UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

Using Inexpensive Software-Defined Radios as GPS Receivers in a Ground-Based Augmentation System

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

Jaxon Taylor
Norman, Oklahoma
2022

Using Inexpensive Software-Defined Radios as GPS Receivers in a Ground-Based Augmentation System

A THESIS APPROVED FOR THE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

BY THE COMMITTEE CONSISTING OF

Acknowledgements

I would like to thank my loving wife, Morgan, for all of her support throughout my studies. I'd also like to thank Dr. Davis and Dr. Zhang for their continuing support and push to get this thesis completed. I would like to thank David Milligan for his all-day support and help with many advanced software issues. I'd like to thank Fernando Soto for his extensive help getting the hardware for this system working. I'd like to thank the rest of the team, Ry Fleming, Nick Schneider, Brandon Mansur, Keegen Hart, Avery Mayfield, and also Jacob Henderson. I'd also like to thank everyone else who was involved that isn't listed here. Lastly, I would like to thank all of my friends and family for their unending support.

Abstract

Ground Based Augmentation Systems (GBAS) are used to augment Global Positioning Systems (GPS) signals to make the position solutions significantly more accurate and precise. The systems have been studied and demonstrated before, however, they would typically use dedicated GPS receivers. These receivers are typically expensive and lack the ability to be customized for different situations. This thesis attempts to use a software-defined radio (SDR) using a software called Global Navigation Satellite Systems-Software Defined Receiver (GNSS-SDR) to replace these dedicated GPS receivers. Doing this requires multiple GNSS-SDR receivers to output the data in real-time to a central GBAS computer for real-time computations. This is done using the User Datagram Protocol (UDP) output functionality of the GNSS-SDR software. The output then needs to be received on the GBAS computer and decoded. A novel method for decoding the Google Protocol Buffer encoded UDP messages is used for efficient LabVIEW decoding. The outputs are then tested using an existing Closed-Loop Ground Based Augmentation System (CL-GBAS) program. An analysis of the raw pseudorange values of two different SDRs and certified GPS receivers is then performed. The research performed in thesis will ideally be used as a stepping stone for more thorough analysis of different SDRs and the GNSS-SDR program in an attempt to make SDRs an effective receiver for GBAS purposes.

Contents

List Of Figures

Chapter 1

Introduction

1.1 Motivations of the Thesis

Most modern GPS receivers only output positional data like the latitude, longitude, and altitude, which is sufficient for the majority of the users. However, high precision applications, like in the Local Area Augmentation System (LAAS) used for landing planes, the raw data from the satellite is required so that the accuracy can be improved to a level beyond what a single receiver can achieve. Modern receivers that can output this raw data are either very expensive or not available for public use. Thus, using low-cost Software-Defined-Radios (SDR), such as the Great Scott Gadgets' HackRF One equipped with the NooElec Tiny TCXO is explored. This software-defined radio is controlled by a computer running open-source software called Global Navigation Satellite System-Software Defined Receiver, or simply, GNSS-SDR [5]. Since the software is open source, it can be studied, fixed, and customized to fit the exact needs of the user. In this case, the software does indeed output the raw data required for the LAAS algorithms.

This thesis presents a comparison of the HackRF One results with the results of a GPS receiver that is certified to work with LAAS. The HackRF One (and other SDRs) results will be compared to the same receivers previously used for the OU LAAS, the Thales GG12. When comparing the receivers, the raw pseudorange data and the positional data will be compared side-by-side. The raw data will also be processed through the algorithms previously developed for the OU CL-GBAS system as described in [4], [6], and [7].

This thesis explores the feasibility of using a low-cost Software-Defined Radio (SDR) as an alternative for expensive and certified GPS receivers for use in local area augmentation systems (LAAS). In particular, this thesis aims to replace the old receivers used in the CL-GBAS system at the University of Oklahoma (OU), with the use of SDR technologies.

1.2 Organization of the Thesis

Chapter 2 covers the background of GPS navigation and introduction of GNSS augmentation, such as GBAS/LAAS.

Chapter 3 gives a background into SDRs, GNSS-SDR, and a description of the testing setup. This chapter describes how the GNSS-SDR software communicates and interfaces with the SDR and how to output that data in real time. It describes how the data is received, decoded, and sent to the main programming work station.

In chapter 4, an analysis of the pseudorange data of the HackRF One alongside a certified GPS receiver is shown. In this chapter, the debugging processes and attempts to fix issues are discussed in detail.

Chapter 5 contains the final summary of the thesis research and also discusses the potential work for future research and improvements.

Chapter 2

Theory and System Architecture

2.1 Basic Principles of GNSS Navigation

GPS uses the concept of satellite ranging. This is a method of determining the position of a receiver by measuring the distance from the receiver to multiple satellites. The distance from the satellite to the receiver is measured indirectly by measuring the time it takes for the GPS satellite message to send to the receiver. This difference in time can be multiplied by the speed of light, c (299,792,458 m/s), to give the distance in meters. Since this measures the distance to the satellite but does not account for all the known errors, it is called the pseudorange.

The pseudorange, a value calculated by the receiver, is susceptible to many errors. These errors include multipath, where signals reflect off of buildings and other objects, solar storms, where the waves get distorted when the Sun emits more charged particles, and even imprecise internal clocks on the receivers. These errors can lead to significantly incorrect position solutions. Thus, it is important that these errors are minimized and the pseudorange is corrected.

When the receiver calculates the pseudorange to a particular satellite, it can map out a sphere around that particular satellite whose radius is equal to the pseudorange. This is because the direction of the signal cannot be determined, but the distance (pseudorange) is reasonably well known. The center of this sphere is a well known location since the location and motion of the satellite are well-known values. This is due to the very accurate position measurements taken from observation sites as well as the use of Kepler's and Newton's Laws to accurately predict the position of the satellite. So, the receiver is somewhere located on the surface of that sphere, as seen in Figure 2.1. When a second pseudorange is calculated, one can similarly draw a sphere around that satellite. Since the receiver has to be on the surface of the sphere around the first satellite and it also has to be on the surface of the sphere around the second satellite, it is necessary to look at where these two spheres intersect. The intersection of these two overlapping spheres is a circle. This refines the position of the receiver to a circle which is shown in Figure 2.2. Using a third satellite and thus a third pseudorange adds another sphere whose intersection with the other two spheres yields two points. One of these two points lies on the surface of the Earth while the other does not. The one not on the surface of the Earth can be discarded as an extraneous solution (Figure 2.3). Thus, using the pseudorange from just three satellites, the rough position of the receiver can be determined. However, using just three satellites is not enough for the accuracy we have become accustomed to.

A fourth satellite is also required which is used to eliminate the clock error. Typically, three spheres will be able to intersect, but when adding a 4th sphere, the spheres might not intersect. However, the receiver can use this information to determine what the clock error is, allowing the four (or more) spheres to intersect [8]. All pseudoranges will then be corrected and the position can be accurately determined. This algorithm alone can calculate the position of a receiver to within about 7 meters 95% of the time [1], [9].

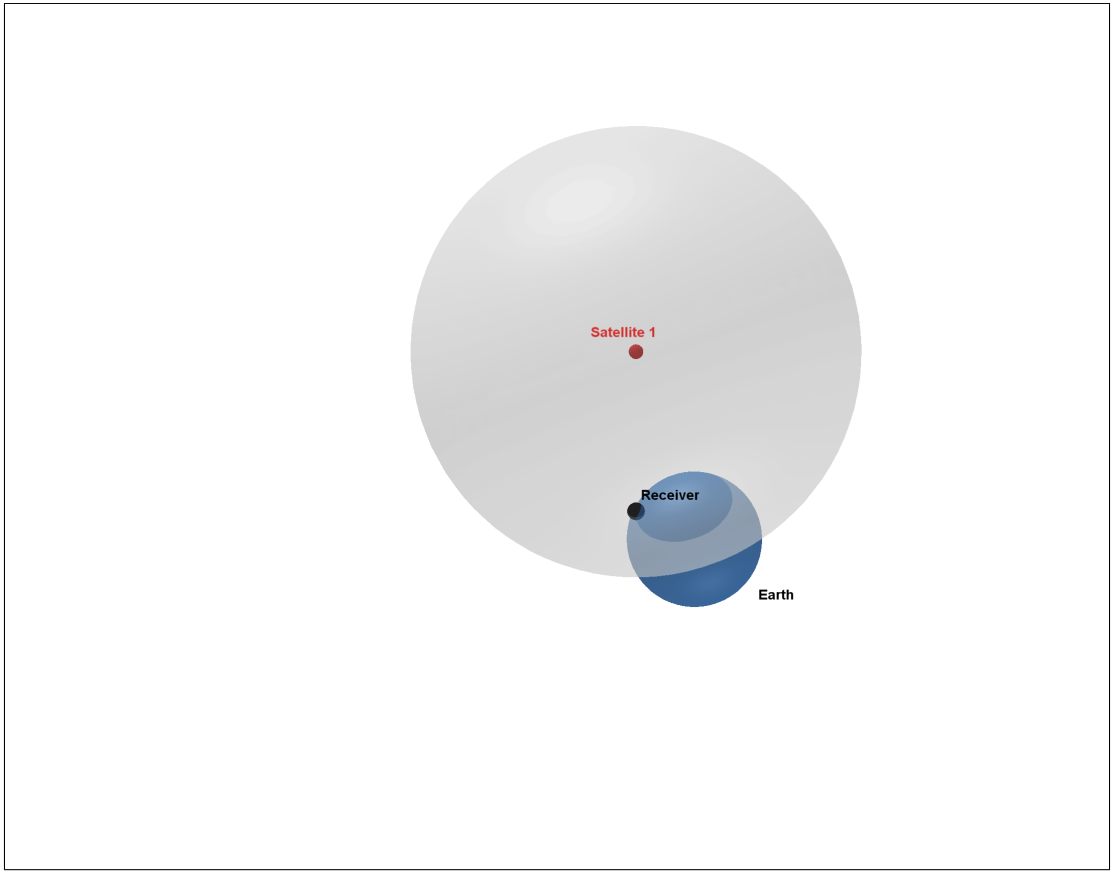


Figure 2.1: The sphere around a GPS satellite whose radius is the pseudorange determined by the receiver. With just one satellite's data, the receiver's location could be anywhere on that sphere.

Although GPS has many obvious applications in the world, one industry that uses GPS signals is the aviation industry. Prior to 1994 [10], airplanes primarily used the Instrument Landing System (ILS) as a means for navigating to a runway of an airport in less than ideal conditions. The ILS, a system developed around WWII, contains two main components, the localizer and the glide slope. The localizer sends out directional radio waves which the plane can pick up so that

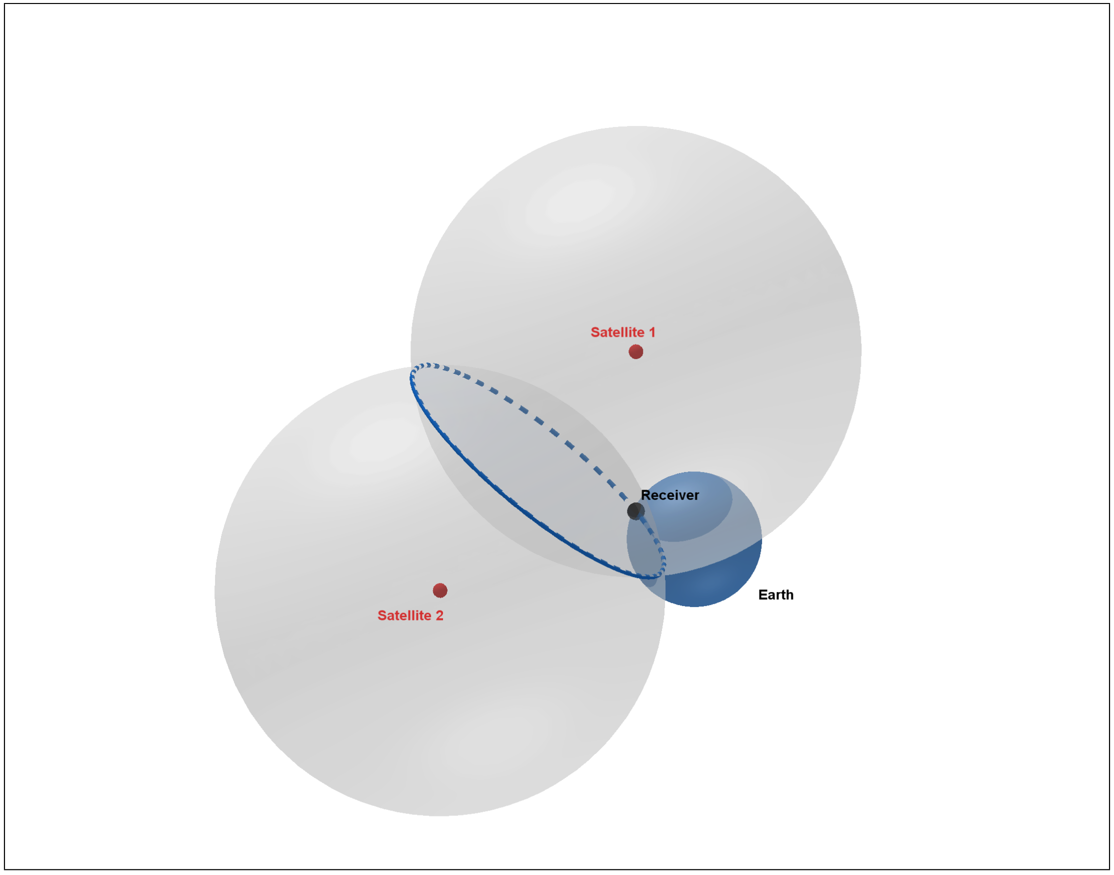


Figure 2.2: When a second satellite, thus a second sphere is added, the receiver must be on the intersection of these two spheres, the circle shown in blue.

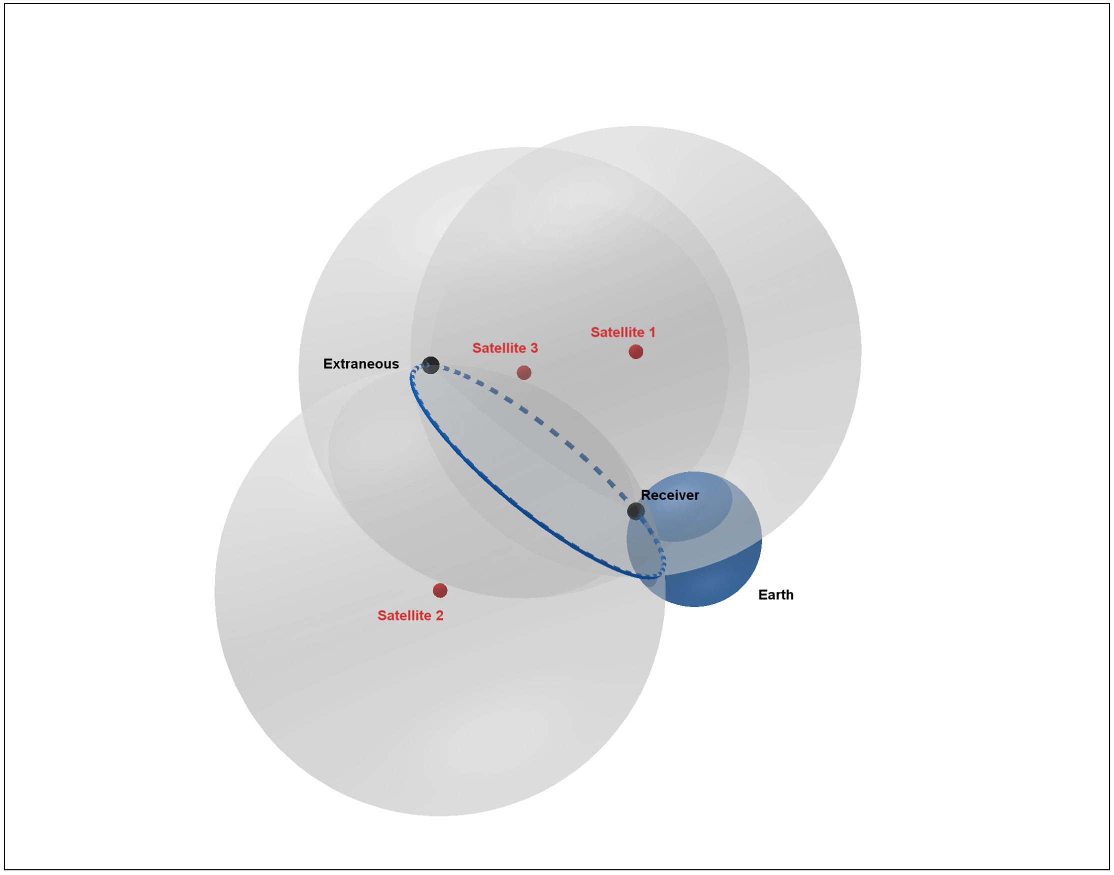


Figure 2.3: When a third satellite is added. The spheres' intersection becomes two points on the same circle above, shown as the black dots above. One of these will be on the surface of the Earth, the other is an extraneous solution.

the plane can get aligned left/right (horizontally) with the center of the runway. The glide slope is similar except that it is used to guide the plane vertically to the runway in a stable approach. These systems work well in a majority of scenarios, but have several drawbacks. Some of these drawbacks include the need for an ILS on each runway, reverse sensing when approaching the runway from the "back course" direction, and signal interference. Thus, the need for cheaper and more reliable systems has risen.

GPS is a good alternative to the ILS because it is cheap and independent of the technology at the airport. However, GPS is subject to many issues in accuracy. These issues are due to things like signal multipath errors, signal distortion due to solar storms, signal distortion due to thunderstorms, and other things. Even in the best case scenario, GPS receivers are typically only reliable to within around 5 meters 95% of the time [9]..

Although standard GPS is not extremely accurate, there are ways to improve the resulting position, sometimes referred to as the position solution, or just solution. One of these methods, which is very effective, is called differential GPS (DGPS). Differential GPS works by having at least two antennas within close proximity where one has a well-known location. When the current position solution is calculated for the known antenna, the algorithm checks to see how far it is from the true location of that antenna, thus determining the error. This error is then converted into a correction that is sent to the other antenna, and the resulting position solution of the second antenna is vastly improved as demonstrated, even with a low-cost DGPS, in [11]. This is valid only when the antennas are nearby, as it assumes the effects due to the atmosphere are similar and can be corrected in the same way. One downside to this technique is that it requires a nearby and well-surveyed antenna with a receiver that can send real-time adjustments to the other antenna's receiver. However, for antennas at an airport, this is not an issue, making it ideal for aviation.

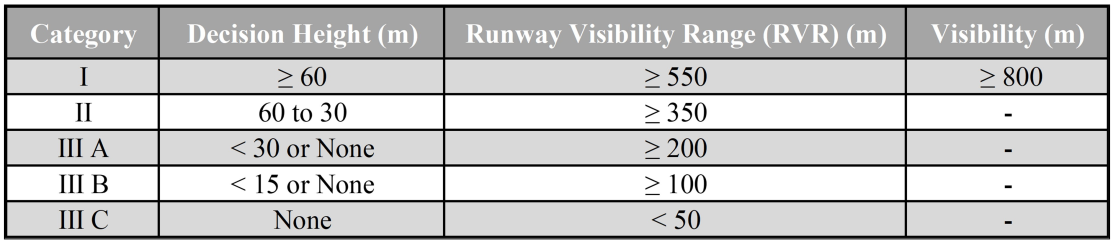
Another method used to improve the accuracy of GPS is to use more than just four satellites. Due to the physics of the atmosphere, satellites lower in the sky toward the horizon typically result in less accurate position solutions. This is due to the fact that signal sent by the satellite must travel through significantly more air molecules than one directly overhead. This results in lower signal strength and higher amounts of noise. This reduces the overall signal-to-noise ratio (SNR). Thus, it is ideal if the receiver can choose more satellites to calculate the position solution, or better yet, select which satellites it wants to use. One such application of this method is called Receiver Autonomous Integrity Monitoring (RAIM). This is a system which allows a receiver to detect poor signal from any particular satellite. If there are more than four satellites available, some receivers can choose to exclude any satellites which the RAIM determines to be faulty. This combined system is called fault detection and exclusion (FDE).

Most GPS receivers typically use a single frequency, however, using more than one frequency (dual/multi-band) can be very beneficial. With the use of two separate frequencies, the effects of signal multipathing and some atmospheric effects can be significantly reduced as seen in [12], [13], [14]. These techniques have been implemented and shown to improve the position solutions in GNSS-SDR as seen in [15]. Further discussion on the advantages of multi-band GNSS receivers can be found in [16].

Since GPS has shown the ability to have its accuracy drastically improved, systems utilizing correction methods that were previously described have been created for use in the aviation industry. One type of system is called space-based augmentation system (SBAS). These systems typically use geostationary satellites to help provide corrections to a particular area on the globe. One such implementation of SBAS is called wide-area augmentation system (WAAS). As the name suggests, this system provides GPS correction data for a very wide-area. Currently WAAS is available to all of North America. Since it covers such a large area, WAAS typically only provides marginal improvements to the position solution, around a 1.6m nominal vertical accuracy as seen in the WAAS performance standard in [17].

Another very effective system for computing and transmitting GPS correction data is the ground-based augmentation system (GBAS), also referred to as local-area augmentation system (LAAS). GBAS implements differential GPS techniques with multiple antennas to send correction data over very-high frequency (VHF) data broadcasters (VDB). These corrections can be received by special receivers on aircraft called multi-mode receivers (MMR). This system is illustrated in Figure 2.5. These GBAS landing systems have many advantages over the ILS including simplicity, annual maintenance and operational costs, and the ability to service multiple runways with one system. These advantages are discussed and highlighted in a case study of Viseu Airfield in [18]. Another article in [19] shows how ILS was outperformed by GBAS.

"GBAS was developed to provide the required accuracy, availability, integrity, coverage, and continuity to initially support CAT I precision approaches and eventually CAT II and III precision approaches. Unlike current ILS, a single GBAS ground station provides precision approach capability to all runway ends at an airport [20]." As stated above in the 2021 Federal Radionavigation Plan, GBAS is currently in use but only for CAT I precision approaches, the exact specifications of which are described in [21]. More research and demonstration of better systems can lead to approval of higher precision approaches. The different types of precision approaches are shown in Figure 2.4, and are described in [22].



Category	Decision Height (m)	Runway Visibility Range (RVR) (m)	Visibility (m)
I	≥ 60	≥ 550	≥ 800
II	60 to 30	≥ 350	-
III A	< 30 or None	≥ 200	-
III B	< 15 or None	≥ 100	-
III C	None	< 50	-

Figure 2.4: The precision approach requirements for different categorical levels.

The traditional way that a GBAS would compute the corrections is by modeling the atmospheric conditions and using statistics to approximate the effects of the atmosphere and any other issues. This approach is good for the most common errors seen in GPS, but is not an accurate way to accurately model all possible effects, which depend not only on location but also time. Two example of these effects are unpredictable weather and solar storms.

A second technique for a GBAS to compute corrections is by using a closed-loop feedback system. Referred to as a closed-loop GBAS (CL-GBAS), this method works by using a well-surveyed antenna and seeing if the solution from the GBAS accurately matches the location of this extra monitoring antenna that was not used in the GBAS position solution. Instead of attempting to correct for all the different atmospheric effects, the closed-loop system adjusts its corrections so that the computed position of the well-surveyed antenna and the actual surveyed position are within some tolerance. Thus, if the computed position for the well-surveyed antenna is accurate, then there is an increased confidence that the corrections are accurate and can be sent to airplanes in the area. This system was successfully theorized and demonstrated in [4].

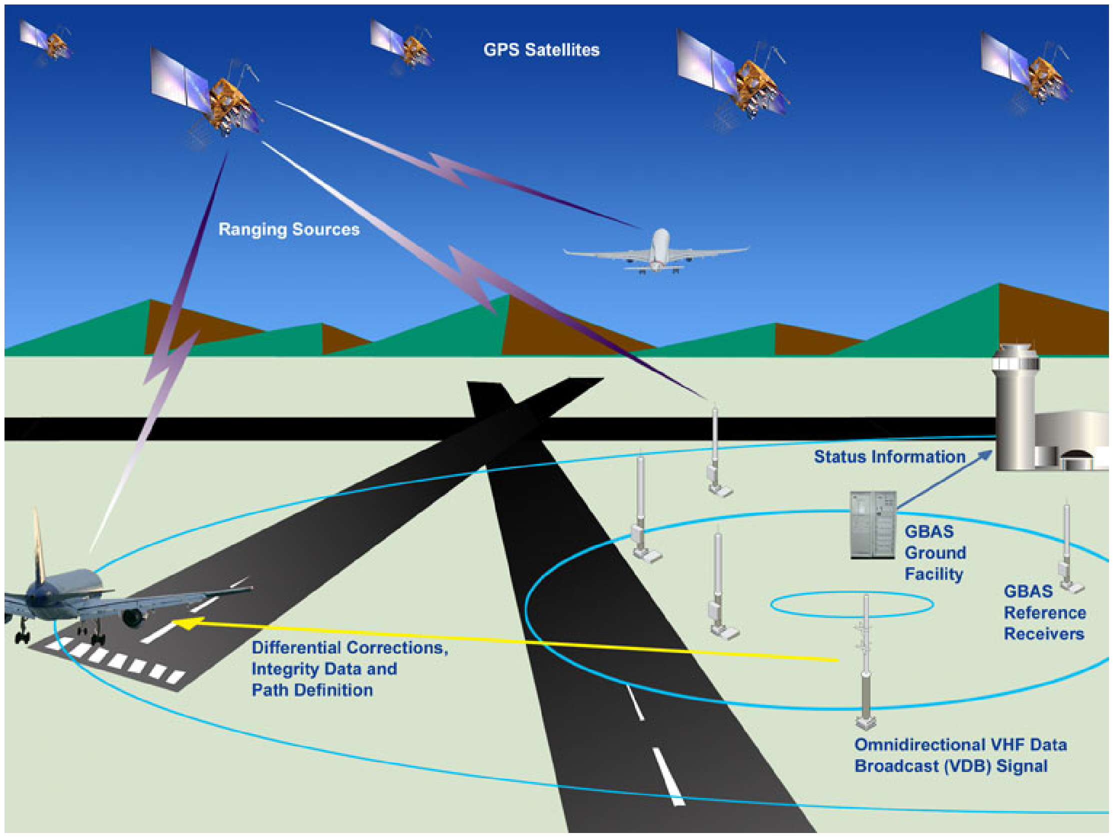

GPS Satellites
Ranging Sources
Status Information
GBAS Ground Facility
GBAS Reference Receivers
Differential Corrections, Integrity Data and Path Definition
Omnidirectional VHF Data Broadcast (VDB) Signal

Figure 2.5: GBAS Architecture image from the FAA website [1].

GPS is just one of many different satellite systems. Although it was the first, there are now at least 4 other Global Navigation Satellite Systems (GNSS) in the sky. These include the Russian GLONASS system, the European GALILEO system, the Chinese BEIDOU system, the Indian IRNSS system, and the Japanese QZSS system [23]. Some systems have been shown to have improvements over others, which can result in more accurate positions. In particular, GALILEO's message structure has been shown to have some advantages over the GPS message structure [24].

Chapter 3

GNSS-SDR: GNSS Implementation in Software Defined Radio

3.1 Software-Defined Radio (SDR)

Software Defined Radio (SDR) is an architecture for a radio frequency (RF) receiver in which the bulk of the processing is done digitally via software and not by the actual hardware. This design simplifies the hardware requirements and allows for the signal processing to be done via the software. This allows changes/fixes with a particular receiver to be fixed without needing to replace the entire radio unlike with traditional radios. With the software controlling the signal processing, it is possible for a single receiver to work within a large range of frequencies. The HackRF One, an open-source SDR by Great Scott Gadgets can be used within the 1 MHz to 6 GHz range. With traditional radios, this amount of wide-band capability would require an expensive and dedicated radio. These radios are much more expensive than the HackRF One which typically retails around $350.

Unlike traditional radios, the hardware on the SDR is simplified. Traditional radios, which typically use the superheterodyne technique, have several hardware components. These components include multiple amplifiers, filters, a demodulator, and a local oscillator. With all these components, the radios can be more complicated to manufacture, more delicate, and thus more expensive overall. SDRs use an RF filter to choose the desired frequency, and then convert those analog signals to a digital signal using an analog to digital convert (ADC). This ADC allows the signal to be further processed using digital techniques on a computer. Having the ability to choose a desired frequency over a very broad range is a major advantage of the SDR. The SDR will then have its software take over to further analyze the signal. The software here also holds an advantage because it can be very easily manipulated and bugs can be easily fixed. Thus, with the rapidly improving digital circuitry, software enhancements, and extreme flexibility, the SDR holds an enormous advantage over traditional radios.

3.2 Great Scott Gadget's HackRF One

The HackRF One, shown in Figure 3.1, is a commercial and open-source SDR originally made by Great Scott Gadgets. This SDR was initially researched and thought to be one of the most cost-effective solutions for a customizable and accurate GPS receiver for GBAS in a graduate special studies course by Gus Azevedo in [25]. It has the capability of operating between 1 MHz and 6 GHz, making it ideal for working with typical GPS frequencies (1.57542 GHz). This SDR is rather inexpensive as the retail cost is around $350, which is ideal for testing the capabilities of GNSS-SDR and GBAS. Typically, GPS signal processing and satellite tracking requires very accurate internal clocks on the order of 1 part-per-million (ppm). Unfortunately, the HackRF One has a lower quality internal clock with a precision of about 20 ppm at 10 MHz. Trying to use the stock HackRF

One as a GPS receiver with the GNSS-SDR software directly will result in failure to lock onto any satellites. The error message shown when using it is the common “Loss of lock in channel ...” message. The software is never able to achieve bit synchronization with the satellite and thus will not output a position solution or any related satellite data. However, an external oscillator can be added to the HackRF One which will vastly improve its performance. Then GNSS-SDR can successfully lock onto the satellites and compute a position solution reliably.

Figure 3.1: Great Scott Gadget’s HackRF One. Image from their website [2].

For this project, the external oscillator used was NooElec’s Tiny TCXO, whose information can be found in [26]. This oscillator has a precision of 0.5ppm at 10 MHz, which is 40 times more precise than the HackRF One’s default clock. The Tiny TCXO is a small chip which plugs directly into the HackRF One circuit board. This is a very convenient upgrade, although the case of the stock HackRF One is too small for the Tiny TCXO chip to fit in. The HackRF One also has an SMA connector for which an external oscillator can be plugged into instead of having to plug it directly into the circuit board. There is also another SMA connector which will send out the clock signal from the HackRF One, allowing multiple devices to be synchronized using the exact same clock signal.

3.3 Ettus Research's USRP E310

To help compare the results of the HackRF One, a more expensive and higher quality SDR was also used. The Ettus Research's USRP E310 SDR, seen in Figure 3.2 is a much more expensive SDR which outputs data via an Ethernet cable instead of a serial cable, which poses new challenges. However, the high quality internal components of this receiver will help serve as a valid comparison against the more affordable HackRF One.

3.4 GNSS-SDR

GNSS-SDR, which stands for Global Navigation Satellite Systems Software Defined Receiver, is an open-source software program that integrates with software defined *radios* (SDRs) and processes the signals. It can use the SDR to acquire signals from satellites, lock onto the signals, and then begin computing navigational solutions. This software can work with a wide variety of receivers that connect typically via USB or Ethernet. GNSS-SDR offers a lot of customizing options for the user. In the configuration file, the user can change many different settings such as sample rate, frequency, max number of satellites, and how/where to output the data. These many customizable settings make GNSS-SDR a very useful tool for a wide variety of applications. For GBAS, GNSS-SDR can control an inexpensive receiver, output the raw data in real-time, and also output the

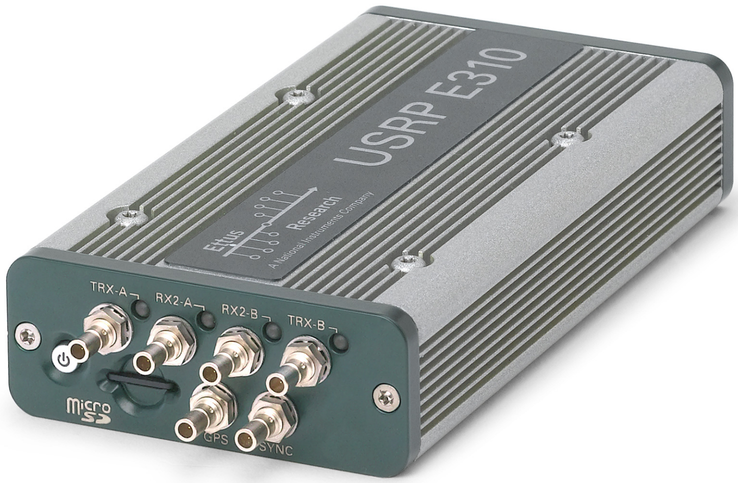


Figure 3.2: Ettus Research's USRP E310. Image from their website [3]

ephemeris data for each satellite, making it a viable alternative to a traditional GPS receiver.

GNSS-SDR, while able to run on Windows in some cases, is typically ran on a Linux machine. This requires the user to clone the open-source software, checkout the latest version using Git, build the software, and then install the program, in order to get the most up-to-date changes. Detailed steps for GNSS-SDR installation are shown in Appendix B. Once GNSS-SDR has been installed, the user must edit the configuration file in order to fit their system. This requires knowing which satellites to use, which frequency to listen on, what to do with the output data, and most importantly, what kind of SDR is being used and how to connect to it. The configuration file used for the GBAS tests in this thesis is shown in Appendix E.

The typical workflow of GNSS-SDR is as follows: the software connects to the SDR via whichever means are defined in the configuration file, and then it has the receiver open a port on the desired frequency. In this work, the frequency used is 1575.42 MHz, the GPS L1 band. At this point, the SDR, which is connected to a GPS antenna, begins listening to that frequency, and down converts the messages and outputs them at a lower frequency to GNSS-SDR. At this point, GNSS-SDR takes these down converted raw satellite messages and begins to sort them into different channels for each different satellite. Once in a specific channel, the software will begin to acquire the signals and eventually begin to lock onto the satellite. This requires several satellite messages and takes a lot of computing power. Since this is an intensive process, the configuration file can be configured to make only one channel acquire and begin tracking a satellite before the next channel begins. Thus, GNSS-SDR can be optimized to run on slower PCs like a Raspberry Pi.

After the GNSS-SDR has a lock on and is tracking more than 4 different satellites, each on a different channel, the software can start taking the navigational data from the satellites and use them to begin computing a position solution. Once a solution has been found, the software will begin outputting the positional data and sending time, location, and GPS satellite information in the format specified in the configuration file.

3.5 Initial Evaluation of GNSS-SNR

Once the GNSS-SDR software has been properly configured with an SDR that is connected to a working antenna, the software will begin working rapidly. Typically, depending on the configuration settings, computer processor speed, and current satellite constellation, the receiver can get a position solution in anywhere between 45 seconds and a few minutes. From experience, the position accuracy improves after about a minute, and the resulting position is typically very accurate within a few meters. A plot of the position of the antenna compared to the surveyed location was generated from a few minutes of data using the HackRF One, the USRP E310, and the GG12s, as seen in Figure 3.3. As can be seen in the graph, all of the receivers were within +/- 5m of the true antenna location. One thing to consider is that the GG12s have a very precise location amongst all three of them, while the GNSS-SDR results varied significantly more. The results of the GNSS-SDR receivers shown here are similar to the ones found in a paper written by some of the creators of GNSS-SDR [27]. Another paper discussing the performance of GNSS-SDR using low-cost receivers in a difficult environment, found that the GNSS-SDR software's customizability made it an extremely useful tool even though it was often limited by the internal clock of the SDR [28].

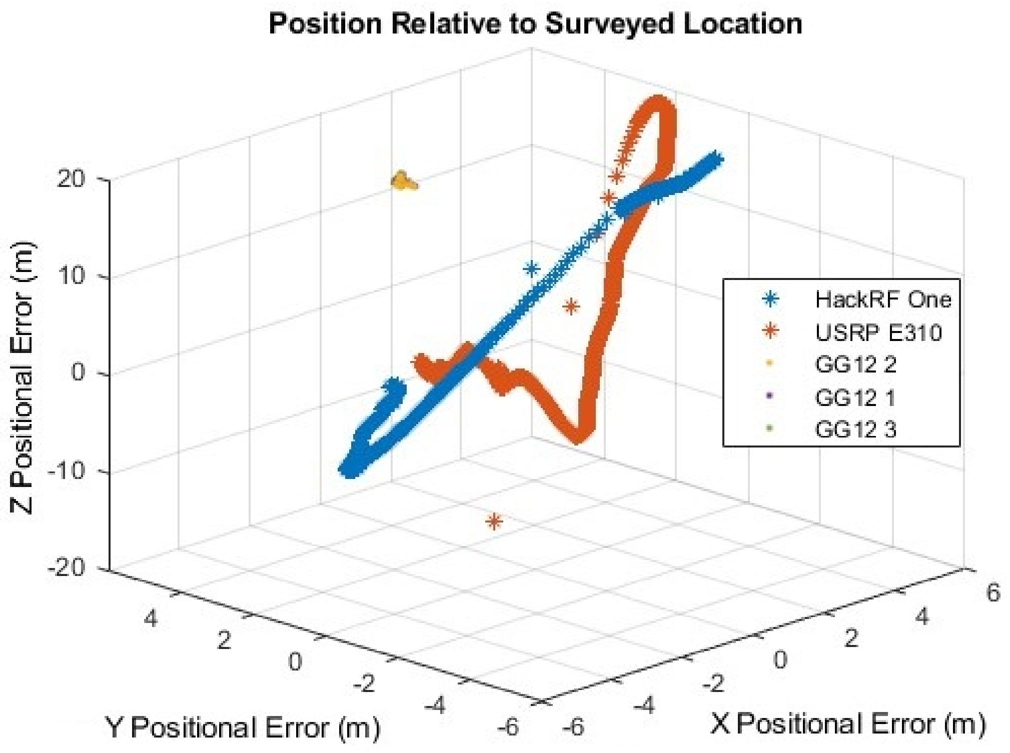


Figure 3.3: The East-North-Up (ENU) coordinates of the antenna location taken from different receivers at the same time. The HackRF One and the USRP E310 are very spread out while the three GG12s have very precise positions all stacked on top of each other at the upper left.

3.6 Receiving GBAS Data from GNSS-SDR

GNSS-SDR collects a vast amount of GPS data when running. Depending on how the configuration is setup, the software can output the data in several formats. Usually, the software outputs data to many different files. For this work, the chosen method for data output is over UDP, which is delivered over the IP protocol. This method was chosen because the desired raw data for the GBAS algorithm was already available for live output using this method. Other methods that output data live did not output the raw pseudorange and phase data that are required. This UDP streaming method can be achieved by connecting two PCs either by Ethernet, Wi-Fi, or through a virtual machine and editing the GNSS-SDR configuration file. This allows data to be transferred at large rates and in one direction. For the base station at OU, the base station computer and the reference station computers were connected using Ethernet cables and a network switch. Information on how the computers are networked can be found in Appendix E. When outputting through UDP, GNSS-SDR encodes the data using the Google Protocol Buffer (Protobuf) standard [29]. This is important because this information will be used to properly decode the data in LabVIEW.

The first step in receiving the proper data is to output the data from GNSS-SDR with the desired parameters. The GNSS-SDR configuration settings to enable UDP streaming of data are shown below.

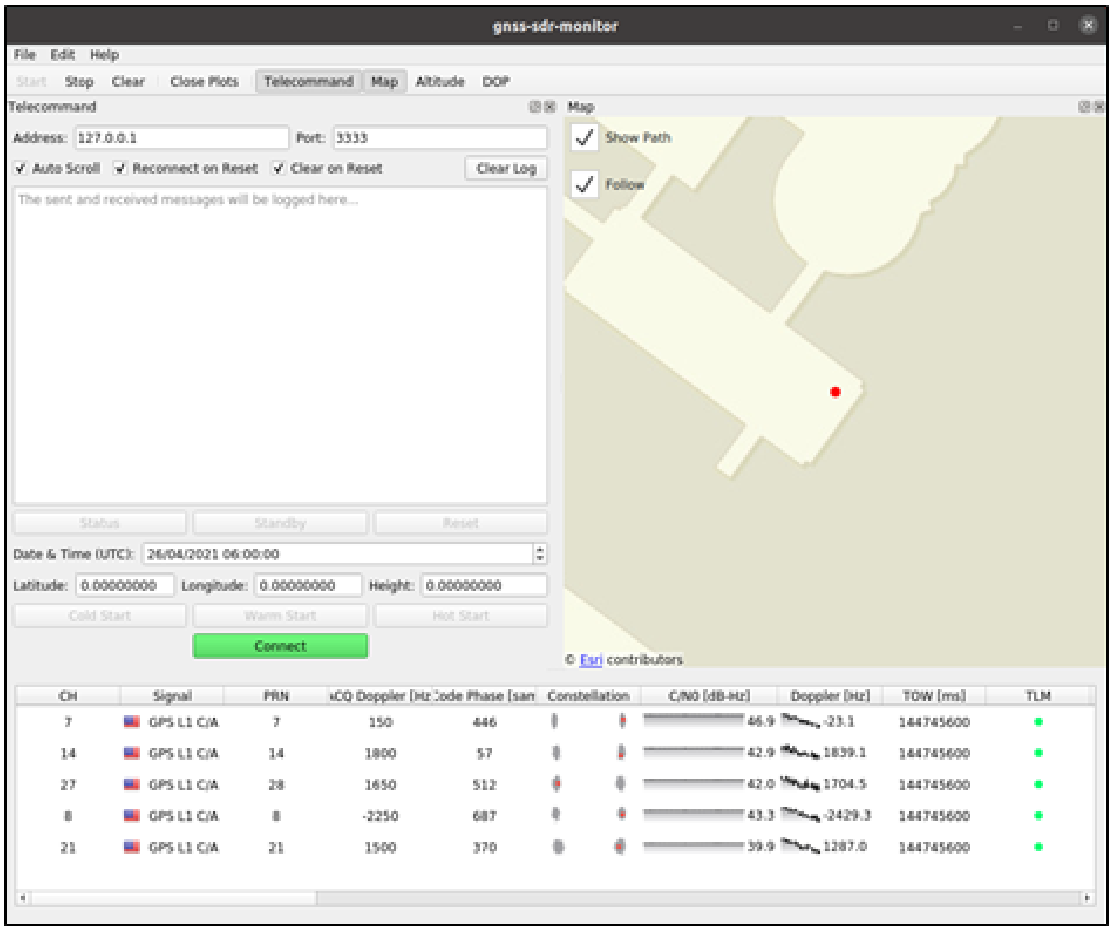
```
PVT.enable_monitor=true
PVT.enable_protobuf=true
PVT.monitor_client_addresses=10.10.10.1_127.0.0.1
PVT.monitor_udp_port=1112
PVT.enable_monitor_ephemeris=true
PVT.monitor_ephemeris_client_addresses=10.10.10.1_127.0.0.1
PVT.monitor_ephemeris_udp_port=1113
;######### MONITOR CONFIG ############
Monitor.enable_monitor=true
Monitor.enable_protobuf=true
Monitor.decimation_factor=1
Monitor.client_addresses=10.10.10.1_127.0.0.1
Monitor.udp_port=1111
```

These lines in the configuration file control whether there is data outputting over UDP, where it is sending the data, and what port it is sending the data over. When running multiple receivers, the ports will need to be changed to be unique so that each receiver's output can be differentiated from each other in LabVIEW. The 127.0.0.1 address just means that it is sending the data to itself, while the 10.10.10.1 is the IP address of the base station computer. It is important that the data is being sent to itself so that the GNSS-SDR-Monitor software, which is useful for debugging, will run properly, which is discussed in the next section.

3.7 Running the GNSS-SDR-Monitor Software

Now that the data is properly outputting data over UDP, the GNSS-SDR-Monitor software can now be used. The installation and execution instructions can be found on the project GitHub in [30]. After the program runs for a while, satellites should populate in the bottom section of the screen and then a position point should appear on the map as seen below in Figure 3.4. This can be done opening another terminal after the main software has started and running the following command:

"sudo /home/gbas/work/gnss-sdr-monitor/build/src/gnss-sdr-monitor".



Address: 127.0.0.1 Port: 3333

✓ Auto Scroll ✓ Reconnect on Reset ✓ Clear on Reset

The sent and received messages will be logged here...

Date & Time (UTC): 26/04/2021 06:00:00

Latitude: 0.00000000 Longitude: 0.00000000 Height: 0.00000000

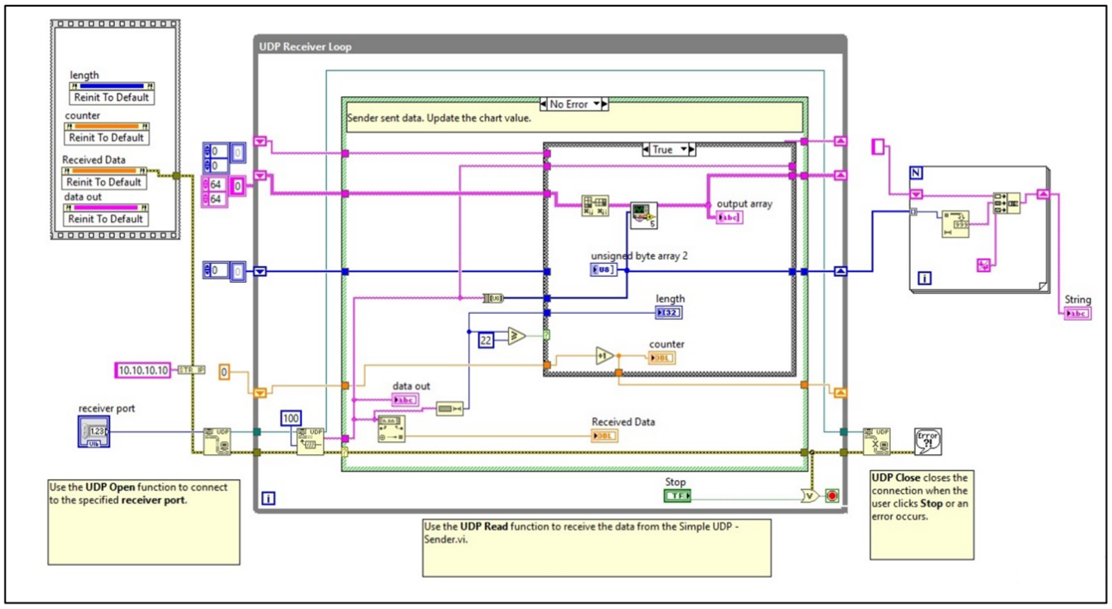
CH	Signal	PRN	ACQ Doppler [Hz	Code Phase [san	Constellation	C/N0 [dB-Hz]	Doppler [Hz]	TOW [ms]	TLM
7	GPS L1 C/A	7	150	446		46.9	-23.1	144745600	●
14	GPS L1 C/A	14	1800	57		42.9	1839.1	144745600	●
27	GPS L1 C/A	28	1650	512		42.0	1704.5	144745600	●
8	GPS L1 C/A	8	-2250	687		43.3	-2429.3	144745600	●
21	GPS L1 C/A	21	1500	370		39.9	1287.0	144745600	●

Figure 3.4: GNSS-SDR-Monitor showing the data from the "parking lot" antenna.

If the monitor software does not appear to work, double check that the configuration file is correct. Also, check that the ports in the monitor software itself are correct by editing the preferences.

3.8 Receiving GNSS-SDR Stream Data in LabVIEW

Since the data is sent over UDP from one PC, the other PC can receive it by listening to UDP messages over the specific port that it is being sent on. This can be done easily in LabVIEW by modifying the example "Simple UDP Receiver.vi" example as seen in Figure 3.5. Once the message is received, it is parsed using the string to byte array function. Now the received messaged is converted from an ASCII string, which is unreadable, to an array of 8-bit numbers which range from 0-255. These 8-bit numbers are also referred to as bytes. Once converted to bytes, the live data is much easier to work with.



Sender sent data. Update the chart value.

Use the **UDP Open** function to connect to the specified **receiver port**.

Use the **UDP Read** function to receive the data from the Simple UDP - Sender.vi.

UDP Close closes the connection when the user clicks **Stop** or an error occurs.

Figure 3.5: UDP Receiver which receives and properly handles the data within the message. This code is modified from the UDP Simple Receiver.vi example LabVIEW code.

After the string message is converted to a byte array, it needs to be decoded. The first step to decoding the data is to first understand how it was encoded. Encoding is done in the GNSS-SDR software using a relatively new method called Google Protocol Buffers or "protobufs" for short. Using this protobuf method allows for extreme lossless compression of the data. This is ideal for software that needs to output large amounts of data quickly. Since there is very little LabVIEW support for Google protobufs, the data needed to be decoded manually. Eventually, two designs were created to decode the data using LabVIEW. One method required learning another programming language and integrating it into LabVIEW while the other was done using more manual techniques. In the end, the first method was extremely slow, not able to keep up with the incoming data, while the manual method worked incredibly well, easily able to keep up with the incoming data. These two methods are explained in more detail in the following two sections.

3.8.1 Decoding Method 1: Using Python in LabVIEW

The first method attempted was to create a Python script that could properly decode the message. This required saving an example protobuf message from the UDP Receiver LabVIEW code. Once a sample message had been acquired, the Python script could be executed to decode the message into its proper data values.

The first stumbling block reached on this method was the actual creation of the protobuf decoding script. This required watching several videos, reading lots of documentation, and using trial and error. According to the Google protobuf documentation website [29], one should be able to easily compile a Python (or any of the other supported languages) script using their downloadable software.

After compiling the Python script, it was successfully tested with a sample protobuf message. The script was able to properly decode and parse the data values from the encoded protobuf message. Now, the next step was to run this script using the test data in LabVIEW. Python scripts can be ran in two different ways in LabVIEW. One way is to run a command in the command prompt, but unfortunately it never properly work. The second way is to use a Python node in LabVIEW, similar to the MathScript node in LabVIEW. This was very user friendly and was very simple to implement in LabVIEW. After testing some extremely basic scripts, it was then able to be successfully tested using the sample protobuf message from before. The sample decoding LabVIEW block diagram that worked for this is shown below in Figure 3.6. Now that it worked in LabVIEW, the next goal was to test it again using live data.

When testing the python decoder on live data, it appeared to slow the computer down. In order to narrow down the issue, the obvious step was to time how long it took a message to get decoded. The average result after several minutes of decoding was that it took this python script 0.25 seconds on average to decode a single message. This was extremely slow, so some code changes were implemented to attempt to optimize the code. The only thing that really helped was to not open and close the Python session each time a message was decoded. This change did indeed speed up the code, however the average message decode time went down to about 0.20 seconds per message. Now this might seem fast, however, it is important to realize that there were 32 data channels, and each had its own message being sent every 20ms. Now, even though maybe just 10 of these channels might be sending useful data, that still leaves 500 messages to decode each second. So, the GBAS PC is receiving 500 useful messages each second, but was only able to decode 5 of them every second using this method. This method is much too slow to be useful. It was also noticed that the code would fail occasionally. After much troubleshooting, it was discovered that if the satellite PRN was greater than 9 (double-digits), then the decoder would fail. Thus, although this may be a preferred method since it used the Google protobuf decoder compiler, using the compiled Python decoding script was not useful for this work, especially in LabVIEW.

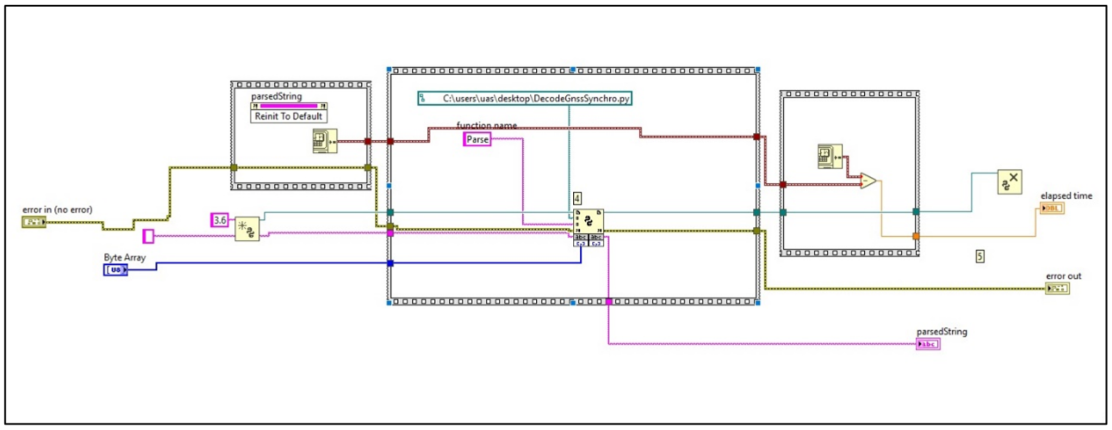


Figure 3.6: Python Protobuf Parser/Decoder in LabVIEW.

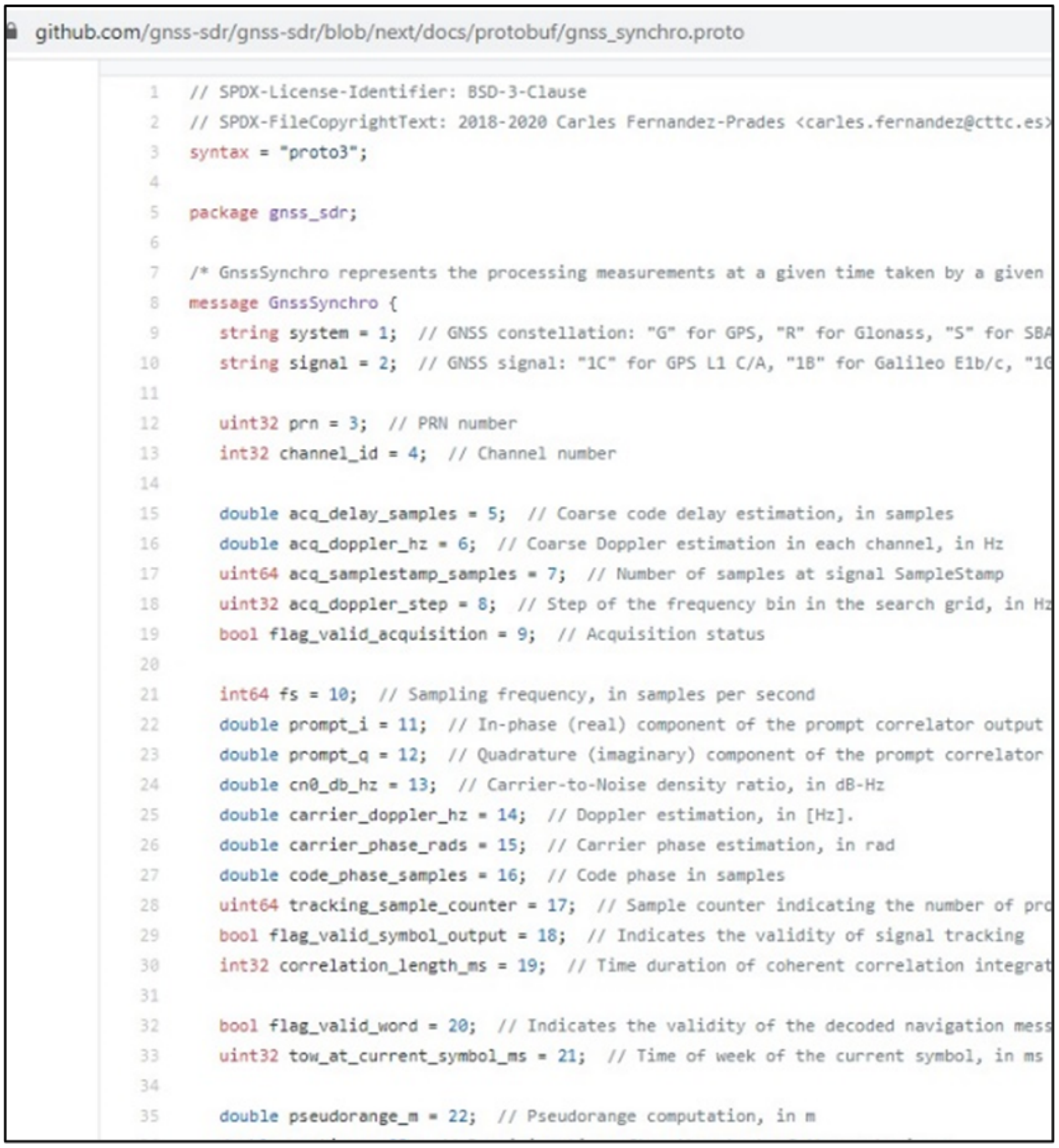
3.8.2 Decoding Method 2: Manual Decoding in LabVIEW

Since the Python script method was unusable, it was decided to decode the protobuf messages manually in LabVIEW. To begin this, an analysis of the array of bytes changing while the UDP receiver was running live was conducted. Although most of the numbers were changing too fast to comprehend anything, some of the numbers remained constant. Further analysis was performed that used those numbers to establish a pattern. Each of the “constant” bytes were increasing by about 8 as it went down the array of bytes. This was curious, leading to more research into the exact encoding method in the Google Protobuf documentation.

In simplifed terms, the protobuf messages are encoded in the following manner. A ".proto" file is required to properly encode a set of values. In this .proto file, one will find a list of key-pair values. Each key is defined with a name and a data type. This key-value pair is important, because in the encoded message, the first byte is usually the key value, which determines what the following data is representing. For a double, the next 8 bytes would be the exact value of the parameter. So, for the very first data point, a 1 is expected, since that is the first one to be sent. However, it is found that the first leading byte is an 8. This is because the number is left-shifted by 3. This means, the binary digit 0000 0001 (1) has every value shifted left by 3, meaning it gets encoded to 0000 1000 (8). This makes the key for the second data value 0001 0000 (16), and the third would be 0001 1000 (24). Thus, to find a specific parameter, one would find the name on the .proto file, look at its mapped value, and then multiply it by 8. Then look for that value in the encoded message. Now depending on the data type, add 1 or 2 after multiplying by 8. After a key value is encoded, the actual value of that parameter is encoded right after the key value. The length of the rest of it is dependent on the data type of the parameter. After the key is larger than 16, the key value is encoded in the same manner, but the next byte is a 1, then the encoded parameter value follows.

For example, in the gnss_synchro.proto file, the double-precision floating point "pseudorange_m" key is mapped to the value of 22. So, in the encoded message, one is looking for $8 \times 22 + 1 = 177$ (add 1 since the value is a double). Once 177 is found, the next byte should be a 1, and then the next 8 bytes are the actual value of the pseudorange in meters.

With a proper understanding of how the encoding worked, the message could be decoded. This was as simple as doing a search for a particular value, taking



github.com/gnss-sdr/gnss-sdr/blob/next/docs/protobuf/gnss_synchro.proto

```
// SPDX-License-Identifier: BSD-3-Clause
// SPDX-FileCopyrightText: 2018-2020 Carles Fernandez-Prades <carles.fernandez@cttc.es>
syntax = "proto3";

package gnss_sdr;

/* GnssSynchro represents the processing measurements at a given time taken by a given
message GnssSynchro {
   string system = 1;  // GNSS constellation: "G" for GPS, "R" for Glonass, "S" for SBA
   string signal = 2;  // GNSS signal: "1C" for GPS L1 C/A, "1B" for Galileo E1b/c, "1G

   uint32 prn = 3;  // PRN number
   int32 channel_id = 4;  // Channel number

   double acq_delay_samples = 5;  // Coarse code delay estimation, in samples
   double acq_doppler_hz = 6;  // Coarse Doppler estimation in each channel, in Hz
   uint64 acq_samplestamp_samples = 7;  // Number of samples at signal SampleStamp
   uint32 acq_doppler_step = 8;  // Step of the frequency bin in the search grid, in Hz
   bool flag_valid_acquisition = 9;  // Acquisition status

   int64 fs = 10;  // Sampling frequency, in samples per second
   double prompt_i = 11;  // In-phase (real) component of the prompt correlator output
   double prompt_q = 12;  // Quadrature (imaginary) component of the prompt correlator
   double cn0_db_hz = 13;  // Carrier-to-Noise density ratio, in dB-Hz
   double carrier_doppler_hz = 14;  // Doppler estimation, in [Hz].
   double carrier_phase_rads = 15;  // Carrier phase estimation, in rad
   double code_phase_samples = 16;  // Code phase in samples
   uint64 tracking_sample_counter = 17;  // Sample counter indicating the number of pro
   bool flag_valid_symbol_output = 18;  // Indicates the validity of signal tracking
   int32 correlation_length_ms = 19;  // Time duration of coherent correlation integrat

   bool flag_valid_word = 20;  // Indicates the validity of the decoded navigation mess
   uint32 tow_at_current_symbol_ms = 21;  // Time of week of the current symbol, in ms

   double pseudorange_m = 22;  // Pseudorange computation, in m
```

Figure 3.7: GNSS_Synchro.proto file.

the next 8 values, and then decoding it. However, there was a problem with this method. If there happened to be the exact value in multiple places in the encoded message, it would completely ruin the message. Thus, like in the above example, if looking for the value 177, it may occur several times in the message. The code needs to know that the 177 found is the exact one referring to the key value 22. After analyzing the problem, a method was created with the idea of checking for nearby known values to verify it is the right data point. For example, with 177, the code will check that the next byte value is 1 and the 9th next byte value is 185. This makes the code extremely robust as the probability of this set of numbers occurring in this exact format anywhere else in the message is extremely unlikely. Once the correct key is found, one can get the decoded value by taking the next 8 bytes, reversing their order (from little endian to big endian) and then flattening them to a float value using the flatten function in LabVIEW. Some data types, like the PRN (a uint32), did not require this extra step of flattening.

Overall, this manual decoding method is extremely robust and fast. It is able to easily keep up with the 500+ messages per second. It is also able easily parse the data, even if the PRN number is greater than 9, which couldn't be done with the first method.

To receive the data in LabVIEW, ensure LabVIEW is not blocked by the Windows Firewall. To do this, requires using admin privileges, opening the Firewall, and adding LabVIEW to the allowed programs list. Next, run the programs created in this work in LabVIEW. When running the program, make sure to have the correct port selected. If it is not, the program will not function properly. If changing ports, make sure to restart the program by selecting the "Stop" button on the Front Panel. Do not use the abort button, as this is bad LabVIEW practice and can cause an assortment of other issues.

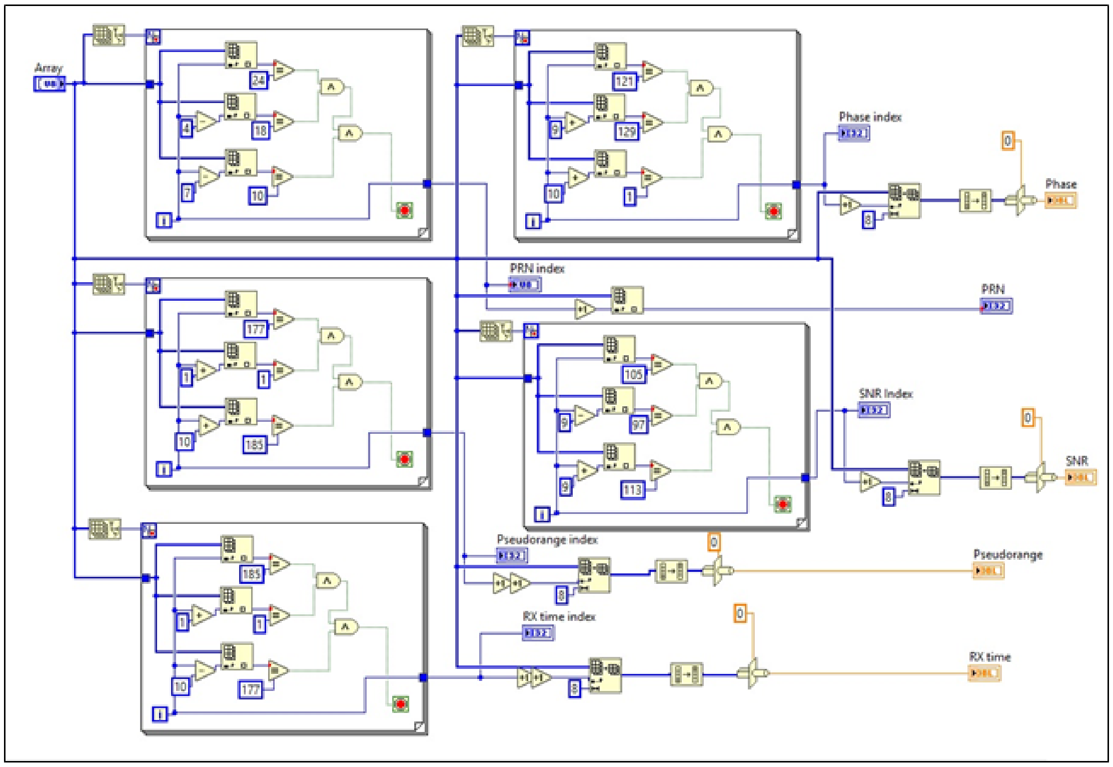


Figure 3.8: GNSS Synchro Protobuf Converter. This LabVIEW code decodes and parses the PRN, SNR, pseudorange, phase, and receive (rx) time of each satellite. As can be seen in each for loop, there are two extra checks to ensure the index of the value in the array is the proper one.

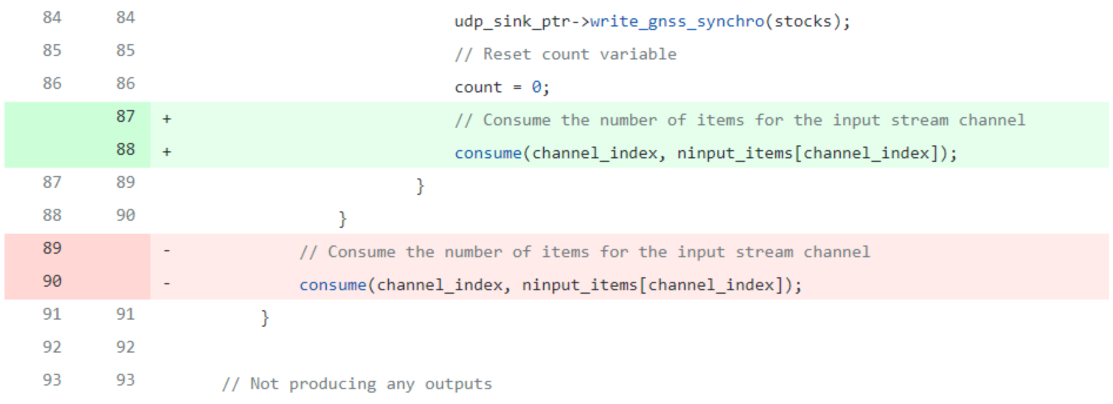
3.9 Synchronization of GNSS-SDR UDP Output

Since the original GBAS program from [4] used the Thales GG12 GPS receivers (specs found in [31]), which output data every half second, the GNSS-SDR software must also output at that rate. Luckily, the GNSS-SDR configuration file comes with a simple parameter called "Monitor.decimation_factor" which defaults to 1 if not specified. Since the software outputs every 20 ms, a decimation factor of 25 would output 2 messages per second, the same rate as the GG12s. Unfortunately, when changing the decimation factor to anything but 1, the program stopped sending message reliably. With a decimation factor of 2, the messages would arrive randomly about every 2 minutes apart. It appeared the open-source GNSS-SDR software had a bug and thus it was pursued further.

With limited background knowledge in C programming, perusing the source code can be a challenge. However, after a bit of searching, it was determined the error had to be caused in the file "src/core/monitor/gnss_synchro_monitor.cc" which was the file determining when to stream the data. After studying the code it was determined that a particular line which contains a consume command to be the cause of the issue. Instead of being after the loop where it is ran every time, it needs to only be run when the decimation value is reached inside the if statement. Testing the software changes showed that this fixed it. The exact code changes can be seen in Figure 3.9.

The following changes were submitted to the GNSS-SDR GitHub project. The team accepted them and added them to the newest version of the software.

After running the software with the decimation factor set to 25, it was immediately noticed that the receive (RX) times between multiple receivers were not the same. For example, the HackRF One attached to the parking lot antenna (C in Figure C.1) would output data whose "rxtime" parameter ended with 0.26

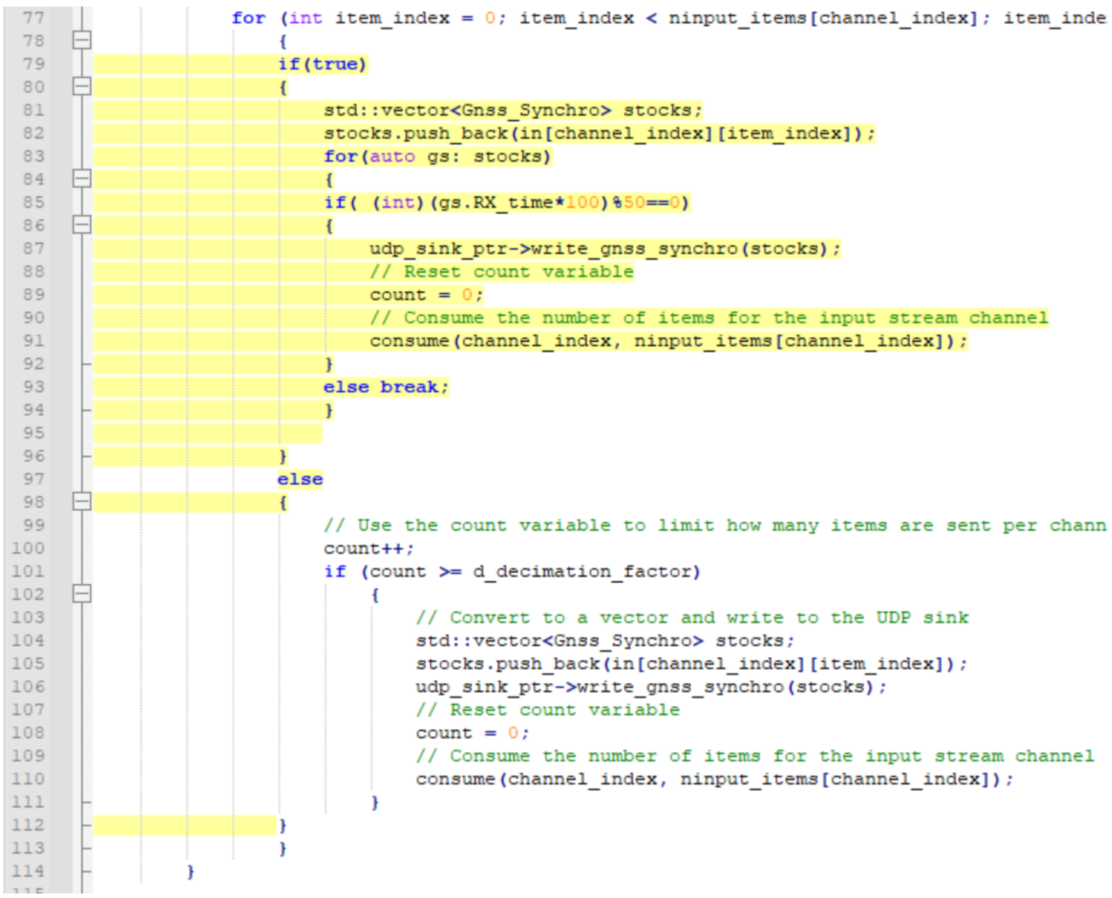


```
                        udp_sink_ptr->write_gnss_synchro(stocks);
                        // Reset count variable
                        count = 0;
+                       // Consume the number of items for the input stream channel
+                       consume(channel_index, ninput_items[channel_index]);
                    }
                }
-               // Consume the number of items for the input stream channel
-               consume(channel_index, ninput_items[channel_index]);
            }

        // Not producing any outputs
```

Figure 3.9: Changes made in the gnss_synchro_monitor.cc file. Green lines show added lines where red lines show removed lines.

and 0.76 every time a message was sent while the another HackRF One attached to the same antenna was outputting messages with an “rxtime” ending with 0.47 and 0.97 each time. After studying the GG12s, it was noticed that each one output with “rxtime” ending in 0.00 and 0.50 each message. This was a problem as one of the goals of this thesis is to run the GG12s and SDRs simultaneously. Using different receive time values would not only break the algorithms in Davis’s original code in [4], but would be meaningless to the GG12s.

At this point, it was back to the GNSS-SDR source code looking for a way to force the data being output to come in at intervals of 0.00 and 0.50 seconds only. Luckily, this was a rather simple fix in the same file that the above change was made. Instead of having the code check for just the decimation factor, one could simply have it check to see if the receive time ended in .00 or .50. So, as a proof of concept and not by using proper coding standards, the if statement checking the decimation factor was simply subverted using an if(true) statement before it as seen in Figure 3.10. The new code includes multiplying the current RX time by 100, to allow for an easy application of modulo 50. Checking to see if this result equals zero implies the RX time ended with .00 or .50. Thus, this code synchronizes the outputs of the data to be sent every half-second at 0 and 500 milliseconds, matching the GG12s.



```
            for (int item_index = 0; item_index < ninput_items[channel_index]; item_inde
                {
                if(true)
                {
                    std::vector<Gnss_Synchro> stocks;
                    stocks.push_back(in[channel_index][item_index]);
                    for(auto gs: stocks)
                    {
                    if( (int)(gs.RX_time*100)%50==0)
                    {
                        udp_sink_ptr->write_gnss_synchro(stocks);
                        // Reset count variable
                        count = 0;
                        // Consume the number of items for the input stream channel
                        consume(channel_index, ninput_items[channel_index]);
                    }
                    else break;
                    }

                }
                else
                {
                    // Use the count variable to limit how many items are sent per chann
                    count++;
                    if (count >= d_decimation_factor)
                        {
                            // Convert to a vector and write to the UDP sink
                            std::vector<Gnss_Synchro> stocks;
                            stocks.push_back(in[channel_index][item_index]);
                            udp_sink_ptr->write_gnss_synchro(stocks);
                            // Reset count variable
                            count = 0;
                            // Consume the number of items for the input stream channel
                            consume(channel_index, ninput_items[channel_index]);
                        }
                }
                }
        }
```

Figure 3.10: The highlighted section shows the new code added.

3.10 Integrating the GNSS-SDR Receiver into the CL-GBAS Base Station Program

Currently, the CL-GBAS program used in [4] is relatively dated and was only designed to work with the Thales GG12s. Thus, getting a comparison between the SDRs and the GG12s requires the SDRs to be integrated into the program. For our comparisons, the almanac data as well as the ephemeris data is acquired by the GG12s for both the SDRs and GG12s. Only the raw data from the SDRs, which contains the pseudorange and phase for each satellite, is used. This is done because the alamanac and ephemeris data are received directly from the satellites and should be the exact same for all receivers. Also, it should be noted that the UDP output of the almanac data is not currently supported in GNSS-SDR.

The first step to get the old program to work with GNSS-SDR receivers is to create a software driver that can take the GNSS-SDR outputs and convert the desired data into the same format as the GG12s. The code to perform this task is shown in Figures 3.11 and 3.12. The framework for this GNSS-SDR driver was originally designed and described in [32].

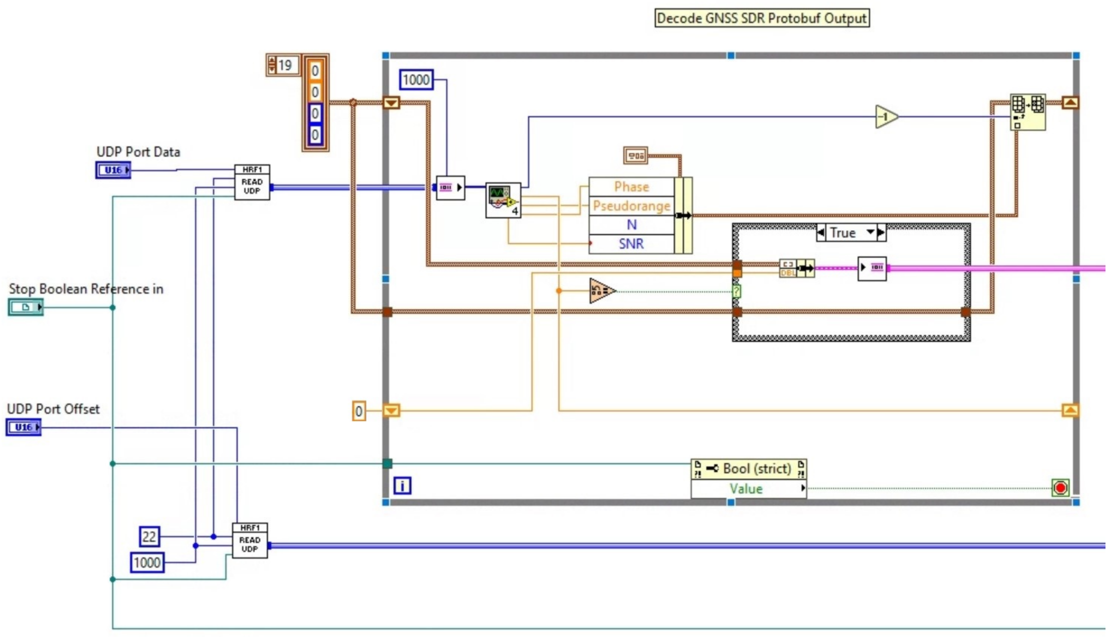


Figure 3.11: This code shows the new GNSS SDR Driver Code. Here the data is received and placed into a map of values.

Once the data has been properly formatted it is sent into the main CL-GBAS program shown in Figure 3.13. After attempting to run the program, it was found

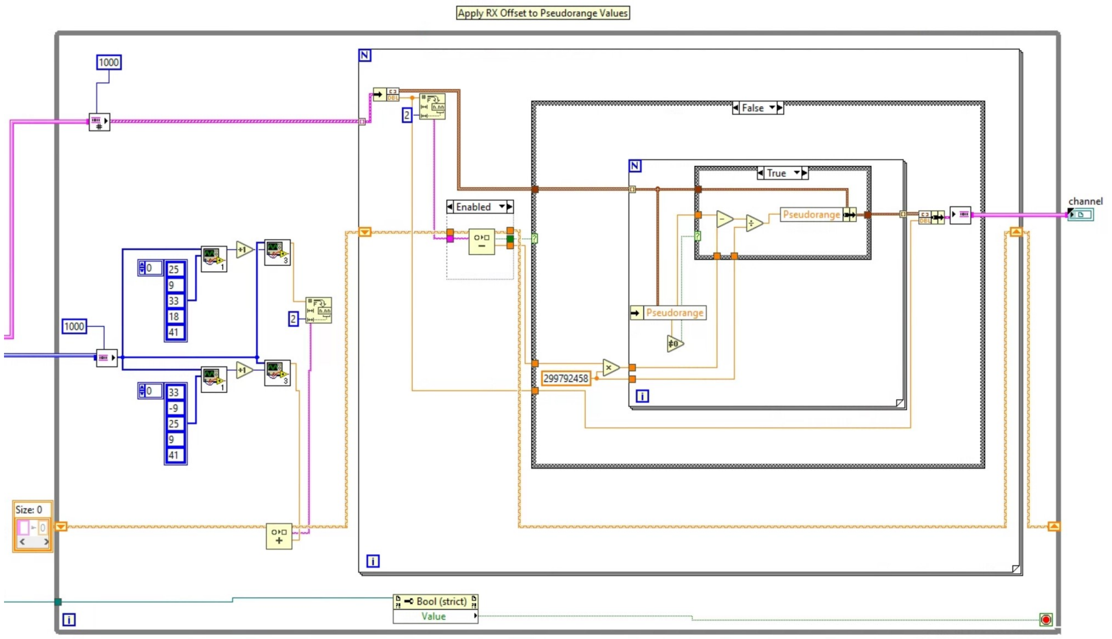


Figure 3.12: This section of the GNSS SDR Driver code shows where the pseudorange is corrected using the proper clock bias.

that not all of the GNSS-SDR data would arrive at the same time. This required more code to be written to ensure strict synchronization between the individual GNSS-SDR receivers. The code created for this task is shown in Figures 3.14 and 3.15.

Finally, after proper formatting and synchronization of the GNSS-SDR receivers, the program began to work and calculate vertical error results. It was noted that the results were not consistent and would sporadically jump. However, it always appeared that the number of satellites used was typically a direct cause of the poor results. Thus, the vertical error was logged using the data logger shown in Figure 3.16. This allowed for further post-processing analysis of the acquired data. After logging the data for approximately 45 minutes, the data was analyzed in MATLAB. It was found that in the 45 minute long test, only 758 out of 2,733 samples had more than 4 valid satellites. Thus, only approximately 28%

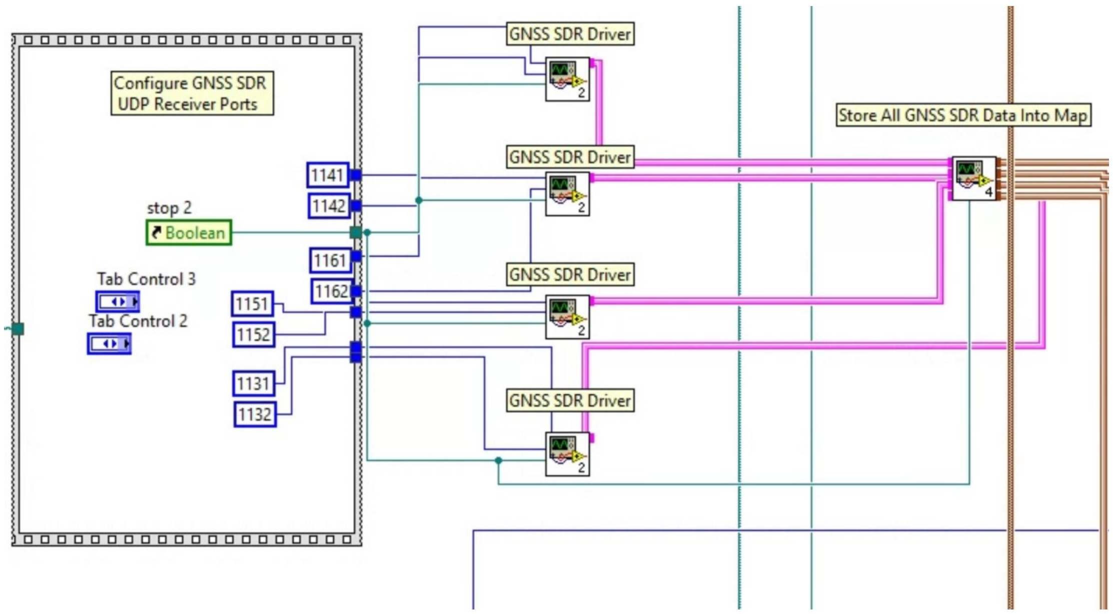


Figure 3.13: This section of code shows where the GNSS-SDR Drivers are configured and how they output into the main GBAS Program developed in [4].

of the time, or 12 minutes, were the results valid. When the results were valid, the vertical error (VE) shown in Figure 3.17 has very good results. Had the B-values been better and more satellites were being used in the solution, the vertical error would likely be much smaller. The code used to generate this plot is found in Appendix K.

Upon further inspection of the HackRF One values going through the modified CL-GBAS program, it was noticed that a lot of the values were rejected, resulting in a lack of valid satellites being used. This happened because the B-Values for the particular satellites were too large. The B-Values represent the errors that cannot be corrected in each receiver. These are values that are not correlated between receivers and can be caused by things like multipath, receiver noise issues, and hardware failure. They are the difference between broadcasted pseudorange corrections and the corrections obtained without the receiver in question [33].

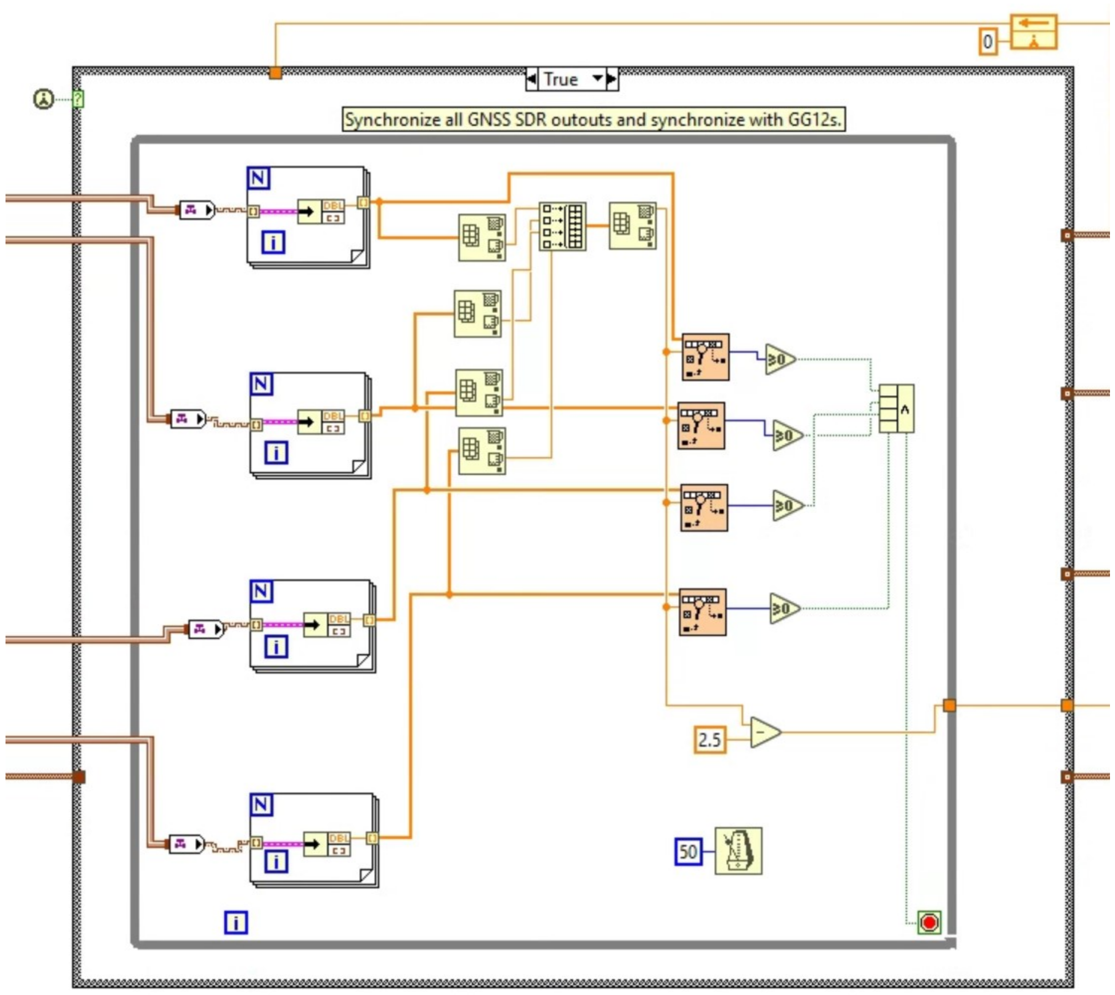

True
Synchronize all GNSS SDR outouts and synchronize with GG12s.
0
2.5
50

Figure 3.14: This section shows how the GNSS-SDR outputs need to be synchronized with each other to be sent to the GBAS program.

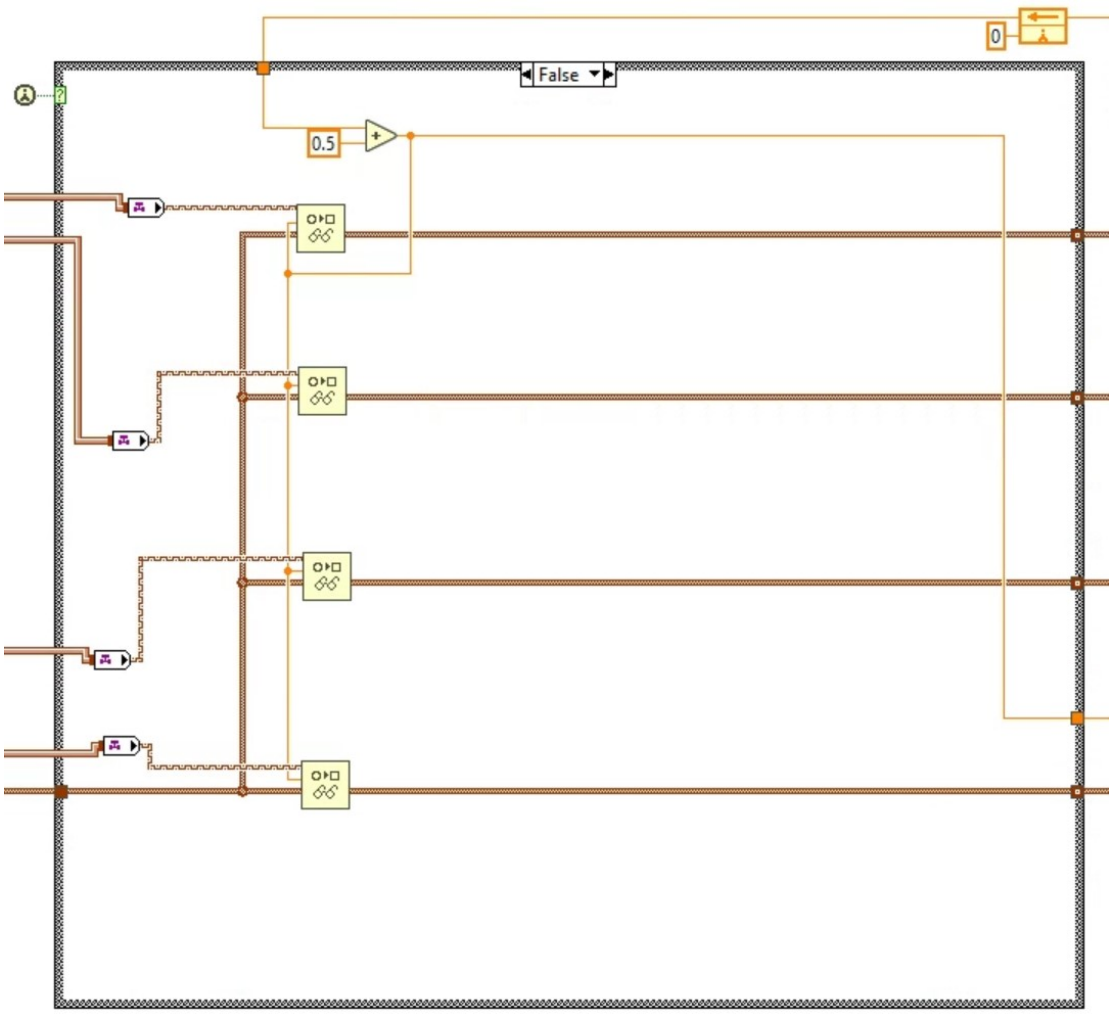


Figure 3.15: This code section again shows how the GNSS-SDR outputs need to be synchronized with each other to be sent to the CL-GBAS program. This time, it shows what happens every other time this case statement is ran.

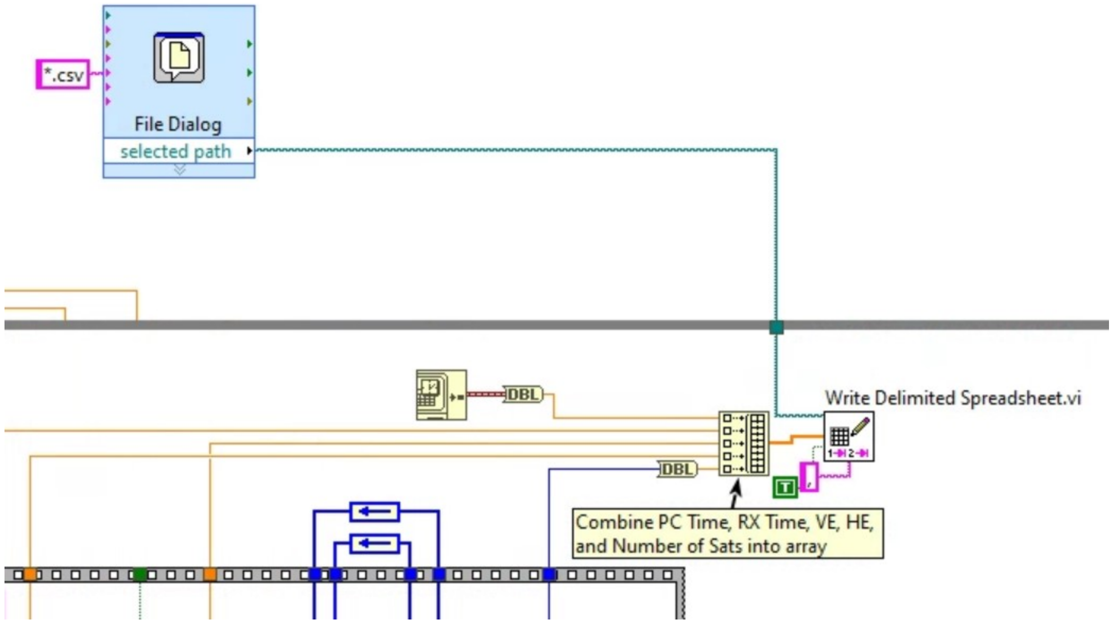

*.csv
File Dialog
selected path
DBL
Write Delimited Spreadsheet.vi
DBL
T
Combine PC Time, RX Time, VE, HE, and Number of Sats into array

Figure 3.16: This plot shows the non-rejected vertical error experimental data with a best fit curve overlaid on top of it. The standard deviation and mean values are displayed as well.

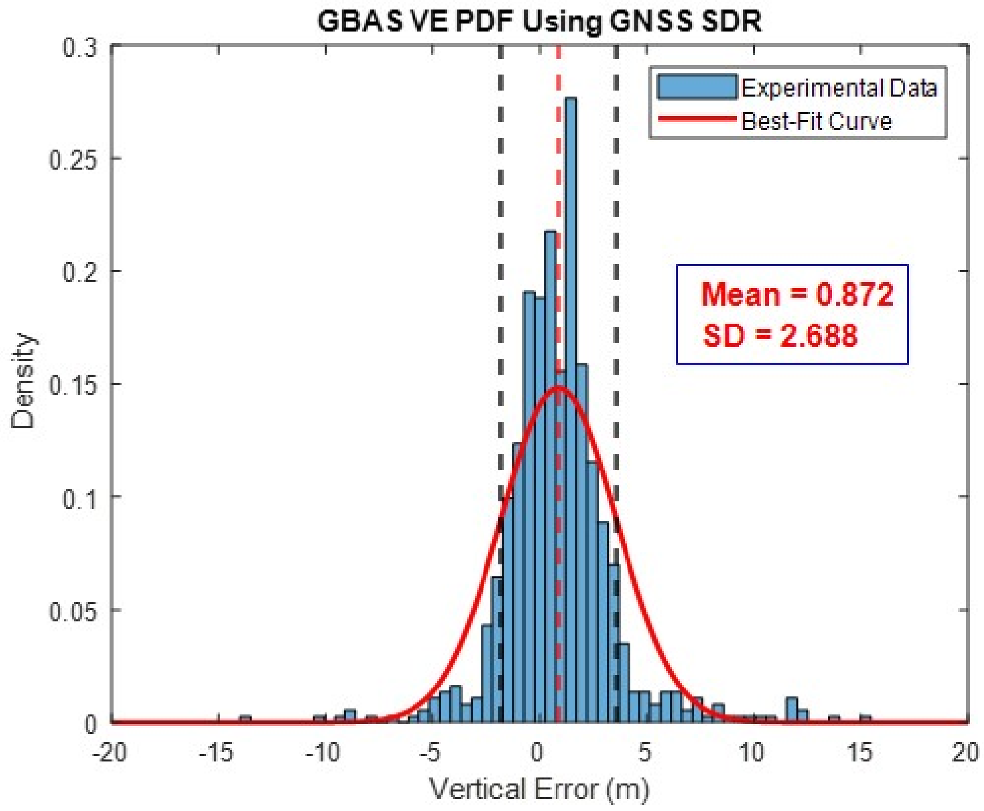


Figure 3.17: This plot shows the probability distribution function of the non-rejected vertical error experimental data with a best fit curve overlaid on top of it. The standard deviation and mean values are displayed as well.

The B-Values in the CL-GBAS are calculated as follows as described in [4]. The initial pseudorange correction per satellite is taken as the average of the carrier smoothed and receiver clock adjusted pseudorange corrections from each of the reference stations. The B-Value for a given reference station is calculated by subtracting this initial pseudorange correction by the average carrier smoothed and receiver clock adjusted pseudorange corrections of the reference stations without including the reference station in question. If this B-Value is higher than the set B-Value limit, then the non-correctable errors from this particular reference station are too large and the corrections for this satellite and this reference station are set to 0. If there are ever any satellites with less than three non-zero carrier smoothed and receiver clock adjusted pseudorange corrections, then that satellite cannot be used in the final solution. Since there were only three reference stations being used in this particular test, when any one of the reference stations had too large of a B-Value, it would result in a correction of 0 and thus the entire satellite could not be used in the final solution. Thus, it would have been more beneficial if four reference stations were used. These large B-Values are likely caused by sporadic pseudorange values which are further investigated in the following chapter.

Chapter 4

Evaluating Pseudorange from GNSS-SDR

4.1 Concept and Algorithm

The distance from the satellite to the antenna which is calculated by the receiver is referred to as the pseudorange. This is called the pseudorange because it is not an exact measurement of the distance, but actually the distance plus the many sources of errors in the measurement. When the satellite sends a navigation message, it includes the exact time which the message was transmitted. The receiver also keeps track of the time at which it received that message. The difference of this time is the time it took for the message to travel from the satellite to the receiver. Using a predetermined value of the speed of light through the atmosphere, one can multiply it by the travel time of the message to get the distance from the satellite to the receiver.

The pseudorange calculation on GNSS-SDR is interesting as GNSS-SDR calculates them all relative to a specific satellite with the most recent TOW (time of week) [34].

4.2 Pseudorange Comparison of SDRs and GG12s

4.2.1 Initial Tests

One of the first tests to verify if the HackRF Ones are suitable for GBAS purposes was to test the pseudorange outputs and compare it to the GG12s' outputs, simultaneously using the same antenna. To do this, one GG12 and three HackRF Ones using GNSS-SDR were connected to the same antenna using powered GPS splitters. The data coming from each of the receivers for over 24 hours was logged, using two LabVIEW logging programs. This gave a lot of sampling points for every GPS satellite for all four receivers. The logged data, which was originally already programmed, output the data in a particular format. This format required the logged data to be ran through another program which would then weave the data together, eventually applying the clock offsets, and outputting the data in a single master spreadsheet, who had 256 columns of data. One column for each of the 32 possible satellites, per eight possible receivers in the system (only seven were used). This additional program, written in LabVIEW, is shown in Figures 4.1, 4.2, and 4.3. The resulting file output of the combining files program is shown below in Figure 4.4.

The pseudorange values were then graphed them with respect to time. The initial results of a single satellite are seen in Figure 4.5. The MATLAB code used to generate these plots is included in the appendix.

As can be seen, the pseudoranges from the HackRF Ones are not close to the level of the certified GPS receiver. Some are over 3000 kilometers distance greater. However, they seem to have very similar shapes which seems to hint that there is a systematic issue going on. After scanning the algorithms for a while, it was seen that the code outputs a variable called "user_clk_offset" in a different

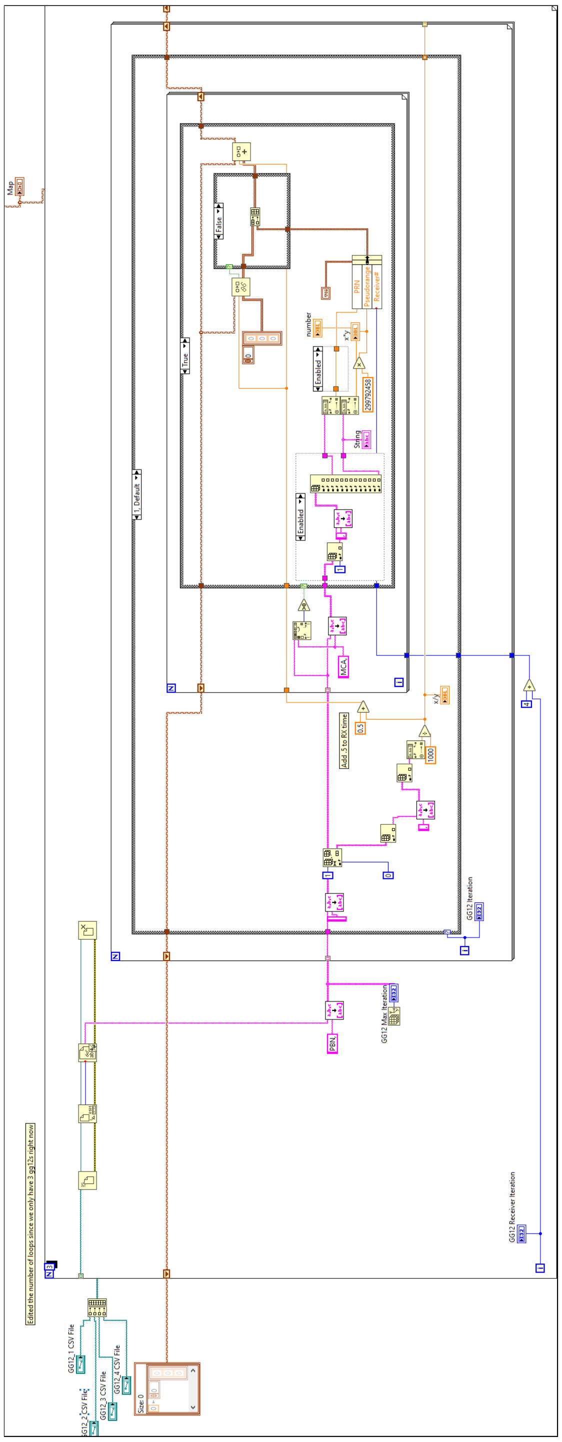


Figure 4.1: Examples of data logged from the GG12s. It then outputs the data into a properly formatted map to synchronize the GNSS-SDR receivers.

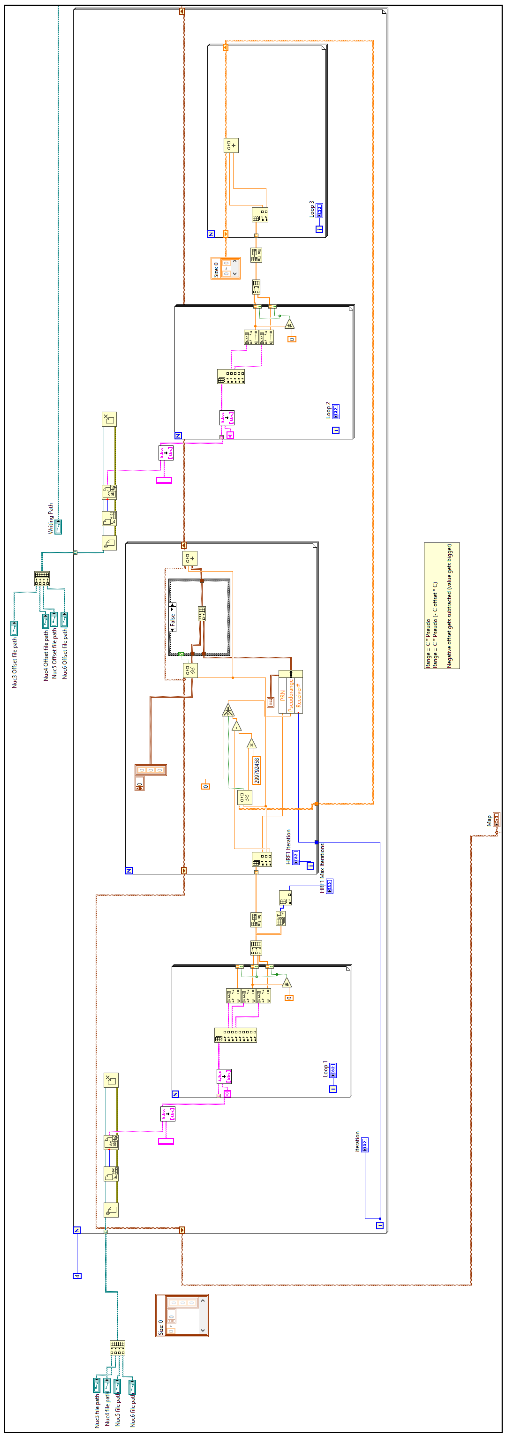


Figure 4.2: Application of the receiver clock offset from a separate file log file for the GNSS-SDR to correct the pseudorange values.

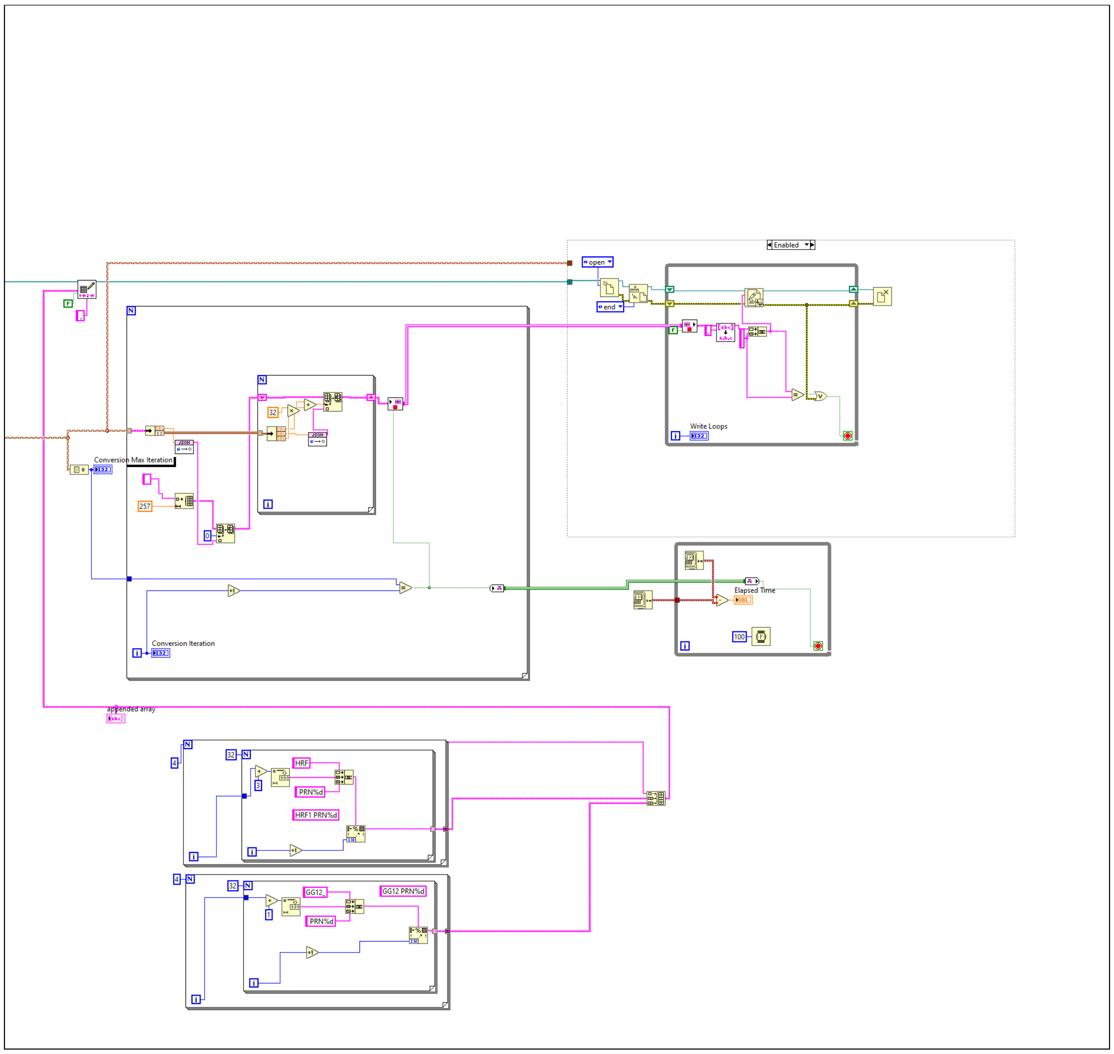
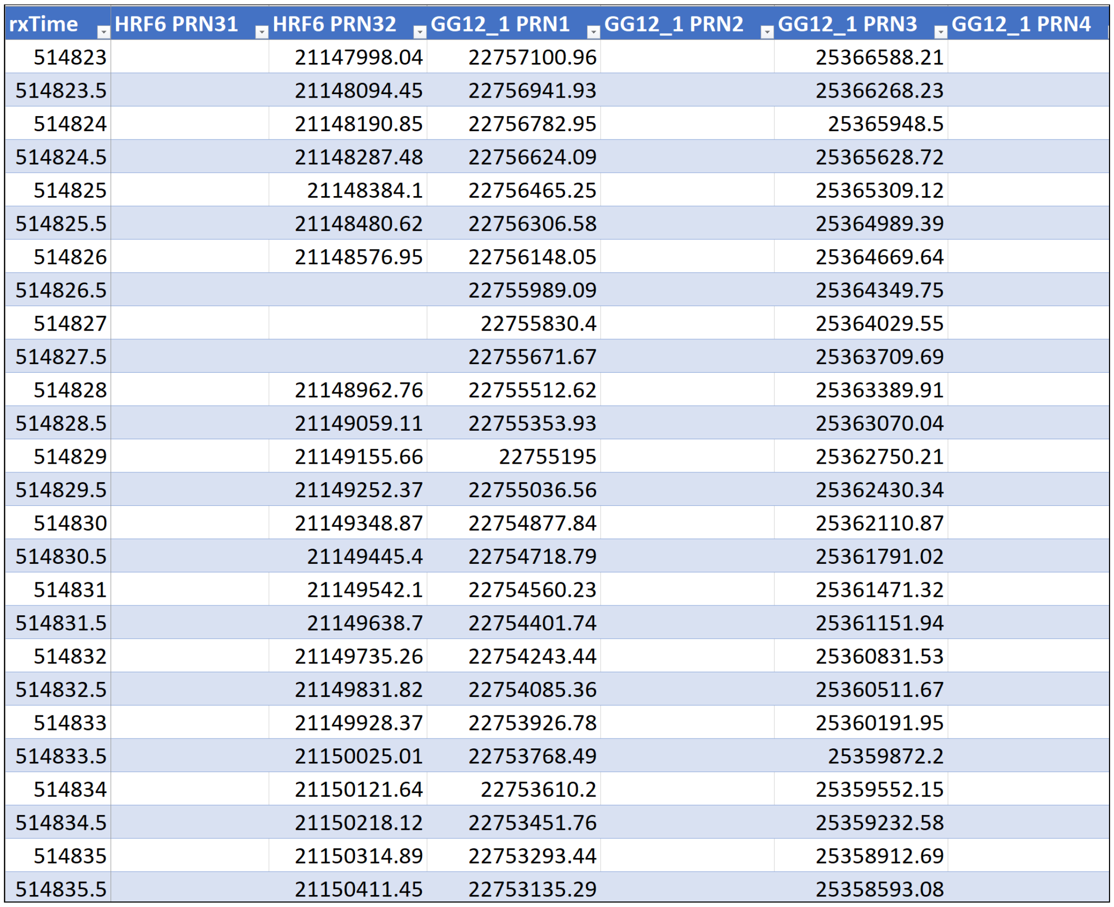


Figure 4.3: Output map from the logged GG12s is synchronized with the GNSS-SDR receiver logged data. This allows the data to be precisely lined up based on the data's receive time value across all of the different receivers.



rxTime	HRF6 PRN31	HRF6 PRN32	GG12_1 PRN1	GG12_1 PRN2	GG12_1 PRN3	GG12_1 PRN4
514823		21147998.04	22757100.96		25366588.21	
514823.5		21148094.45	22756941.93		25366268.23	
514824		21148190.85	22756782.95		25365948.5	
514824.5		21148287.48	22756624.09		25365628.72	
514825		21148384.1	22756465.25		25365309.12	
514825.5		21148480.62	22756306.58		25364989.39	
514826		21148576.95	22756148.05		25364669.64	
514826.5			22755989.09		25364349.75	
514827			22755830.4		25364029.55	
514827.5			22755671.67		25363709.69	
514828		21148962.76	22755512.62		25363389.91	
514828.5		21149059.11	22755353.93		25363070.04	
514829		21149155.66	22755195		25362750.21	
514829.5		21149252.37	22755036.56		25362430.34	
514830		21149348.87	22754877.84		25362110.87	
514830.5		21149445.4	22754718.79		25361791.02	
514831		21149542.1	22754560.23		25361471.32	
514831.5		21149638.7	22754401.74		25361151.94	
514832		21149735.26	22754243.44		25360831.53	
514832.5		21149831.82	22754085.36		25360511.67	
514833		21149928.37	22753926.78		25360191.95	
514833.5		21150025.01	22753768.49		25359872.2	
514834		21150121.64	22753610.2		25359552.15	
514834.5		21150218.12	22753451.76		25359232.58	
514835		21150314.89	22753293.44		25358912.69	
514835.5		21150411.45	22753135.29		25358593.08	

Figure 4.4: A sample of the format of the master spreadsheet generated in the above LabVIEW programs. This image demonstrates how the table is structured and that not all satellites have data. This screenshot only shows 7 of the 256 total columns in the file.

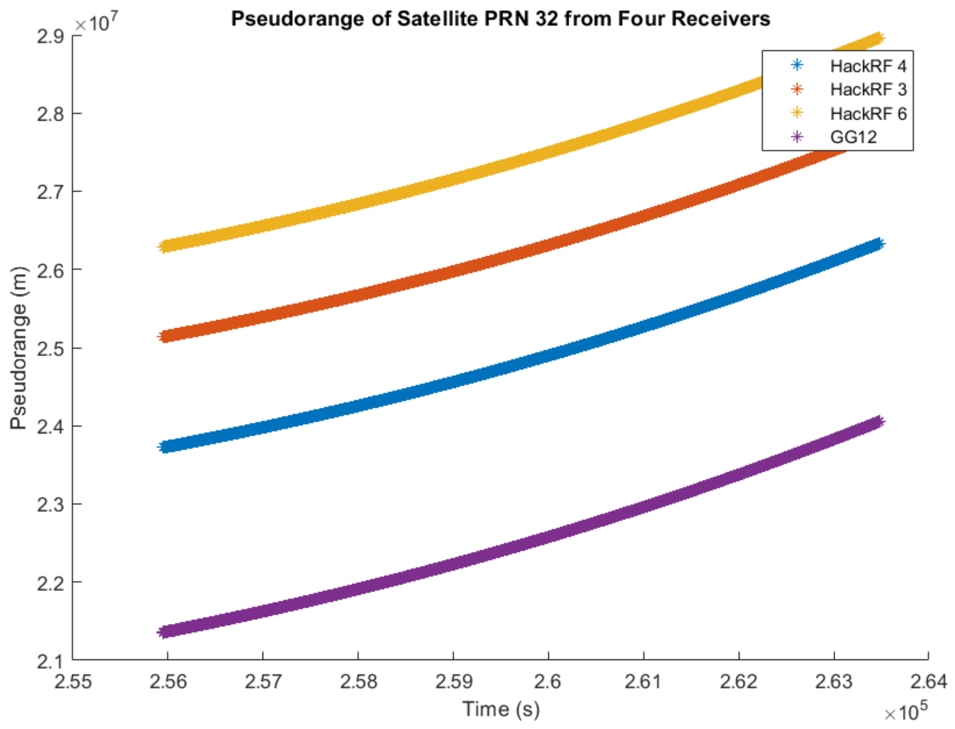
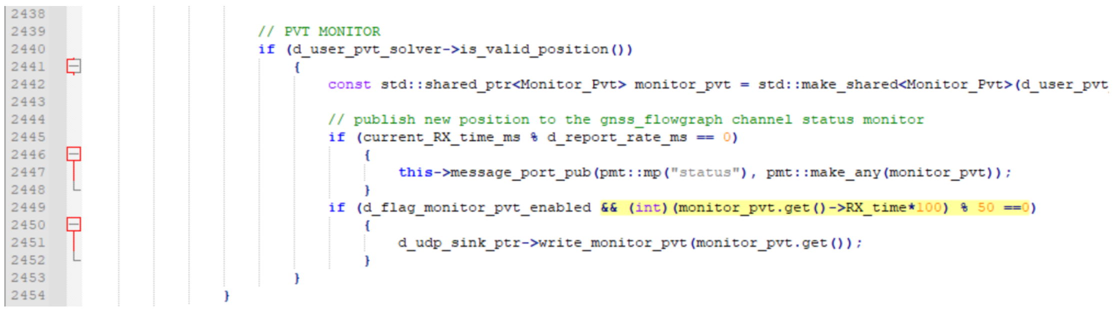


Figure 4.5: Pseudoranges generated from three HackRF Ones and one GG12.

UDP stream than the pseudorange. Multiplying this offset value by the speed of light and subtracting the product from the corresponding pseudorange with the same receive time results in a value that is substantially closer to the GG12s' pseudorange.

One issue associated with the offsets being sent out by the GNSS-SDR software is that they are sent using a different UDP stream than the other data and are not exactly synchronized with it. This means that while the GNSS-SDR pseduorange data is being sent twice a second with receive times at every 0 and 500 millisecond mark, the offset data is getting sent for completely different receive times. This makes the offset data not useful as they cannot be matched up one-to-one. A quick solution to this is found back in the GNSS-SDR source code. In one particular file, a check, similar to the check made in chapter three, can be applied to force the output to be synchronized to every 0 and 500 milliseconds. The code changes are shown in Figure 4.6.



```
            // PVT MONITOR
            if (d_user_pvt_solver->is_valid_position())
                {
                    const std::shared_ptr<Monitor_Pvt> monitor_pvt = std::make_shared<Monitor_Pvt>(d_user_pvt

                    // publish new position to the gnss_flowgraph channel status monitor
                    if (current_RX_time_ms % d_report_rate_ms == 0)
                        {
                            this->message_port_pub(pmt::mp("status"), pmt::make_any(monitor_pvt));
                        }
                    if (d_flag_monitor_pvt_enabled && (int)(monitor_pvt.get()->RX_time*100) % 50 ==0)
                        {
                            d_udp_sink_ptr->write_monitor_pvt(monitor_pvt.get());
                        }
                }
        }
```

Figure 4.6: GNSS-SDR code change to force the output to be sent at every 0 and 500 millisecond mark is highlighted above.

After successfully synchronizing the clock offset and pseudorange data, the pseudorange values can now be corrected. The results of the clock bias-corrected pseudoranges are shown below in Figure 4.7.

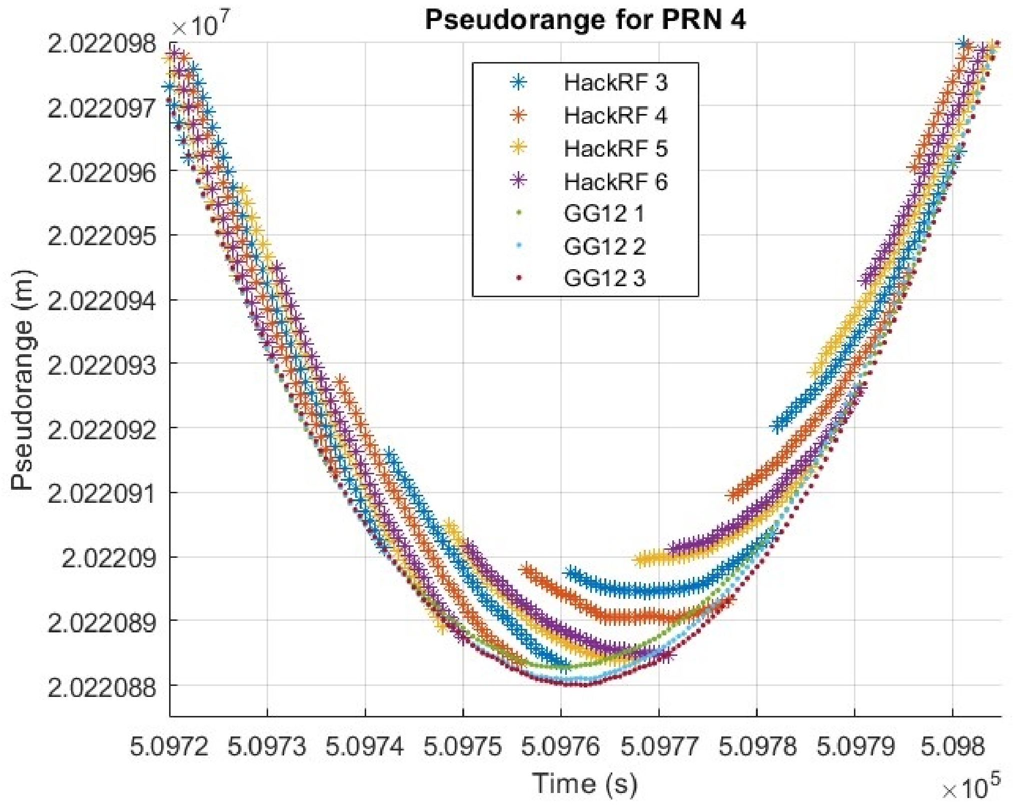


Figure 4.7: Pseudorange plots of four HackRF Ones and three GG12s after clock offsets have been applied.

The clock bias is the calculated difference in time between the GPS and satellite clocks. All GPS receivers have a difference in internal clock time. It would appear that these biases are automatically removed from the GG12s but are not removed in the GNSS-SDR software. This offset must be applied during the real-time algorithms in order for the GBAS to be effective. In GNSS-SDR, the RTKLIB is used to compute the position solution. In the process of computing the position, the RTKLIB software also calculates the clock bias of the receiver. The calculation for the clock bias is dependent on which type of positioning mode is configured in the GNSS-SDR configuration file. For the tests shown here, the static Precise Point Positioning (PPP) positioning mode is used. The exact calculations for this method can be found in the RTKLIB manual [35].

In Figure 4.8, the pseudorange values of each receiver are subtracted from the average GG12 pseduorange value. This results in a much simpler graph than the data in Figure 4.7. One interesting thing to note is that the pseudoranges of the HackRF Ones differ from the average GG12 value by +/-30m and it appears that the HackRF One pseudoranges have a periodic nature to them as they appear to oscillate relative to the average GG12 value. The fact that it is oscillatory is important as it may be something that can be optimized using filtering techniques in the future.

To determine if the low-cost SDRs were the problem, the Ettus Research USRP E310 SDR was used alongside the HackRF Ones and GG12s in the pseudorange test. This USRP is an expensive software-defined radio and has many performance improvements over the HackRF One. Getting this SDR to work with the GNSS-SDR software was a considerable challenge and was eventually solved. The details of the integration of the USRP into the system are included in Appendix F. For the same comparison as above but on a different day with different satellites, the

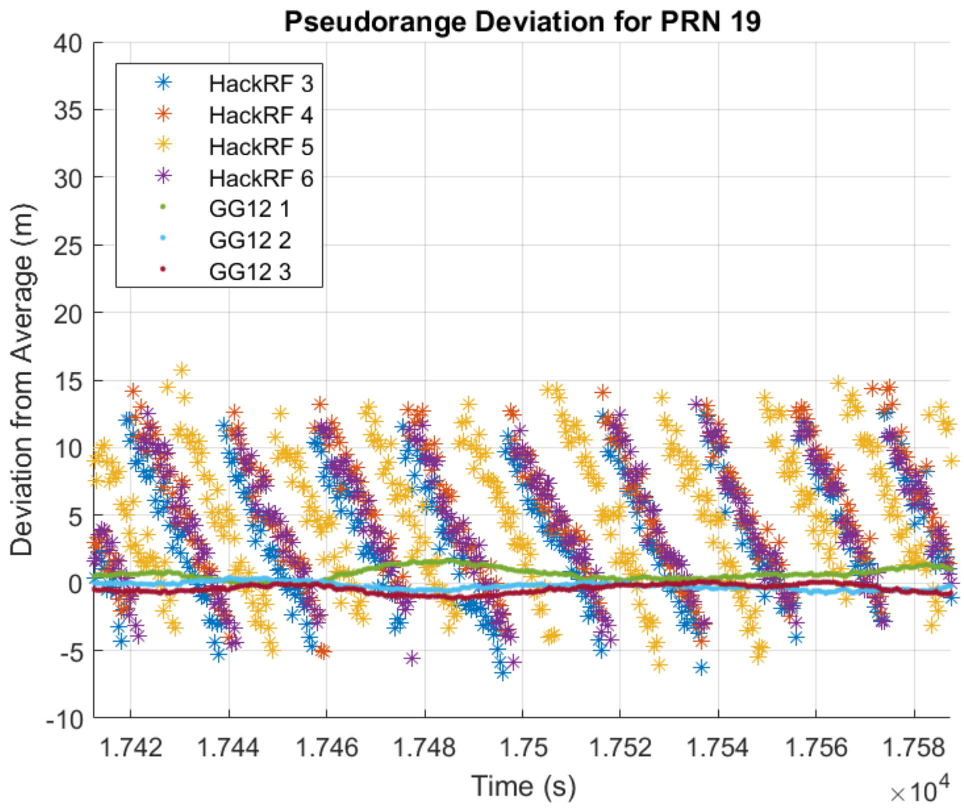


Figure 4.8: Four HackRF Ones' and three GG12s' pseudoranges are plotted against the average GG12 pseudorange for the given satellite.

resulting graph in Figure 4.9 was generated. As can be seen, the HackRF Ones still maintain their oscillating pseudorange values while the E310's values are quite sporadic with values in similar ranges to the HackRF Ones. There appears to be no obvious oscillation, but it seems like the E310 may benefit from GNSS-SDR's built-in smoothed pseudorange calculations.

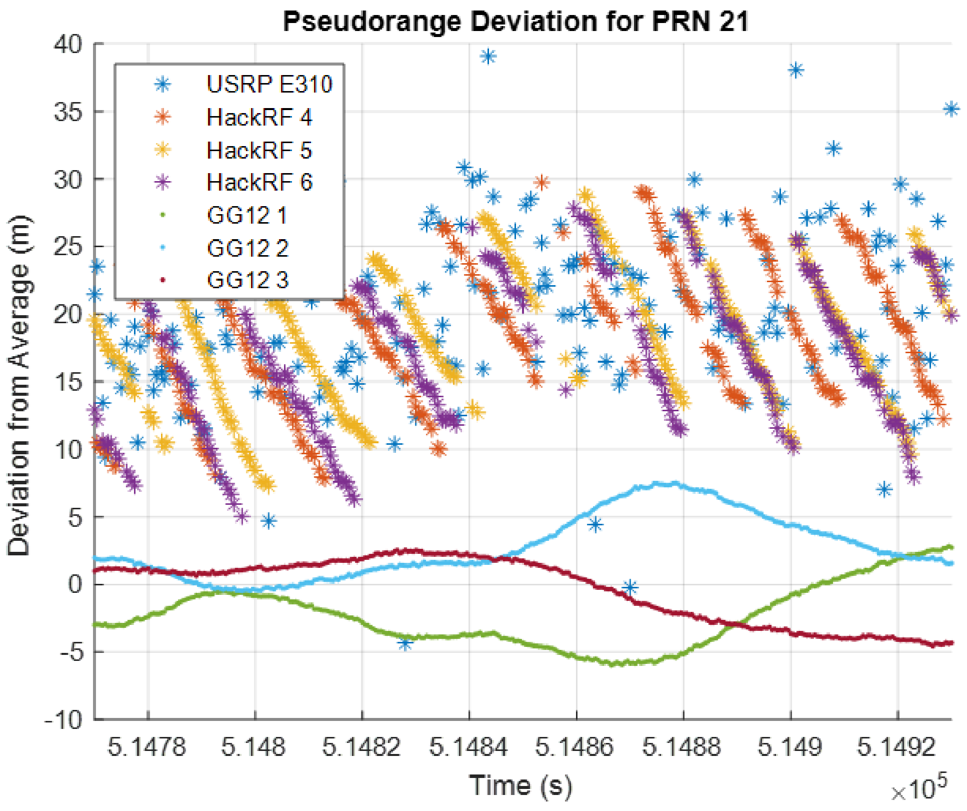


Figure 4.9: The USRP E310, three HackRF Ones' and three GG12s' pseudoranges are plotted against the average GG12 pseudorange for the given satellite PRN21.

A second set of data was collected after attempting to improve upon the original results with the E310. This time, GNSS-SDR's pseudorange smoothing was enabled, whose algorithm is described in [36]. As can be seen in Figure 4.10, the E310 pseudorange values are significantly smoother. These results are encouraging because they do not appear to show any oscillations like the HackRF Ones.

It should be noted that the E310 results could likely be improved upon if the GNSS-SDR configuration parameters are carefully tuned.

Another issue that was discovered is that the network switch in the testing setup was not adequate and lead to frequent loss of lock on satellites. This was discovered by directly connecting the E310 to the Intel NUC host computer. However, in this setup, the GNSS-SDR had no way of communicating to the base station computer since it was not connected through the network switch. It may be possible to increase the kernel and NIC buffer size on the Intel NUC, and that may solve this issue altogether, or simply a better network setup with better cables and a network switch with higher throughput. All this being said, it may not improve the accuracy of the E310, but it will certainly make it more reliable.

While testing the data, a plot of just the clock errors for the HackRF One was produced. After zooming in on the graph, “jumps” in the clock error became immediately evident. This plot is shown in Figure 4.11. After carefully analyzing the clock error graph, it appeared that these “jumps” in the error occurred at about the same time that the pseudorange values did. Next, a second plot, Figure 4.12 was generated overlaying the two graphs with the same receive times. It becomes clear that the time when clock error “jumps” and the time when pseudorange values “jumps” are closely related. Next, the clock error for the E310 is plotted in Figure 4.13. Unlike the HackRF One, the USRP E310 does not have any jumps in its clock error plots. This shows that the E310, which has non-oscillating pseudorange values, is likely to have better results in a GBAS setup than the HackRF One using GNSS-SDR.

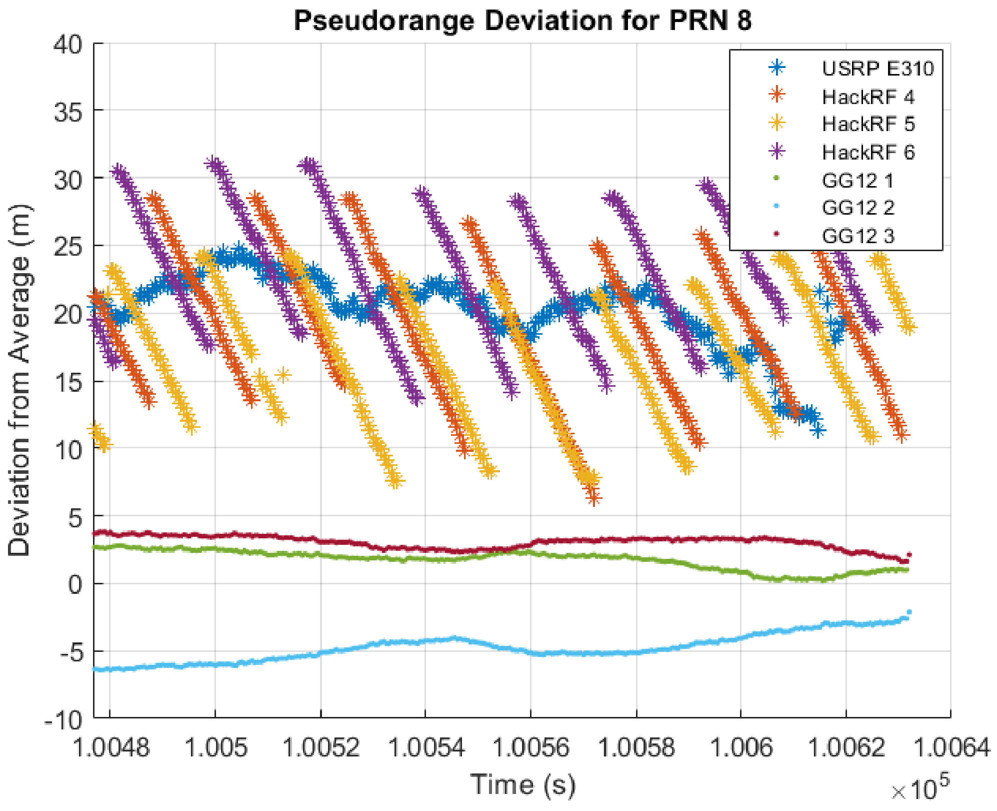


Figure 4.10: The USRP E310, three HackRF Ones' and three GG12s' pseudoranges are plotted against the average GG12 pseudorange for the given satellite PRN8. The pseudoranges from the E310 has much smoother values.

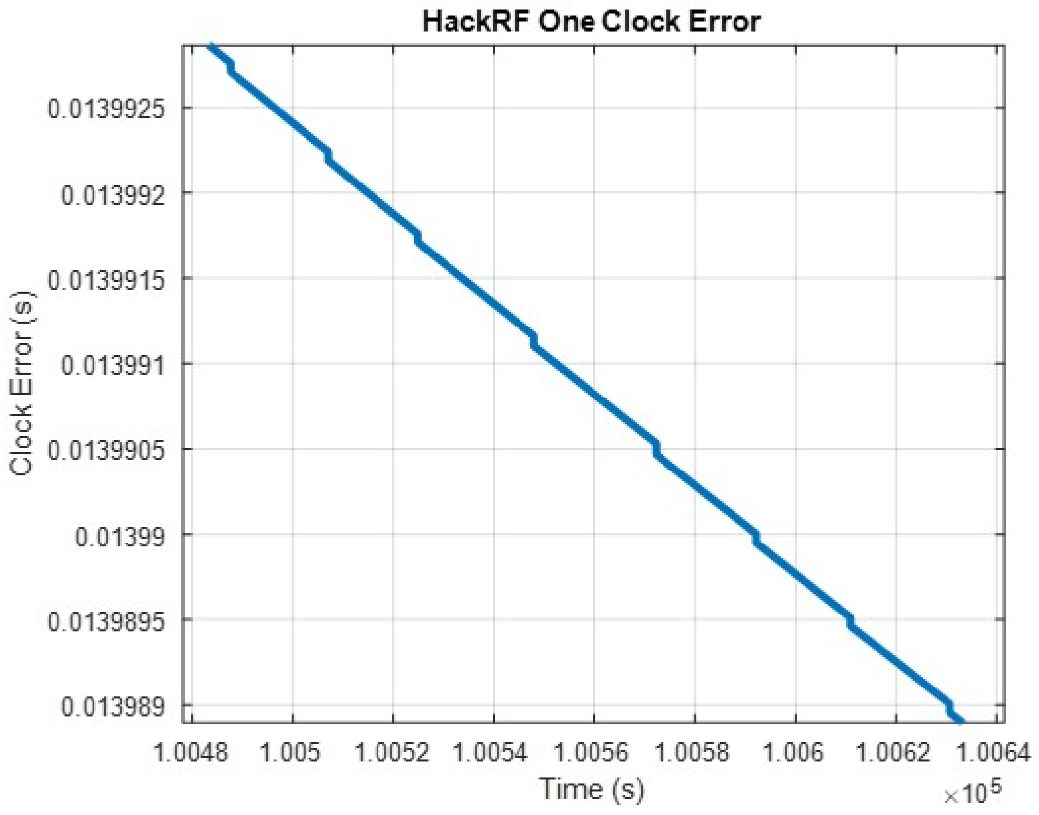


Figure 4.11: The clock error corresponding to the HackRF 4 in the graph above. As can be seen, there are consistent jumps in the clock error.

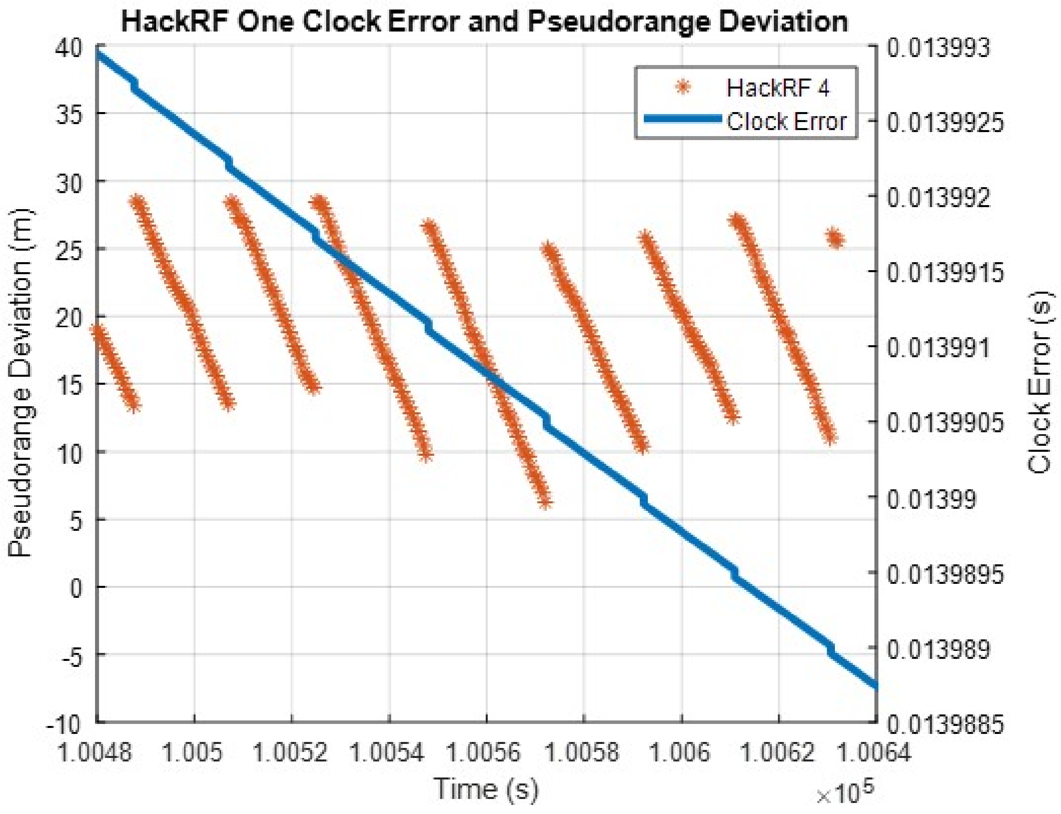


Figure 4.12: The clock error of the HackRF One overlaid with its previous pseudorange values to improve clarity. It can be clearly seen that the clock error and the pseudorange values are correlated.

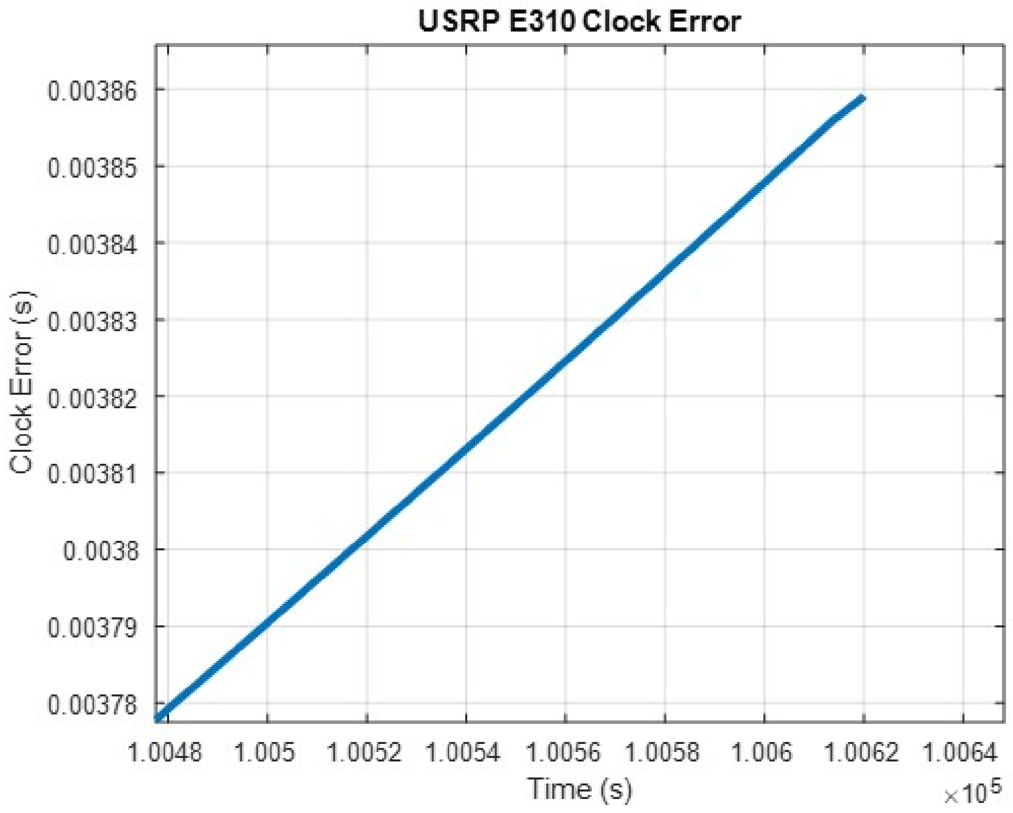


Figure 4.13: The clock error of the USRP E310. Contrary to the HackRF One's clock error above, there are no noticeable jumps.

Chapter 5

Summary and Conclusions

5.1 Summary

This thesis describes the application of using low-cost software-defined radios using the GNSS-SDR software for a ground-based augmentation system. The GNSS-SDR software can produce very accurate positional data on its own, but the performance demand greatly increases when using it for a GBAS. Getting the HackRF One with GNSS-SDR to run through the GBAS program required a substantial amount of software debugging and changes to reach the point where the data being streamed into the base station computer was formatted and synchronized correctly to produce a successful result. These changes include debugging the GNSS-SDR source code, changing the configuration file for GNSS-SDR, and correcting the navigation data using data from multiple UDP streams after the data is received at the GBAS station.

Although the HackRF Ones suffered from a lack of reliability in the GBAS program, they were indeed able to produce a good user position vertical error, but only when there were sufficient usable satellites in the CL-GBAS algorithm. Our results showed that the vertical error produced was only a useful result about

28% of the time. The reason for the lack of reliability is likely the inconsistent and jumping pseudorange values. These are likely caused by the lack of a high quality internal oscillator within the HackRF One, even with a high quality external oscillator attached. These results confirm that inexpensive software-defined radios *could* be used as receivers for a CL-GBAS, but the receivers will need significant improvements to improve reliability and accuracy of the resulting CL-GBAS corrections.

5.2 Contributions

The contributions of this thesis work include developing a method for retrieving and decoding live data from GNSS-SDR using programs written in LabVIEW. This method includes a unique way of quickly and reliably decoding Google Protocol Buffer messages, since there are no other supported ways to do this in LabVIEW. Another contribution includes debugging the GNSS-SDR open-source code and submitting a bug fix in the way that the data is output via UDP, which allows users to properly apply a decimation factor to the output. Another contribution is the unique way of acquiring and synchronizing the outputs from GNSS-SDR, so that multiple receivers would output their data at the same receive times, allowing for better consistency and the ability to run in a GBAS program. Since there are no differences in the outputs from different SDRs through GNSS-SDR, any SDR can be used, it is not limited to the HackRF One or the USRP E310. The next contribution is the study and analysis of the pseudorange of the GNSS-SDR SDRs and the Thales GG12s. Along with the pseudorange study was the study of the vertical error produced by the GNSS-SDR receivers. The final contribution is the creation of a complete lab setup which included the antennas, the network configuration, and the connection of many receivers allowing for consistent and reliable tests to be ran.

5.3 Future Works

To improve the capabilities of software-defined radios (SDRs) in a ground-based augmentation system, future work may include improving the GNSS-SDR firmware and software to better synchronize the SDRs, while improving sending the information from the reference station computer to the base station computer. Currently that data link sends the information over a network in the form of UDP packets. It may be advantageous to send it over using different methods, such as serial communications.

Another improvement would be to properly package the data with the reference station computer so that the navigation messages can be sent over a long distance using wireless communication links.

It would also simplify the base station programming if the GNSS-SDR program sent the clock data offset with the pseudorange measurement. This would reduce the complications found above where it was necessary to wait for the clock offset data to come in before the navigational data so that the offset could be properly applied. This could also be an avenue used to speed up the processing in the base station codes. Additional analysis of the USRP E310 or other high-quality SDRs is needed to help rule out any issues that may be with just the HackRF One. If a better network switch is acquired, the USRP E310 can be tested within the GBAS program as well. Lastly, the output and formatting of the GPS almanac data through GNSS-SDR is needed to be developed so that the almanac data can be acquired without a dedicated GPS receiver or the internet.

Bibliography

[1] FAA. (2022) Satellite navigation - gps - how it works. [Online]. Available: https://www.faa.gov/about/office_org/headquarters_offices/ato/service_units/techops/navservices/gnss/gps/howitworks

[2] G. S. Gadgets. (2022) Hackrf one. [Online]. Available: https://greatscottgadgets.com/hackrf/one/

[3] E. Research. (2022) Usrp e310. [Online]. Available: https://www.ettus.com/all-products/e310/

[4] C. Davis, "Conceptualization and implementation of a new and novel ground based augmentation system utilizing feedback control," Dissertation, University of Oklahoma, Norman, Oklahoma, 2007.

[5] G.-S. Team. (2022) Gnss-sdr docs. [Online]. Available: https://gnss-sdr.org/docs/

[6] R. Pendergraft, "A new adaptive integrity monitor for the local area augmentation system utilizing closed loop feed back," Dissertation, University of Oklahoma, Norman, Oklahoma, 2013.

[7] C. Sherrell, "Development of a robust ground based satellite augmentation system software architecture," Thesis, University of Oklahoma, Norman, Oklahoma, 2006.

[8] M. B. . T. Harris. (2006, Sep.) How gps receivers work. [Online]. Available: https://electronics.howstuffworks.com/gadgets/travel/gps.htm

[9] FAA. (2022) Global positioning system standard positioning service performance analysis report. [Online]. Available: https://www.nstb.tc.faa.gov/reports/2020_Q4_SPS_PAN_v2.0.pdf#

[10] R. Connor. (2014, Feb.) Twenty years of gps and instrument flight. [Online]. Available: https://airandspace.si.edu/stories/editorial/twenty-years-gps-and-instrument-flight

[11] M. Matosevic, Z. Salcic, and S. Berber, "A comparison of accuracy using a gps and a low-cost dgps," *IEEE Transactions on Instrumentation and Measurement*, vol. 55, no. 5, pp. 1677–1683, 2006.

[12] S. Datta-Barua, P. Doherty, S. Delay, T. Dehel, and J. Klobuchar, "Ionospheric scintillation effects on single and dual frequency gps positioning," *Proceedings of the 16th International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GPS/GNSS 2003)*, vol. 16, pp. 336–346, Sept 2003.

[13] D. Skournetou and E.-S. Lohan, "Comparison of single and dual frequency gnss receivers in the presence of ionospheric and multipath errors," in *Personal Satellite Services*, G. Giambene and C. Sacchi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 402–410.

[14] R. Odolinski and P. J. G. Teunissen, "Single-frequency, dual-gnss versus dual-frequency, single-gnss: a low-cost and high-grade receivers gps-bds rtk analysis," *Journal of Geodesy*, vol. 90, no. 11, pp. 1255–1278, Nov 2016. [Online]. Available: https://doi.org/10.1007/s00190-016-0921-x

[15] D. Skufca, "Dual frequency gps receiver implementation in gnss-sdr," 2018. [Online]. Available: https://sites.tufts.edu/eeseniordesignhandbook/files/2018/05/Skufca-EJ-edit.pdf

[16] I. S. Team. (2022) Multi-band gnss. [Online]. Available: https://docs.inertialsense.com/user-manual/gnss/multi_band_gnss/#advantages

[17] FAA. (2008) Global positioning system wide area augmentation system (waas) performance standard. [Online]. Available: https://www.gps.gov/technical/ps/2008-WAAS-performance-standard.pdf

[18] O. Pedro, S. Jorge, and S. Paulo, "A comparative study between ils and gbas approaches: The case of viseu airfield," *Journal of Airline and Airport Management*, vol. 10, no. 2, pp. 65–75, 09 2020.

[19] M. T. and et al., "Approach with precision," *GPS World*, vol. 17, pp. 32–38, 2006.

[20] D. of Defense, D. of Homeland Security, and D. of Transportation. (2021) 2021 federal radionavigation plan.

[21] U. D. of Transportation ederal Aviation Administration. (2005, Oct) Non-fed specification: Category i local area augmentation system ground facility. faa-e-ajw44-2937a.

[22] R. (Firm), *Minimum Aviation System Performance Standards for the Local Area Augmentation System (Laas)*. RTCA, Washington, DC, 2004.

[23] GPS.gov. (2021, Oct.) Other global navigation satellite systems (gnss). [Online]. Available: https://www.gps.gov/systems/gnss/#navic

[24] C. Gioia, D. Borio, A. Angrisano, S. Gaglione, and J. Fortuny, “A galileo iov assessment: Measurement and position domain,” *GPS Solutions*, vol. 19, pp. 187–199, 04 2015.

[25] G. B. H. de Azevedo, “Software defined receiver for global navigation satellite systems,” Norman, Oklahoma, May 2020, final Report for ECE-5990 course. Unpublished.

[26] N. Team. (2022) Hackrf one - ’tiny tcxo’ 10mhz 0.5ppm tcxo module. [Online]. Available: https://www.nooelec.com/store/tiny-tcxo.html

[27] C. Fernández-Prades, J. Arribas, and P. Closas, “Turning a television into a gnss receiver,” in *ION GNSS+At: Nashville, Tennessee*, 09 2013.

[28] M. Cutugno, U. Robustelli, and G. Pugliano, “Low-cost gnss software receiver performance assessment,” *Geosciences*, vol. 10, no. 2, 2020. [Online]. Available: https://www.mdpi.com/2076-3263/10/2/79

[29] G. Team. (2022) Protocol buffers overview. [Online]. Available: https://developers.google.com/protocol-buffers/docs/overview

[30] Álvaro Cebrián Juan, “Gnss-sdr-monitor,” https://github.com/acebrianjuan/gnss-sdr-monitor, 2022.

[31] T. Navigation, “Gg12 gps board,” 2002.

[32] D. M. II, “A real time ground based augmentation system implemented with labview software,” Thesis, University of Oklahoma, Norman, Oklahoma, 2022.

[33] FAA. (2017, Oct) Gbas performance analysis and activities report. [Online]. Available: https://laas.tc.faa.gov/documents/GPAR/2017/GPAR_2017_Q3.pdf

[34] G. F. Mark Petovello, Marco Rao, “Code tracking and pseudoranges: How can pseudorange measurements be generated from code tracking?” *Inside GNSS*, vol. 7, no. 1, pp. 26–33, Feb/Jan 2012.

[35] R. Team. (2013, Apr) Rtklib ver. 2.4.2 manual. [Online]. Available: https://www.rtklib.com/prog/manual_2.4.2.pdf

[36] M. V. Mark Petovello, Letizia Lo Presti, “Can you list all the properties of the carrier-smoothing filter?” *Inside GNSS*, vol. 10, no. 4, pp. 26–33, Jul/Aug 2015.

[37] A. Prakash. (2020, Nov.) How to install ubuntu along with windows. [Online]. Available: https://itsfoss.com/install-ubuntu-dual-boot-mode-windows/

[38] N. J. Cannon, "Electrical development of a reference receiver system for use on the ou local area augmentation system," Thesis, University of Oklahoma, Norman, Oklahoma, 2009.

[39] J. M. Harkness, "Mechanical development of a reference receiver system for use on the ou local area augmentation system," Thesis, University of Oklahoma, Norman, Oklahoma, 2009.

[40] J. Dyer, "A new siting model for reference station placement in a ground based augmentation system," Dissertation, University of Oklahoma, Norman, Oklahoma, 2008.

[41] G. Networking, "Aldcbs1x4," 2020. [Online]. Available: https://www.gpsnetworking.com/system/datasheets/46/original/DS-ALDCBS1X4.pdf?1631560437

[42] E. Research. (2022) Usrp hardware driver and usrp manual. [Online]. Available: https://files.ettus.com/manual_archive/release_003_009_007/html/page_usrp_e3x0.html

[43] T. Julian, "Getting started with the usrp e310 and gnss-sdr," Norman, Oklahoma, May 2022, final Report for ECE-5970 course. [Online]. Available: https://github.com/TylerJulian/gnss/blob/main/TylerJulian_5970.pdf

[44] A. Bityutskiy. (2022) Bmap-tools documentation. [Online]. Available: https://manpages.ubuntu.com/manpages/xenial/man1/bmaptool.1.html\#author

Appendix A

List Of Acronyms and Abbreviations

ADC	Analog to Digital Converter
CL-GBAS	Closed-Loop Ground Based Augmentation System
DGPS	Differential Global Positioning System
FAA	Federal Aviation Administration
FDE	Fault Detection and Inclusion
GB	Gigabyte
GBAS	Ground-Based Augmentation System
GNSS	Global Navigation Satellite System
GNSS-SDR	Global Navigation Satellite Systems Software-Defined Receiver
GPS	Global Positioning System
ILS	Instrument Landing System
IP	Internet Protocol
LAAS	Local Area Augmentation System
MMR	Multi-Mode Receiver
OU	University of Oklahoma
PC	Personal Computer
PM	Phase Modulation

PRN	Pseudorandom Noise
Protobufs	Google Protocol Buffers
RAIM	Receiver Autonomous Integrity Monitoring
RF	Radio Frequency
SBAS	Space-Based Augmentation System
SDR	Software-Defined Radio
SNR	Signal-to-Noise Ratio
TCP	Transmission Control Protocol
TCXO	Temperature Controlled/Compensated Crystal Oscillator
Tx/Rx	Transmit/Receive
UDP	User Datagram Protocol
UHD	USRP Hardware Driver
USRP	Universal Software Radio Peripheral
VDB	VHF Data Broadcaster
VE	Vertical Error
VHF	Very High Frequency
WAAS	Wide Area Augmentation System

Appendix B

Installing GNSS-SDR for GBAS

B.1 Installing Dual-Boot Ubuntu

In this section, I will go over how to properly install the GNSS-SDR software on Ubuntu Linux in order for it to work with the HackRF One. Most of these steps were taken from and can be found in [37]. The first step is to get an installation of Ubuntu running on your computer.Ideally, the computer will only be running the Ubuntu OS. However, it can be useful to run multiple operating systems from one machine. From my own experience, running Ubuntu through a virtual machine does not work well as stability issues begin to arise and the software does not always run as expected. I was able to solve this on my current computer by dual-booting Linux, that way the software can access the full power of your computer.

To dual-boot Linux on a Windows 10 PC, it requires you to have enough storage space on your hard drive, probably at least 50GB. Ideally, I set 200 GB for my dual-boot partition. However, I will list the important steps below as well. Consider backing up your hard drive before attempting the following steps.

1. Download the latest version of Ubuntu from their website.
2. Download Universal USB Installer
3. Create a bootable Linux thumb drive
4. While on windows, make sure there is an empty partition with at least 20GB, (I used 20GB)
5. Restart your PC and select from "boot from removable disk"
6. (WARNING) You will need to create your own partition under installation type to boot the device to. Do not select an option that deletes the rest of your disk, this would not be ideal for a "dual-boot".
   (a) Create memory of type "Ext 4 journaling file system" called "/". This is the root directory. It should be at least 15GB.

   (b) Create a directory called “swap”. This should be twice the size of your system’s RAM.
   (c) Create a directory called /home for everything else. 20GB should be fine here.

7. Once all three directories are created, click “install now” and follow the rest of the directions.

Congratulations! You have now installed dual-boot Ubuntu. Every time you start your computer, you will be prompted to select the operating system which you wish to boot to. Tip: if you use “hibernate” when shutting down Windows 10, you can boot to Ubuntu, then restart and boot back into Windows 10 and all your applications will still be running. Installing GNSS-SDR After you have installed Ubuntu, you can start to install the GNSS-SDR software. To do this, you will need to be connected to the internet. Once connected to the internet, proceed with the following steps to install the software.

1. Open a terminal using Ctrl + Alt + T.
2. Run the following series of commands:
   (a) sudo apt-get update
   (b) sudo apt-get upgrade
   (c) sudo apt-get install build-essential cmake git pkg-config libboost-dev libboost-date-time-dev libboost-system-dev libboost-filesystem-dev libboost-thread-dev libboost-chrono-dev libboost-serialization-dev libboost-program-options-dev libboost-test-dev liblog4cpp5-dev libuhd-dev gnuradio-dev gr-osmosdr libblas-dev liblapack-dev libarmadillo-dev libgflags-dev libgoogle-glog-dev libhdf5-dev libgnutls-openssl-dev libmatio-dev libpugixml-dev libpcap-dev libprotobuf-dev protobuf-compiler libgtest-dev googletest python3-mako python3-six
   (d) mkdir work (if not already created)
   (e) cd work
   (f) git clone https://github.com/gnss-sdr/gnss-sdr
   (g) cd gnss-sdr/build
   (h) git checkout next (this gives you the newest/updated “next” branch)
   (i) cmake -DENABLE_OSMOSDR=ON ../ (osmo sdr is required for the HRF1)
   (j) make (takes a long time)
   (k) sudo make install
   (l) sudo apt-get install hackrf

Congratulations! You have successfully installed the HackRF One and GNSS-SDR software. To check that the HackRF One is detected and working properly, you can run the "hackrf_info" command in the terminal. Installing GNSS-SDR-Monitor GNSS-SDR-Monitor is a helpful visual program to ensure that the GNSS-SDR software is working properly as intended. It shows each satellite on each channel and several pieces of data about each satellite. It also has a map showing the position of the antenna. To install it, run the following commands (assuming GNSS-SDR is already installed):

1. sudo apt install build-essential cmake git libboost-dev libboost-system-dev libprotobuf-dev protobuf-compiler qtbase5-dev qtdeclarative5-dev qtpositioning5-dev libqt5charts5-dev qml-module-qtquick2 qml-module-qtquick-controls2 qml-module-qtquick-window2 qml-module-qtlocation qml-module-qtpositioning qml-module-qtquick-layouts

2. cd work

3. git clone https://github.com/acebrianjuan/gnss-sdr-monitor

4. cd gnss-sdr-monitor/build

5. cmake ..

6. make

7. sudo make install

B.2 Running GNSS-SDR and Monitor

Now, to test the software. One will need a proper working antenna (grounded and powered if necessary) connected to the HRF1's antenna connector. Next, copy the already created configuration file to the work directory. Then run the following command in the terminal:

1. "gnss-sdr –config_file=/home/gbas/work/hackrf_GPS_L1.conf"

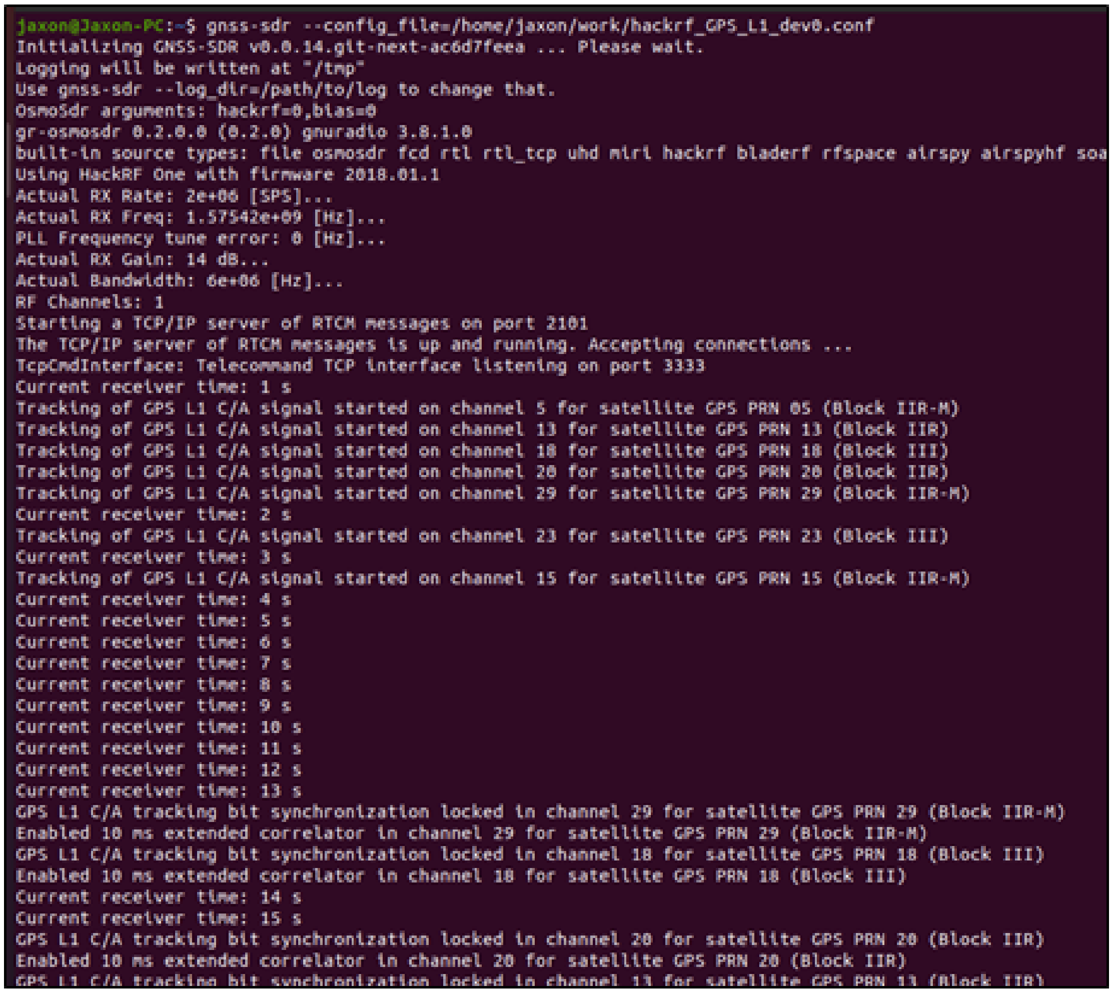
Several lines of text should start to appear. Proper working of the software will show several of the following lines after a given amount of time:

1. Tracking of GPS L1 C/A signal . . .

2. GPS L1 C/A tracking bit synchronization locked. . .

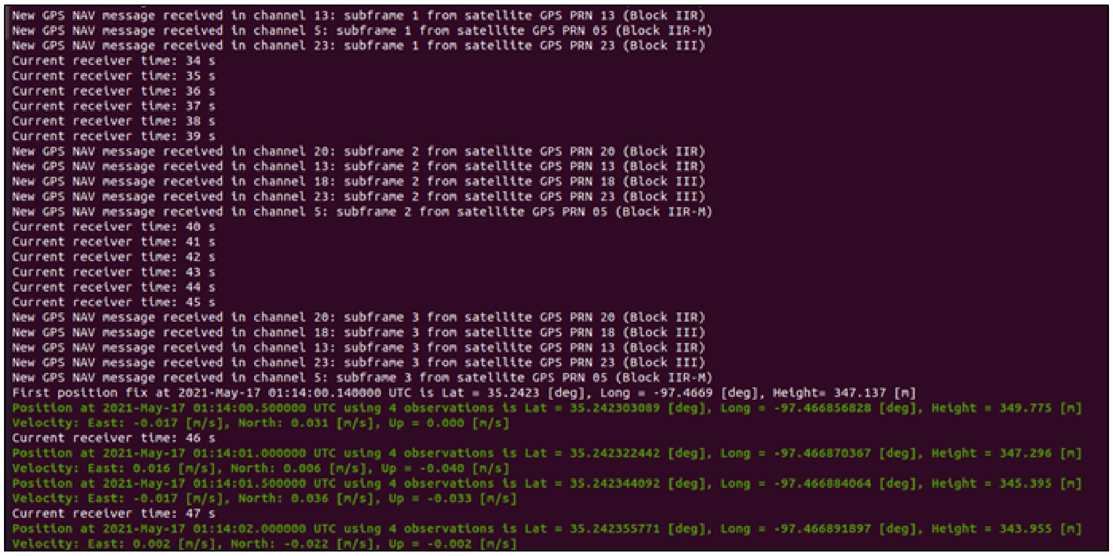
3. New GPS NAV message received. . .

4. Position at: . . .



```
jaxon@Jaxon-PC:~$ gnss-sdr --config_file=/home/jaxon/work/hackrf_GPS_L1_dev0.conf
Initializing GNSS-SDR v0.0.14.git-next-ac6d7feea ... Please wait.
Logging will be written at "/tmp"
Use gnss-sdr --log_dir=/path/to/log to change that.
OsmoSdr arguments: hackrf=0,bias=0
gr-osmosdr 0.2.0.0 (0.2.0) gnuradio 3.8.1.0
built-in source types: file osmosdr fcd rtl rtl_tcp uhd miri hackrf bladerf rfspace airspy airspyhf soa
Using HackRF One with firmware 2018.01.1
Actual RX Rate: 2e+06 [SPS]...
Actual RX Freq: 1.57542e+09 [Hz]...
PLL Frequency tune error: 0 [Hz]...
Actual RX Gain: 14 dB...
Actual Bandwidth: 6e+06 [Hz]...
RF Channels: 1
Starting a TCP/IP server of RTCM messages on port 2101
The TCP/IP server of RTCM messages is up and running. Accepting connections ...
TcpCmdInterface: Telecommand TCP interface listening on port 3333
Current receiver time: 1 s
Tracking of GPS L1 C/A signal started on channel 5 for satellite GPS PRN 05 (Block IIR-M)
Tracking of GPS L1 C/A signal started on channel 13 for satellite GPS PRN 13 (Block IIR)
Tracking of GPS L1 C/A signal started on channel 18 for satellite GPS PRN 18 (Block III)
Tracking of GPS L1 C/A signal started on channel 20 for satellite GPS PRN 20 (Block IIR)
Tracking of GPS L1 C/A signal started on channel 29 for satellite GPS PRN 29 (Block IIR-M)
Current receiver time: 2 s
Tracking of GPS L1 C/A signal started on channel 23 for satellite GPS PRN 23 (Block III)
Current receiver time: 3 s
Tracking of GPS L1 C/A signal started on channel 15 for satellite GPS PRN 15 (Block IIR-M)
Current receiver time: 4 s
Current receiver time: 5 s
Current receiver time: 6 s
Current receiver time: 7 s
Current receiver time: 8 s
Current receiver time: 9 s
Current receiver time: 10 s
Current receiver time: 11 s
Current receiver time: 12 s
Current receiver time: 13 s
GPS L1 C/A tracking bit synchronization locked in channel 29 for satellite GPS PRN 29 (Block IIR-M)
Enabled 10 ms extended correlator in channel 29 for satellite GPS PRN 29 (Block IIR-M)
GPS L1 C/A tracking bit synchronization locked in channel 18 for satellite GPS PRN 18 (Block III)
Enabled 10 ms extended correlator in channel 18 for satellite GPS PRN 18 (Block III)
Current receiver time: 14 s
Current receiver time: 15 s
GPS L1 C/A tracking bit synchronization locked in channel 20 for satellite GPS PRN 20 (Block IIR)
Enabled 10 ms extended correlator in channel 20 for satellite GPS PRN 20 (Block IIR)
GPS L1 C/A tracking bit synchronization locked in channel 13 for satellite GPS PRN 13 (Block IIR)
```

Figure B.1: GNSS window setup.



```
New GPS NAV message received in channel 13: subframe 1 from satellite GPS PRN 13 (Block IIR)
New GPS NAV message received in channel 5: subframe 1 from satellite GPS PRN 05 (Block IIR-M)
New GPS NAV message received in channel 23: subframe 1 from satellite GPS PRN 23 (Block III)
Current receiver time: 34 s
Current receiver time: 35 s
Current receiver time: 36 s
Current receiver time: 37 s
Current receiver time: 38 s
Current receiver time: 39 s
New GPS NAV message received in channel 20: subframe 2 from satellite GPS PRN 20 (Block IIR)
New GPS NAV message received in channel 13: subframe 2 from satellite GPS PRN 13 (Block IIR)
New GPS NAV message received in channel 18: subframe 2 from satellite GPS PRN 18 (Block III)
New GPS NAV message received in channel 23: subframe 2 from satellite GPS PRN 23 (Block III)
New GPS NAV message received in channel 5: subframe 2 from satellite GPS PRN 05 (Block IIR-M)
Current receiver time: 40 s
Current receiver time: 41 s
Current receiver time: 42 s
Current receiver time: 43 s
Current receiver time: 44 s
Current receiver time: 45 s
New GPS NAV message received in channel 20: subframe 3 from satellite GPS PRN 20 (Block IIR)
New GPS NAV message received in channel 18: subframe 3 from satellite GPS PRN 18 (Block III)
New GPS NAV message received in channel 13: subframe 3 from satellite GPS PRN 13 (Block IIR)
New GPS NAV message received in channel 23: subframe 3 from satellite GPS PRN 23 (Block III)
New GPS NAV message received in channel 5: subframe 3 from satellite GPS PRN 05 (Block IIR-M)
First position fix at 2021-May-17 01:14:00.140000 UTC is Lat = 35.2423 [deg], Long = -97.4669 [deg], Height= 347.137 [m]
Position at 2021-May-17 01:14:00.500000 UTC using 4 observations is Lat = 35.242303089 [deg], Long = -97.466856828 [deg], Height = 349.775 [m]
Velocity: East: -0.017 [m/s], North: 0.031 [m/s], Up = 0.000 [m/s]
Current receiver time: 46 s
Position at 2021-May-17 01:14:01.000000 UTC using 4 observations is Lat = 35.242322442 [deg], Long = -97.466870367 [deg], Height = 347.296 [m]
Velocity: East: 0.016 [m/s], North: 0.006 [m/s], Up = -0.040 [m/s]
Position at 2021-May-17 01:14:01.500000 UTC using 4 observations is Lat = 35.242344092 [deg], Long = -97.466884064 [deg], Height = 345.395 [m]
Velocity: East: -0.017 [m/s], North: 0.036 [m/s], Up = -0.033 [m/s]
Current receiver time: 47 s
Position at 2021-May-17 01:14:02.000000 UTC using 4 observations is Lat = 35.242355771 [deg], Long = -97.466891897 [deg], Height = 343.955 [m]
Velocity: East: 0.002 [m/s], North: -0.022 [m/s], Up = -0.002 [m/s]
```

Figure B.2: GNSS window with position data.

Refer to Figures B.1 and B.2 to see a screenshot of what the software looks like in the command terminal.

If you see a lot of "Loss of Lock" messages, that generally means you do not have a reliable lock on the satellite. This can result from faulty antenna connection/setup or a slow PC. Make sure to check all wired connections going to the antenna and that the antenna is getting the proper DC bias voltage sent to it (if needed). If you see lots of "OOO..." messages, that means you are receiving messages but are unable to process them fast enough. This is a sign of a slower PC and you may need to upgrade it. These "OOO..." messages are a common cause of "Loss of Lock" issues.

Appendix C

Hardware Setup

C.1 Antennas

The hardware used in this work consists of four antennas on the rooftop of one of the OU buildings at the local Max Westheimer Airport in Norman, Oklahoma. The antenna labeled “D” in Figure C.1 is the original OU local monitor antenna used in the early GBAS work found in [4], [6]. This antenna is a simple GPS antenna and is seen in Figure C.2. The other three antennas are fully described in [38] and [39], although the antennas have been stripped of their electrical component boxes and their solar panels. This was done to lower their cross-sectional area and was combined with several weather-proof sandbags weighing them down, allowing the antennas to withstand several severe storms. An example of these antennas, the “Parking Lot Antenna” labeled “C” in Figure C.1 is shown in Figure C.3. Using the given space on the rooftop, the antennas were spread out as much as possible to get the best results, similarly described in [40].

C.2 Cables

LMR-400 coaxial cable, selected for its great combination of price and performance, was ran from each antenna along the roof, down the side of the building, through multiple walls, and into the GBAS room, shown in Figure C.4. At this point, a type-N connector was attached on both ends of each cable. These cables were then connected to their own active GPS splitter [41], as shown in Figure C.5. Additional cables were ran from these splitters to the desired receivers.

C.3 Receivers and Computers

The cables from the antennas were ran into the GPS splitters and then from there routed into the desired receivers. The GG12 receivers were connected to their own custom breakout board and then connected to the main GBAS PC via serial cables. The GNSS-SDR receivers were connected to their host PC either by USB

(HackRF One) or via Ethernet (USRP E310). The host PCs were then connected to a simple network switch via Ethernet, which also connected to the GBAS PC. The entire setup can be seen in Figure C.7.

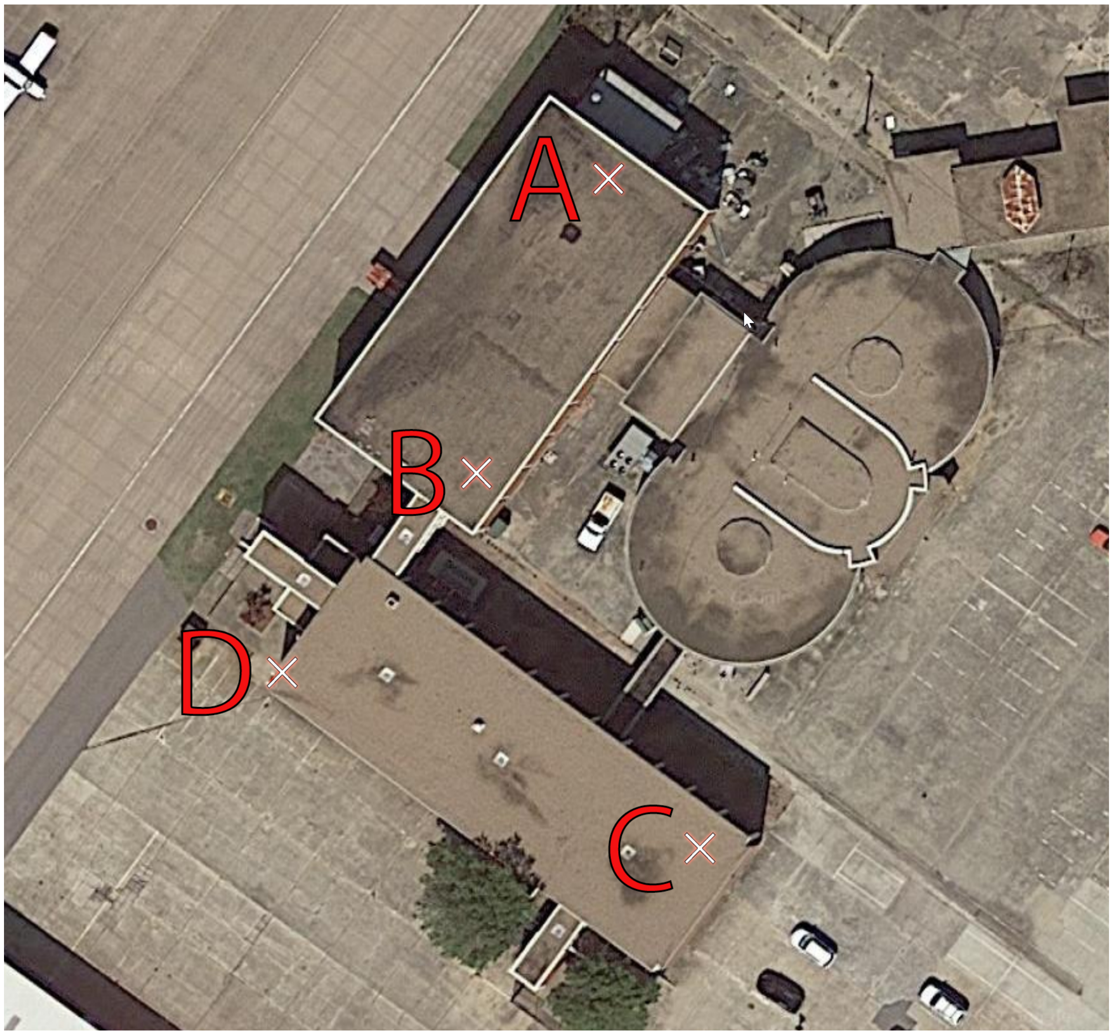


Figure C.1: Locations of the OU GBAS Reference Station Antennas.

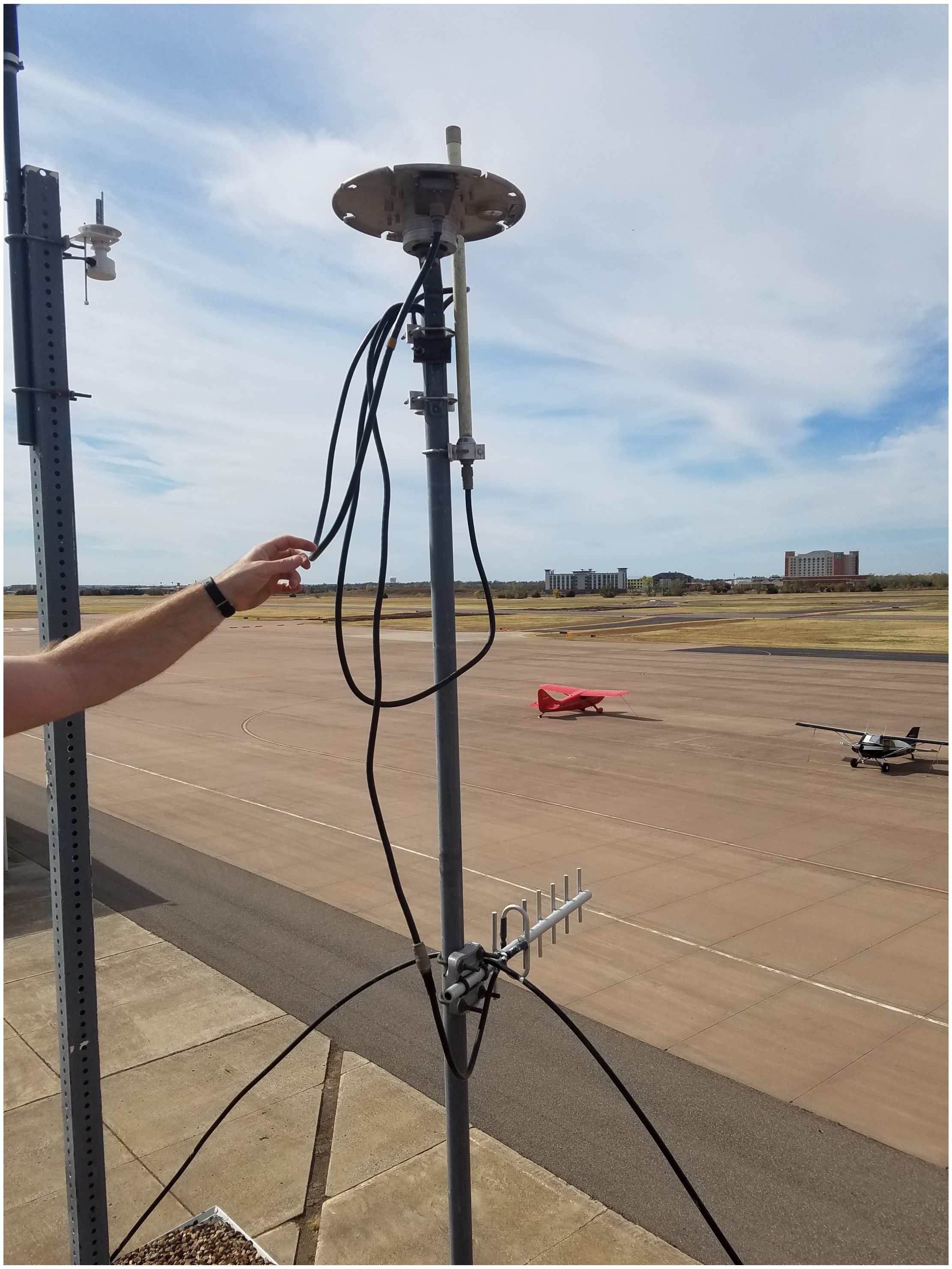


Figure C.2: The original OU GBAS local monitor antenna. This antenna is labeled “D” in Figure C.1.

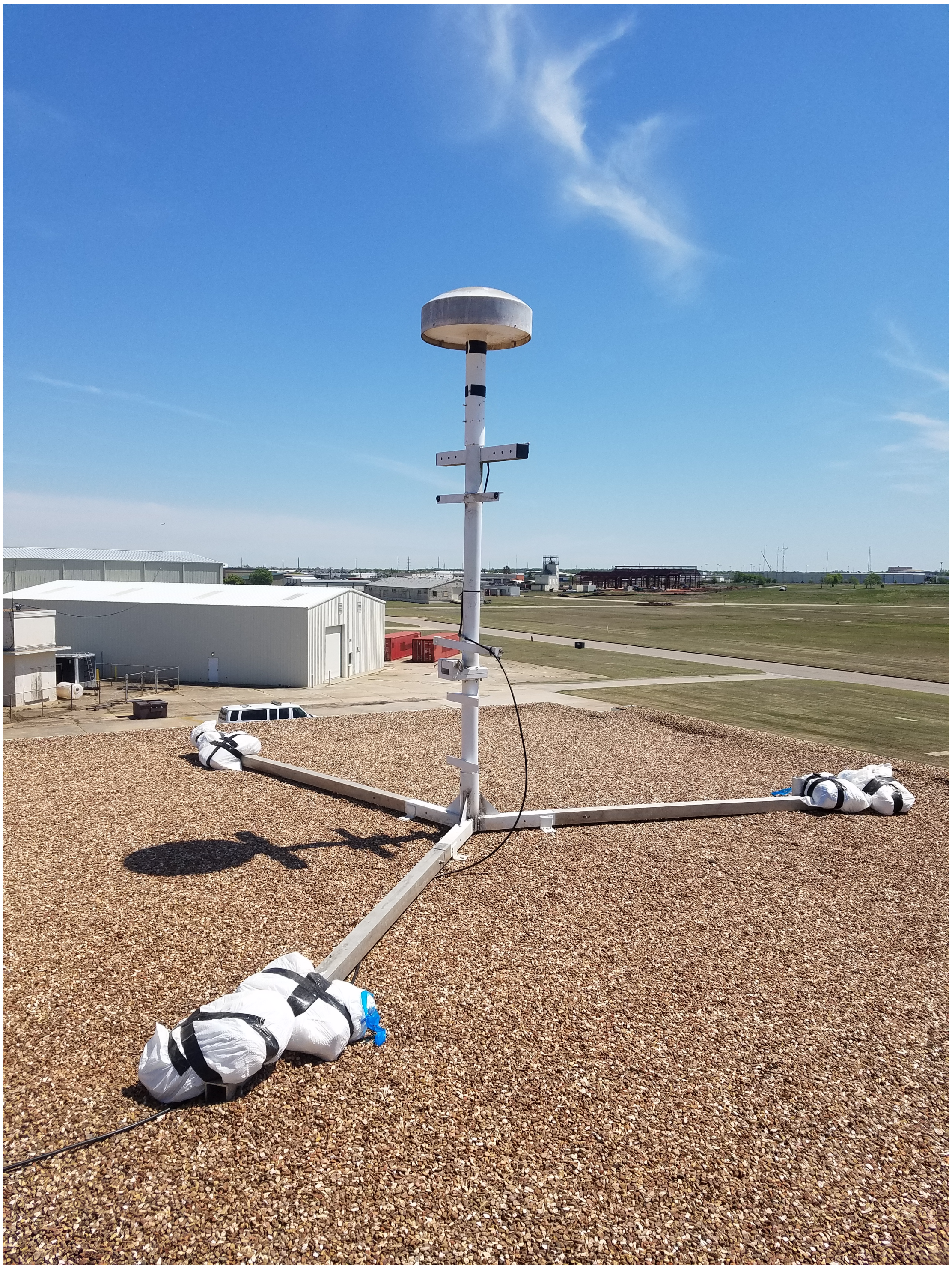


Figure C.3: The “Parking Lot Antenna” as labeled “A” in Figure C.1. Antennas labeled “B” and “C” are the same type of antenna.

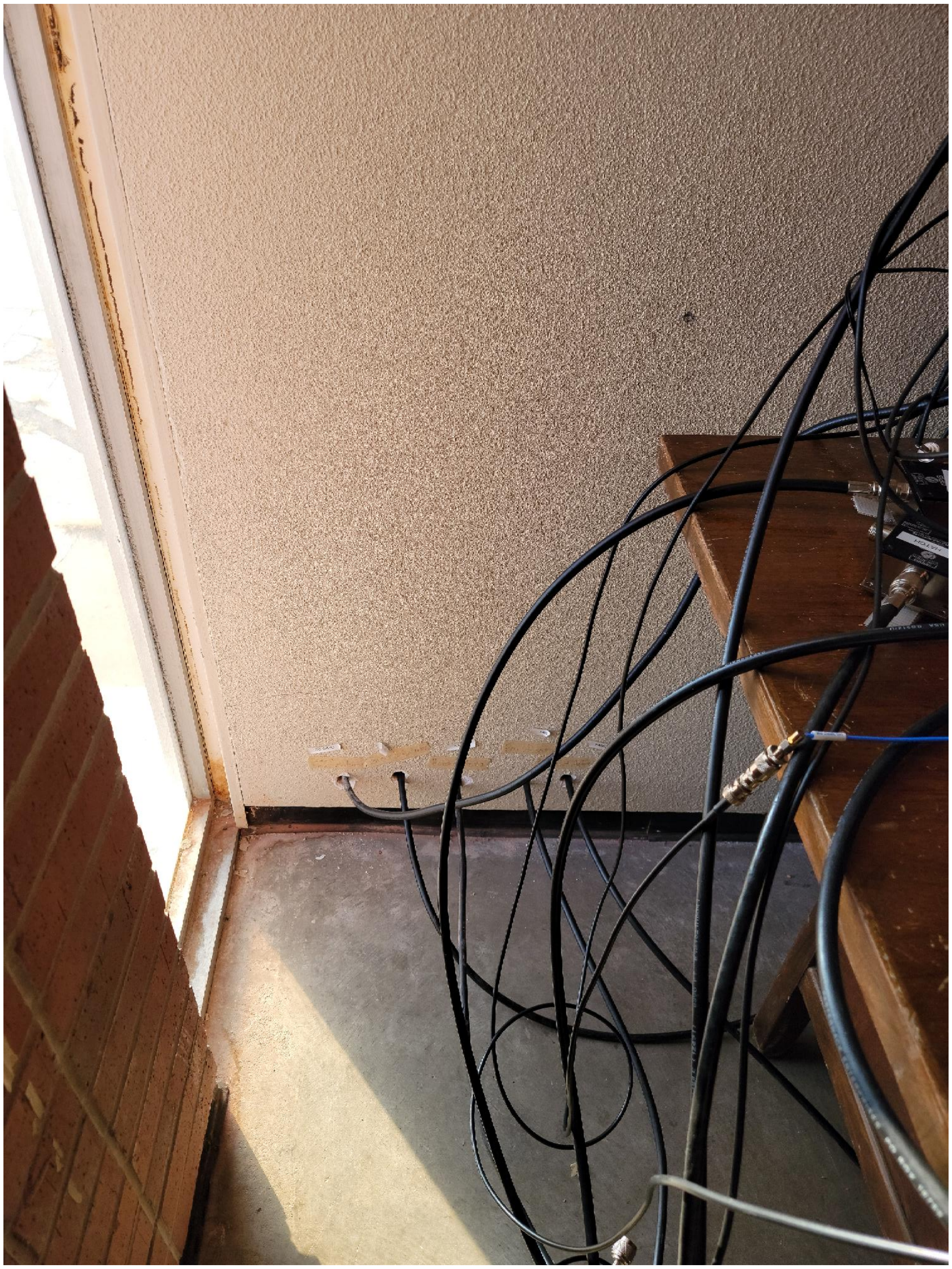


Figure C.4: Cables from the antennas located on the roof running into the GBAS room.

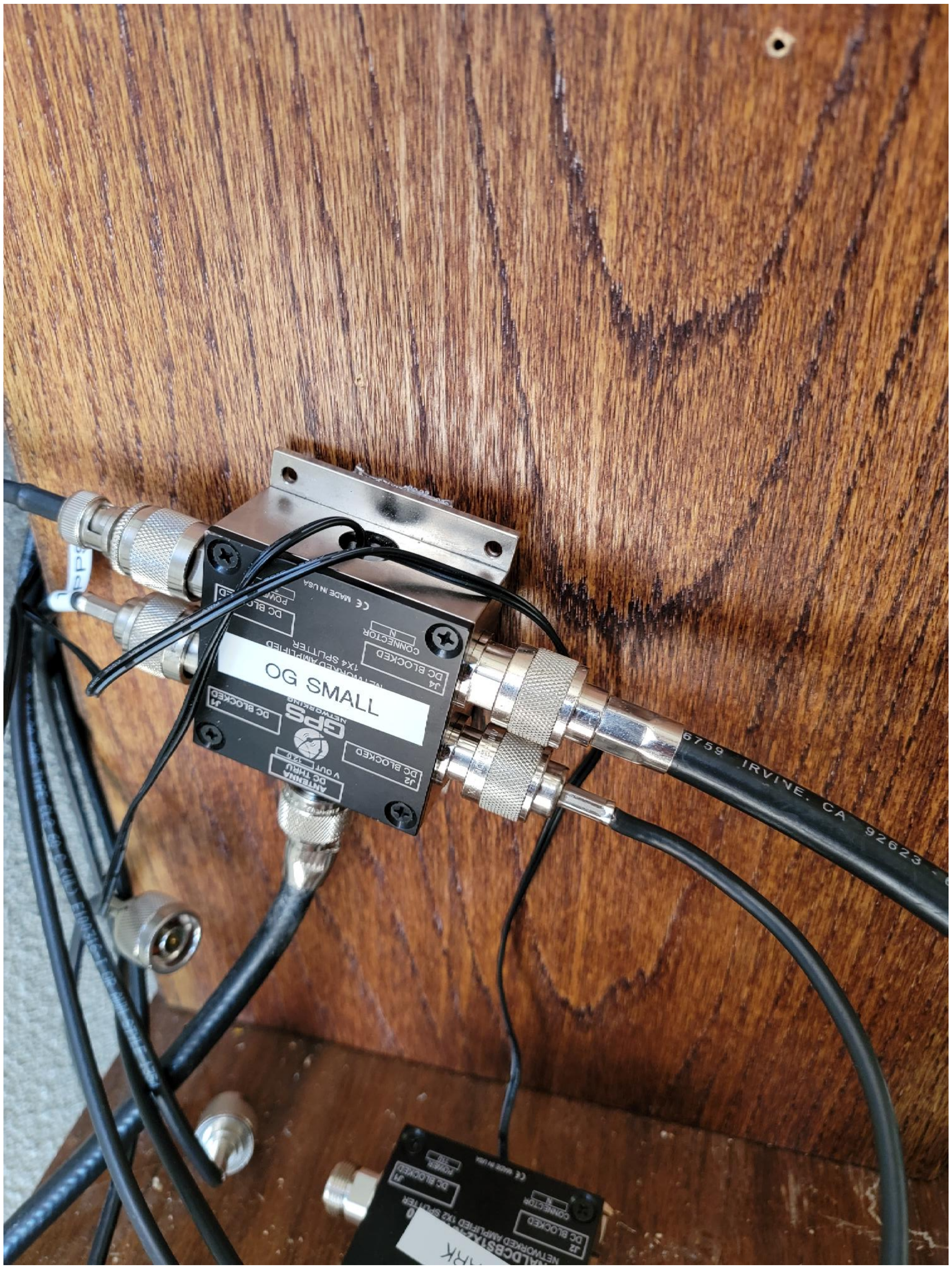


Figure C.5: Cables from the walls run directly to these GPS Networking active antenna splitters. In some instances, multiple 4x1 splitters are connected together to allow up to 7 different receivers to use the same antenna, as seen in the pseudorange testing in Chapter 4.

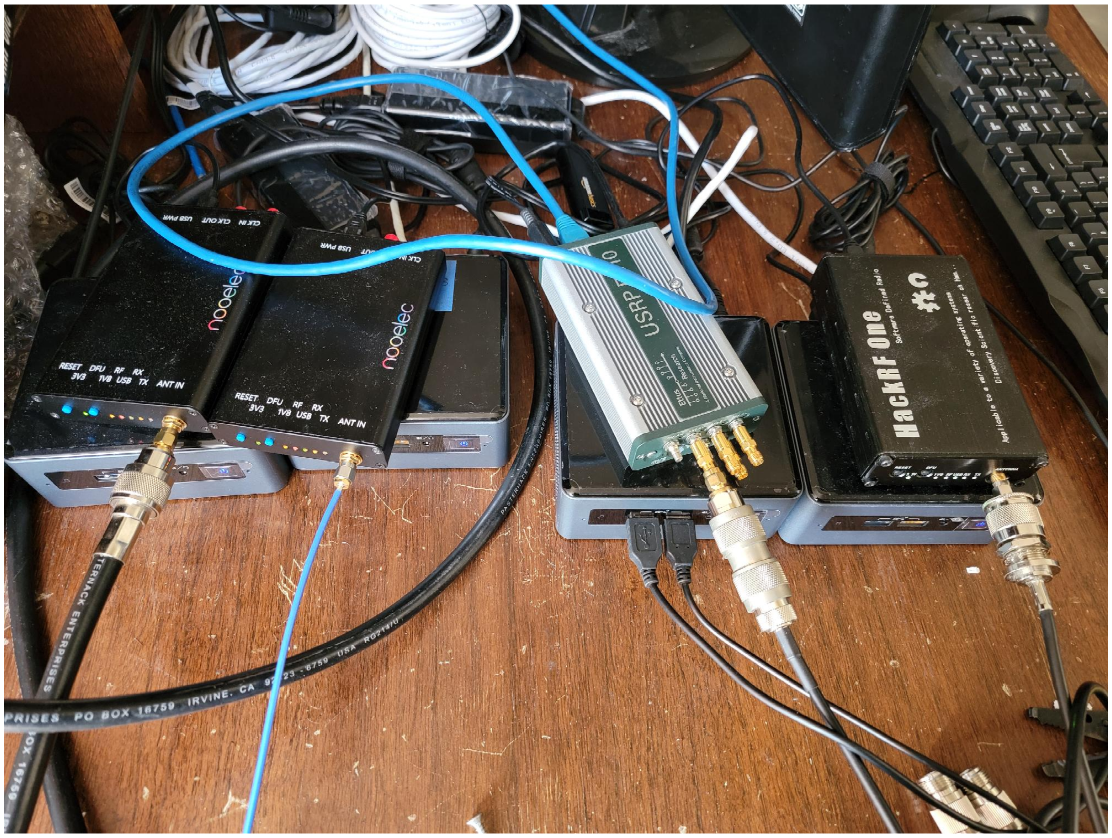


Figure C.6: GBAS SDR Setup. Four Intel NUCs are seen with three HackRF Ones and one USRP E310 on top of them.

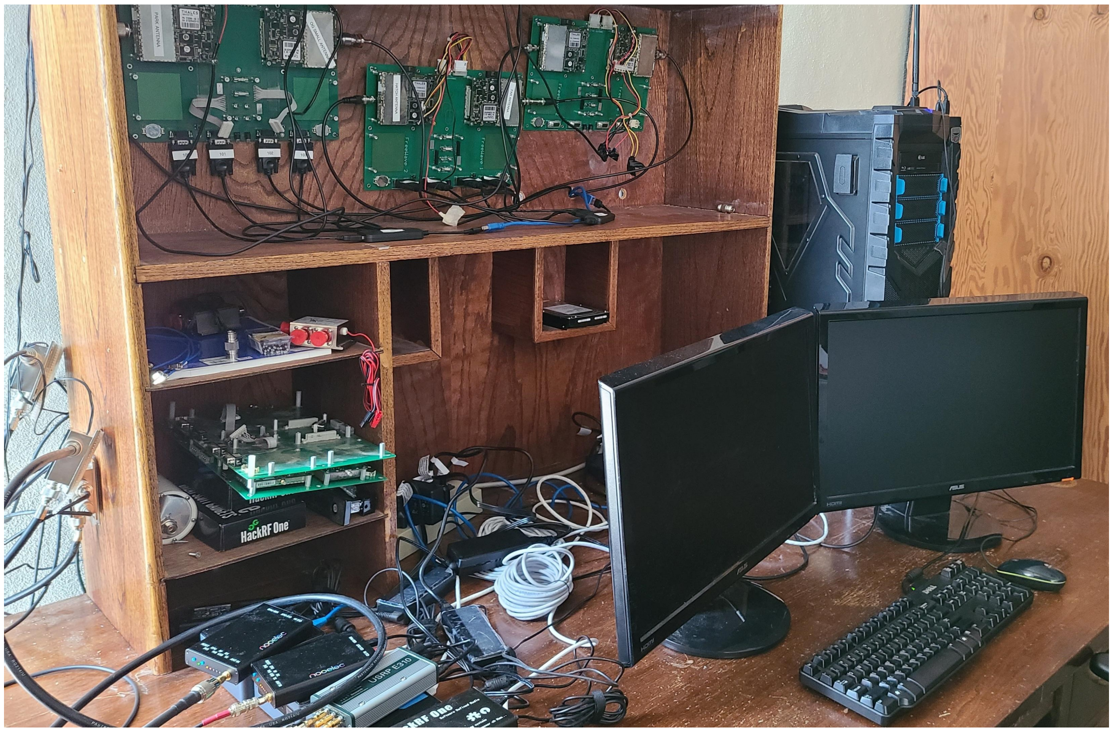


Figure C.7: GBAS Ground Base Station setup. Mounted on the back wall are the GG12s connected to the GBAS PC via multiple serial to USB cables. The SDRs are connected to their respective Intel NUCs at the lower left which are connected via a network switch to the GBAS PC.

Appendix D

HackRF One Configuration File

```
[GNSS-SDR]

;######### GLOBAL OPTIONS ##################
GNSS-SDR.internal_fs_sps=2000000
GNSS-SDR.telecommand_enabled=true
GNSS-SDR.telecommand_tcp_port=3333

;######### SIGNAL_SOURCE CONFIG ############
SignalSource.implementation=Osmosdr_Signal_Source
SignalSource.item_type=gr_complex
SignalSource.sampling_frequency=2000000
SignalSource.freq=1575420000
SignalSource.gain=40
SignalSource.rf_gain=40
SignalSource.if_gain=50
SignalSource.AGC_enabled=false
SignalSource.samples=0
SignalSource.repeat=false
;# Next line enables the internal HackRF One bias (3.3 VDC)
SignalSource.osmosdr_args=hackrf=0,bias=0
SignalSource.enable_throttle_control=false
SignalSource.dump=false
SignalSource.dump_filename=./signal_source.dat

;######### SIGNAL_CONDITIONER CONFIG ############
SignalConditioner.implementation=Signal_Conditioner

;######### DATA_TYPE_ADAPTER CONFIG ############
DataTypeAdapter.implementation=Pass_Through

;######### INPUT_FILTER CONFIG ############
```

```
InputFilter.implementation=Freq_Xlating_Fir_Filter
InputFilter.decimation_factor=1
InputFilter.input_item_type=gr_complex
InputFilter.output_item_type=gr_complex
InputFilter.taps_item_type=float
InputFilter.number_of_taps=5
InputFilter.number_of_bands=2
InputFilter.band1_begin=0.0
InputFilter.band1_end=0.85
InputFilter.band2_begin=0.9
InputFilter.band2_end=1.0
InputFilter.ampl1_begin=1.0
InputFilter.ampl1_end=1.0
InputFilter.ampl2_begin=0.0
InputFilter.ampl2_end=0.0
InputFilter.band1_error=1.0
InputFilter.band2_error=1.0
InputFilter.filter_type=bandpass
InputFilter.grid_density=16
InputFilter.dump=false
InputFilter.dump_filename=../data/input_filter.dat

;######### RESAMPLER CONFIG ############
Resampler.implementation=Pass_Through

;######### CHANNELS GLOBAL CONFIG ############
Channels_1C.count=32
Channels.in_acquisition=1
Channel.signal=1c

;######### ACQUISITION GLOBAL CONFIG ############
Acquisition_1C.implementation=GPS_L1_CA_PCPS_Acquisition
Acquisition_1C.item_type=gr_complex
Acquisition_1C.coherent_integration_time_ms=1
Acquisition_1C.pfa=0.01
Acquisition_1C.doppler_max=30000
Acquisition_1C.doppler_step=150
Acquisition_1C.max_dwells=1
Acquisition_1C.dump=false
Acquisition_1C.dump_filename=./acq_dump.dat

;######### TRACKING GLOBAL CONFIG ############
Tracking_1C.implementation=GPS_L1_CA_DLL_PLL_Tracking
```

```
Tracking_1C.item_type=gr_complex
Tracking_1C.extend_correlation_symbols=10
Tracking_1C.early_late_space_chips=0.5
Tracking_1C.early_late_space_narrow_chips=0.15
Tracking_1C.pll_bw_hz=40
Tracking_1C.dll_bw_hz=2.0
Tracking_1C.pll_bw_narrow_hz=5.0
Tracking_1C.dll_bw_narrow_hz=1.50
Tracking_1C.fll_bw_hz=10
Tracking_1C.enable_fll_pull_in=true
Tracking_1C.enable_fll_steady_state=false
Tracking_1C.dump=false
Tracking_1C.dump_filename=tracking_ch_

;######### TELEMETRY DECODER GPS CONFIG ############
TelemetryDecoder_1C.implementation=GPS_L1_CA_Telemetry_Decoder
TelemetryDecoder_1C.dump=false

;######### OBSERVABLES CONFIG ############
Observables.implementation=Hybrid_Observables
Observables.dump=false
Observables.dump_filename=./observables.dat
Observables.enable_carrier_smoothing=true

;######### PVT CONFIG ############
PVT.implementation=RTKLIB_PVT
PVT.positioning_mode=PPP_Static
PVT.output_rate_ms=100
PVT.display_rate_ms=500
PVT.iono_model=Broadcast
PVT.trop_model=Saastamoinen
PVT.flag_rtcm_server=true
PVT.flag_rtcm_tty_port=false
PVT.rtcm_dump_devname=/dev/pts/1
PVT.rtcm_tcp_port=2101
PVT.rtcm_MT1019_rate_ms=5000
PVT.rtcm_MT1077_rate_ms=1000
PVT.rinex_version=2
PVT.output_enabled=true
PVT.enable_monitor=true
PVT.enable_protobuf=true
PVT.monitor_client_addresses=127.0.0.1_10.10.10.1
PVT.monitor_udp_port=1142
PVT.enable_monitor_ephemeris=true
PVT.monitor_ephemeris_client_addresses=127.0.0.1_10.10.10.1
PVT.monitor_ephemeris_udp_port=1143
PVT.elevation_mask=0



;######### MONITOR CONFIG ############
Monitor.enable_monitor=true
Monitor.enable_protobuf=true
Monitor.decimation_factor=25
Monitor.client_addresses=127.0.0.1_10.10.10.1
Monitor.udp_port=1141
```

Appendix E

GBAS Network Configuration

To properly send UDP messages from one PC to another, the PCs need to have an established network connection. The simplest way to do this is by connecting them by ethernet cables. The only issue with this is that you have to manually configure the IP addresses so that the PCs actually see each other. You will need to do this on both the Ubuntu and Windows 10 PC, so here are the steps to do it:

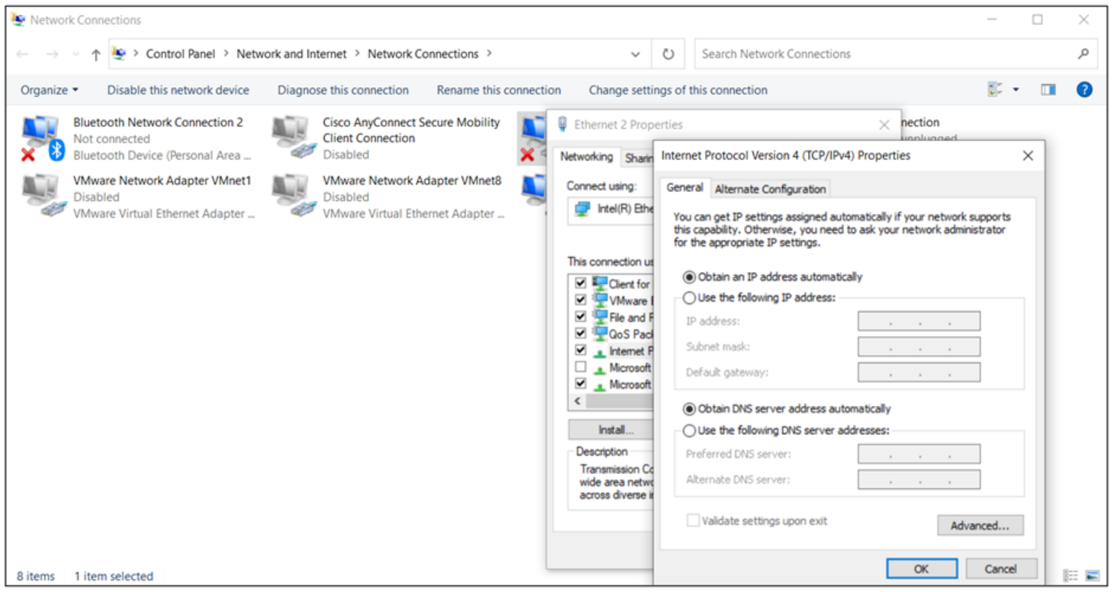
Windows 10:

1. Click the Windows logo at the bottom right and type "control panel" and select the result to open the Control Panel.
2. Click Network and Internet and then Networking and Sharing Center.
3. On the left side of the window, click "Change adapter settings".
4. Double-click the Ethernet adapter to bring up its options.
5. In the scroll list, double-click "Internet Protocol Version 4 (TCP/IPv4)".
6. Select "Use the following IP address" and enter the following as seen in Figure E.1:
    (a) IP address: 10.10.10.1
    (b) Subnet Mask: 255.255.255.0
    (c) Default gateway: 10.10.10.10
7. Click "OK" on every window to save the settings.

Ubuntu:

1. On the top right corner, click the network interface icon and choose "Wired settings" in the dropdown box.

2. Underneath “Wired”, choose the little gear icon to open the settings.
3. Click “Manual” for IPv4 Method and then enter the following as seen in Figure E.1:
    (a) Address: 10.10.10.2 (must be different from every other PC)
    (b) Netmask: 255.255.255.0
    (c) Gateway: 10.10.10.10
4. Click “Apply” to save the settings.


Network Connections
Control Panel > Network and Internet > Network Connections >
Search Network Connections
Organize
Disable this network device
Diagnose this connection
Rename this connection
Change settings of this connection
Bluetooth Network Connection 2
Not connected
Bluetooth Device (Personal Area ...
Cisco AnyConnect Secure Mobility Client Connection
Disabled
VMware Network Adapter VMnet1
Disabled
VMware Virtual Ethernet Adapter ...
VMware Network Adapter VMnet8
Disabled
VMware Virtual Ethernet Adapter ...
8 items
1 item selected
Ethernet 2 Properties
Networking
Connect using:
Intel(R) Ethe
Install...
Internet Protocol Version 4 (TCP/IPv4) Properties
General
Alternate Configuration
Advanced...
OK
Cancel

You can get IP settings assigned automatically if your network supports this capability. Otherwise, you need to ask your network administrator for the appropriate IP settings.

[✓ Obtain an IP address automatically]
Use the following IP address:
IP address: . . .
Subnet mask: . . .
Default gateway: . . .

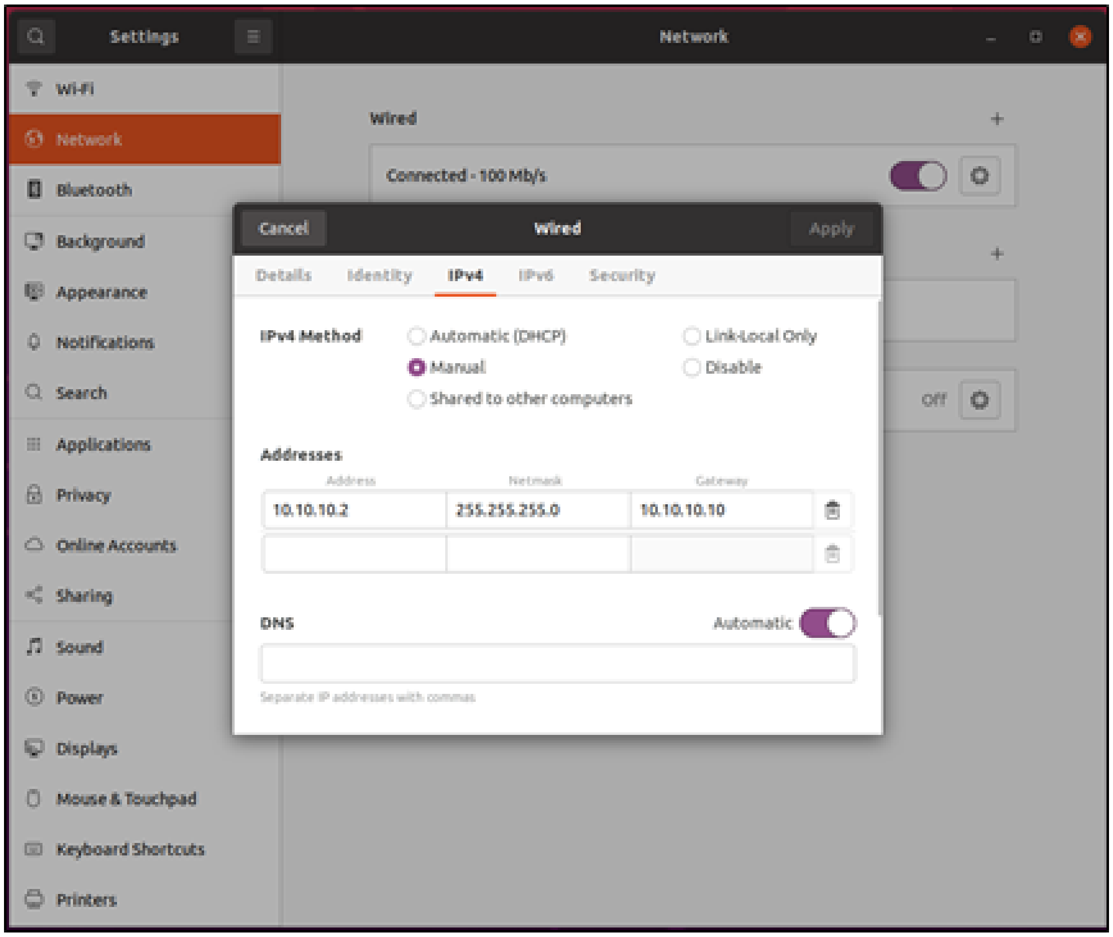
[✓ Obtain DNS server address automatically]
Use the following DNS server addresses:
Preferred DNS server: . . .
Alternate DNS server: . . .

☐ Validate settings upon exit

Figure E.1: Properly configuring the IP address on Windows 10.

To check that the computers are connected and configured properly, you can open the command prompt (Windows) or terminal (Linux) and ping the other’s IP address by entering “ping #.#.#.#”. For example, if I am on the computer with IP address 10.10.10.2 and I want to check the connection to 10.10.10.1, then I’d type “ping 10.10.10.1”. You should receive data back. Sometimes the ping command on Windows needs to be allowed through. You can temporarily disable the Firewall to check the ping commands are being sent and received between the computers. Make sure to re-enable the Firewall afterwards.

Once done changing the IP addresses, these computers will not be able to connect to the internet via ethernet cable. This is because their IP addresses and gateways are not correct for connecting to the internet properly. To change this back, follow the previous steps for changing the IPv4 settings but choose automatic instead of manual. However, it is still possible to connect to the internet through Wi-Fi after manually changing the IP settings. This is because the Wi-Fi



IPv4 Method: Automatic (DHCP) [✓ Manual] Shared to other computers Link-Local Only Disable

Addresses

Address	Netmask	Gateway
10.10.10.2	255.255.255.0	10.10.10.10

DNS Automatic [✓ On]

Separate IP addresses with commas

Figure E.2: Properly configuring the IP address on Ubuntu.

connection uses a different network adapter than the ethernet port and both can run at the same time, although you may have to select the Wi-Fi as your active connection.

Appendix F

Setting Up the USRP E310 and GNSS-SDR

The starting point for several of the steps taken in this setup procedure starts with the setup manual on the Ettus Research website in [42]. Although the website had very detailed instructions, additional steps were required due to errors that were encountered attempting to integrate the USRP E310 with the GNSS-SDR software. The E310 is ran on a Ubuntu Linux operating system so that it would be able to easily run alongside GNSS-SDR with no compatibility issues. The majority of the following information is from the final paper written for an undergraduate class research project in [43].

The first step in setting up the E310 is to build and install the UHD software on the host computer controlling the E310. This requires opening the command terminal and running the following commands:

1. sudo apt-get update
2. sudo apt-get upgrade
3. sudo apt-get -y install autoconf automake build-essential ccache cmake cpufrequtils doxygen ethtool fort77 g++ gir1.2-gtk-3.0 git gobject-introspection gpsd gpsd-clients inetutils-tools libasound2-dev libboost-all-dev libcomedi-dev libcppunit-dev libfftw3-bin libfftw3-dev libfftw3-doc libfontconfig1-dev libgmp-dev libgps-dev libgsl-dev liblog4cpp5-dev libncurses5 libncurses5-dev libpulse-dev libqt5opengl5-dev libqwt-qt5-dev libsdl1.2-dev libtool libudev-dev libusb-1.0-0 libusb-1.0-0-dev libusb-dev libxi-dev libxrender-dev libzmq3-dev libzmq5 ncurses-bin python3-cheetah python3-click python3-click-plugins python3-click-threading python3-dev python3-docutils python3-gi python3-gi-cairo python3-gps python3-lxml python3-mako python3-numpy python3-numpy-dbg python3-opengl python3-pyqt5 python3-requests python3-scipy python3-setuptools python3-six python3-sphinx python3-yaml python3-zmq python3-ruamel.yaml swig wget

4. sudo apt-get install build-essential cmake git pkg-config libboost-dev libboost-date-time-dev libboost-system-dev libboost-filesystem-dev libboost-thread-dev libboost-chrono-dev libboost-serialization-dev libboost-program-options-dev libboost-test-dev liblog4cpp5-dev libblas-dev liblapack-dev libarmadillo-dev libgflags-dev libgoogle-glog-dev libhdf5-dev libgnutls-openssl-dev libmatio-dev libpugixml-dev libpcap-dev libprotobuf-dev protobuf-compiler libgtest-dev googletest python3-mako python3-six
5. git clone https://github.com/EttusResearch/uhd
6. cd uhd/host
7. git switch UHD-3.9.LTS
8. mkdir build
9. cd build
10. cmake -DENABLE_E300=ON ../
11. sake
12. sake test
13. sudo make install
14. sudo ldconfig

At this point, if everything has been installed correctly, you should be able to run the following command and get the version number of the UHD software installed:

1. uhd_find_devices

The next major step is to flash the SD card with the correct image for E310. This step requires downloading the correct driver for the E310, which can be found online at the USRP website: `https://files.ettus.com/e3xx\_images/e3xx-release-4/ettus-e3xx-sg3/`. The image used in this project is "sdimage-gnuradio-demo.direct.xz".

Once that image has been downloaded, the SD card must be flashed with the image using whichever software tools are available. For this setup, we used the tool Bmap-tool whose documentation can be found on the Ubuntu website in [44]. The following commands will install bmap-tool and also flash the SD with the downloaded image, which may take up to thirty minutes to complete.

1. sudo apt-get install -y bmap-tools

2. sudo bmaptool copy sdimage-gnuradio-demo.direct.xz /dev/<SD DEVICE NAME> –nobmap

After inserting the properly configured SD card into the E310, powering it on, and connecting it the host PC via serial cable, it can be connected to and configured. The following command from above should show the E310 which is attached to the host PC.

1. uhd_find_devices

Now it is important to configure the E310 properly so that it will send its data over the network to the host PC. To connect to it, run the following command in the terminal (assuming /dev/tty/USB0 is the hardware location of the E310):

1. sudo screen /dev/ttyUSB0 115200

At this point, you may need to press the enter key, but eventually a new command prompt should appear asking for a username and password. The default username is root. There is not a default password. At this point, run the following commands to configure the E310's network settings and to turn on the network mode:

1. ifconfig eth0 10.10.10.4 netmask 255.255.255.0 up

2. usrp_e3x0_network_mode

This now has the E310 output over the IP address 10.10.10.4, which is desired for the network setup for this project.

The last thing required is to ensure GNSS-SDR has been built using the proper settings. To build.install it for the E310, ensure the following commands are ran:

1. cmake -DENABLE_E300=ON ../

2. make

3. sudo make install

4. sudo ldconfig

Now just ensure the GNSS-SDR configuration file has been setup properly to find the E310 which should be connected via Ethernet cable at the proper IP address.

Appendix G

USRP E310 Configuration File

```
[GNSS-SDR]

;######### GLOBAL OPTIONS ##################
GNSS-SDR.internal_fs_sps=1000000
GNSS-SDR.telecommand_enabled=true
GNSS-SDR.telecommand_tcp_port=3333

;######### SIGNAL_SOURCE CONFIG ############
SignalSource.implementation=UHD_Signal_Source
SignalSource.device_address= 10.10.10.103
SignalSource.item_type=cshort
SignalSource.sampling_frequency=1000000
SignalSource.freq=1575420000
SignalSource.gain=40
SignalSource.subdevice=A:A ; <- Can be A:0 or B:0
SignalSource.samples=0

;######### SIGNAL_CONDITIONER CONFIG ############
SignalConditioner.implementation=Signal_Conditioner

;######### DATA_TYPE_ADAPTER CONFIG ############
DataTypeAdapter.implementation=Pass_Through
DataTypeAdapter.item_type=cshort

;######### INPUT_FILTER CONFIG ############
InputFilter.implementation=Fir_Filter
InputFilter.input_item_type=cshort
InputFilter.output_item_type=gr_complex
InputFilter.taps_item_type=float
InputFilter.number_of_taps=11
InputFilter.number_of_bands=2
InputFilter.band1_begin=0.0
InputFilter.band1_end=0.48
InputFilter.band2_begin=0.52
InputFilter.band2_end=1.0

InputFilter.ampl1_begin=1.0
InputFilter.ampl1_end=1.0
InputFilter.ampl2_begin=0.0
InputFilter.ampl2_end=0.0

InputFilter.band1_error=1.0
InputFilter.band2_error=1.0

InputFilter.filter_type=bandpass
InputFilter.grid_density=16
InputFilter.sampling_frequency=1000000
InputFilter.IF=0

;######### RESAMPLER CONFIG ############
Resampler.implementation=Pass_Through

;######### CHANNELS GLOBAL CONFIG ############
Channels_1C.count=16
Channels.in_acquisition=1

;######### ACQUISITION GLOBAL CONFIG ############
Acquisition_1C.implementation=GPS_L1_CA_PCPS_Acquisition
Acquisition_1C.item_type=gr_complex
Acquisition_1C.coherent_integration_time_ms=1
Acquisition_1C.pfa=0.01
Acquisition_1C.doppler_max=5000
Acquisition_1C.doppler_step=250
Acquisition_1C.max_dwells=1
Acquisition_1C.dump=false
Acquisition_1C.dump_filename=./acq_dump.dat

;######### TRACKING GLOBAL CONFIG ############
;Tracking_1C.implementation=GPS_L1_CA_DLL_PLL_Tracking
;Tracking_1C.item_type=gr_complex
;Tracking_1C.extend_correlation_symbols=10
;Tracking_1C.early_late_space_chips=0.5
;Tracking_1C.early_late_space_narrow_chips=0.15
```

```
;Tracking_1C.pll_bw_hz=40
;Tracking_1C.dll_bw_hz=2.0
;Tracking_1C.pll_bw_narrow_hz=5.0
;Tracking_1C.dll_bw_narrow_hz=1.50
;Tracking_1C.fll_bw_hz=10
;Tracking_1C.enable_fll_pull_in=true
;Tracking_1C.enable_fll_steady_state=false
;Tracking_1C.dump=false
;Tracking_1C.dump_filename=tracking_ch_
Tracking_1C.implementation=GPS_L1_CA_DLL_PLL_Tracking
Tracking_1C.item_type=gr_complex
Tracking_1C.pll_bw_hz=30.0;
Tracking_1C.dll_bw_hz=4.0;
Tracking_1C.order=3;
Tracking_1C.early_late_space_chips=0.5;
Tracking_1C.dump=false
Tracking_1C.dump_filename=./tracking_ch_


;######### TELEMETRY DECODER GPS CONFIG ############
TelemetryDecoder_1C.implementation=GPS_L1_CA_Telemetry_Decoder

;######### OBSERVABLES CONFIG ############
Observables.implementation=Hybrid_Observables

;;######### PVT CONFIG ############
PVT.implementation=RTKLIB_PVT
PVT.positioning_mode=PPP_Static
;PVT.enable_rx_clock_correction=true
PVT.output_rate_ms=100
PVT.display_rate_ms=500
PVT.iono_model=Broadcast
PVT.trop_model=Saastamoinen
PVT.flag_rtcm_server=true
PVT.flag_rtcm_tty_port=false
PVT.rtcm_dump_devname=/dev/pts/1
PVT.rtcm_tcp_port=2101
PVT.rtcm_MT1019_rate_ms=5000
PVT.rtcm_MT1077_rate_ms=1000
PVT.output_enabled=true
PVT.rinex_version=2
PVT.enable_monitor=true
PVT.enable_protobuf=true
PVT.monitor_client_addresses=127.0.0.1_10.10.10.1
PVT.enable_monitor_ephemeris=true
PVT.monitor_ephemeris_client_addresses=127.0.0.1_10.10.10.1
PVT.monitor_ephemeris_udp_port=1133
PVT.elevation_mask=0
PVT.monitor_udp_port=1132


;######### MONITOR CONFIG ############
Monitor.enable_monitor=true
Monitor.enable_protobuf=true
Monitor.decimation_factor=25
Monitor.client_addresses=127.0.0.1_10.10.10.1
Monitor.udp_port=1131
```

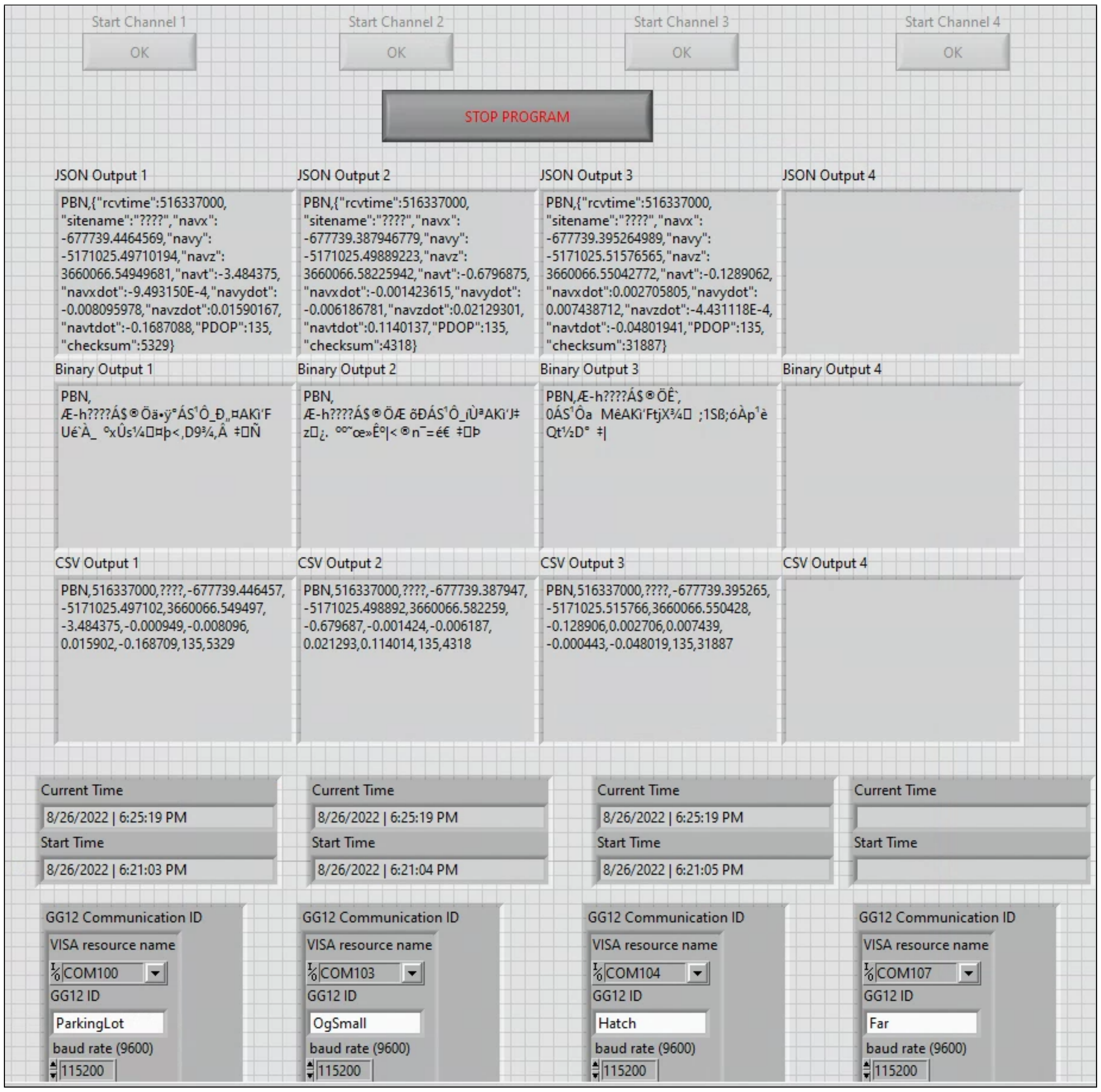
Appendix H

Combined Pseudorange Plots MATLAB Code

```
close all;
tbl = readtable("Combined File_20220826.csv");
tblArr = table2array(tbl);
x = tblArr(:,1);
FirstColumn_HRF3 = 2;
FirstColumn_HRF4 = FirstColumn_HRF3 + 32;
FirstColumn_HRF5 = FirstColumn_HRF4 + 32;
FirstColumn_HRF6 = FirstColumn_HRF5 + 32;
FirstColumn_GG1 = FirstColumn_HRF6 + 32;
FirstColumn_GG2 = FirstColumn_GG1 + 32;
FirstColumn_GG3 = FirstColumn_GG2 + 32;
FirstColumn_GG4 = FirstColumn_GG3 + 32;
% Index in the loop represents that satellite number: PRN<index>

for index = 1:32
    GG_avg(:,index) = mean([tblArr(:,FirstColumn_GG1 + index - 1)
    tblArr(:,FirstColumn_GG2 + index - 1)
    tblArr(:,FirstColumn_GG3 + index - 1)
    tblArr(:,FirstColumn_GG4 + index - 1)],2,"omitnan");
    y1 = tblArr(:,FirstColumn_HRF3 + index - 1);
    windowSize = 10;
    b = (1/windowSize)*ones(1,windowSize);

    y1 = y1 - GG_avg(:,index);
    y2 = tblArr(:,FirstColumn_HRF4 + index - 1) - GG_avg(:,index);
    y3 = tblArr(:,FirstColumn_HRF5 + index - 1) - GG_avg(:,index);
    y4 = tblArr(:,FirstColumn_HRF6 + index - 1) - GG_avg(:,index);
    y5 = tblArr(:,FirstColumn_GG1 + index - 1) - GG_avg(:,index);
    y6 = tblArr(:,FirstColumn_GG2 + index - 1) - GG_avg(:,index);
    y7 = tblArr(:,FirstColumn_GG3 + index - 1) - GG_avg(:,index);
    y8 = tblArr(:,FirstColumn_GG4 + index - 1) - GG_avg(:,index);
    figure(index);
    scatter(x,y1,'DisplayName','USRP E310','Marker','*');
    hold on;
    scatter(x,y2,'DisplayName','HackRF 4','Marker','*');
    scatter(x,y3,'DisplayName','HackRF 5','Marker','*');
    scatter(x,y4,'DisplayName','HackRF 6','Marker','*');
    scatter(x,y5,'DisplayName','GG12 1','Marker','.');
    scatter(x,y6,'DisplayName','GG12 2','Marker','.');
    scatter(x,y7,'DisplayName','GG12 3','Marker','.');
    scatter(x,y8,'DisplayName','GG12 4','Marker','.');
    hold off;
    legend('Location','northeast');

    title(['Pseudorange for PRN ', num2str(index)]);
    xlabel('Time (s)');
    ylabel('Pseudorange (m)');

    ylim([-10,40]);
xlim([514800,514900]);
    grid on;
end
```

Appendix I

GG12 Logger LabVIEW Code



JSON Output 1

```
PBN,{"rcvtime":516337000,
"sitename":"????","navx":
-677739.4464569,"navy":
-5171025.49710194,"navz":
3660066.54949681,"navt":-3.484375,
"navxdot":-9.493150E-4,"navydot":
-0.008095978,"navzdot":0.01590167,
"navtdot":-0.1687088,"PDOP":135,
"checksum":5329}
```

JSON Output 2

```
PBN,{"rcvtime":516337000,
"sitename":"????","navx":
-677739.387946779,"navy":
-5171025.49889223,"navz":
3660066.58225942,"navt":-0.6796875,
"navxdot":-0.001423615,"navydot":
-0.006186781,"navzdot":0.02129301,
"navtdot":0.1140137,"PDOP":135,
"checksum":4318}
```

JSON Output 3

```
PBN,{"rcvtime":516337000,
"sitename":"????","navx":
-677739.395264989,"navy":
-5171025.51576565,"navz":
3660066.55042772,"navt":-0.1289062,
"navxdot":0.002705805,"navydot":
0.007438712,"navzdot":-4.431118E-4,
"navtdot":-0.04801941,"PDOP":135,
"checksum":31887}
```

JSON Output 4

Binary Output 1

```
PBN,
Æ-h????Á$®Öä•ÿ°ÁS¹Ô_Ð„¤AKí'F
Ué`À_ °xÛs¼□¤þ<,D9¾,Â ‡□Ñ
```

Binary Output 2

```
PBN,
Æ-h????Á$®ÖÆ õÐÁS¹Ô_ıÙªAKí'J‡
z□¿. °°˜œ»Ê°|<®n¯=é€ ‡□Þ
```

Binary Output 3

```
PBN,Æ-h????Á$®ÕÊ`,
0ÁS¹Ôa MêAKí'FtjX¾□ ;1Sß;óÀp¹è
Qt½D° ‡|
```

Binary Output 4

CSV Output 1

```
PBN,516337000,????,-677739.446457,
-5171025.497102,3660066.549497,
-3.484375,-0.000949,-0.008096,
0.015902,-0.168709,135,5329
```

CSV Output 2

```
PBN,516337000,????,-677739.387947,
-5171025.498892,3660066.582259,
-0.679687,-0.001424,-0.006187,
0.021293,0.114014,135,4318
```

CSV Output 3

```
PBN,516337000,????,-677739.395265,
-5171025.515766,3660066.550428,
-0.128906,0.002706,0.007439,
-0.000443,-0.048019,135,31887
```

CSV Output 4

Current Time	8/26/2022 | 6:25:19 PM	8/26/2022 | 6:25:19 PM	8/26/2022 | 6:25:19 PM	
Start Time	8/26/2022 | 6:21:03 PM	8/26/2022 | 6:21:04 PM	8/26/2022 | 6:21:05 PM	

GG12 Communication ID

VISA resource name	COM100	COM103	COM104	COM107
GG12 ID	ParkingLot	OgSmall	Hatch	Far
baud rate (9600)	115200	115200	115200	115200

Figure I.1: Front panel of the GG12 Logger. This program gives live output feed to the controls in the front panel while also logging all of the GPS data to files. Each channel can be customized and turned on/off during runtime.

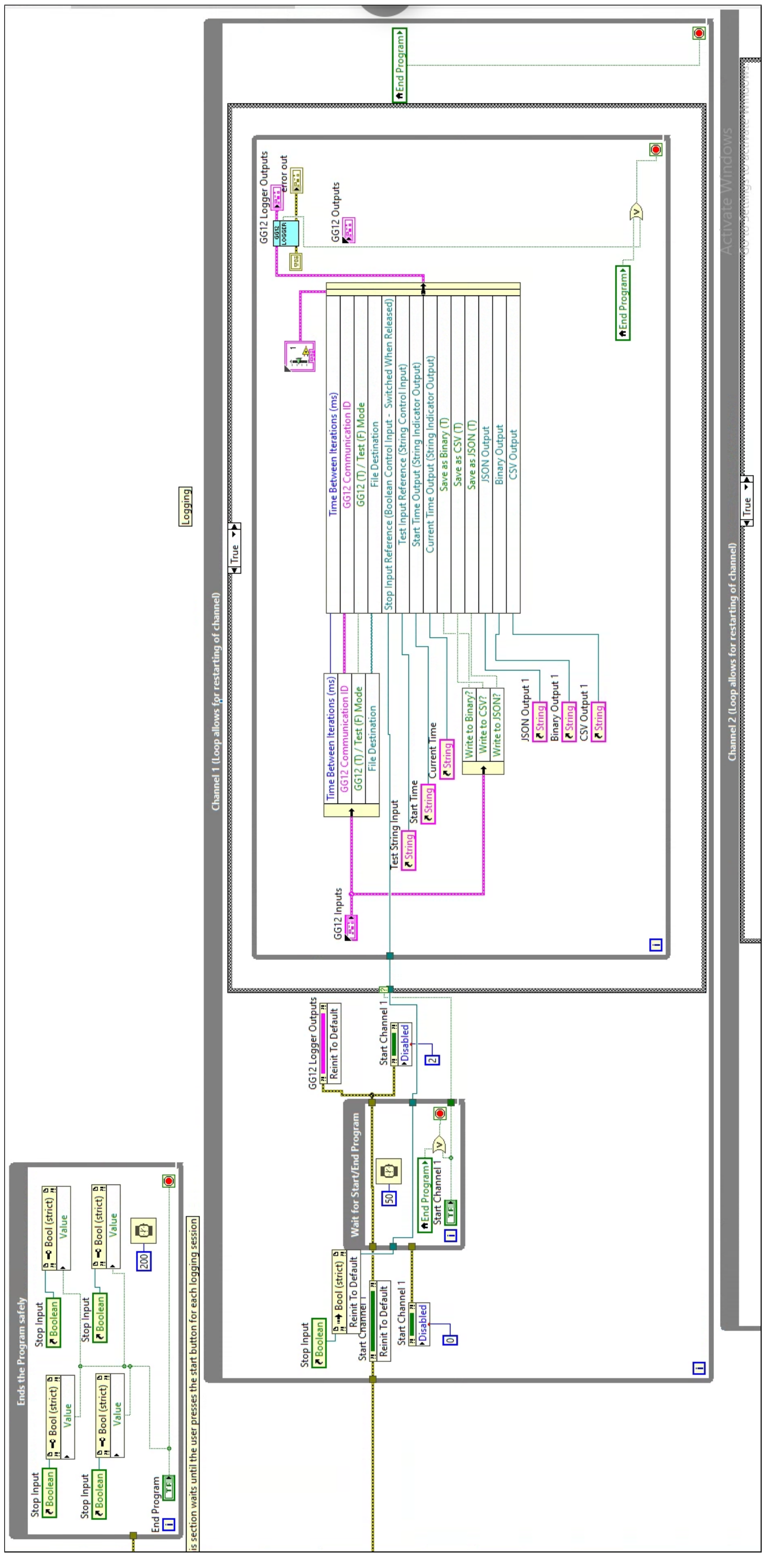
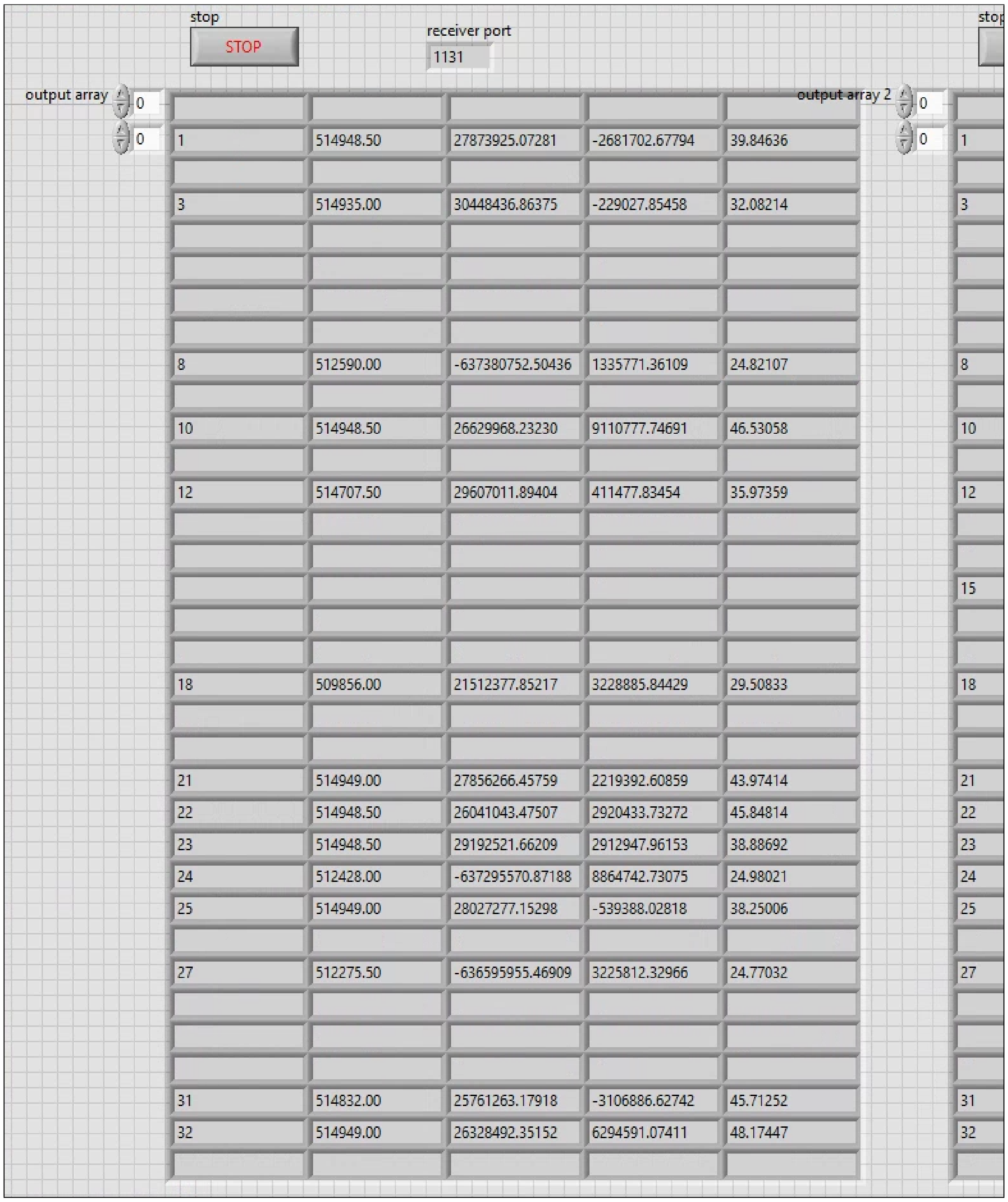


Figure I.2: The block diagram of the GG12 logger, showing only the code for the first channel.

Appendix J

GNSS-SDR Logger Code

The following LabVIEW screenshots show the code used to log the output from GNSS-SDR. The code first reads the UDP message, decodes it, reformats it, and then outputs the resulting data to a log file with the date, time, and receiver number.



receiver port 1131

output array 0 0

1	514948.50	27873925.07281	-2681702.67794	39.84636
3	514935.00	30448436.86375	-229027.85458	32.08214
8	512590.00	-637380752.50436	1335771.36109	24.82107
10	514948.50	26629968.23230	9110777.74691	46.53058
12	514707.50	29607011.89404	411477.83454	35.97359
18	509856.00	21512377.85217	3228885.84429	29.50833
21	514949.00	27856266.45759	2219392.60859	43.97414
22	514948.50	26041043.47507	2920433.73272	45.84814
23	514948.50	29192521.66209	2912947.96153	38.88692
24	512428.00	-637295570.87188	8864742.73075	24.98021
25	514949.00	28027277.15298	-539388.02818	38.25006
27	512275.50	-636595955.46909	3225812.32966	24.77032
31	514832.00	25761263.17918	-3106886.62742	45.71252
32	514949.00	26328492.35152	6294591.07411	48.17447

output array 2 0 0

1
3
8
10
12
15
18
21
22
23
24
25
27
31
32

Figure J.1: Front panel of the GNSS-SDR Logger. This program gives live output feed to the indicators in the front panel while also logging all of the GPS data to files.

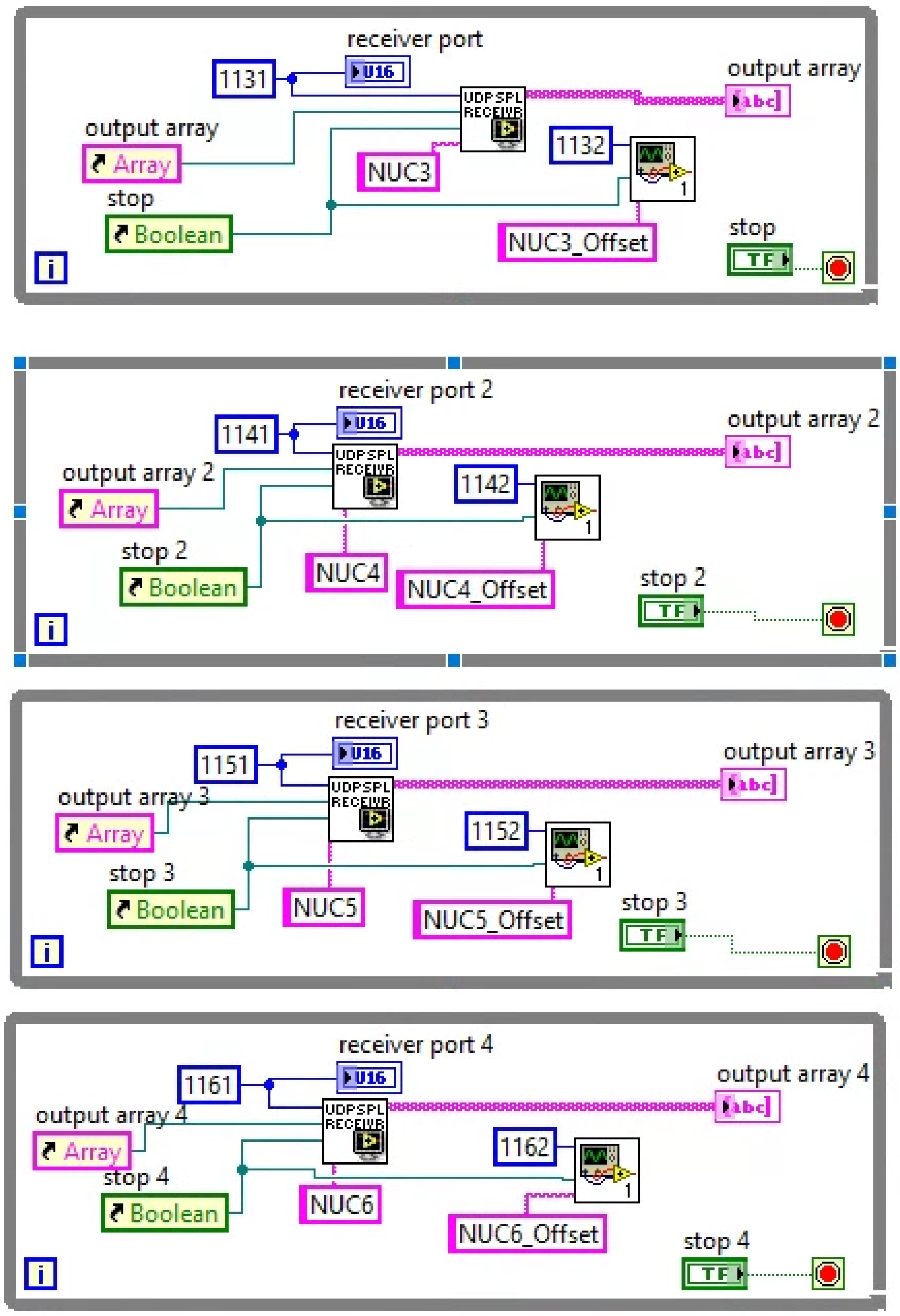
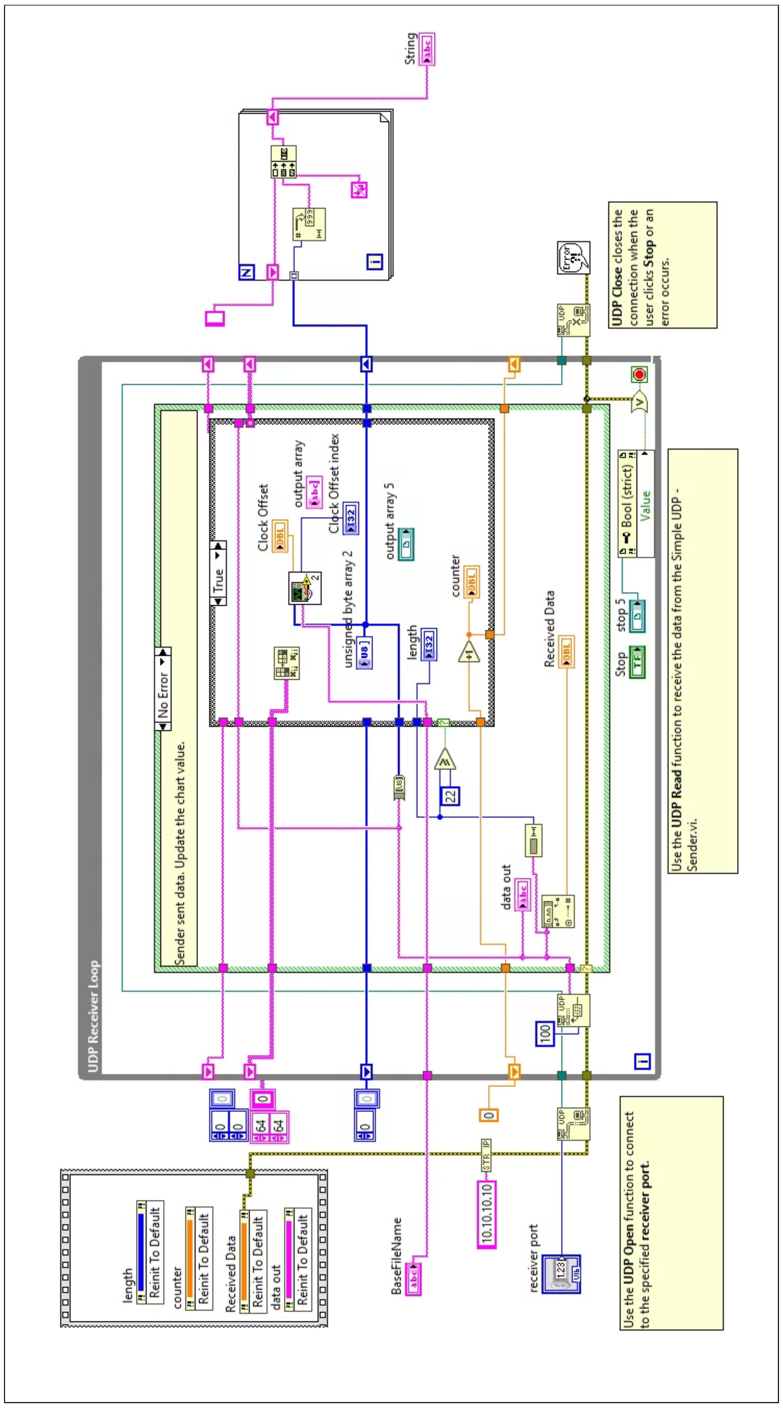


Figure J.2: Uppermost block diagram of the GNSS-SDR Logger. As can be seen, each channel is broken into four identical sections to utilize existing code. Each section runs the GNSS Synchro and GNSS Offset loggers.


Use the UDP Open function to connect to the specified receiver port.

Use the UDP Read function to receive the data from the Simple UDP - Sender.vi.

UDP Close closes the connection when the user clicks Stop or an error occurs.

Sender sent data. Update the chart value.

Figure J.3: This section reads the UDP messages from a specific port, parses the data, and then logs it in a specific format to a file. There is another similar file for parsing the GNSS Synchro data.

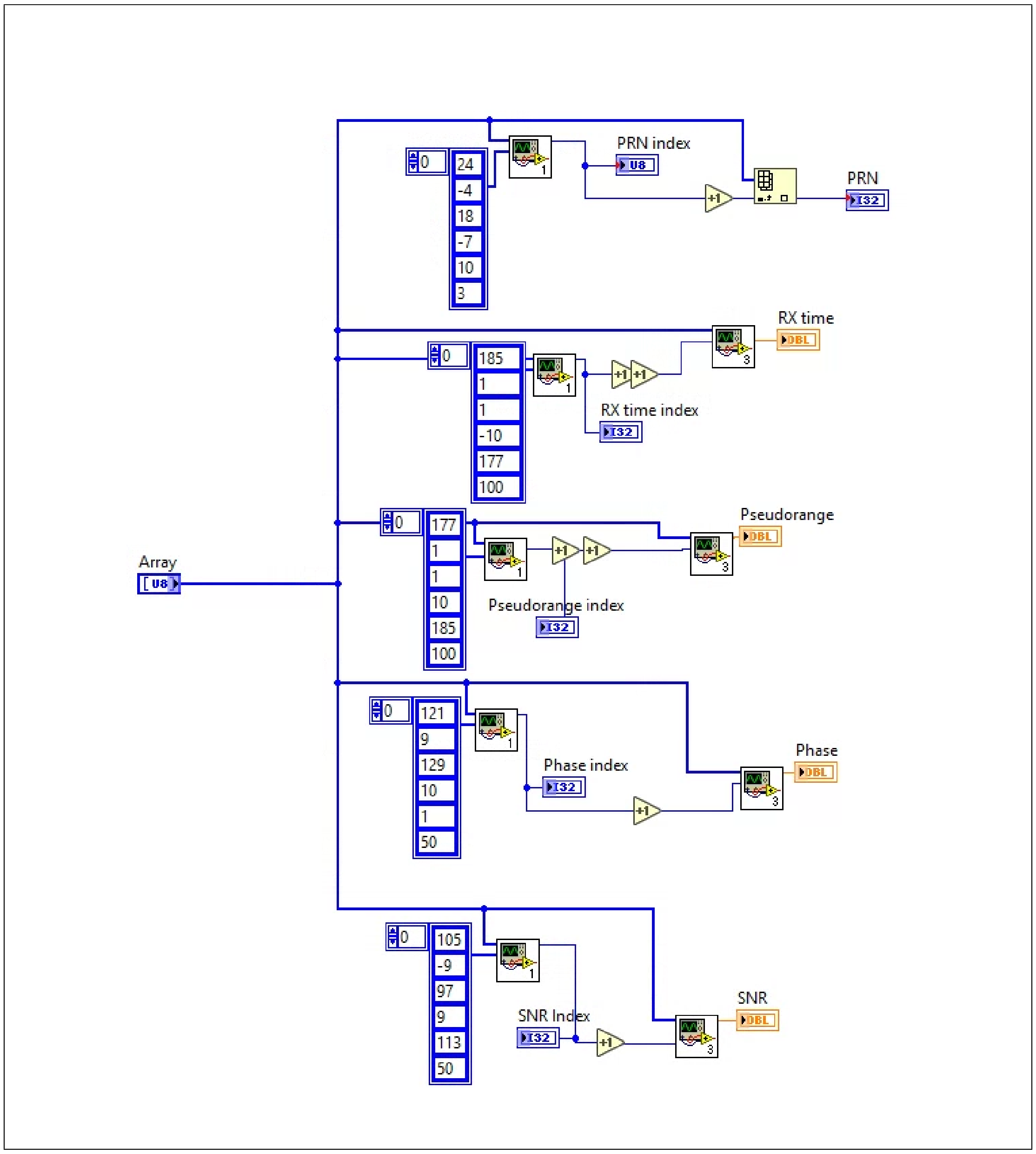


Figure J.4: This code will parse the protobuf message for the desired values.

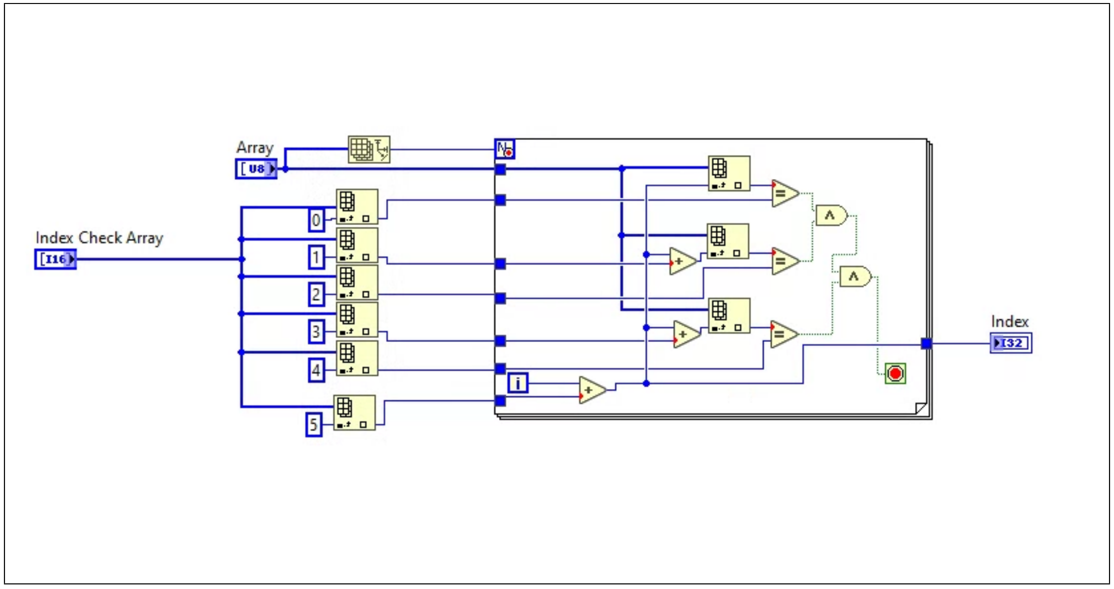


Figure J.5: This program takes a given protobuf message and finds the index of the desired data values within.

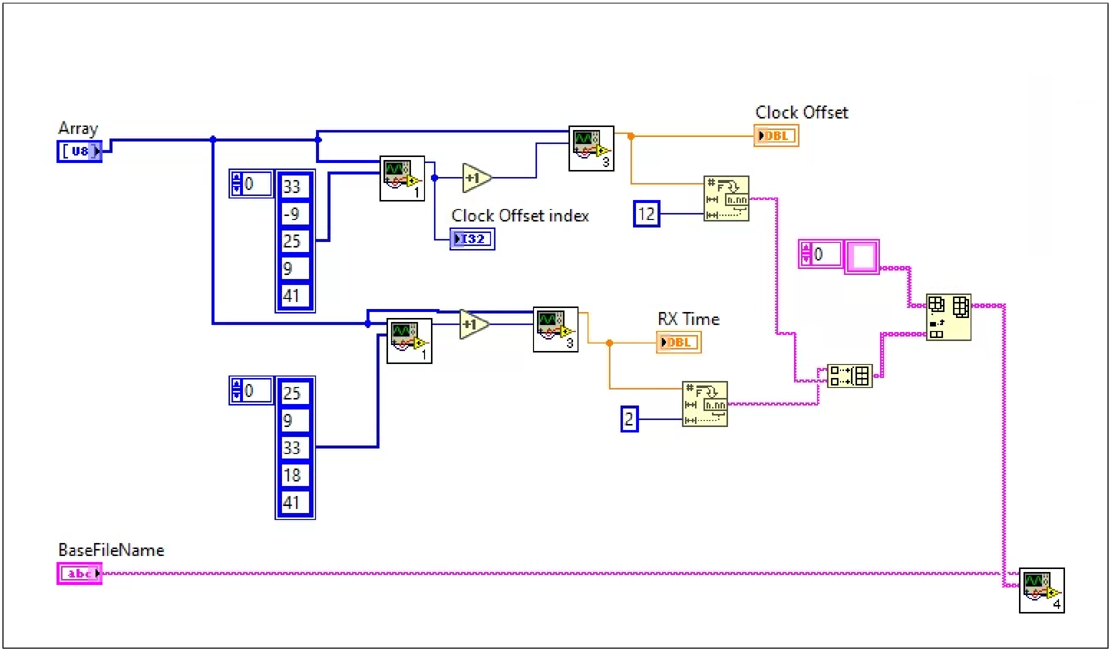


Figure J.6: This section uses the indexes found from the above program and converts the specific protobuf values to the proper types. In this case, the program converts the offset and receive time.

Appendix K

VE Plot MATLAB Code

```
%Grab data from imported log file
pcTime = GNSSSDROutputs.PCTime;
rxTime = GNSSSDROutputs.RXTime;
ve = GNSSSDROutputs.VE;
he = GNSSSDROutputs.HE;
sats = GNSSSDROutputs.SatellitesUsed;

bins = [-100000,x_pdf,100000];
x_output = [-14999:1:15000]
ve_real=[];
%this plot
for i=1:length(ve)
    j=0;
    v = ve(i);
    if (sats(i)<4)
        continue %Reject data not using at least 4 satellites.
    end
    if (isnan(v))
        continue; %Reject NaN values.
    end
    ve_real = [ve_real v];
end

h=histogram(ve_real,60,'Normalization','pdf');

%calculate pdf best-fit
pd = fitdist(ve_real',"Normal");
x_pdf = [-40:0.1:40];
y = pdf(pd,x_pdf);

line(x_pdf,y, 'Color',"red", 'LineWidth', 2)
```

```
title("GBAS VE PDF Using GNSS SDR");
xlabel("Vertical Error (m)");
ylabel("Density");
xlim([-20, 20]);
legend("Experimental Data","Best-Fit Curve",'','','');

data = ve_real;
numBins = 60;
% Compute mean and standard deviation.
mu = mean(data)
sigma = std(data)
% Indicate those on the plot.
xline(mu, 'Color', 'red', 'LineWidth', 2, 'LineStyle', '--', ...
    'HandleVisibility','off');
xline(mu - sigma, 'Color', 'black', 'LineWidth', 2, 'LineStyle', ...
    '--', 'HandleVisibility','off');
xline(mu + sigma, 'Color', 'black', 'LineWidth', 2, 'LineStyle', ...
    '--', 'HandleVisibility','off');
sMean = sprintf('  Mean = %.3f\n  SD = %.3f', mu, sigma);
% Position the text on the plot.
text(mu, .2, sMean, 'Color', 'r', ...
'FontWeight', 'bold', 'FontSize', 12, ...
'EdgeColor', 'b');
sMean2= sprintf('Histogram with %d bins.  Mean = %.3f.  , ...
   SD = %.3f' numBins, mu, sigma);
```