMENTATION OF A SOFTWARE DEFINED RADIOMETER

One of the principal goals with this research was to implement a fully functioning total power radiometer within software. The N200 provides us the link between the RF signal captured by the antenna and converts that to a format that the computer can now use to manipulate the signal. Once the signal has been passed to the computer, GNURadio will implement the correct algorithms to detect the power within the signal, filter and output the information. One of the advantages of course with a SDR is that filtering can also be done within the software. In addition, thanks to the WxGUI that GNURadio uses, we can also build a user interface that can control several key variables that are useful for us. This includes controlling the gain on the programmable gain amplifier on the DBSRX2 card, the sampling or bandwidth of the signal, the center frequency, and also the integration time. All of these can now be controlled in real time as well. GNURadio will also store the power data so that we are able to do further analysis of the data using a software program like Matlab. Because GNURadio is a very flexible system, much more can be done with the signal if needed and future updates may add more capabilities or additional analysis on the signal can be achieved.

3.1 Requirements

Requirements for this system was based on information provided by Dr. Brian Hornbuckle and was also based on requirements of a typical radiometer. These requirements are outlined in table 3.1 below.

Table 3.1 ISU Radiometer requirements

Requirement	Value	Units
Frequency Range	1400 - 1425	MHz
Bandwidth	25	MHz
Polarization	Dual	
Sensitivity	-30	dBm
Accuracy	1	Kelvin

3.1.1 Hardware Requirements

The selection of the N200 SDR from Ettus Research was based on many of the requirements outlined in table 3.1. In addition, we wanted the hardware to be flexible but also affordable. There are many different kinds of software defined radios on the market. However, we choose the N200 based on the availability of the device, the large community support, especially with regards to support by GNURadio, and because it meet and often exceeded the requirements stated above.

Flexibility was another key aspect of the N200 that made it an excellent selection for this research. The N200 uses a daughter board setup for bridging the RF interface to the rest of the electronics. Several different daughter boards are available that have different frequency ranges and offer both receive, transmit and transceiver designs. For the radiometer we need to operate around 1.4 GHz and receive only. Based on those requirements we choose the DBSRX2 daughter board. This board is designed from 800 MHz to 2.3 GHz and has a fairly low noise figure.

The current RF front end to the radiometer is designed for a 20 MHz wide signal. This requirement was one of the main driving points for selecting the N200 SDR as it can support up to 50 MHz in bandwidth between the N200 and the host computer. This is accomplished by using a 1 Gbps Ethernet connection between the N200 and the host computer.

3.1.2 Software Requirements

The driving force for the software requirement was to have a system that was easy to use. One reason for the development of this platform is to make radiometers more accessible to other

researchers and other programs such as education and even amateur radiometer work. Therefore ease of use was taken into consideration when selecting the hardware and the associated software used with it.

With the switch to a software defined radio platform, we switched to having most of the processing done by the host computer with digitization and low level processing accomplished by the software defined radio's FPGA. However, we still wanted to keep the interface hardware fairly simple. GNURadio was selected as it includes GNURadio Companion (GRC) which uses a graphical interface for creating the radio environment. It also includes options to create a user interface during the operation of the N200 as well. This allowed us to rapidly create both the critical radio components needed for the radiometer and also a control interface.

3.2 Power detection

Power detection is a key ability that allows a radiometer to function. At its core a radiometer is a power detector. Therefore the implementation of power detection is a crucial function of a software defined radio radiometer.

To implement a total power radiometer in software, we first need to look at how we implement a total power radiometer traditionally. Traditionally, a square law detector is used to detect the average power that is seen by the radiometer. This simple device uses a diode that gives a small voltage output based on the RF power present. This small voltage is then amplified and can now be calibrated with a known source to give us a noise temperature.

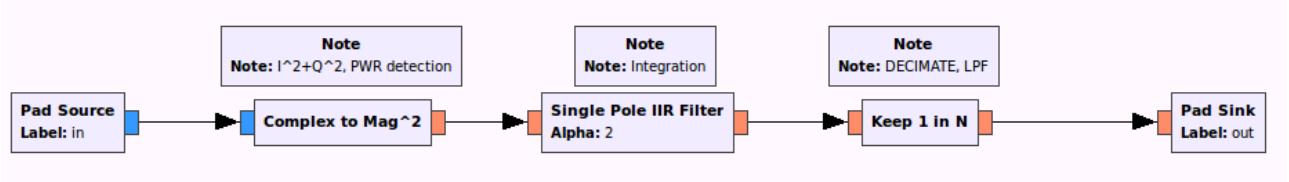

Figure 3.1 A block diagram showing how the radiometer performs the equivalent square law detector in software.

To implement this in software, we need to build a square law detector mathematically. We can then give this to the software defined radio to process the information accordingly. A square

law detector mathematically, is the sum of the squares. Once the signal has been digitized, it is expressed in data bits of I and Q, which represent in-phase and quadrature-phase of the signal. By squaring each term, we get the desired result of the power of the signal [**?**)][**?**)].

3.3 Integrator through a IIR Filter

Another step that we typically do in a traditional radiometer is to integrate the signal over time. This gives us an average of the signal and helps to smooth out the output. In addition, we will show later that the integration time can be adjusted to help improve our sensitivity of the radiometer.

In a traditional radiometer, we can integrate by using a simple integrator circuit, which consists of an op-amp, resistor and capacitor. This type of circuit is also equivalent to a low pass filter circuit as well, and the two are interchangeable. We can then look at how we filter in the digital domain, and this is down with a infinite impulse response filter or IIR. We can use this type of digital filter to then integrate the signal for our total power radiometer.

3.4 GNURadio Blocks

A software defined radio by its definition requires software for the proper operation of the radio. In our case there is the software or firmware that resides on the FPGA and then the software that runs on the computer. For this thesis, we focused on the software running on GNURadio. This software package provides us with the ability to write the code needed to implement a total power radiometer but also gives us a method to control the SDR as well. It also allows for us to record the information for performing post-processing on the data.

3.5 Control of the SDR Hardware through GNURadio

The N200 sends all data across the 1 Gbps connection to be read in by a host computer running GNURadio. This data is the raw I/Q values that is read by the on board A/D and processed by the on board FPGA. An example of a very simple GNURadio software implementation would simply take this data and store the data to a hard drive in a file. This

can be very handy if we want to simply record the data and then process it later. However, depending on the sample rate, it can consume a large amount of storage. A short recording can easily consume 1-2 GB with a sample rate of 10 Msps. It also does not give us any immediate feedback on the radiometer and it does not give us controls of the radiometer such as frequency, integration time or other key variables. Fortunately GNURadio has tools that allows us to build up a very rich application that is able to give us the data we need and control the software defined radio as well.

The GNURadio Companion allows us to create python code that is used to not only receive the data from the SDR but also perform calculations, signal processing and also to send control information to the SDR. This allows us to build up an application that can be run on any computer that is capable of running GNURadio.

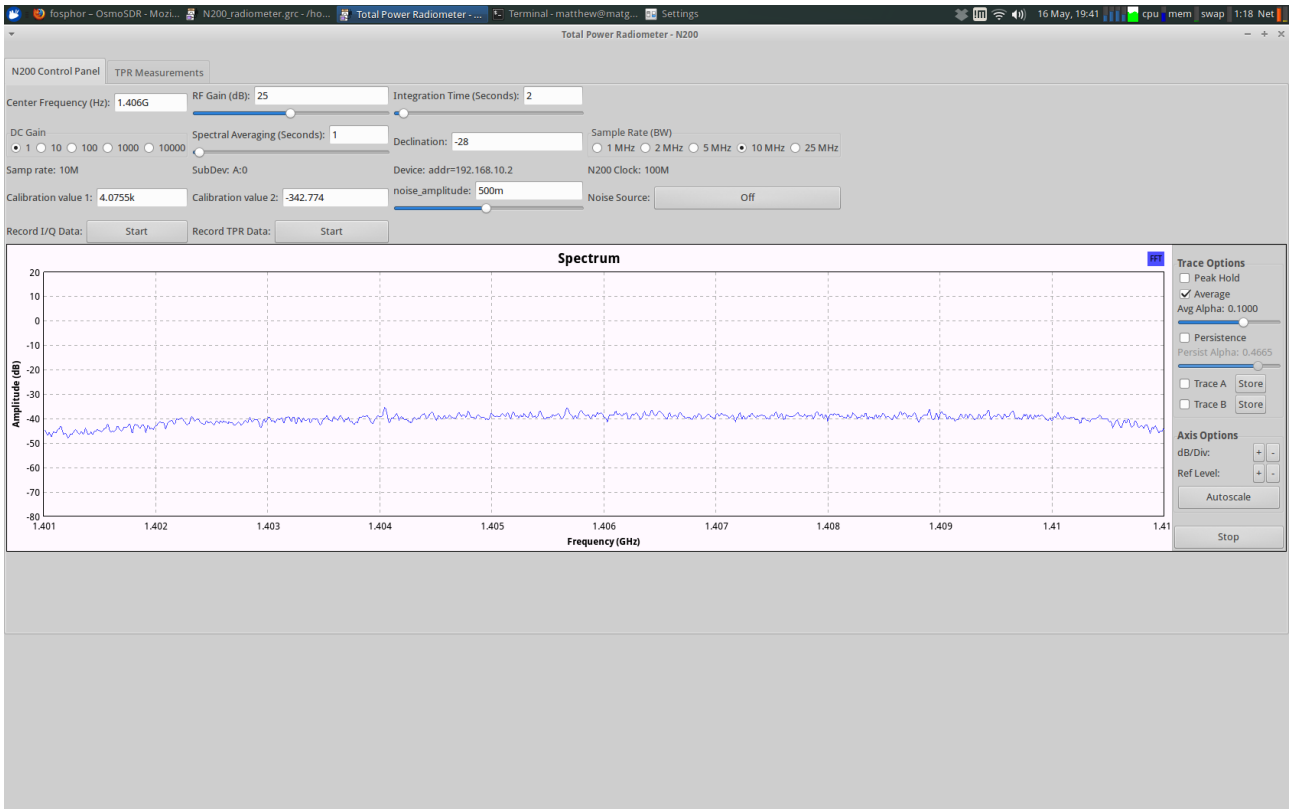

Figure 3.2 A screenshot of the interface made for communication with and controlling the software defined radio

Through this interface we are able to control several key aspects of not only the radio

hardware within the SDR but also with the behavior of the radiometer as well. Hardware control of the SDR hardware allows us to change frequency and also adjust gain values within the N200. Bandwidth is another parameter that can be altered from here as well. Bandwidth affects the bandwidth for the RF signal, but also has an impact on the radiometer sensitivity as well. Additional controls allows for alterations to the radiometer and includes being able to adjust the integration time of the radiometer.

3.5.1 Impact of the Controls Related to Radiometry

The controls that have been added for controlling the radiometer can have a large impact on the performance of the radiometer. There is a reason why these controls were added to the GUI for the radiometer, they play key roles in how the radiometer performs.

For any radiometer noise temperature is a large consideration and is critical to the design of the radiometer. One method to determine how well a radiometer is to look at the sensitivity of the radiometer. We can do this by looking at the smallest change in temperature the radiometer can see. We will call this the Noise Equivalent ΔT or $NE\Delta T$ of the radiometer. The equation for this is shown below.

$$NE\Delta T = \frac{T_A + T_{sys}}{\sqrt{\beta * \tau}} \tag{3.1}$$

β can be changed by changing the sample rate of the SDR. The sample effectively controls the bandwidth in which the SDR is operating at. This also gives us a band-pass filter as well, since the SDR will not respond to frequencies outside of this bandwidth.

τ is the integration time for the radiometer. This parameter is set by the user through the GUI and allows us to change the integration time in seconds.

3.6 GNURadio Data Handling

Once we have the data that has been processed by the software defined radio, we will want to display this information and be able to store the data so we can analyze it later if needed. Data display can be handled GNURadio, where we can plot the total power over time. This

allows the user to be able to visualize the total power and be able to determine if the total power has increased or decreased over the time window shown.

Although not usually needed for a total power radiometer, we also have the ability to look at the signal in terms of a frequency versus amplitude. This allows to look for any unusual signals that may be interfering with or causing erroneous data with our radiometer.

Finally we will want to store the data so we can do additional analyses on it at a later time. The GNURadio program allows us to store the data in two formats. The first format is storing the raw I/Q data from the radiometer. This format allows us to "playback" the data through GNURadio at a later time. This can be useful for if we wish to change parameters in GNURadio such as bandwidth or integration time. It can also be a good diagnostic tool as we can check that the signal coming in is clean or if we need to apply additional filters to remove an unwanted signal. It should be noted that this file can be quite large, consuming several gigabytes of data for a 20 MHz wide signal in a matter of minutes of record time.

The second format is the total power that has been calculated by the radiometer. This file is much smaller in size since much of the signal information has now been reduced to simple power versus time information. This allows for easy manipulation through any type of math program such as Matlab for analysis.

3.6.1 GNURadio Data Display

The information from the software defined radio can be displayed through GNURadio to show a number of things. Since we have both frequency and magnitude information we can display this information. We are able to also display the information that shows the total power that is being seen by the radiometer as well.

We are not limited to just total power from the radiometer. If the radiometer has been calibrated, those calibration points can be entered and GNURadio can calculate the calibrated noise temperature. Additional information may also be added as needed. For example, we are able to view the full spectrum that the radiometer sees. This can be a useful tool for looking at potential RFI issues.

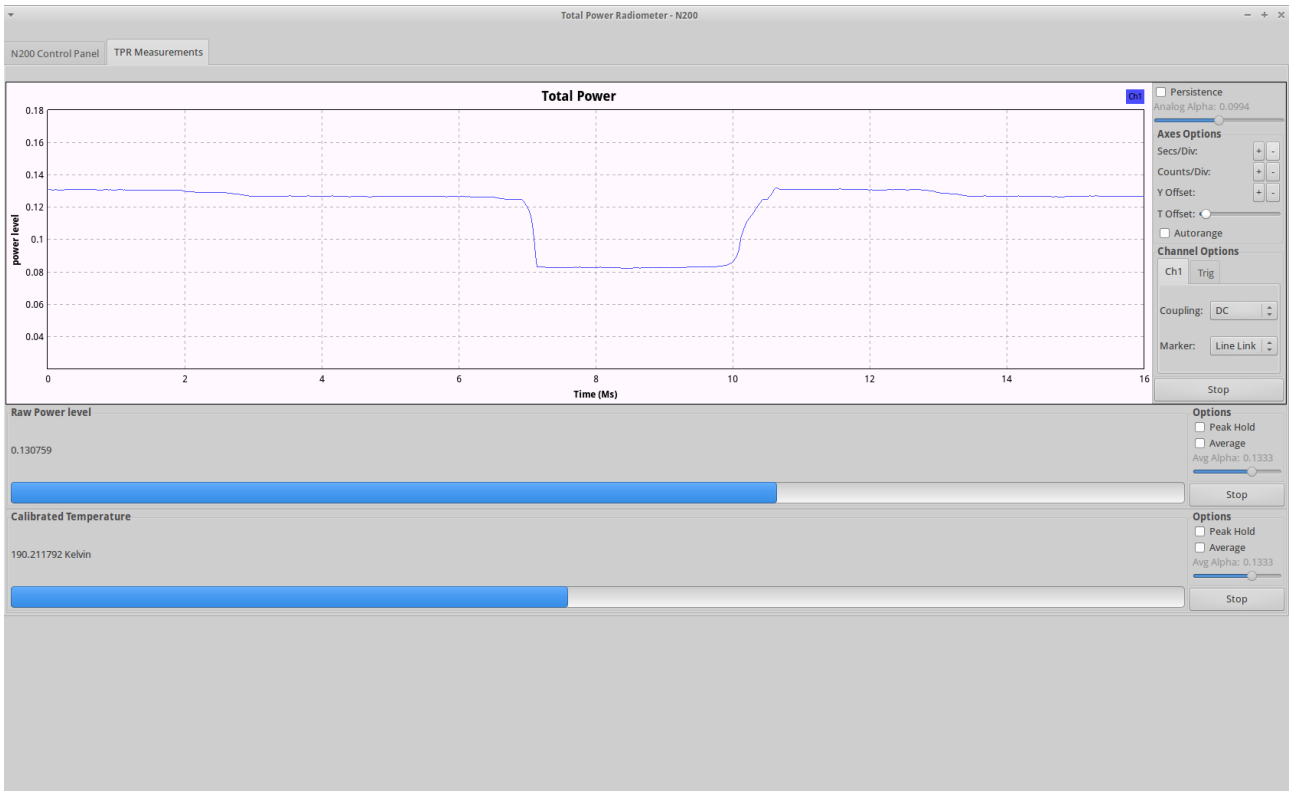

Figure 3.3 A screenshot showing the ticker tape display for the total power readings. In addition raw and calibrated noise temperature is shown below.

3.6.2 GNURadio Data Processing

GNURadio is capable of storing information in files that can be processed later. These files are binary formats that are stored in a little-endian format and can be a character, integer, float or complex. A simple example is storing the raw I/Q values to a file. This file can then be processed by Python, GNURadio, or even Matlab. For example, we can store the raw I/Q values and then play them back though GNURadio. Of course other information can be stored as well and we use this same method to store the total power radio values to a file as well.

Matlab is one tool we can use to process the information that is stored by GNURadio. Appendix A contains the Matlab source that will read the total power file generated by GNU-Radio. It then calculates information such as the NEΔT and the calibration points based on the user input. We can also use Matalb to graph this information as well.

While Matlab is one tool, other tools can be used. Python for example is also capable of

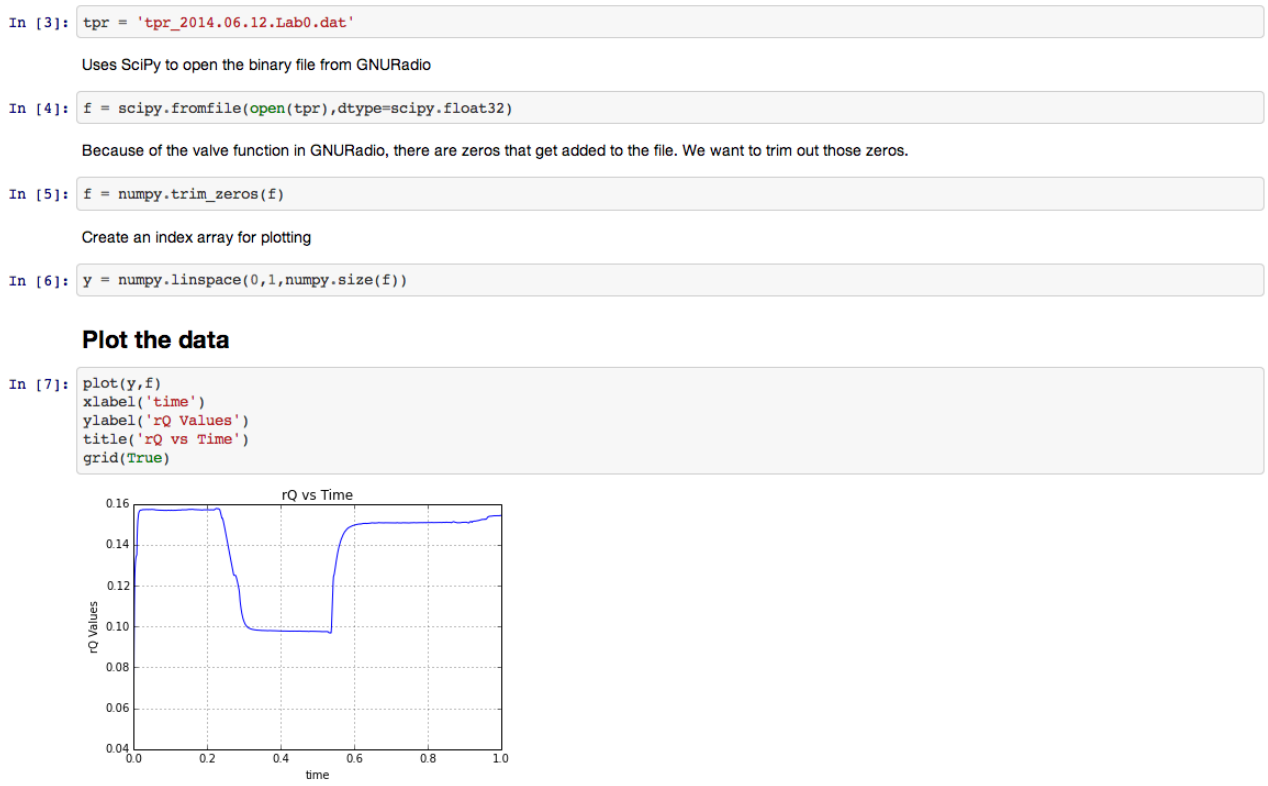

Figure 3.4 A screenshot showing the Python code and related graphs generated for parsing GNURadio data

reading in these files and when paired with NumPy and SciPy can be used to perform analysis on the data as well. In addition, the open source mathematical program Octave should also be able to read and work with these files. For this thesis both Matlab and Python was used to provide analysis on the data.

CHAPTER 4. EVALUATION AND EXPERIMENTAL SETUP

Testing and verification ensures that the new components that we have added to the system are working as intended and has not caused a negative impact on the overall system performance. To do this, several tests were conducted and verification was obtained by using a well known method of detecting power in a radiometer, a square-law detector.

Testing began with the square-law detector as this was one of the first components that was obtained. Next, testing was done with the the software defined radio as a whole. Finally a cold bath test was done with both the software defined radio and with the square law detector as a system to verify both functionality and to compare the results from both devices.

Additional testing was also performed to test additional functions that are not found on a typical radiometer. This test involved injecting a signal and having the software defined radio remove the offending signal. A comparison was then performed to show the difference between the SDR radiometer which is able to remove the offending signal and the square-law detector which is not able to remove it unless additional hardware filters are placed in front of it.

In addition to these tests, a few real world tests were also done in the Spring of 2012 and the Spring of 2014. These tests were conducted by Dr. Brian Hornbuckle's E E 518 class was conducted as part of their lab requirement for the test. These test results can be found in Appendix 2.

4.1 Square-law Detector

To verify the results of the information that the software defined radio is obtaining, a square-law detector is used to measure the power of the incoming signal in parallel to the software defined radio. The incoming signal is split using a power divider so that the signal

will be the same to both devices with the exception of the 3 dB plus insertion loss the power divider adds. This allows us to verify the software defined radio with a proven system. Square-law detectors have traditionally been used in radiometers and have been proven to work in radiometer applications. They are also a very simple device which means there is little that can go wrong with using them.

4.1.1 Analog Devices ADL5902

The square-law detector that we obtained is the Analog Devices ADL5902. The ADL5902 is a true rms responding power detector that has a square law detector, a variable gain amplifier and an output driver. It also has a temperature sensor and will compensate for temperature variations. The output driver allows for the small signal from the square law detector to be amplified to a level that most analog to digital devices can detect. This driver however does have low noise and has a noise output of approximately $25nV/\sqrt{Hz}$ at 100 kHz. The ADL5902 operates from 50 MHz to 9 GHz and in most cases can detect down to -60 dBm. This works well in our application since the radiometer operates at 1.4 GHz and after the amplification stage we usually see between -40 to -30 dBm of power.

The specifications of the ADL5902 can be seen in Table 4.1 shown below.

Table 4.1 ADL5902 Specifications

Parameter @ 900 MHz	**Value**	**Units**
Frequency Range	50 to 9000	MHz
Dynamic Range	61	dB
Minimum Input Level, ±1.0 dB	60	dB
Maximum Input Level, ±1.0 dB	1	db
Logarithmic Slope	53.7	mV/dB
Output Voltage Range	0.03 to 4.8	V

The ADL5902 outputs an analog signal that falls between 0.03 volts and 4.8 volts. It outputs a change of 53.7 millivolt per dB detected by the ADL5902.

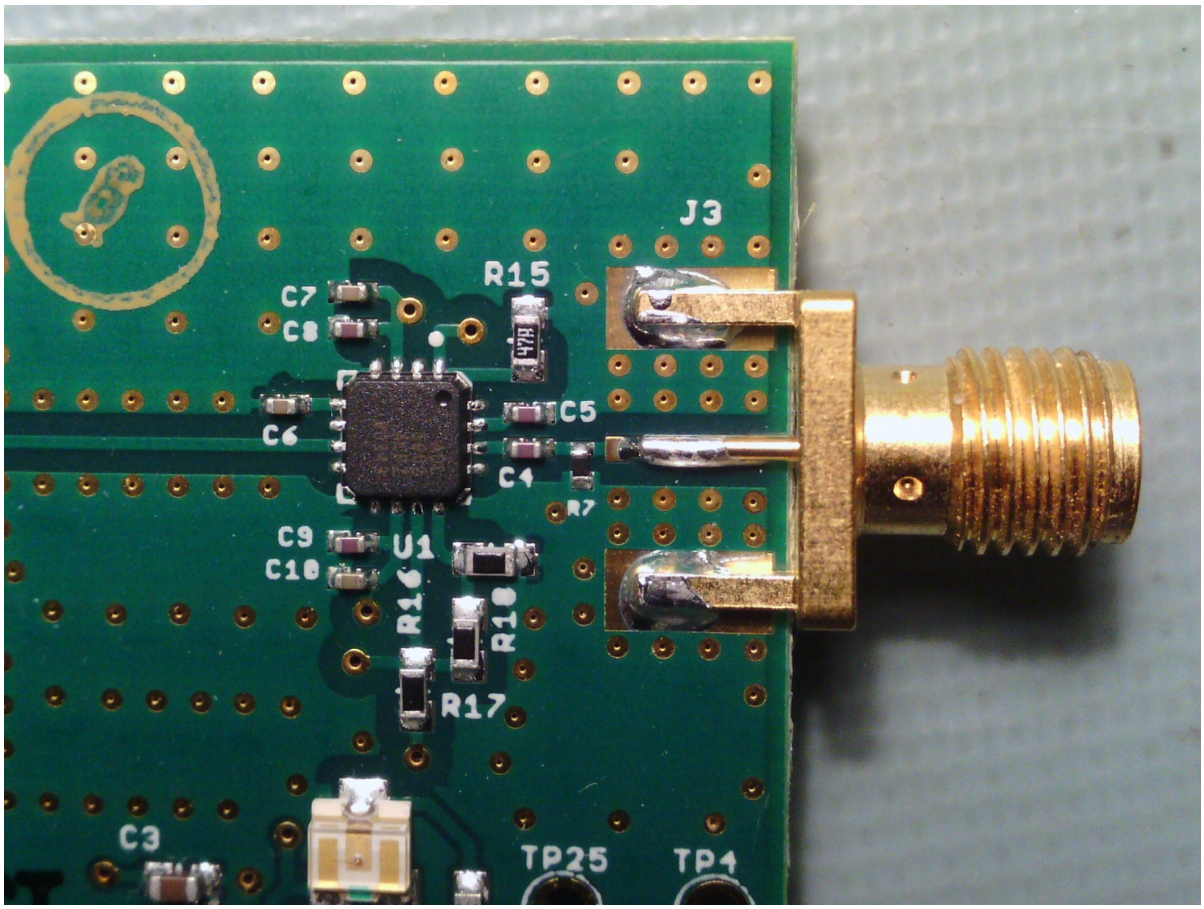

Figure 4.1 An image of the ADL5902 square-law detector used in these experiments

4.1.2 Data acquisition and storage

In order to analyze the data, a method was developed to acquire the data and store it for later use. For the N200 software defined radio, this is done automatically by the GNURadio program. Both the complete signal and the power information is stored to a file for later analysis. The square-law detector however outputs information as an analog voltage that is linearly proportional to the RF power measured. This required a system that can capture the analog signal from the ADL5902 and then send the data to be stored. This was accomplished by using a National Instruments USB-6009 data acquisition unit. This unit has 8 analog inputs that can sample up to 48 ksps with a resolution of 14-bits. This was more than adequate for our needs. To use the USB-6009 a fairly simple Lab View program was created. This program

retrieved the information from the USB-6009 and stored the data in both Lav View's binary format and in a more human friendly ASCII format. The USB-6009 then connects to a host computer through the USB interface. This made obtaining the data and using device fairly straightforward to use.

4.1.2.1 Labview Acquisition program

The Labview program was created to talk to the USB-6009 and then store the data. In addition, it is able to display in real time the current readings coming from the USB-6009. This was beneficial during testing of the program but is also good information to have while running an experiment.

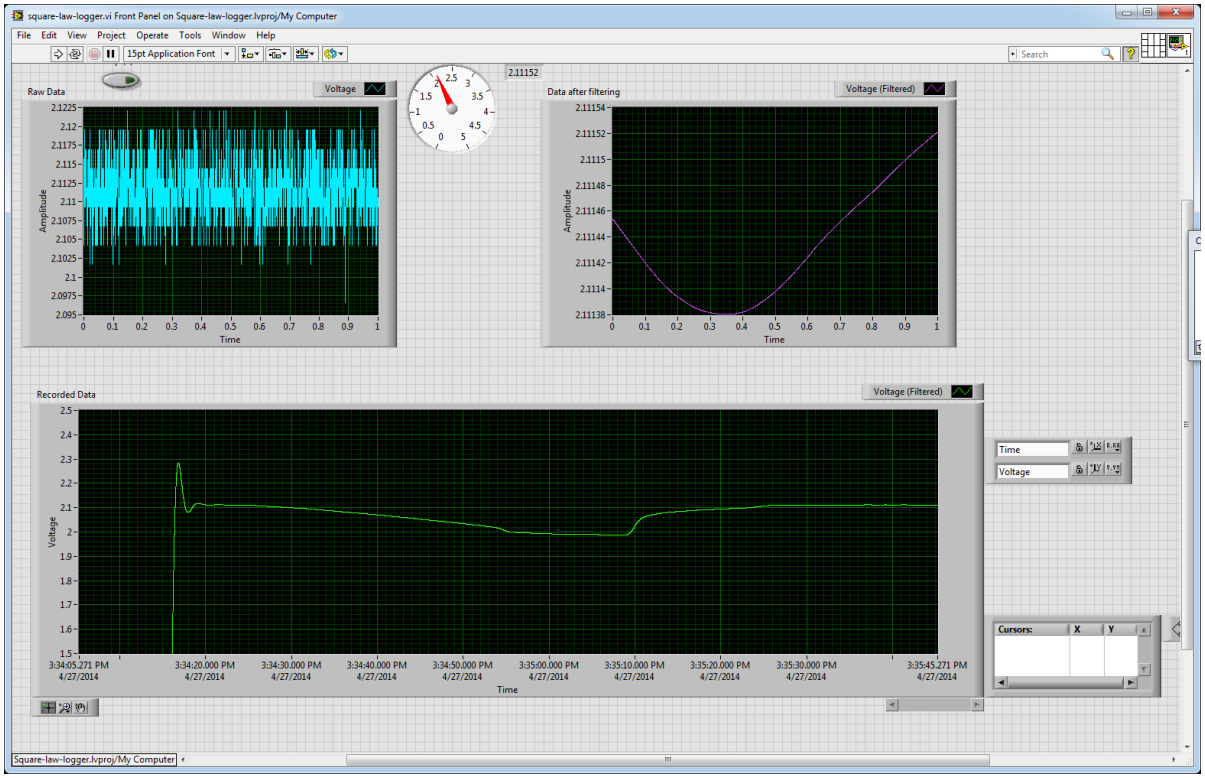

Figure 4.2 A screenshot of the Labview GUI interface

The Labview program uses National Instruments DAQ assistant which allows for quick configuration and setup for the computer to talk to a number of NI devices. Labview also includes blocks that allows us to easily record the data to a file. These blocks made up most

of the program and resulted in a program that was quickly made.

While the USB-6009 allows up to 48,000 samples per second, we don't need such a high rate. A rate of 1,000 samples per second was determined to provide more than enough samples of data. With such a high rate of samples however, the resulting output will be noisy due to the natural noise found in the RF signal. Like the software defined radio, we want to filter this noise as well to produce a smoother output. This was accomplished using a filter block in Labview to help reduce the noise.

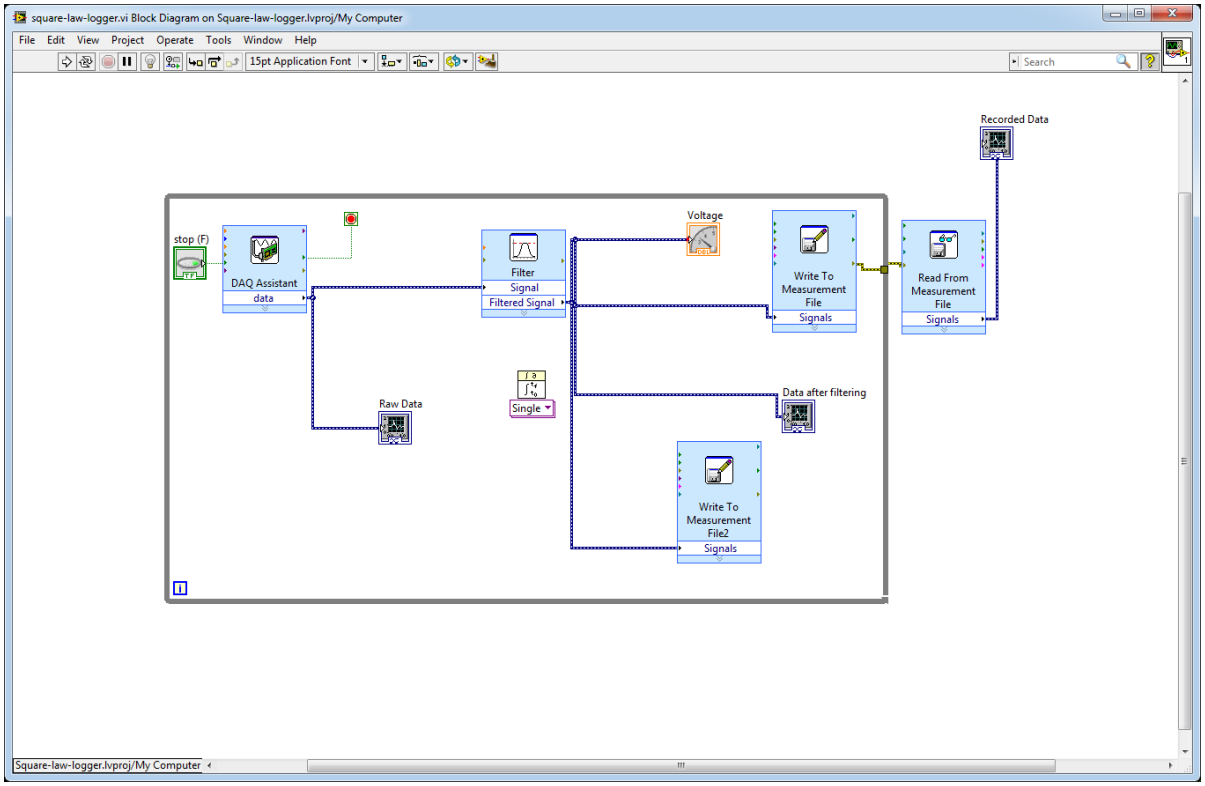

Figure 4.3 A screenshot of the Labview block diagram

This program allowed for the data from the square law detector to be easily recorded. Using Labview also allows us to customize the interface more to our liking and other features such as adding calibration information can also be added as well.

4.1.3 Tests on the ADL5902

To test the ADL5902 a signal generator was used that had a controlled output. The specific signal generator that was used was an older model that allowed us to change the output in 10 dBm increments. The signal generator was also configured for 1.4 GHz as that is the frequency the ISU RF front end is configured to listen at. Ideally a noise generator would have been desirable, however a noise generator with adjustable power output could not be located on campus.

The ADL5902 is available from Analog Devices in an evaluation board. This board pairs the ADL5902 with a AD7466 12-bit analog to an analog to digital converter. This board can then be mated with Analog Devices BlackFin processor which acts as a USB gateway for the AD7466 data. A test program written in LabView is also provided as well.

The test program provided by Analog Devices allows us to query the ADL5902 and record the raw ADC value. The test program also allows us to enter in the frequency used during testing and the temperature during the test. The test program also allows us to calibrate the system as well. All of the data is then stored into an Excel spreadsheet which can be accessed later.

For this test we used the signal generator set to 1.4 GHz and started at -60 dBm for the output signal. This was selected as this is the lowest the square law detector can detect. The output power was then incremented on the signal generator in 10 dBm steps. There was no change to any other parameter. This was done up to 0 dBm. Any higher and there was risk of damage to the ADL5902. This test was then repeated several times and was done with the signal generator stepping up from -60 dBm to 0 dBm and from 0 dBm to -60 dBm.

The data collected was then graphed using Excel. The graph shown in 4.4 Shows that the ADL5902 has a linear output and matches the expected value based on the input.

Once it was confirmed the ADL5902 does in fact have a linear response, we then looked at ways to work with the ADL5902. The graph above was generated using Analog Devices program that talked to the Blackfin processor on the daughter-board. However, the information for talking to the Blackfin is proprietary. Therefore the daughter-board was removed for future

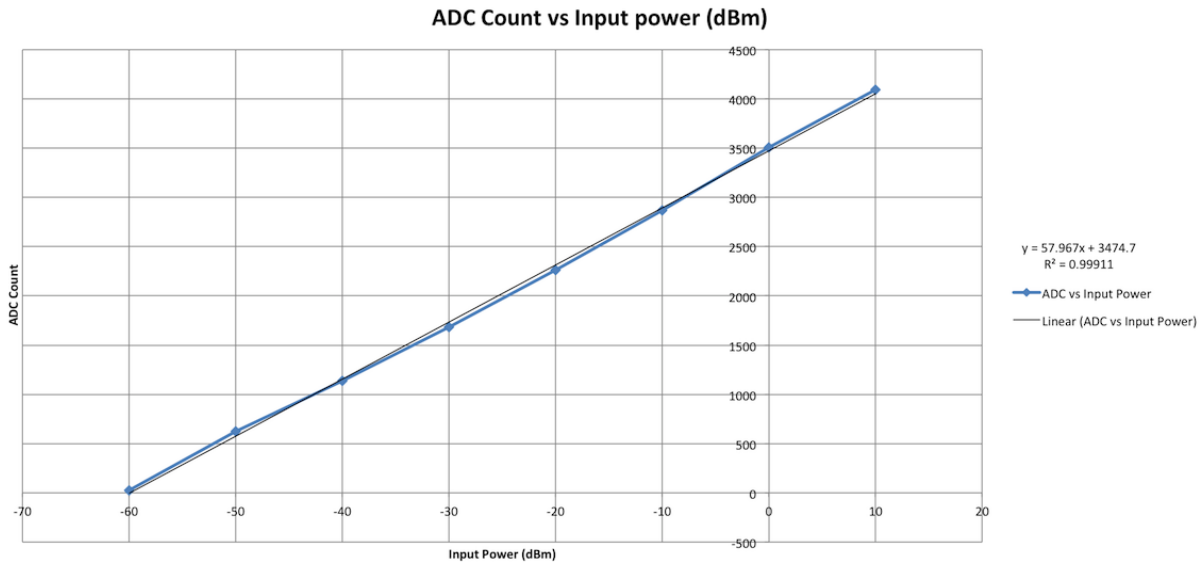

Figure 4.4 Graph showing the linearity of the ADL5902

tests and instead we used the USB-6009 data acquisition board to read the raw analog voltages from the ADL5902. This information was then stored later for additional analysis.

4.2 Software Defined Radio tests

Once it was confirmed that the square-law detector was working within the specifications that were given, testing then moved to the software defined radio. Once the Python program was established for replicating a total power radiometer in software, this was then loaded into GNURadio and used to control the N200 SDR.

Before testing the program with the N200, testing was done with the built in noise generator in GNURadio. This was used to test it's ability to measure small changes in noise. This simulated noise verified that the program written was able to detect changes in noise power using a simulated Gaussian source. It was also desired to also use a hardware based noise generator, however a suitable noise generator could not be located on campus.

4.3 Total Power Radiometer Test with Ice Water Bath

To fully test both the total power radiometer in the software defined radio and the square-law detector, a cold water bath experiment was established to verify that both the square-law

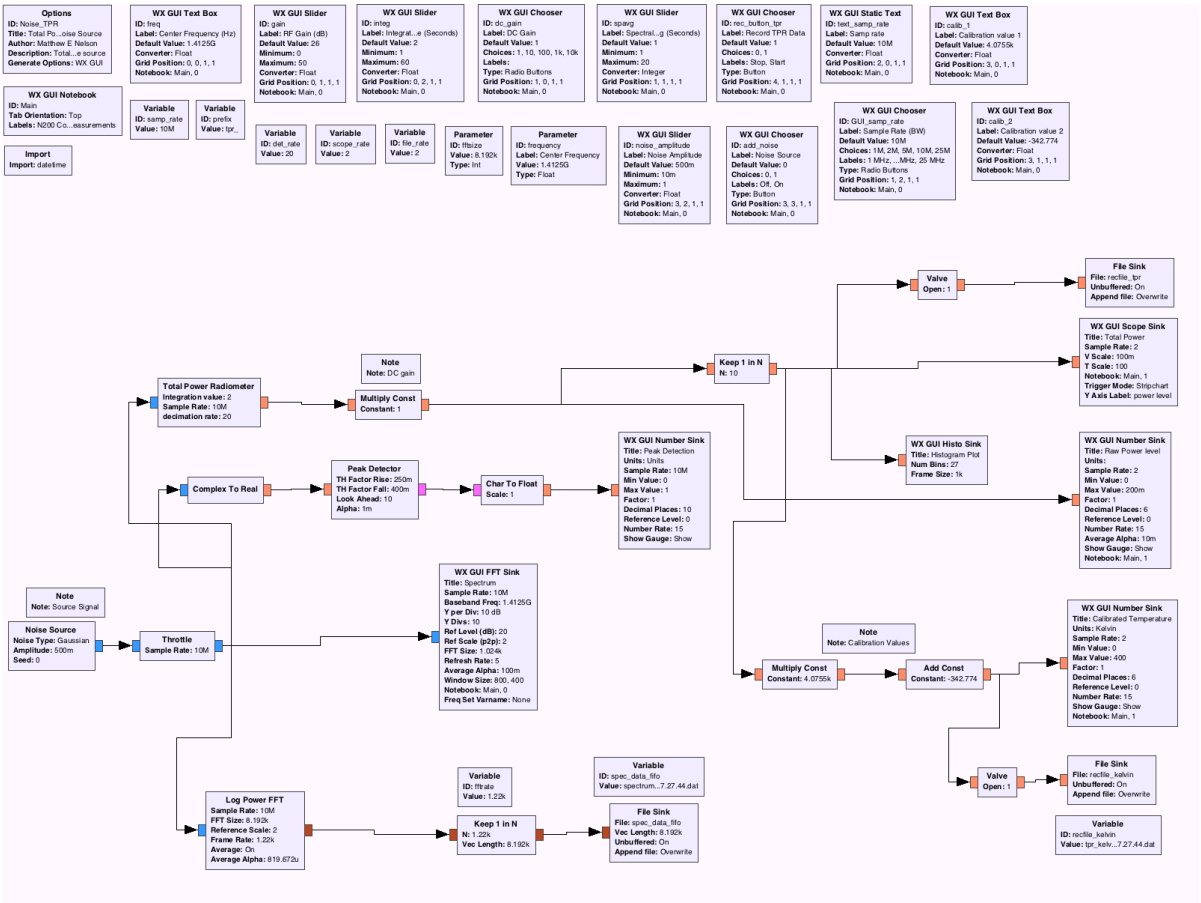

Figure 4.5 A screenshot of the GNURadio program using a noise generator block.

detector and SDR was able to measure real world changes in the noise. In addition, this would also give us a calibration line to to calibrate the radiometer.

4.3.1 Laboratory Setup

For this experiment the, ISU radiometer was setup in a laboratory that had access to measurement tools. To measure the change in noise, a 50 ohm matched load was attached to the radiometer. The ISU radiometer, with the current filters and LNAs would then amplify and filter the signal. This signal was then measured by the square-law detector and the N200 SDR. The 50 ohm load was then submersed into a hot or cold bath. These baths were temperature controlled first by using Liquid Nitrogen (LN2) which is known to boil at 77 K. Second a ice water bath and a boiling water bath was also used. The temperature of these could easily be

monitored and maintained. The load was submersed in each bath for 2 minutes to allow for it to reach the same temperature as the bath. The noise measured was then recorded using GNURadio with the N200 SDR and with the USB-6009 to record the square-law data. In addition to the total power measurements, the raw I-Q data was also recorded. This allowed us to replay the experiment through GNURadio for further study.

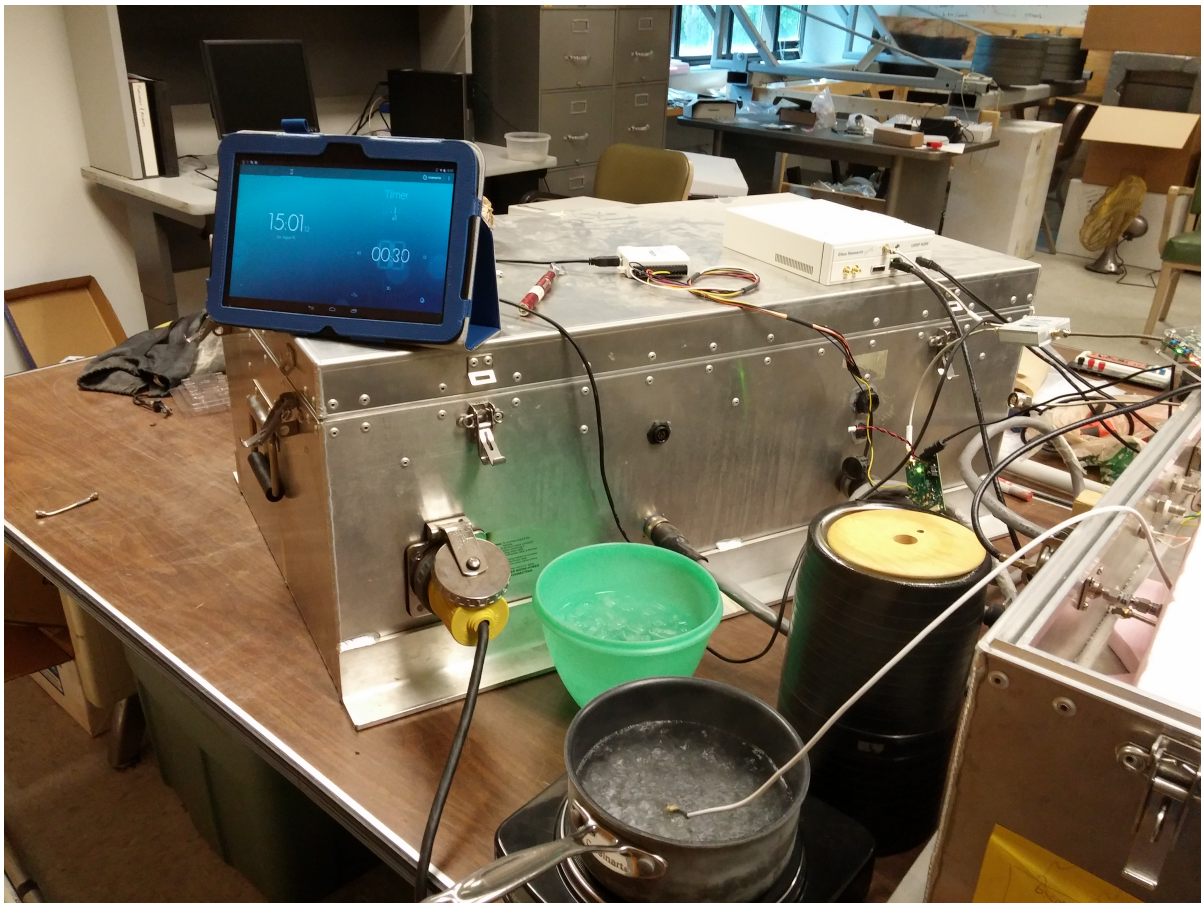

Figure 4.6 An image of the typical lab setup used to test the software defined radiometer

4.3.2 Test Results

Several experiments were conducted with this cold-bath configuration to establish that the results were reproducible. In each case, the experiment showed that both the square-law detector and the SDR were able to measure a change of noise temperature. When the results were calibrated to the known temperatures, both the square-law detector and the N200

showed identical readings. In terms of performance, the N200 showed slightly higher sensitivity, however this was expected as the N200 does have additional amplifiers which are not available to the square-law detector.

4.4 Liquid Nitrogen Test

The Liquid Nitrogen Test was conducted to verify that the radiometer was able to operate within the specifications that were given for the radiometer. This test is also a fairly common test for testing and calibrating a traditional type of radiometer. With this test, we can test the radiometer by measuring extreme values for the noise temperature. To do this, we submerge a matched load attached to the radiometer into a liquid nitrogen bath. This cools the load, which represents our source, to a physical temperature of approximately 77 Kelvin. We can then select several "warmer" temperatures such as a ice water bath, room temperature or even boiling water. Since we expect that the radiometer is linear in how it responds to these noise temperatures, we can then build a calibration line for the radiometer.

Additional tests were conducted with keeping the matched load submerged for an extended period of time. This was to determine the stability of the radiometer as it is expected that as long as the LN2 is covering the matched load, it should maintain the temperature at 77 K.

4.4.1 Testing Apparatus

To run this test, we need to have some equipment for this. First and foremost, we need a radiometer. For this test we used the ISU Radiometer front end, with the LNAs and bandpass filters in place. The noise diode was turned off for these experiments. A fifty ohm matched load was then attached to low loss coax and represented our source to the radiometer. Finally, the output of the radiometer was run to the Ettus N200 software defined radio instead of running the on-board Analog to Digital converter and FPGA. The data from the N200 was then sent to a computer running GNURadio and the custom software that I had written. This data is sent to the computer through a 1 gigabit Ethernet connection due to the large bandwidth coming from the N200.

4.4.2 Calibration and verification experiment

The first test run was conducted to help verify that the radiometer was operating correctly and to start to build up data for calibration. This test was conducted by placing the matched load into the liquid nitrogen bath, which has a boiling temperature of 77 K and then putting the matched load into a ice water bath which has a temperature of 273.15 K. This would give us two points to figure out a calibration point.

During this experiment both the square law data and data from the software defined radio was recorded. This information was then feed into a Matlab script that is able to read the data files, perform the needed calculations and then plot the data.

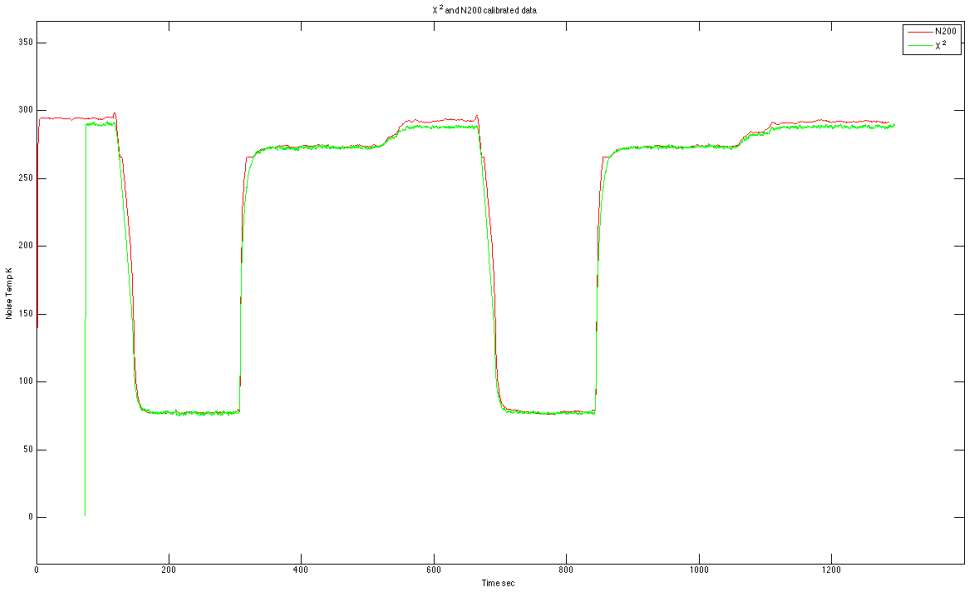

Figure 4.7 Graph showing both the square-law detector and the software defined radio total power readings.

Data from the N200 was stored to file using the GNURadio file sink which stores the data in a binary format. The square-law detector was feed into a LabView DAQ which then stored the data in both an ASCII and LabView's TDMS file format. In addition the LabView program displayed the square-law data in real time

This experiment showed that both the square-law detector and the software defined radio

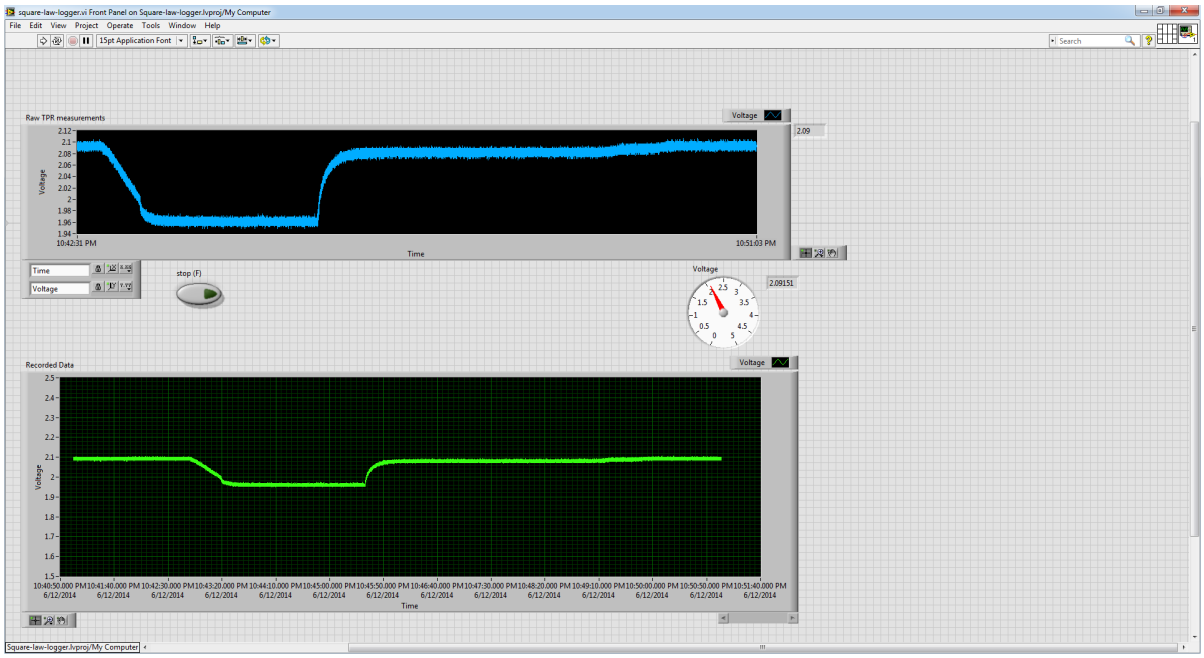

Figure 4.8 Screen shot of the Labview program capturing total power information from the radiometer

was able to detect changes in the noise temperature. It was also able to show that once calibrated, both showed identical results.

4.5 Testing with an injected noise source

The addition of an unwanted signal can be a determinant to the radiometer and has an adverse affect on how the radiometer operates. In today's world though, it is getting more difficult to control intentional radiators as the RF spectrum becomes crowded with more devices. This problem becomes even a greater problem with radiometers used in orbiting spacecrafts as they are able to "see" large areas[**?**)]. Even though the band we are working in of 1.4 GHz is an internationally protected frequency, there can still be both intentional and unintentional radiators that cause interference. This has been seen in some of the SMOS satellite data.

RFI detection and mitigation is not a new topic in radiometry and there have been other methods in both the detection and mitigation of these signals[**?**)][**?**)][**?**)][**?**)].

This test was designed to test a problem that a software defined radio would be to cope with while a square law detector would not be able to cope with it. This test injected a known

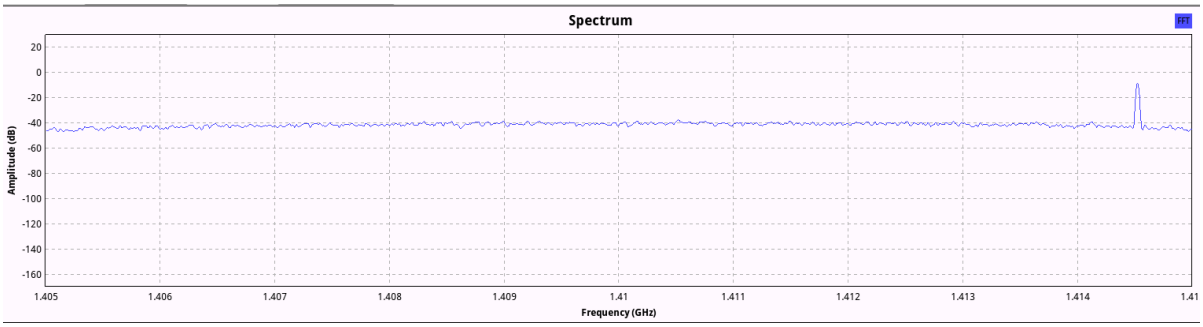

Figure 4.9 Image of the interfering signal appearing on the spectrum display of the SDR Radiometer.

signal at 1.406 GHz to interfere with the normal operation of the radiometer. In this test, the square law detector, since it is a wide band device, would not be able to accurately measure the change in the total power of the signal. However, the SDR is able to create a digital filter to filter out the offending signal. Since this is done in software, the SDR is able to adapt to changes much faster than an analog radiometer.

For this test however, we kept the experiment simple and fixed the interfering signal to a known frequency and amplitude. The filter was then designed with these parameters in mind. This type of radiometer operating in a real world environment would need some additional programming to identify the offending signal and then mitigate. However this test shows that the SDR is capable of responding to an offending signal.

4.5.1 Test setup

To test this theory, a similar setup was used as with previous tests. The ISU RF front end was once again used to amplify the signal. One difference however is that a power combiner, which is the same as a power divider, was used to hook up to a signal generator. The signal generator then provided the offending signal. By using the signal generator we can control how much power and what frequency the offending signal is at. In this case we are performing this in a controlled setup and would know ahead of time where the offending signal is. In the future the SDR defined could identify the offending signal and then filter it out.

Like the other tests, a power divider is used to split the signal after the ISU RF front end

which was then feed to a square-law detector and then the N200 SDR. The square-law detector was then hooked up to a National Instruments USB-6009 DAQ. The DAQ was then feed into a Labview program to be processed and recorded. The N200 continued to use the GNURadio program that was created to record the total power coming out. However, it has been modified with a band reject filter to filter the offending signal.

4.5.2 Test results

Results of this test showed two key things with the SDR. The first being that the SDR is capable of filtering out an offending signal. This can be seen in figure 4.10.

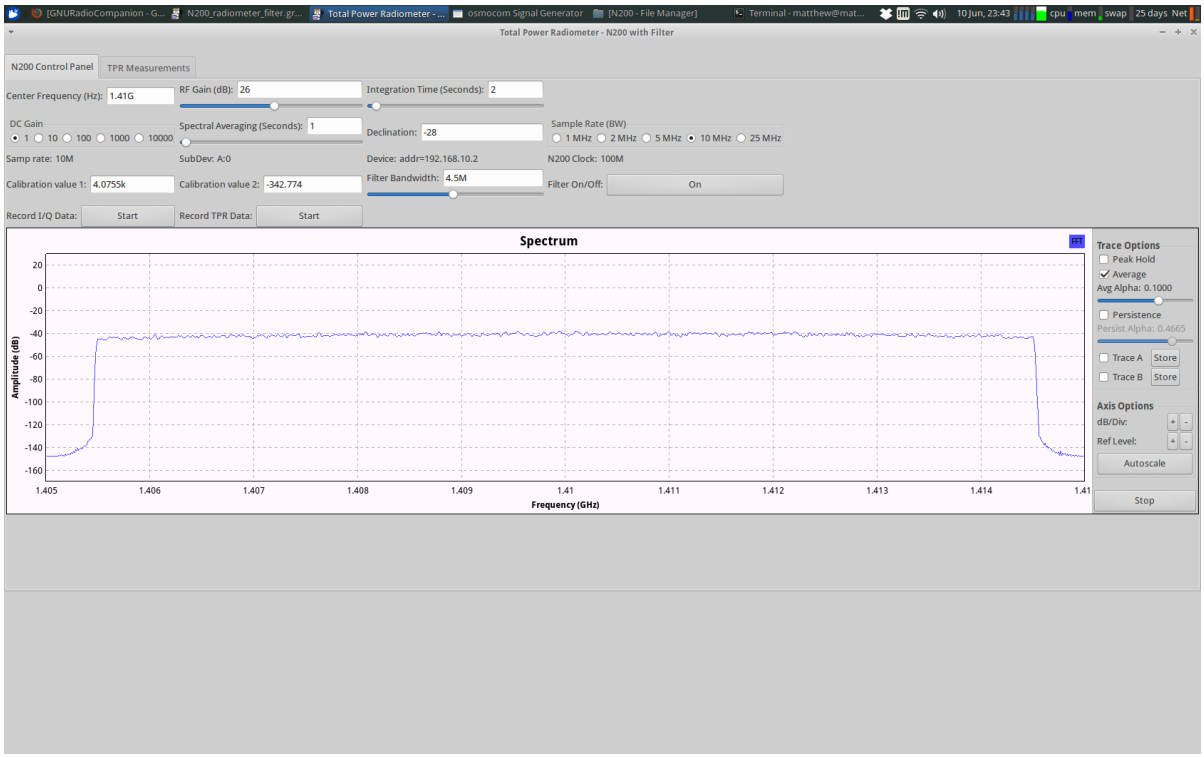

Figure 4.10 Image of the offending signal being filtered out by the SDR. It can be seen that the signal is no longer visible.

The second result is with the SDR still being able to take total power measurements.

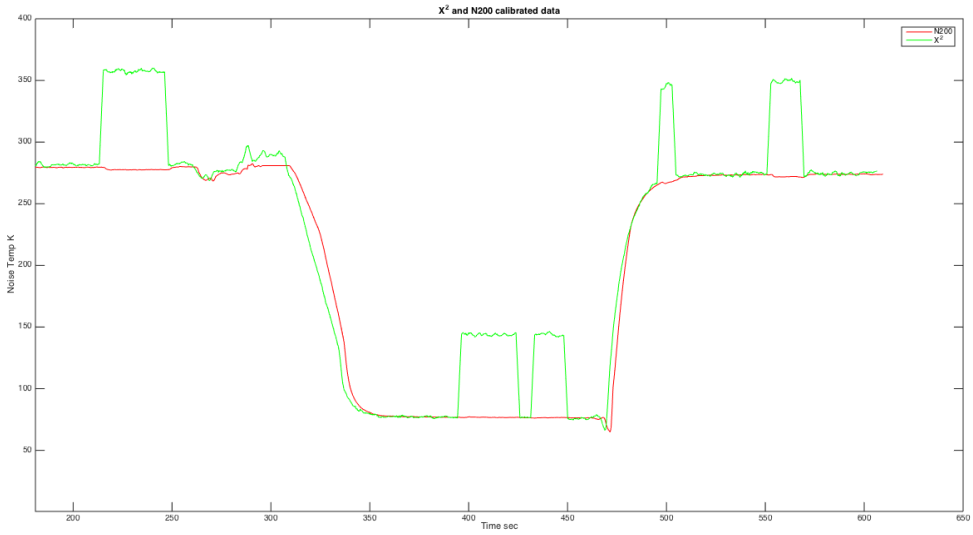

Figure 4.11 Image showing both the SDR and square-law detector recording TPR, but with the SDR using a filter to remove the offending signal.

4.6 Further testing

The software defined radio radiometer can be used in other configurations as well. Further testing is needed to compare some of these other modes to a traditional radiometer to verify their operation, but in theory these additional configurations should operate the same in a software defined radio as they do with a traditional radiometer.

CHAPTER 5. RESULTS AND ANALYSIS

Radiometers measure power and it is this information that we can use to determine information about a certain target. This power however is often expressed in terms of an equivalent temperature and if we are looking at an object, this is the brightness temperature of that object. The primary function of the radiometer is to measure the power that is seen at the radiometer's antenna. Ideally this is the brightness temperature of an object of interest. In our case, this is usually looking at the ground or soil sample. Thus the goal of the radiometer is to measure this antenna temperature with sufficient resolution and accuracy that a correlation can be made between the antenna temperature and the object's temperature that we are studying.

5.1 Performance of a radiometer

The performance of a radiometer can be measured by looking at both the sensitivity and the accuracy of the radiometer. In addition, we need to be concerned with the stability of the radiometer as this affects the accuracy of the radiometer.

5.1.1 Sensitivity

Sensitivity of the radiometer relates to the amount of power that the radio selects from the antenna. This selection is then dependent on the bandwidth that the radiometer is able to listen to. The radiometer however detects not only the signal of interest but also receives a noise signal as well. This noise is added to the signal and can not be separated from the signal. Because this noise is added to the signal, we must be able to determine a change in the signal while the noise signal is also present.

Power from the radiometer can be expressed in equation 2.2 from chapter 2, where we take into consideration bandwidth, gain and the input from the antenna plus the noise added from

the antenna.

Sensitivity relates to being able to distinguish one signal from the other. In other words, we must be able to distinguish, or detect, our wanted signal from noise that is present in the signal. To demonstrate this, consider a system that has a system noise temperature of 700K and an antenna temperature of 300K. This gives us a total noise temperature of 1000K. Therefore, if we wish to detect a change of 1K, we need to be able to detect between 1000K and 1001K.

Since our signal is really random fluctuations about a mean power input to our system, we can reduce the fluctuations by averaging or integrating the signal. This results in equation 5.1 which is derived in chapter 2.

$$\Delta T = \frac{T_A + T_N}{\sqrt{\beta * \tau}} \tag{5.1}$$

This equation gives us the radiometer sensitivity based on the input, which is both the noise and input signal with consideration to the bandwidth and integration time, or averaging, that is done to the signal.

5.1.2 Accuracy and Stability

Stability and accuracy are additional problems that need to be considered when looking at the radiometer system. To begin we can once again look at the power received equation that we discussed in Chapter 2 and is equation 2.2.

As we look at this equation, we can see that if k, B, G, and T_N are constant, then stability can be assured. k is a known constant and we can also assume that our bandwidth, B, will also remain constant, or at least while we are taking our measurements. Gain and the noise temperature however can vary.

Gain is usually our largest factor that can change on a radiometer and even with a software defined radio this is still a large source of variation. This is due to the analog nature of the amplifiers that affect a large portion of the gain in our system. Various things can affect our gain, but the two largest factors is the physical temperature of the amplifier and the voltage that feeds the amplifier. Voltage can be controlled to a degree. High accuracy voltage regulators can help control fluctuations in voltages that can in turn affect the gain. A factor however

that is harder to control is temperature. It is because of this that the current ISU radiometer has gone to great strides to control the temperature of the amplifiers to maintain a constant temperature.

5.2 Required Performance Requirements

To help us quantify the required performance of the radiometer, we referred to information provided to us by Dr. Brian Hornbuckle but also derived from existing radiometers. As stated earlier, although a radiometer measures power, we often convert this to an equivalent brightness temperature. Specifically, we are looking at the brightness temperature of the antenna added to the brightness temperature of the object of interest. We also have to be able to detect a minimum amount of power. Since changes in noise can be small, the better the sensitivity of the radiometer, the better we can detect these small changes.

The requirements given are outlined in the table below.

Table 5.1 Required Radiometer performance

Parameter	**Value**	**Units**
Minimum bandwidth	20	MHz
Operational frequency	1400 - 1425	MHz
$NE\Delta T$	1	Kelvin

5.3 Square-law Detector Performance

The Square-law detector was added to our system in order to give us another reference point and to help verify the power output that the software defined radio. The performance of our square-law detector is based on two items. The first is the actual square-law detector itself. The sensitivity of this device accounts for most of the performance factor of the system. In our system the output of this square-law detector is then feed directly into an analog to digital converter. Therefore the performance of this A/D converter needs to be accounted for as well [**?**)].

For our square-law detector, it has a noise output of $25nV/\sqrt{Hz}$ at 100 kHz and will detect

a signal as low as −60 dBm. This works will with our needs since the RF front end brings the noise floor to approximately −30 dBm.

5.4 Software Defined Radio Performance

Performance of the software defined radio is governed by the system that takes in the RF signal and then digitizes the signal. Once the signal is in digital form, we no longer are concerned about loss of performance due to additional noise that may get added to the system. For this reason, we attempt to digitize the signal as soon as possible.

For the N200 this is done by the DBSRX2 daughter-board that plugs into the N200 base system. While this board does play a part in the overall system performance, because this sits later in the chain however the impact to the performance is low as shown in equation 2.16. However, this module does need to be in the frequency range of the radiometer, or in our case 1.4 GHz. This board meets that criteria. Ideally, we still want the noise figure on this board as low as possible, even though the impact is low. The DBSRX2 does meet this having a noise figure of approximately 5 dBm.

5.5 Benefits to Software Defined Radio Radiometer

A study was conducted on what benefits a software defined radio radiometer would have over a more traditional radiometer. This was focused on looking at three main areas; cost, weight and size, and the value a SDR radiometer can add over traditional radiometers.

5.5.1 Cost Benefits

Software defined radios have become more commonplace in recent years and this has generated a number of COTS solutions. A COTS solution is often a lower cost solution due to the mass manufacturing that takes place. This has driven the cost of many SDRs to under one thousand dollars while still having excellent performance characteristics. The N200 SDR purchased for this research cost fifteen hundred dollars and the daughter-board cost one hundred and fifty dollars approximately. Many other software defined radios however have come out on

the market since then. Ettus for example has some that are below one thousand dollars and the author has also obtained the HackRF One SDR that now sells for three hundred dollars. The main difference with these software defined radios is with both the resolution, or how many bits the ADC is, and the bandwidth they are able to handle.

Table 5.2 Cost Analysis

Device	**Quantity**	**Cost**
SDR Solution		
N200 SDR	1	$1515
LNA at $60 ea.	3	$180
DBSRX2 Daughter-board	1	$152
GNURadio	1	$0
Total		$1847
ISU Radiometer		
LNA, FPGA, ADC, Microcontroller and power supplies	1	$10,000[1]
Commercial Off the Shelf Unit		
Spectracyber 1420 MHz Hydrogen Line Spectrometer	1	$2,650

As see in table 5.2, even the higher cost Ettus research equipment is a lower cost option than the custom built ISU radiometer purchased from University of Michigan and even a comparable off the shelf radiometer. It should be noted that the radiometer from the University of Michigan is also a dual polarization radiometer so there are two RF front ends and two ADCs that feed into a FPGA board. It would be quite easy to add dual polarization to the Ettus N200 SDR as it does support two daughter-boards. This would increase the cost to $2,179 for the additional LNAs and daughter-board.

The largest cost benefit is that key components that you find in a radiometer, the filters and square-law detector can now be all done in software instead of needed additional equipment. The system is also much more frequency agile, which means it can work on a broader range of frequencies than most traditional radiometers with very little change in hardware and in some cases may require no change in hardware. Some of this does depend on the SDR hardware however. The Ettus N200 for example uses daughter-boards to provide the RF interface. While these boards provide a high quality in the RF signal, it does come at a cost and are

[1] Purchase price in 2005

usually designed for certain bands of frequencies. Other low cost SDRs however are also very wide range in the frequencies they will work in. The HackRF for example works from 10 MHz to 6 GHz, but does so at the cost of lower resolution, less gain in its front end and a lower bandwidth that it can handle.

5.5.2 Weight and component size benefits

A typical radiometer has many components that are involved in the design of the radiometer. This includes filters, LNAs and the power detection or square-law detector used. These components add both weight, size and costs to the radiometer. A software defined radio however digitizes the signal and we are able to replace the filters and square-law detector with their software equivalent. While a software defined radio does add both the ADC and usually a FPGA to do the processing on the signal, advances in semiconductor technology has continued to shrink these components. These components are also lighter than the filters often used in radiometers.

Table 5.3 Weight Analysis

Device	**Mass**
SDR Solution	
N200 SDR	1.2 kg
LNA at .03 kg ea	.09 kg
DBSRX2 Daughter-board	.1 kg
Total	1.39 kg
ISU Radiometer	
LNA, FPGA, ADC, Microcontroller and power supplies	22.7 kg
Commercial Off the Shelf Unit	
Spectracyber 1420 MHz Hydrogen Line Spectrometer	6 kg[2]

Size is another benefit as since semiconductor technology has continued to shrink components. Again, since items like the filters and square-law detector are removed and done in software this helps to reduce the overall size.

[2]Estimated, no data available

5.5.3 Value added benefits

A software defined radio radiometer adds additional value for two reasons. One, it is able to work with both frequency and magnitude where most radiometers do not. This allows for additional analysis on the signal and can help identify issues such as an interfering signal that was demonstrated in this thesis.

Second, we are able to have an agile system that is able to adapt to changing conditions with very little or no change to hardware. Different types of radiometers can be implemented such as a Dicke radiometer, dual polarization radiometer or a radiometer that can perform Stokes parameters. In addition, since we have both frequency and power information we can create a system that is able to adapt to changing conditions such as dealing with an interfering signal.

5.6 Disadvantages of a SDR Radiometer

Although we have outlined a number of advantages of using a COTS SDR Radiometer and how a SDR can add additional value to the radiometer system, there are some disadvantages to a SDR Radiometer.

5.6.1 Power Consumption

One of the largest drawbacks to a SDR radiometer can be in the power consumption of the SDR. With the move to perform functions such as power detection and filtering we now require additional computational power to perform these tasks. With those computational cycles additional power is now required. The use of FPGAs and SoC however can help to minimize these power concerns as they are more efficient than using a full scale x86 based processor and on board computer system.

Power and CPU requirements also increase as we add additional functionality such as filtering an offending signal. While these additions may not require additional hardware, it can require additional processor or computational requirements. This will cause additional strain on the processor and also in the memory requirements for the SDR as well.

5.6.2 Bandwidth constraints

While SDR technology has advanced, bandwidth is still a constraint that affects SDRs and in turn a SDR Radiometer. Bandwidth plays a critical role in the radiometers sensitivity as explained in this thesis, therefore the fact that many SDRs are limited in bandwidth does create a disadvantage. In many cases this bottle neck takes place in both the transport and processing of large bandwidth systems. This also relates to the power consumption disadvantage since larger bandwidth also means requiring additional computational cycles as well.

In contrast, a square-law detector usually has a very large bandwidth, as much as one gigahertz, and is why we usually need to filter to the frequency band of interest.

CHAPTER 6. CONCLUSION AND FUTURE WORK

6.1 Conclusion

In this thesis we have shown that an off the shelf SDR can be used to perform as a radiometer. Using a SDR has several advantages such as a more flexible system and can result in a less expensive system. Since a SDR offers high flexibility, changes to the system can be done very quickly and helps in future proofing the system. Since off the shelf components were used, it also allowed for a lower cost system that is able to perform variations of power detection in a radiometer without adding more costly RF components. Finally, this type of radiometer is very flexible and allows for it to adapt to possible changing conditions. This again happens with no change to the RF hardware and instead happens within the software of the system.

6.2 Future work

The main purpose of this research is to demonstrate and prove that an off the shelf software defined radio is capable of operating like a radiometer and can do so within the same or better specifications that are seen in most radiometers today. To do that, we used a very basic radiometer setup and configured our SDR to behave as a single input radiometer. It was also assumed that the input from the RF front end of the radiometer was stable, which in our case was accomplished by stabilizing the temperature of the active components in the RF Front end.

However, this temperature stabilization requires extra weight and bulk to be added to the radiometer. In the application that the ISU radiometer was designed for, this was not a major drawback to the system. However other applications may require a system that does not have this type of temperature stabilization. For those type of systems, other radiometers use different

methods to account for and adjust for fluctuations in the RF front end.

6.2.1 Improving on the FPGA firmware

For this thesis we focused on the software that would run on a PC or comparable computer system running a full OS like Linux. While this aids to speed up development and works just fine for testing the theory on a off the shelf SDR acting as a radiometer, it does require additional hardware. For some applications of a radiometer, this is not a huge concern. In the case for the ISU radiometer, the concern is not that large since the radiometer is not designed to be "portable" and requires additional support equipment such as a generator anyway. However, other remote sensing applications, such as space based applications, would require a more efficient method. It is very possible to move the software generated in GNURadio into the firmware of the N200. This will help to offload the work needed by the computer and would allow for the computer or similar system to act as more of a control method and for data storage.

6.2.2 Correlation

One such method is to correlate the information with another input which can be another antenna looking at the same source or can be two polarization from the same antenna[**?**)][**?**)]. This results in two receiver systems looking at the same source and you have two signals, S_1 and S_2. Since we are looking at the same source, both signals will be correlated in time, and when multiplied they will provide an output proportional to the strength of the source signal. The noise introduced by each receiver will then have a lower correlation due to the random nature of the noise. This results in a radiometer with a greater sensitivity due to the reduction of the noise even though two receivers are used [**?**)].

The N200 software defined radio was chosen as it is capable of having two different daughter-cards plugged in. Therefore it is possible to have both sources enter the software defined radio and once digitized we can sum the magnitudes of the two incoming sources. This is quite easy to do and is shown in figure 6.1.

Although figure 6.1 shows a correlating software defined radio radiometer, it has not been

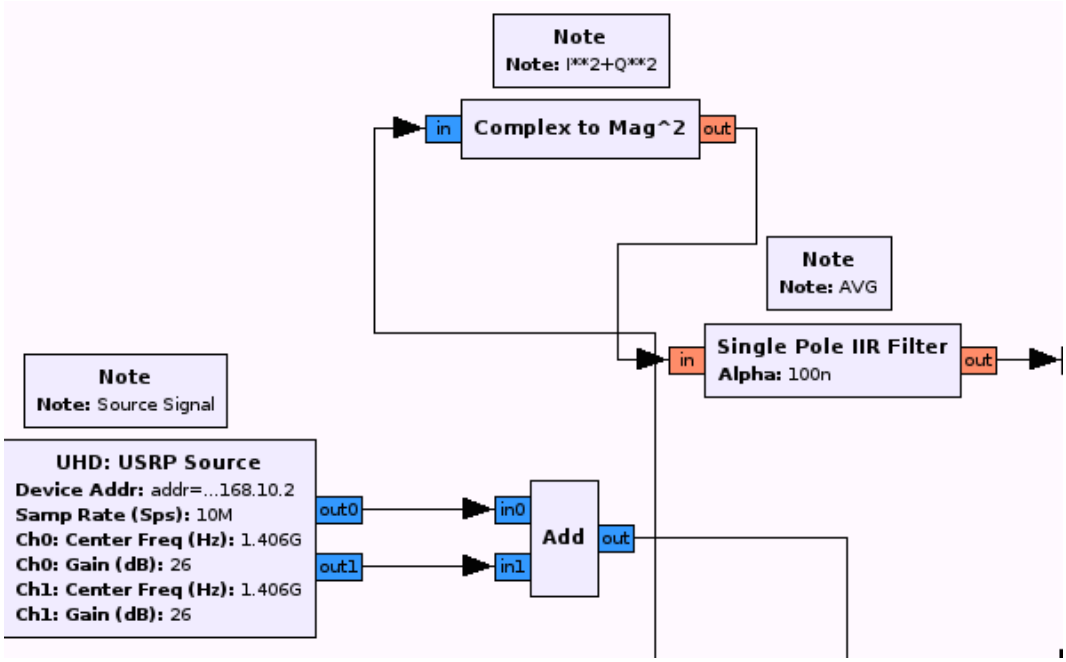

Figure 6.1 The key blocks used for creating a correlating radiometer in software. The key blocks is the USRP source, which allows us to address both daughter-boards and the ADD block which sums the signals.

tested. In theory, this should correlate the signal and improve the sensitivity of the radiometer, however further testing is needed.

6.2.3 Improving stability

One of the largest challenges with a radiometer is improving the stability of the radiometer. Drifts in temperature can greatly affect the gains from the LNAs and also change how much noise all of the components in the radiometer contribute. A software defined radio can help as we are digitizing the signal as soon as possible. This helps in eliminating the analog components for power detection and even for filtering, but does not eliminate all of the physical hardware, mainly the LNAs. In this thesis, we did not focus on this issue since the RF front end of the ISU radiometer is temperature controlled. All components are mounted to an aluminum block which is attached to a thermal electric cooler. This system then maintains the temperature within 1 degree C.

However a more compact, lower cost and easier setup would be to just have the LNAs

attach directly to the SDR without any temperature compensation. While this can be done, we have now lost stability in the LNAs and we need to compensate for that. One method that is discussed by William Goggins is to use a feedback loop to continuously adjust a variable attenuator [**?**)]. In Goggins paper, he discusses using a servo that mechanically controls the attenuator. However since we are in the digital domain, we can control this all in software, and doing a feedback loop is quite easy for a computer to do.

Another method uses multiple temperature points that can be referenced at any time. By using two known temperature references, we can quickly calibrate the radiometer[**?**)].

APPENDIX A. Source code

The following is the source code to several programs that make the radiometer possible. The first is the Python code used with the Ettus N200 software defined radio. This python code should work on any platform that has the GNURadio libraries installed and also the USRP drivers for communicating with the N200. This code can also be easily modified to communicate with any SDR that GNURadio can communicate with. This code was generated using GNURadio Companion.

The second code is the Matlab script that can be used to parse the output from GNURadio. This code will plot the total power output as well as perform some other functions such as a NEδT calculation and also allows for calibration points to be entered as well. This code can be used as a foundation for other programs.

The third code supplied is Python code that can be used to read the data generated and plot it. In many ways it mimics the functions of the Matlab script buy uses Python, NumPy and SciPy to perform the mathematical functions. This may be a better option for those that wish to look at the data but do not have access to Matlab since Python is free to download.

Finally, this code has been included in this thesis as a point of reference. It may be out of date and some other pieces of code was also used for the experimentation used in this thesis. Copies of this thesis source LaTeXcode, and any other code used can be found on the author's GitHub repository, https://github.com/matgyver/Radiometer-SDR-Thesis.

Python code for total power radiometer

Total Power Radiometer

```
#!/usr/bin/env python
##################################################
# Gnuradio Python Flow Graph
# Title: Total Power Radiometer - N200
# Author: Matthew E Nelson
# Description: Total power radiometer connecting to a N200 SDR
# Generated: Tue Feb 24 17:35:30 2015
##################################################

# Call XInitThreads as the _very_ first thing.
# After some Qt import, it's too late
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 datetime import datetime
from gnuradio import blocks
from gnuradio import eng_notation
from gnuradio import filter
from gnuradio import gr
from gnuradio import uhd
from gnuradio import wxgui
from gnuradio.eng_option import eng_option
```

```
from gnuradio.fft import logpwrfft
from gnuradio.fft import window
from gnuradio.filter import firdes
from gnuradio.wxgui import fftsink2
from gnuradio.wxgui import forms
from gnuradio.wxgui import numbersink2
from gnuradio.wxgui import scopesink2
from grc_gnuradio import blks2 as grc_blks2
from grc_gnuradio import wxgui as grc_wxgui
from optparse import OptionParser
import time
import wx

class N200_TPR(grc_wxgui.top_block_gui):

    def __init__(self, subdev="A:0", devid="addr=192.168.10.2",
       frequency=1.4125e9, fftsize=8192):
        grc_wxgui.top_block_gui.__init__(self, title="Total Power
            Radiometer - N200")

        ##################################################
        # Parameters
        ##################################################
        self.subdev = subdev
        self.devid = devid
        self.frequency = frequency
        self.fftsize = fftsize
```

```
##################################################
# Variables
##################################################
self.GUI_samp_rate = GUI_samp_rate = 10e6
self.samp_rate = samp_rate = int(GUI_samp_rate)
self.prefix = prefix = "tpr_"
self.text_samp_rate = text_samp_rate = GUI_samp_rate
self.text_deviceID = text_deviceID = subdev
self.text_Device_addr = text_Device_addr = devid
self.spec_data_fifo = spec_data_fifo = "spectrum_" +
   datetime.now().strftime("%Y.%m.%d.%H.%M.%S") + ".dat"
self.spavg = spavg = 1
self.scope_rate = scope_rate = 2
self.recfile_tpr = recfile_tpr = prefix + datetime.now().
   strftime("%Y.%m.%d.%H.%M.%S") + ".dat"
self.recfile_kelvin = recfile_kelvin = prefix+"kelvin" +
   datetime.now().strftime("%Y.%m.%d.%H.%M.%S") + ".dat"
self.rec_button_tpr = rec_button_tpr = 1
self.rec_button_iq = rec_button_iq = 1
self.noise_amplitude = noise_amplitude = .5
self.integ = integ = 2
self.gain = gain = 26
self.freq = freq = frequency
self.file_rate = file_rate = 2.0
self.fftrate = fftrate = int(samp_rate/fftsize)
self.det_rate = det_rate = int(20.0)
self.dc_gain = dc_gain = 1
self.calib_2 = calib_2 = -342.774
```

```
self.calib_1 = calib_1 = 4.0755e3
self.add_noise = add_noise = 0

##################################################
# Blocks
##################################################
self.Main = self.Main = wx.Notebook(self.GetWin(), style=
   wx.NB_TOP)
self.Main.AddPage(grc_wxgui.Panel(self.Main), "N200
   Control Panel")
self.Main.AddPage(grc_wxgui.Panel(self.Main), "TPR
   Measurements")
self.Add(self.Main)
_spavg_sizer = wx.BoxSizer(wx.VERTICAL)
self._spavg_text_box = forms.text_box(
    parent=self.Main.GetPage(0).GetWin(),
    sizer=_spavg_sizer,
    value=self.spavg,
    callback=self.set_spavg,
    label="Spectral Averaging (Seconds)",
    converter=forms.int_converter(),
    proportion=0,
)
self._spavg_slider = forms.slider(
    parent=self.Main.GetPage(0).GetWin(),
    sizer=_spavg_sizer,
    value=self.spavg,
    callback=self.set_spavg,
```

```
        minimum=1,
        maximum=20,
        num_steps=20,
        style=wx.SL_HORIZONTAL,
        cast=int,
        proportion=1,
    )
    self.Main.GetPage(0).GridAdd(_spavg_sizer, 1, 1, 1, 1)
    self._rec_button_tpr_chooser = forms.button(
        parent=self.Main.GetPage(0).GetWin(),
        value=self.rec_button_tpr,
        callback=self.set_rec_button_tpr,
        label="Record TPR Data",
        choices=[0,1],
        labels=['Stop','Start'],
    )
    self.Main.GetPage(0).GridAdd(self._rec_button_tpr_chooser
       , 4, 1, 1, 1)
    self._rec_button_iq_chooser = forms.button(
        parent=self.Main.GetPage(0).GetWin(),
        value=self.rec_button_iq,
        callback=self.set_rec_button_iq,
        label="Record I/Q Data",
        choices=[0,1],
        labels=['Stop','Start'],
    )
    self.Main.GetPage(0).GridAdd(self._rec_button_iq_chooser,
        4, 0, 1, 1)
```

```
_integ_sizer = wx.BoxSizer(wx.VERTICAL)
self._integ_text_box = forms.text_box(
    parent=self.Main.GetPage(0).GetWin(),
    sizer=_integ_sizer,
    value=self.integ,
    callback=self.set_integ,
    label="Integration Time (Seconds)",
    converter=forms.float_converter(),
    proportion=0,
)
self._integ_slider = forms.slider(
    parent=self.Main.GetPage(0).GetWin(),
    sizer=_integ_sizer,
    value=self.integ,
    callback=self.set_integ,
    minimum=1,
    maximum=60,
    num_steps=100,
    style=wx.SL_HORIZONTAL,
    cast=float,
    proportion=1,
)
self.Main.GetPage(0).GridAdd(_integ_sizer, 0, 2, 1, 1)
_gain_sizer = wx.BoxSizer(wx.VERTICAL)
self._gain_text_box = forms.text_box(
    parent=self.Main.GetPage(0).GetWin(),
    sizer=_gain_sizer,
    value=self.gain,
```

```
        callback=self.set_gain,
        label="RF Gain (dB)",
        converter=forms.float_converter(),
        proportion=0,
    )
    self._gain_slider = forms.slider(
        parent=self.Main.GetPage(0).GetWin(),
        sizer=_gain_sizer,
        value=self.gain,
        callback=self.set_gain,
        minimum=0,
        maximum=50,
        num_steps=100,
        style=wx.SL_HORIZONTAL,
        cast=float,
        proportion=1,
    )
    self.Main.GetPage(0).GridAdd(_gain_sizer, 0, 1, 1, 1)
    self._freq_text_box = forms.text_box(
        parent=self.Main.GetPage(0).GetWin(),
        value=self.freq,
        callback=self.set_freq,
        label="Center Frequency (Hz)",
        converter=forms.float_converter(),
    )
    self.Main.GetPage(0).GridAdd(self._freq_text_box, 0, 0,
       1, 1)
    self._dc_gain_chooser = forms.radio_buttons(
```

```
        parent=self.Main.GetPage(0).GetWin(),
        value=self.dc_gain,
        callback=self.set_dc_gain,
        label="DC Gain",
        choices=[1, 10, 100, 1000, 10000],
        labels=[],
        style=wx.RA_HORIZONTAL,
    )
    self.Main.GetPage(0).GridAdd(self._dc_gain_chooser, 1, 0,
        1, 1)
    self._calib_2_text_box = forms.text_box(
        parent=self.Main.GetPage(0).GetWin(),
        value=self.calib_2,
        callback=self.set_calib_2,
        label="Calibration value 2",
        converter=forms.float_converter(),
    )
    self.Main.GetPage(0).GridAdd(self._calib_2_text_box, 3,
       1, 1, 1)
    self._calib_1_text_box = forms.text_box(
        parent=self.Main.GetPage(0).GetWin(),
        value=self.calib_1,
        callback=self.set_calib_1,
        label="Calibration value 1",
        converter=forms.float_converter(),
    )
    self.Main.GetPage(0).GridAdd(self._calib_1_text_box, 3,
       0, 1, 1)
```

```
self._GUI_samp_rate_chooser = forms.radio_buttons(
    parent=self.Main.GetPage(0).GetWin(),
    value=self.GUI_samp_rate,
    callback=self.set_GUI_samp_rate,
    label="Sample Rate (BW)",
    choices=[1e6,2e6,5e6,10e6,25e6],
    labels=['1 MHz','2 MHz','5 MHz','10 MHz','25 MHz'],
    style=wx.RA_HORIZONTAL,
)
self.Main.GetPage(0).GridAdd(self._GUI_samp_rate_chooser,
    1, 3, 1, 1)
self.wxgui_scopesink2_2 = scopesink2.scope_sink_f(
    self.Main.GetPage(1).GetWin(),
    title="Total Power",
    sample_rate=2,
    v_scale=.1,
    v_offset=0,
    t_scale=100,
    ac_couple=False,
    xy_mode=False,
    num_inputs=1,
    trig_mode=wxgui.TRIG_MODE_STRIPCHART,
    y_axis_label="power level",
)
self.Main.GetPage(1).Add(self.wxgui_scopesink2_2.win)
self.wxgui_numbersink2_2 = numbersink2.number_sink_f(
    self.GetWin(),
    unit="Units",
```

```
        minval=0,
        maxval=1,
        factor=1.0,
        decimal_places=10,
        ref_level=0,
        sample_rate=GUI_samp_rate,
        number_rate=15,
        average=False,
        avg_alpha=None,
        label="Number Plot",
        peak_hold=False,
        show_gauge=True,
    )
    self.Add(self.wxgui_numbersink2_2.win)
    self.wxgui_numbersink2_0_0 = numbersink2.number_sink_f(
        self.Main.GetPage(1).GetWin(),
        unit="",
        minval=0,
        maxval=.2,
        factor=1,
        decimal_places=6,
        ref_level=0,
        sample_rate=scope_rate,
        number_rate=15,
        average=True,
        avg_alpha=.01,
        label="Raw Power level",
        peak_hold=False,
```

```
        show_gauge=True,
    )
    self.Main.GetPage(1).Add(self.wxgui_numbersink2_0_0.win)
    self.wxgui_numbersink2_0 = numbersink2.number_sink_f(
        self.Main.GetPage(1).GetWin(),
        unit="Kelvin",
        minval=0,
        maxval=400,
        factor=1,
        decimal_places=6,
        ref_level=0,
        sample_rate=scope_rate,
        number_rate=15,
        average=False,
        avg_alpha=None,
        label="Calibrated Temperature",
        peak_hold=False,
        show_gauge=True,
    )
    self.Main.GetPage(1).Add(self.wxgui_numbersink2_0.win)
    self.wxgui_fftsink2_0 = fftsink2.fft_sink_c(
        self.Main.GetPage(0).GetWin(),
        baseband_freq=freq,
        y_per_div=10,
        y_divs=10,
        ref_level=20,
        ref_scale=2.0,
        sample_rate=GUI_samp_rate,
```

```
        fft_size=1024,
        fft_rate=5,
        average=True,
        avg_alpha=0.1,
        title="Spectrum",
        peak_hold=False,
        size=(800,400),
    )
    self.Main.GetPage(0).Add(self.wxgui_fftsink2_0.win)
    self.uhd_usrp_source_0 = uhd.usrp_source(
        ",".join((devid, "")),
        uhd.stream_args(
            cpu_format="fc32",
            channels=range(1),
        ),
    )
    self.uhd_usrp_source_0.set_samp_rate(GUI_samp_rate)
    self.uhd_usrp_source_0.set_center_freq(freq, 0)
    self.uhd_usrp_source_0.set_gain(gain, 0)
    (self.uhd_usrp_source_0).set_processor_affinity([0])
    self._text_samp_rate_static_text = forms.static_text(
        parent=self.Main.GetPage(0).GetWin(),
        value=self.text_samp_rate,
        callback=self.set_text_samp_rate,
        label="Samp rate",
        converter=forms.float_converter(),
    )
```

```
self.Main.GetPage(0).GridAdd(self.
   _text_samp_rate_static_text, 2, 0, 1, 1)
self._text_deviceID_static_text = forms.static_text(
    parent=self.Main.GetPage(0).GetWin(),
    value=self.text_deviceID,
    callback=self.set_text_deviceID,
    label="SubDev",
    converter=forms.str_converter(),
)
self.Main.GetPage(0).GridAdd(self.
   _text_deviceID_static_text, 2, 1, 1, 1)
self._text_Device_addr_static_text = forms.static_text(
    parent=self.Main.GetPage(0).GetWin(),
    value=self.text_Device_addr,
    callback=self.set_text_Device_addr,
    label="Device Address",
    converter=forms.str_converter(),
)
self.Main.GetPage(0).GridAdd(self.
   _text_Device_addr_static_text, 2, 2, 1, 1)
self.single_pole_iir_filter_xx_0 = filter.
   single_pole_iir_filter_ff(1.0/((samp_rate*integ)/2.0),
    1)
(self.single_pole_iir_filter_xx_0).set_processor_affinity
   ([1])
_noise_amplitude_sizer = wx.BoxSizer(wx.VERTICAL)
self._noise_amplitude_text_box = forms.text_box(
    parent=self.Main.GetPage(0).GetWin(),
```

```
        sizer=_noise_amplitude_sizer,
        value=self.noise_amplitude,
        callback=self.set_noise_amplitude,
        label='noise_amplitude',
        converter=forms.float_converter(),
        proportion=0,
    )
    self._noise_amplitude_slider = forms.slider(
        parent=self.Main.GetPage(0).GetWin(),
        sizer=_noise_amplitude_sizer,
        value=self.noise_amplitude,
        callback=self.set_noise_amplitude,
        minimum=.01,
        maximum=1,
        num_steps=100,
        style=wx.SL_HORIZONTAL,
        cast=float,
        proportion=1,
    )
    self.Main.GetPage(0).GridAdd(_noise_amplitude_sizer, 3,
       2, 1, 1)
    self.logpwrfft_x_0 = logpwrfft.logpwrfft_c(
        sample_rate=samp_rate,
        fft_size=fftsize,
        ref_scale=2,
        frame_rate=fftrate,
        avg_alpha=1.0/float(spavg*fftrate),
        average=True,
```

```
)
self.blocks_peak_detector_xb_0 = blocks.peak_detector_fb
   (0.25, 0.40, 10, 0.001)
self.blocks_multiply_const_vxx_1 = blocks.
   multiply_const_vff((calib_1, ))
self.blocks_multiply_const_vxx_0 = blocks.
   multiply_const_vff((dc_gain, ))
self.blocks_keep_one_in_n_4 = blocks.keep_one_in_n(gr.
   sizeof_float*1, samp_rate/det_rate)
self.blocks_keep_one_in_n_3 = blocks.keep_one_in_n(gr.
   sizeof_float*fftsize, fftrate)
self.blocks_keep_one_in_n_1 = blocks.keep_one_in_n(gr.
   sizeof_float*1, int(det_rate/file_rate))
self.blocks_file_sink_5 = blocks.file_sink(gr.
   sizeof_float*fftsize, spec_data_fifo, False)
self.blocks_file_sink_5.set_unbuffered(True)
self.blocks_file_sink_4 = blocks.file_sink(gr.
   sizeof_float*1, recfile_tpr, False)
self.blocks_file_sink_4.set_unbuffered(True)
self.blocks_file_sink_1 = blocks.file_sink(gr.
   sizeof_gr_complex*1, prefix+"iq_raw" + datetime.now().
   strftime("%Y.%m.%d.%H.%M.%S") + ".dat", False)
self.blocks_file_sink_1.set_unbuffered(False)
self.blocks_file_sink_0 = blocks.file_sink(gr.
   sizeof_float*1, recfile_kelvin, False)
self.blocks_file_sink_0.set_unbuffered(True)
self.blocks_complex_to_real_0 = blocks.complex_to_real(1)
```

```
self.blocks_complex_to_mag_squared_1 = blocks.
    complex_to_mag_squared(1)
self.blocks_char_to_float_0 = blocks.char_to_float(1, 1)
self.blocks_add_const_vxx_1 = blocks.add_const_vff((
    calib_2, ))
self.blks2_valve_2 = grc_blks2.valve(item_size=gr.
    sizeof_gr_complex*1, open=bool(rec_button_iq))
self.blks2_valve_1 = grc_blks2.valve(item_size=gr.
    sizeof_float*1, open=bool(0))
self.blks2_valve_0 = grc_blks2.valve(item_size=gr.
    sizeof_float*1, open=bool(rec_button_tpr))
self._add_noise_chooser = forms.button(
     parent=self.Main.GetPage(0).GetWin(),
     value=self.add_noise,
     callback=self.set_add_noise,
     label="Noise Source",
     choices=[0,1],
     labels=['Off','On'],
)
self.Main.GetPage(0).GridAdd(self._add_noise_chooser, 3,
    3, 1, 1)

##################################################
# Connections
##################################################
self.connect((self.blks2_valve_0, 0), (self.
    blocks_file_sink_4, 0))
```

```
self.connect((self.blks2_valve_1, 0), (self.
    blocks_file_sink_0, 0))
self.connect((self.blks2_valve_2, 0), (self.
    blocks_file_sink_1, 0))
self.connect((self.blocks_add_const_vxx_1, 0), (self.
    blks2_valve_1, 0))
self.connect((self.blocks_add_const_vxx_1, 0), (self.
    wxgui_numbersink2_0, 0))
self.connect((self.blocks_char_to_float_0, 0), (self.
    wxgui_numbersink2_2, 0))
self.connect((self.blocks_complex_to_mag_squared_1, 0), (
    self.single_pole_iir_filter_xx_0, 0))
self.connect((self.blocks_complex_to_real_0, 0), (self.
    blocks_peak_detector_xb_0, 0))
self.connect((self.blocks_keep_one_in_n_1, 0), (self.
    blks2_valve_0, 0))
self.connect((self.blocks_keep_one_in_n_1, 0), (self.
    blocks_multiply_const_vxx_1, 0))
self.connect((self.blocks_keep_one_in_n_1, 0), (self.
    wxgui_scopesink2_2, 0))
self.connect((self.blocks_keep_one_in_n_3, 0), (self.
    blocks_file_sink_5, 0))
self.connect((self.blocks_keep_one_in_n_4, 0), (self.
    blocks_multiply_const_vxx_0, 0))
self.connect((self.blocks_multiply_const_vxx_0, 0), (self
    .blocks_keep_one_in_n_1, 0))
self.connect((self.blocks_multiply_const_vxx_0, 0), (self
    .wxgui_numbersink2_0_0, 0))
```

```
        self.connect((self.blocks_multiply_const_vxx_1, 0), (self
            .blocks_add_const_vxx_1, 0))
        self.connect((self.blocks_peak_detector_xb_0, 0), (self.
            blocks_char_to_float_0, 0))
        self.connect((self.logpwrfft_x_0, 0), (self.
            blocks_keep_one_in_n_3, 0))
        self.connect((self.single_pole_iir_filter_xx_0, 0), (self
            .blocks_keep_one_in_n_4, 0))
        self.connect((self.uhd_usrp_source_0, 0), (self.
            blks2_valve_2, 0))
        self.connect((self.uhd_usrp_source_0, 0), (self.
            blocks_complex_to_mag_squared_1, 0))
        self.connect((self.uhd_usrp_source_0, 0), (self.
            blocks_complex_to_real_0, 0))
        self.connect((self.uhd_usrp_source_0, 0), (self.
            logpwrfft_x_0, 0))
        self.connect((self.uhd_usrp_source_0, 0), (self.
            wxgui_fftsink2_0, 0))


    def get_subdev(self):
        return self.subdev

    def set_subdev(self, subdev):
        self.subdev = subdev
        self.set_text_deviceID(self.subdev)

    def get_devid(self):
```

```
        return self.devid

    def set_devid(self, devid):
        self.devid = devid
        self.set_text_Device_addr(self.devid)

    def get_frequency(self):
        return self.frequency

    def set_frequency(self, frequency):
        self.frequency = frequency
        self.set_freq(self.frequency)

    def get_fftsize(self):
        return self.fftsize

    def set_fftsize(self, fftsize):
        self.fftsize = fftsize
        self.set_fftrate(int(self.samp_rate/self.fftsize))

    def get_GUI_samp_rate(self):
        return self.GUI_samp_rate

    def set_GUI_samp_rate(self, GUI_samp_rate):
        self.GUI_samp_rate = GUI_samp_rate
        self.set_samp_rate(int(self.GUI_samp_rate))
        self.set_text_samp_rate(self.GUI_samp_rate)
        self._GUI_samp_rate_chooser.set_value(self.GUI_samp_rate)
```

```
        self.uhd_usrp_source_0.set_samp_rate(self.GUI_samp_rate)
        self.wxgui_fftsink2_0.set_sample_rate(self.GUI_samp_rate)

    def get_samp_rate(self):
        return self.samp_rate

    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate
        self.set_fftrate(int(self.samp_rate/self.fftsize))
        self.logpwrfft_x_0.set_sample_rate(self.samp_rate)
        self.single_pole_iir_filter_xx_0.set_taps(1.0/((self.
            samp_rate*self.integ)/2.0))
        self.blocks_keep_one_in_n_4.set_n(self.samp_rate/self.
            det_rate)

    def get_prefix(self):
        return self.prefix

    def set_prefix(self, prefix):
        self.prefix = prefix
        self.set_recfile_kelvin(self.prefix+"kelvin" + datetime.
            now().strftime("%Y.%m.%d.%H.%M.%S") + ".dat")
        self.set_recfile_tpr(self.prefix + datetime.now().
            strftime("%Y.%m.%d.%H.%M.%S") + ".dat")
        self.blocks_file_sink_1.open(self.prefix+"iq_raw" +
            datetime.now().strftime("%Y.%m.%d.%H.%M.%S") + ".dat")

    def get_text_samp_rate(self):
```

```
        return self.text_samp_rate

    def set_text_samp_rate(self, text_samp_rate):
        self.text_samp_rate = text_samp_rate
        self._text_samp_rate_static_text.set_value(self.
            text_samp_rate)

    def get_text_deviceID(self):
        return self.text_deviceID

    def set_text_deviceID(self, text_deviceID):
        self.text_deviceID = text_deviceID
        self._text_deviceID_static_text.set_value(self.
            text_deviceID)

    def get_text_Device_addr(self):
        return self.text_Device_addr

    def set_text_Device_addr(self, text_Device_addr):
        self.text_Device_addr = text_Device_addr
        self._text_Device_addr_static_text.set_value(self.
            text_Device_addr)

    def get_spec_data_fifo(self):
        return self.spec_data_fifo

    def set_spec_data_fifo(self, spec_data_fifo):
        self.spec_data_fifo = spec_data_fifo
```

```
        self.blocks_file_sink_5.open(self.spec_data_fifo)

    def get_spavg(self):
        return self.spavg

    def set_spavg(self, spavg):
        self.spavg = spavg
        self.logpwrfft_x_0.set_avg_alpha(1.0/float(self.spavg*
           self.fftrate))
        self._spavg_slider.set_value(self.spavg)
        self._spavg_text_box.set_value(self.spavg)

    def get_scope_rate(self):
        return self.scope_rate

    def set_scope_rate(self, scope_rate):
        self.scope_rate = scope_rate

    def get_recfile_tpr(self):
        return self.recfile_tpr

    def set_recfile_tpr(self, recfile_tpr):
        self.recfile_tpr = recfile_tpr
        self.blocks_file_sink_4.open(self.recfile_tpr)

    def get_recfile_kelvin(self):
        return self.recfile_kelvin
```

```
def set_recfile_kelvin(self, recfile_kelvin):
    self.recfile_kelvin = recfile_kelvin
    self.blocks_file_sink_0.open(self.recfile_kelvin)

def get_rec_button_tpr(self):
    return self.rec_button_tpr

def set_rec_button_tpr(self, rec_button_tpr):
    self.rec_button_tpr = rec_button_tpr
    self._rec_button_tpr_chooser.set_value(self.
       rec_button_tpr)
    self.blks2_valve_0.set_open(bool(self.rec_button_tpr))

def get_rec_button_iq(self):
    return self.rec_button_iq

def set_rec_button_iq(self, rec_button_iq):
    self.rec_button_iq = rec_button_iq
    self.blks2_valve_2.set_open(bool(self.rec_button_iq))
    self._rec_button_iq_chooser.set_value(self.rec_button_iq)

def get_noise_amplitude(self):
    return self.noise_amplitude

def set_noise_amplitude(self, noise_amplitude):
    self.noise_amplitude = noise_amplitude
    self._noise_amplitude_slider.set_value(self.
       noise_amplitude)
```

```
        self._noise_amplitude_text_box.set_value(self.
            noise_amplitude)

    def get_integ(self):
        return self.integ

    def set_integ(self, integ):
        self.integ = integ
        self._integ_slider.set_value(self.integ)
        self._integ_text_box.set_value(self.integ)
        self.single_pole_iir_filter_xx_0.set_taps(1.0/((self.
            samp_rate*self.integ)/2.0))

    def get_gain(self):
        return self.gain

    def set_gain(self, gain):
        self.gain = gain
        self._gain_slider.set_value(self.gain)
        self._gain_text_box.set_value(self.gain)
        self.uhd_usrp_source_0.set_gain(self.gain, 0)

    def get_freq(self):
        return self.freq

    def set_freq(self, freq):
        self.freq = freq
        self._freq_text_box.set_value(self.freq)
```

```
        self.uhd_usrp_source_0.set_center_freq(self.freq, 0)
        self.wxgui_fftsink2_0.set_baseband_freq(self.freq)

    def get_file_rate(self):
        return self.file_rate

    def set_file_rate(self, file_rate):
        self.file_rate = file_rate
        self.blocks_keep_one_in_n_1.set_n(int(self.det_rate/self.
            file_rate))

    def get_fftrate(self):
        return self.fftrate

    def set_fftrate(self, fftrate):
        self.fftrate = fftrate
        self.logpwrfft_x_0.set_avg_alpha(1.0/float(self.spavg*
            self.fftrate))
        self.blocks_keep_one_in_n_3.set_n(self.fftrate)

    def get_det_rate(self):
        return self.det_rate

    def set_det_rate(self, det_rate):
        self.det_rate = det_rate
        self.blocks_keep_one_in_n_4.set_n(self.samp_rate/self.
            det_rate)
```

```
        self.blocks_keep_one_in_n_1.set_n(int(self.det_rate/self.
            file_rate))

    def get_dc_gain(self):
        return self.dc_gain

    def set_dc_gain(self, dc_gain):
        self.dc_gain = dc_gain
        self._dc_gain_chooser.set_value(self.dc_gain)
        self.blocks_multiply_const_vxx_0.set_k((self.dc_gain, ))

    def get_calib_2(self):
        return self.calib_2

    def set_calib_2(self, calib_2):
        self.calib_2 = calib_2
        self._calib_2_text_box.set_value(self.calib_2)
        self.blocks_add_const_vxx_1.set_k((self.calib_2, ))

    def get_calib_1(self):
        return self.calib_1

    def set_calib_1(self, calib_1):
        self.calib_1 = calib_1
        self._calib_1_text_box.set_value(self.calib_1)
        self.blocks_multiply_const_vxx_1.set_k((self.calib_1, ))

    def get_add_noise(self):
```

```
        return self.add_noise

    def set_add_noise(self, add_noise):
        self.add_noise = add_noise
        self._add_noise_chooser.set_value(self.add_noise)

if __name__ == '__main__':
    parser = OptionParser(option_class=eng_option, usage="%prog:
       [options]")
    parser.add_option("", "--subdev", dest="subdev", type="string
       ", default="A:0",
        help="Set USRP Subdevice ID [default=%default]")
    parser.add_option("", "--devid", dest="devid", type="string",
        default="addr=192.168.10.2",
        help="Set USRP Device ID [default=%default]")
    parser.add_option("", "--frequency", dest="frequency", type="
       eng_float", default=eng_notation.num_to_str(1.4125e9),
        help="Set Center Frequency [default=%default]")
    parser.add_option("", "--fftsize", dest="fftsize", type="intx
       ", default=8192,
        help="Set fftsize [default=%default]")
    (options, args) = parser.parse_args()
    tb = N200_TPR(subdev=options.subdev, devid=options.devid,
       frequency=options.frequency, fftsize=options.fftsize)
    tb.Start(True)
    tb.Wait()
```

Matlab code for reading and displaying data from GNURadio

GNURadio Parsing Code

```
%Radiometer Parsing script
%Matthew E. Nelson
%Updated 6/8/2014
%Rev. 2.15
%
    ------------------------------------------------------------------------


%Revision History
%1.7 - Added CSV input file format.  Gave up on reading LVM
%1.8 - Added User input box
%1.9 - Added Calibration points for square law detector
%1.91 - Cleaned up some code
%1.92 - Futher clean up of unused code
%1.93 - Fixed dialog boxes not showing the entire title
%2.0 - Added filter to clean up noisy x^2 data
%2.1 - Added NEdeltaT (NEAT) calculation, minor change to plot
    labels
%2.11 - Added square law voltage to dBm conversion
%2.12 - Fixed bug on when doing NEAT calc, added a time base
    calculation
%for plots
%2.13 - Added calibration line plots
%2.14 - changed graphs to a time base
```

```
%2.15 - Added plotting of both x^2 and N200 on same graph for
    comparasion

%This script uses the read_float_binary.m file to read in a file
    written by
%GNURadio.  This data can then be manipulated by Matlab and
    graphed.  This
%script can also accept calibration coefficients in order to
    calculate the
%calibrated noise temperature.  This requires that two data
    points are
%recorded to known sources such as LN2.

%Data files using the GNURadio flow diagram written by Matthew
    Nelson use a
%valve block to turn "on and off" recording.  However, data is
    still
%written to the file, but they will be all zeros.  This script
    has a simple
%routine to remove all zeros.  If this is not desired (for
    example turning
%on and off the recording) please comment it out.

%In April I switched from a UBW32 board to a NI USB DAQ to obtain
     the
%square law data.  This changed the output file from a csv file
    to NI's
```

```
%TDMS binary format.  The original CSV code will remain but will
   be
%commented out.

%In theory, this script may also run on Octave, but the file I/O
   package
%will be needed to use the file dialog box.
%
   ------------------------------------------------------------------------

%Clear the workspace
clear all;
%
   ------------------------------------------------------------------------

%Constants
%set's the window size to filter the square-law data
windowSize = 200;
%Receiver Noise Temperature for NEAT calc
Trec = 400;
%Integration Time for NEAT calc
tau = 2;
%Bandwidth for NEAT calc
beta = 10e6;
%
   ------------------------------------------------------------------------

%User Dialog entry
```

```
%The User input box will accecpt the calibration points from the
   user and
%will output the calibration data.

%Setup dialog options
options.Resize='on';
options.WindowStyle='normal';
options.Interpreter='tex';
%Setup MsgBox options
CreateStruct.Interpreter = 'tex';
CreateStruct.WindowStyle = 'modal';
%
   ------------------------------------------------------------------------


%Ask for user input for calibration
prompt = {'                          Enter calibration temp 1 (K)',
   'Enter calibration temp 2 (K)','Enter Calibration value 1:','
   Enter Clibration value 2:'};
dlg_title = 'Calibration for N200';
num_lines = 1;
def = {'371','77','.170','.103'};
N200answer = inputdlg(prompt,dlg_title,num_lines,def,options);


prompt = {'                          Enter calibration temp 1 (K):'
   ,'Enter calibration temp 2 (K):','Enter Calibration value 1:',
   'Enter Clibration value 2:'};
```

```
dlg_title = 'Calibration for X^2';
num_lines = 1;
def = {'371','77','2.1','1.9'};
x2answer = inputdlg(prompt,dlg_title,num_lines,def,options);

%Calibration variables based on two temperature points

N200temp1 = N200answer(1);
N200temp2 = N200answer(2);

%Enter the measured data points for temp1 and temp2
N200data1 = N200answer(3);
N200data2 = N200answer(4);

%Calibration variables based on two temperature points

x2temp1 = x2answer(1);
x2temp2 = x2answer(2);

%Enter the measured data points for temp1 and temp2
x2data1 = x2answer(3);
x2data2 = x2answer(4);

%Store the values into a and b
syms a b;

%Solve our two calibration points for the SDR
y = solve(N200data1*a+b==N200temp1,N200data2*a+b==N200temp2);
```

```
N200calib1 = double(y.a);
N200calib2 = double(y.b);

msgbox(sprintf('The calibrations points for the N200 is: %f and %
   f',N200calib1,N200calib2));

N200calibration = [y.a y.b];
fprintf('N200 Coefficient 1: %.2f N200 Coefficent 2: %.2f \r\n',
   N200calib1, N200calib2);

%Store the values into a and b
syms a b;

%Solve our two calibration points for the X^2
y = solve(x2data1*a+b==x2temp1,x2data2*a+b==x2temp2);

x2calib1 = double(y.a);
x2calib2 = double(y.b);

msgbox(sprintf('The calibrations points for the X^2 is: %f and %f
   ',x2calib1,x2calib2),CreateStruct);

x2calibration = [y.a y.b];
fprintf('X^2 Coefficient 1: %.2f X^2 Coefficent 2: %.2f \r\n',
   x2calib1, x2calib2);
```

```
%
   ------------------------------------------------------------------------

%Read data files.
%GNURadio outputs a binary file and LabView outputs a TDMS file


%Ask for the filename that has the TPR data from GNURadio
gnuradio_file = uigetfile('*.*','Select the GNURadio data file');
disp('Importing Radiometer data...')

%Ask for the filename of the Square law detector.  Comment out if
    not using
square_law = uigetfile('*.tdms','Select the Square_law data file'
   );
disp('Importing Square Law data...')

%Call the program that will convert the TDMS file format to a .
   mat file

tdms = convertTDMS2(true,square_law);

%The data created is nested in the array, we need to pull the
   data we want
x2=tdms.Data.MeasuredData(1,4).Data;

%Call the read_float_binary script.  This scripts reads the
   GNURadio
```

```
%binary protocol
gnuradio = read_float_binary(gnuradio_file);

%
   ------------------------------------------------------------------

%Remove zeros which is common in files that use the valve feature
    to
%control flow
gnuradio = gnuradio(gnuradio~=0);

%Create a time index
timen200=(1:length(gnuradio))*.5;
timex2=(1:length(x2))./100;

%
   ------------------------------------------------------------------

%Calculate the calibrated noise temperature for the SDR
N200calib_data = ((gnuradio*N200calibration(1))+N200calibration
   (2));

%Calculate the calibrated noise temperature for the X^2
x2calib_data = ((x2*x2calibration(1))+x2calibration(2));

%
   ------------------------------------------------------------------
```

```
%The square-law data is fairly noise, so we will filter it to
   smooth
%it out.
%First, convert from a sym matrix to a double
temp1=double(x2calib_data);
temp2=double(x2);
%Now filter it
avgx2_calib=filter(ones(1,windowSize)/windowSize,1,temp1);
avgx2=filter(ones(1,windowSize)/windowSize,1,temp2);
%
   ------------------------------------------------------------------


%Convert voltage from square-law to dBm.  53 mV is 1 dBm
dbmx2=x2./.053;
avgdbmx2=avgx2./.053;

%Calculate N E Delta T (NEAT)
%First calculatation is the NEAT expected based on BW and other
   parameters
%NEAT = (Ta+Tsys)/SQRT(tau+beta)

NEAT = (Trec)./sqrt(tau+beta);

%Now we can calculate the actual NEAT
%Look at a sample that is stable, in the LN2 area

for n = 1:100;
    rQ_stdev(n) = gnuradio(n+610);
```

```
end
estNEAT = std(rQ_stdev);

%Now print this information out
fprintf('Calculated NEAT: %.2f Actual NEAT: %.2f \r\n',NEAT,
   estNEAT);
%
   ------------------------------------------------------------------------



%Plot the calibrated data
figure;
subplot(2,1,1);
plot(N200calib_data);
title('N200 TPR Calibrated Data');

% Create xlabel
xlabel('Time');

% Create ylabel
ylabel('Calibrated Noise Temperature in K');
subplot(2,1,2);
plot(avgx2_calib);
title('x^2 Calibrated Data');
figure
timeshift = timex2+75;
plot(timen200,N200calib_data,'r');
hold on;
```

```
plot(timeshift,avgx2_calib,'g');
title('X^2 and N200 calibrated data');
ylabel('Noise Temp K');
xlabel('Time sec');
legend('N200','X^2');
%
   ------------------------------------------------------------------------

%Plot the raw data
figure;
subplot(2,1,1);
plot(timen200,gnuradio);
title('N200 TPR Raw Data');
xlabel('Time Sec');
ylabel('rQ Value');
subplot(2,1,2);
plot(timex2,avgx2);
title('x^2 Raw Data');
ylabel('Raw Voltage');
xlabel('Time Sec');
axis([-inf inf 2.1 2.4]);

%
   ------------------------------------------------------------------------

%Plot the calibration line
figure;
x=linspace(str2double(N200data2),str2double(N200data1),200);
```

```
y=linspace(str2double(x2data2),str2double(x2data1),200);
subplot(2,1,1);
plot(x, N200calib1*x+N200calib2,'-r');
title('Calibration line for N200 SDR');
xlabel('raw value');
ylabel('Noise Temperature K');
subplot(2,1,2);
plot(y,x2calib1*y+x2calib2,'-b');
title('Calibration line for X^2 detector');
xlabel('raw value');
ylabel('Noise Temperature K');
%
   ------------------------------------------------------------------------


%plot x2 with dBm
figure
plot(timex2,avgdbmx2);
title('x^2 power readings');
ylabel('dBm');
xlabel('Time Sec');
axis([-inf inf 37 40]);
```

Python code for analyzing data

Total Power Radiometer

```
#-*- coding: utf-8
#Radiometer Parsing Function

#This code shows an example of reading in and plotting data that
   is outputted from a GNURadio GRC file.
#In this example a Total Power Radiometer is developed in
   GNURadio GRC and uses the File Sink function to store the data
   .
#The plot then shows the total power output from the radiometer
   as a matched load is submerged in Liquid Nitrogen,
#then Ice Water and then left to dry.
# - - -
#
### Read the data

# Import Needed functions

# Import needed libraries
from pylab import *
import pylab
import scipy
import numpy
import scipy.io as sio
import csv
```

```
# Use this to set the filename for the data file and CSV
   Calibration file.


tpr = 'tpr_2014.06.12.Lab0.dat'
calib = 'tpr_calib_2014.06.12.Lab0.csv'
x2_data = 'tpr_x2_2014.06.12.Lab0.csv'


# Uses SciPy to open the binary file from GNURadio


f = scipy.fromfile(open(tpr),dtype=scipy.float32)


# Because of the valve function in GNURadio, there are zeros that
    get added to the file.  We want to trim out those zeros.

# In[5]:

f = numpy.trim_zeros(f)


# Create an index array for plotting.  Also, since we know the
   interval the data is taken, we can convert this to an actual
   time.
```

```
# In[6]:

y = numpy.linspace(0,(len(f)*.5),numpy.size(f))


### Plot the data

# In[7]:

plot(y,f)
xlabel('Time (sec)')
ylabel('rQ Values')
title('rQ vs Time')
grid(True)

pylab.show()

# ## Calibration

# The rQ values are the raw values from the total power
   radiometer and are uncalibrated.  While the graph shows the
   change in the total power recorded and shows that the
   radiometer can detect changes in noise temperature, it has no
   other meaning than that.  What we want is to show what the
   total power is in relation to a noise temperature.  Since we
   have recorded the values of the rQ at fixed and known
   teperatures, we can create a calibration line and calibrate
```

```
   the radiometer.  For this experiment, we found that the
   following values matched our two known temperatures.
#
# |rQ Value|X^2 Voltage|Temperature
# |--------|-----------------------
# |.0977   | 1.9617    |77 K
# |.1507   | 2.085     |273.15 K
#
# We can now solve for y = mx + b since we have two equations and
    two unknowns.
#
# To work with this, a calibration file is created.  This is a
   very simple CSV file that contains 3 values: The raw rQ value,
    the raw voltage from the square-law detector (discussed later
   ) and the observed temperature.  The table above would then
   look like the following in the file.
# ```
# .0977,1.9617,77
# .1507,2.085,273.15
# ```
# - - -

# We need to read in the values from our CSV file that contains
   the values

# In[67]:

read_csv = open(calib, 'rb')
```

```
csvread = csv.reader(read_csv)
rQ_values = []
temp_values = []
voltage = []

for row in csvread:
    rQ,volt,temp = row
    rQ_values.append(float(rQ))
    voltage.append(float(volt))
    temp_values.append(float(temp))
read_csv.close()

a = numpy.array([[rQ_values[0],1.0],[rQ_values[1],1.0]],numpy.
   float32)
b = numpy.array([temp_values[0],temp_values[1]])

z = numpy.linalg.solve(a,b)
print z


# Now we apply these values to the array that holds our raw rQ
   values

g = f*z[0]+z[1]


# Now we can re-plot the graph but this time with the calibrated
   noise temperatures
```

```
plt.figure()
plot(y,g)
xlabel('Time')
ylabel('Noise Temperature (K)')
title('Temp vs Time')
grid(True)

pylab.show()

# This is looking better, but the time at the bottom doesn't have
    much meaning.  Since we know the sample rate of the Software
   Defined Radio, we can calculate the time interval between each
    sample.

# - - -
# # Square-law data
#
# We now want to look at the data from the Square-Law detector to
    verify the operation of the SDR.  In the experiment that was
   conducted above, a power splitter was used to split the RF
   signal so that one went to the SDR and the other to a square-
   law detector (with a 3.1 dB loss though).  Therefore both data
    should be the same.  Let's read and then plot this data.

read_csv = open(x2_data, 'rb')
csvread = csv.reader(read_csv)
dummy = []
```

```
x2_voltage = []

for row in csvread:
    dummy,x2voltage = row
    x2_voltage.append(float(x2voltage))
read_csv.close()


# Like the SDR data, we want to have a time reference at the
   bottom.

w = numpy.linspace(0,(len(x2_voltage)*.01),numpy.size(x2_voltage)
   )

plt.figure()
plot(w,x2_voltage)
xlabel('Time (sec)')
ylabel('Voltage (V)')
title('X^2 Voltage vs Time (Noisy)')
grid(True)

pylab.show()

# The Square-law detector doesn't have a filter on it unlike the
   data we get from the SDR.  The GNURadio program takes the data
    and applies a Low Pass Filter to "clean up" the information.
    We need to do the same with our Square-law data.
```

```
from scipy import signal
N=100
Fc=2000
Fs=1600
h=scipy.signal.firwin(numtaps=N, cutoff=40, nyq=Fs/2)
x2_filt=scipy.signal.lfilter(h,1.0,x2_voltage)



plt.figure()
plot(w,x2_filt)
xlabel('Time (sec)')
ylabel('Voltage (V)')
title('X^2 Voltage vs Time')
axis([0, 610, 1.94, 2.12])
grid(True)

pylab.show()

# Now we wish to calibrate this data as well.  We will use the
    same file and use the calibration points in that file.

a = numpy.array([[voltage[0],1.0],[voltage[1],1.0]],numpy.float32
    )
b = numpy.array([temp_values[0],temp_values[1]])

z = numpy.linalg.solve(a,b)
print z
```

```
x2_calib = x2_filt*z[0]+z[1]


plt.figure()
plot(w,x2_calib)
xlabel('Time (sec)')
ylabel('Voltage (V)')
title('X^2 Noise Temp vs Time')
axis([0, 610, 70, 300])
grid(True)

pylab.show()

# This looks to be the same as our SDR graph, but let's overlay
    them to make sure

plt.figure()
plot(w,x2_calib,'r',label='X^2')
plot(y,g,'b',label='SDR')
xlabel('Time (sec)')
ylabel('Voltage (V)')
title('Noise Temperature vs Time')
axis([0, 610, 70, 300])
grid(True)
legend(loc='lower right')


# We have some timeshift due to two reasons.  One, the timing isn
    't always perfect when starting the collection of the two data
```

sets. And two, we get a timeshift from filtering the square-law data

```
pylab.show()
```

APPENDIX B. EE 518 Test Results

E E 518 Lab Tests

Further testing of the square-law detector was performed by using the radiometer in a real world event. This test was exercised in conjunction with the Microwave Remote Sensing class (E E 518) under Dr. Brian Hornbuckle. In the Spring of 2012 the radiometer was moved to the roof of agronomy and the EE 518 students conducted a number of tests using the N200 software defined radio to collect the data.

The E E 518 test however showed that there were additional problems with the radiometer. While the test showed that the SDR could in fact read data, the data was skewed. It was later found out that the radiometer was generating an interfering signal that caused the power readings to be elevated. It was also found that the interfering signal was also flucatating and seemed to correspond somewhat to the physical position of the radiometer. This was found as a result of having the SDR record the raw I/Q values and then was analyzed later. Through this analysis, we found that a strong harmonic was developed and caused a spike in signal being recorded. Although the exact reason for this has not been found, the problem has been isolated to something in the RF Front End of the ISU radiometer.

In the Spring of 2014 the E E 518 class ran another experiment, but this time a different one. This experiment mimicked the experiments that the author ran to calibrate and test the radiometer. This allowed an outside source to validate the results that I was getting with running the radiometer. Like my tests, the students submerged a matched load into liquid nitrogen and then in boiling water. The liquid nitrogen was assumed to be at 77 K and the boiling water was measured to be at 99 C or 372 K. The students were then given a mystery sample in which they had to determine the temperature of a water sample. Since the students had to points,

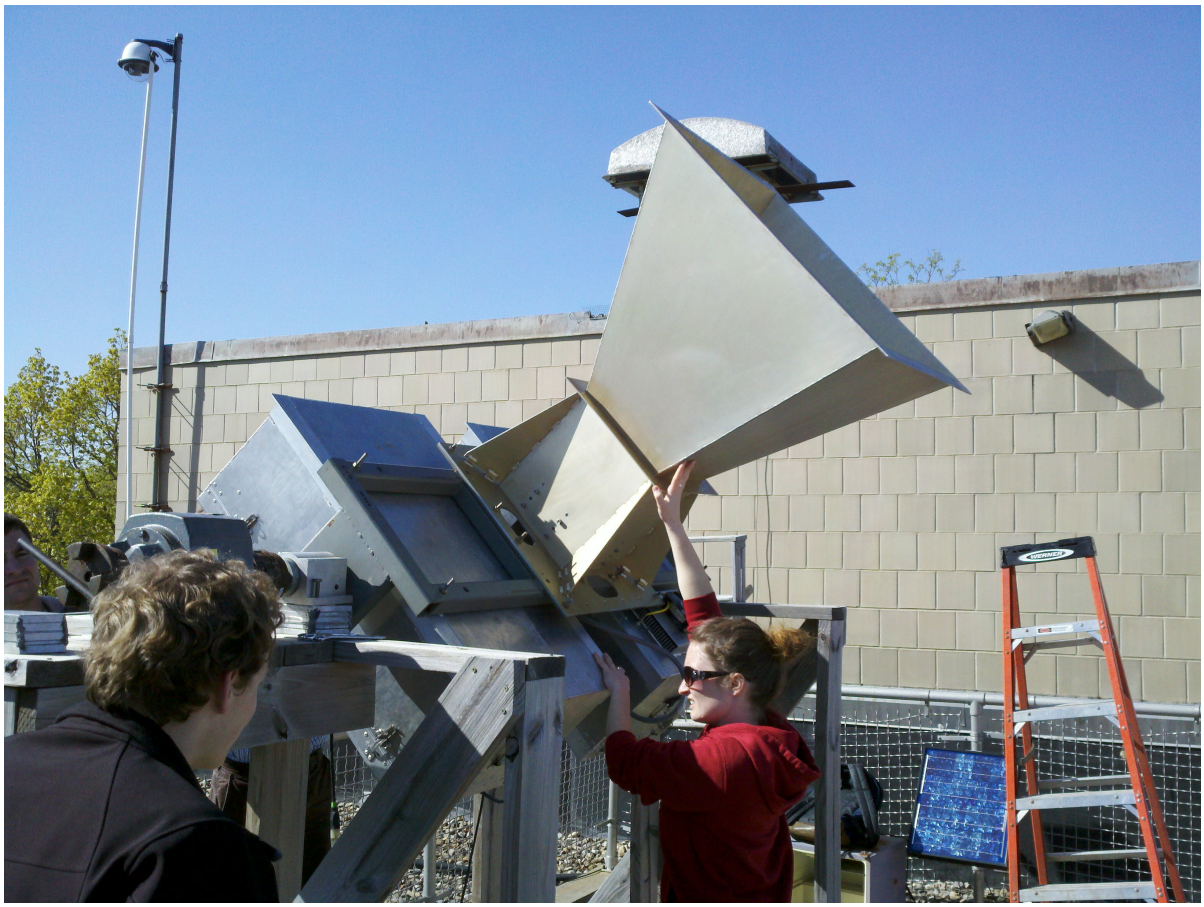

Figure B.1 Students rotating the radiometer for an experiment on Agronomy Hall

they could find the calibration line and determine the temperature. The experiment was run twice as we expected the radiometer may have drifted between measurements.

B.0.4 Test setup

For the E E 518 class, the setup was similar to experiments that was run to test the results in this thesis. For the E E 518 however the students only used the N200 SDR for recording the total power readings. The ISU RF front end was used only to provide the RF front end as in other tests and the output from that RF front end was feed into the N200. A matched load was attached to the input of the RF front end. This matched load was then submersed into either the liquid nitrogen or water baths.

B.0.5 Lab Experiment

Students conducted the experiment by submerging the matched load into Liquid Nitrogen, then boiling water, then an unknown sample. The LN2 was believed to be at 77 K. The boiling water was measured by the students and was recorded to be 99 C. The mystery water was then measured by myself and recorded both before the students placed the match load in there and after to see if it changed temperature between the experiments. During this time the GNURadio program that I wrote recorded the total power information to the hard drive. This file was then given to the tests and an example Matlab script was also provided for them to use to read the data file.

B.0.6 Lab Results

Two labs were conducted, however the data from the first lab was more stable and had better results than the second lab. Therefore the students only used the data set from the first lab experiment. Students were then asked to write a lab report that calculated the temperature of the mystery water, write up what each component in the radiometer does, plot the raw rQ values, calculate and plot the calibration lines and finally calculate the NEδT for the system by doing a standard deviation on a section of the data that was at a stable temperature.

The results of the lab experiments were mixed in the students reports. Almost all of the students reported a temperature cooler than the actual recorded temperature. After looking at the data and the student reports, there is a couple of reasons this may have happened. First, it would appear that there may not have been enough time to allow the matched load to equalize. The graph of the rQ values does not show a stable line as expected with the LN2 and hot water baths. It appears that there was a dip in the rQ values and that the students picked that lowest point as the mystery water temperature. It is uncertain what caused the dip. Second, it is safe to say that the temperature of the mystery water changed due to the matched load being inserted in it. There was nothing to keep the mystery water at a fixed temperature as it was simply water at room temperature. These may have caused some of the odd readings and made it difficult for the sample to stabilize unless it was allowed to sit longer.

B.0.7 Conclusion

While the students were not able to get an accurate reading of the mystery water, the calibration points, and NEδT values were within what was expected of the system and matched fairly close to those calculated by the author. Overall, this experiment proved to be a good exercise for the students as it allowed them to have a hands on exercise and see and operate a radiometer for themselves.

APPENDIX C. Direct-Sampling Digital Correlation Radiometer

Theory of Operation

The original ISU Radiometer was constructed by the University of Michigan and was based on Dr. Mark Fischman's dissertation on a Direct-Sampling Digital Correlation Radiometer [**?**)]. This type of radiometer amplifies the signal but then immediately digitizes the signal to be processed. In many ways, this is how a software defined radio works, however in this case the ISU radiometer is under-sampled. However, as explained by Dr. Fischman this is acceptable for a total power radiometer since the only information we need is power information. The radiometer that Dr. Fischman proposes also correlates the signals, one from the vertical polarization and the other from the horizontal polarization from the antenna.

C.0.8 Implantation in the ISU Radiometer

The original ISU radiometer worked by under-sampling the RF signal coming into the analog to digital converters. This under-sampled signal lacks enough sample to recreate the full signal, however the power information can be extracted. The original ISU radiometer also used both the vertical and the horizontal polarization coming from the antenna. This allowed the radiometer to correlate between the vertical and horizontal polarization.