# System Manual

Assignment #7

Computer Science 481 - Spring 2018
**Instructor**: Yenemula Reddy
**Mentor**: Kevin Bandura
**Sponsor**: Luke Hawkins

Group 2: Synthetic Pulsar for Instrument Testing
**Members**: Dillion Bowman, Caleb McHenry, Adam Powell, Andrew Suggs

# Abstract of Project

A pulsar is one of the most dense objects in the universe. Imagine the mass of a red supergiant star squeezed into a ball with a diameter of about 10 kilometers. These extreme conditions cause the star to spin while emitting radio frequency waves from its two poles. These signals create a regular, repeating cadence of pulses that are observable on Earth. The Green Bank Observatory in Green Bank West Virginia is just one of many radio observatories that can detect these signals. The team at GBO wanted a way to simulate a pulsar signal using only software. The purpose was to test the hardware that is being used on the GBT (Green Bank Telescope). The significance of a software versus a hardware defined pulsar signal is that it can be run against their hardware for hours on end while requiring minimal storage. The trick to accomplish this is to create a single pulse, requiring up to about one Gigabyte of storage, that is infinitely repeatable while maintaining its absolute periodicity. Then sending that signal through a Digital to Analog Converter to simulate a pulsar cadence rather than storing a complete cadence of pulses that would potentially require Terabytes of storage. In order to best simulate a pulsar, there needs to be the ability to create different pulses with different properties such as the observed frequencies, the length of the pulse in time, dispersion measure, etc. Otherwise, there would only be a single, useless, use case. This project sets out to create a single, customizable, repeatable, pulse using only software.

# Table of Contents

# Introduction

The main aim of this project was to work with the PSS (PulsarSignalSimulator) framework to make repeatable snippets of pulsar signals, something which was titled “cadence snippets”. Along with this, a custom, “non-PSS”, simplified pulsar signal generator was created for the project. PSS is a collaboration to develop a software-based synthetic pulsar with a WVU LCSEE senior design group. It generates a cadence of pulses that need to be snipped into a single pulse.

# Design Achievements

## CLI

We created a simple command line interface for the user to use in order to create their pulsar file. The CLI is equipped with flags to aid with their user input. Details of the cli are provided in the user’s manual.

## Generate Repeatable Pulse

We were able to create a simulated pulse that was created with a normal distribution to represent the signal of a pulsar. This was done using PSS firstly, but the data from the PSS module seemed to be erroneous. Thus, we created our own pulsar generation module, achieving the same result with more customization and reliable data.

## Disperse Pulse

We currently only allow a dispersion that is locked into the period, but user is able to input what frequency they would like the pulse to start and end at to simulate different dispersion measures.

## Write Pulse to File

We write the pulse to a file to be compatible with GNU radio. It is 32 bit interleaved real and imaginary floats written from an array as binary values.

## Test with GNU Radio and a DAC

We were able to transmit the pulse file with the help of GNUradio and a DAC. GNUradio reads the binary data and sends it to the DAC which transmits the data as an analog receiver. We were also able to receive the signal with a separate receiver to confirm that everything functions properly and as expected.

# Complete Software design

## spit_arg_parser.py

In order to use command line flags with our project, we needed to have an argument parser. One parser for spit and another for pss

- Import dependencies
    - argparse
- Instantiate a new parser
    - argparse.ArgumentParser()
- Add all flags
    - add_argument()
    - samples, amplitude, begin, end, offset, file, peak, width, version, plot
- Parse the flags into a namespace
    - parse_args()

## pss_arg_parser.py

- Import dependencies
    - argparse
- Instantiate a new parser
    - argparse.ArgumentParser()
- Add all flags
    - add_argument()
    - frequency, bandwidth, period, file, version, plot
- Parse the flags into a namespace
    - parse_args()

## unit_exponent_parser.py

Allowing the user to use units in the input was important as these values can be very large. Units like MHz and 10e6 were necessary luxuries.

- Import dependencies
    - re
- Create a key value map
    - Keys are prefixes like “M” for Mega or “T” for Tera
    - Values are integer exponents associated with those prefixes
- Modify input String

    - 10 MHz => 10MHz
    - 20e2 => 20e2
    - 5e2 GHz => 500,000 MHz
- Check if input string matches one of two cases
    - A number followed by a prefix and a unit
        - 10MHz (prefix = M, unit = Hz)
        - 12ms (prefix = m, unit = s)
    - A number followed by e exponentiation
        - 10e6
        - 12e-3
- Apply exponentiation to the number if match occurs

## pss.py

The project began using solely PsrSigSim. Once the team learned how to use the module, it was a matter of a few lines of python code to snip this cadence of pulses down into one pulse based on its phase.

- Import dependencies
    - datetime, math, numpy, scipy, matplotlib.pyplot, VersionZeroPointZero, pss_arg_parser, unit_exponent_parser
- Parse through user input using pss_arg_parser
    - parse_args()
- Modify the arguments given using unit_exponent_parser
    - apply_unit_and_exponentiation()
- Create a signal with the modified arguments using VersionZeroPointZero
    - Signal(f0, bw, TotTime, SignalType, data_type)
- Turn that signal into a pulsar using VersionZeroPointZero
    - Pulsar(Signal)
- Make pulses from that pulsar using VersionZeroPontZero
    - make_pulses()
- Store the pulsar in a numpy array, convert to a byte array, write to binary file

## spit.py

After seeing how PSS behaved, we determined that it was not doing exactly what we wanted it to do. It seems that the sample rate of the signal is a static value and if you increase the frequency or period of the pulse, you lose an immense amount of data.

- Import dependencies
    - datetime, math, numpy, matplotlib.pyplot, spit_arg_parser, unit_exponent_parser
- Parse through user input using pss_arg_parser
    - parse_args()
- Modify the arguments given using unit_exponent_parser
    - apply_unit_and_exponentiation()

- Create a quadrature chirp of using a frequency swept sine/cosine wave
    - Over an array with a size of the number of samples the user inputs, starting at index “0”, apply an instantaneous frequency (ft) at each data point based on the beginning and ending frequencies the user gives.
    - Create a real and imaginary component at each data point
    - Convert the real and imaginary components to complex numbers and multiply each number by the amplitude the user gives.
- Create a gaussian pulse based on the quadrature chirp
    - Multiply the chirp by a normal/gaussian distribution based on a mean and standard deviation representing the peak and width of the pulse respectively.
    - Store the gaussian pulse in an array, convert to a byte array, write to binary file

# Software Diagrams

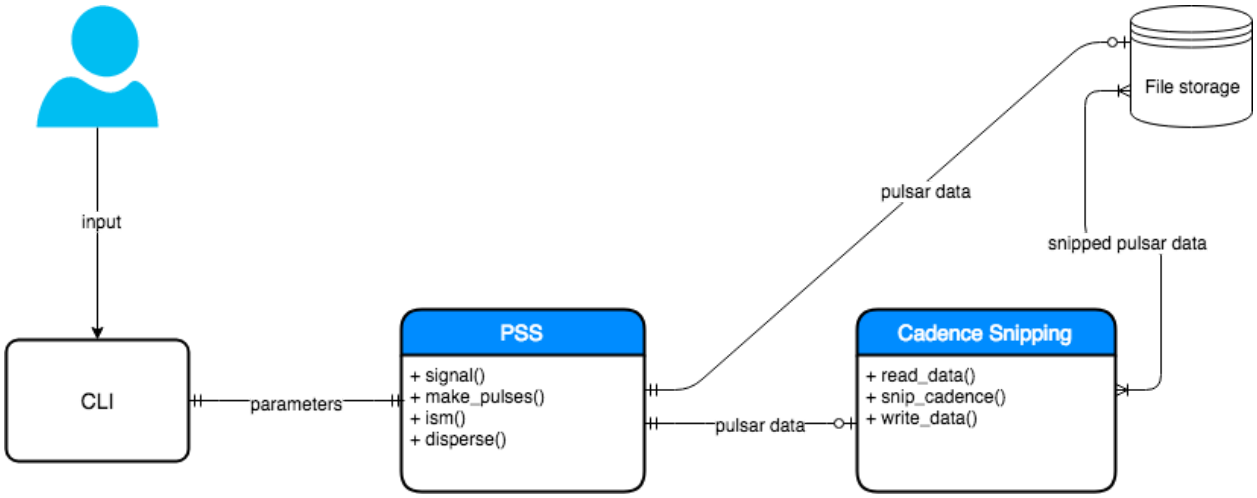


**Fig. 1: Original System architecture**

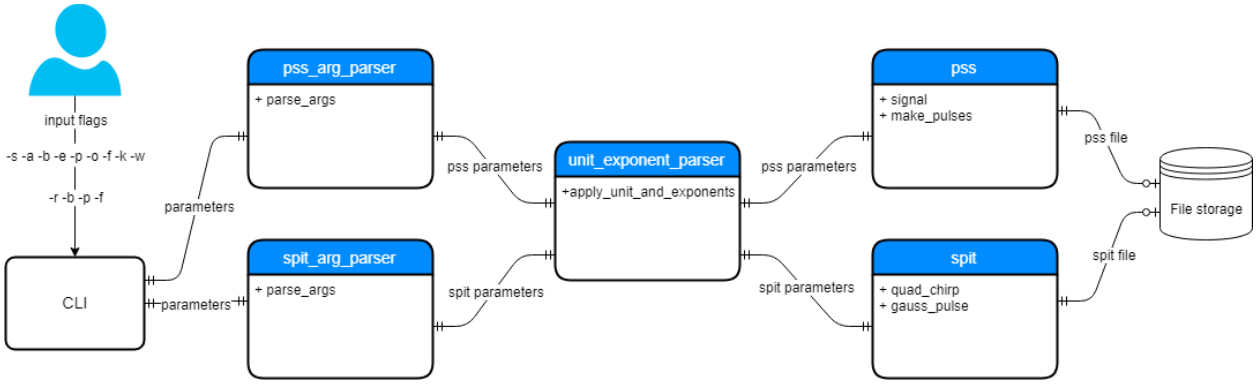


**Fig. 2: Actual System architecture**

# Source Code

https://github.com/amsuggs/SPIT

# Test Results

We tested in three main ways: matplotlib, GNU radio, and transmission with a DAC. For images of all three of these methods reference the Maintenance manual in Appendix 2. Matplotlib is python library that allows the user to visualize data with graphs. This is the first part of testing in our project. From here we can confirm the structure of our pulse. The second for of testing was by using a free software called GNU radio. This software allows you to read in binary files and it simulates pulses in real time. With this we could not only confirm that the pulses we wrote to file were correct, we could also test to make sure the pulse was realistic and repeatable. Finally the last for of testing we used was to physically transmit the signal with the help of a DAC. We could the receive the transmitted signal with a receiver and visual what was received. By comparing all three of these methods we are able to check the synthetic pulse at various stages of its life cycle. With these methods we were able to confirm that spit is capable of producing a signal that the green bank observatory can use to test their receivers.

# Safety Precautions

The signals created by this project are capable of being in a very wide range of frequencies and thus are possible, even likely, to be outside of legal range to be broadcast by individuals without proper licenses and could incur criminal charges against the broadcaster. Besides that, if the signal is transmitted in a bandwidth that requires certification, radio astronomers could be fooled into receiving this signal as a real extraterrestrial object.

# Reflections

Looking back on the project one of the largest issues occuring throughout the semester was working with pre-existing code in the PSS library. This is not fully to blame on either party, the developers of PSS nor our group of developers. First most of the domain of the project was unknown to our developers and there was a steep learning curve of the group to get up to speed to work with PSS. On the other side of the coin, PSS is still in a very early version (self titled: zero point zero) and because of this much of the functionality that was wanted with the SPIT project (high customizable sample rates for example) were not yet created and available for use. Moving ahead with the project we would highly lean toward development of a separate set of tools (as we have begun to do with our simple pulse generation) unless there is adequate development toward the PSS library being more usable for our specific uses. Along with development of a seperate, specialized toolset we would highly recommend onboarding

developers that have a larger knowledge of the domain. Software developers with knowledge of signals, physics, or higher level mathematics would be very beneficial to the project.

# Appendix 1 - User Manual

## Dependencies

1. python
2. numpy
3. scipy
4. matplotlib

To install python onto your machine go here https://www.python.org/downloads/ .
We recommend getting all the required python packages at once using anaconda https://www.anaconda.com/download/

## Getting Started

1. Clone the SPIT repository https://github.com/amsuggs/SPIT
2. Change to the root directory of the SPIT repository(cd)
3. Run the command "python spit.py" (note, this will use default values)
    a. Run "python spit.py" followed by optional flags
    b. Run "python pss.py" followed by optional flags
4. Output signal file will be in the "OutputFile" directory

## Command Line Flags

**Table 1: Command Line Flags used in spit.py**

| | | | |
|---|---|---|---|
| -s | --sample | Number of samples in the specified period (-p) | 8192 |
| -a | --amplitude | Amplitude of of the cosine/sine wave | 1.0 |
| -b | --begin | Beginning frequency of the frequency sweep. Defaultly measured in Hz but units can be added to specify a different unit prefix. Available unit prefixes: Y, Z, E, P, T, G, M, k, h, D, d, c, m, u, n, p, f, a, z, y | 20Hz |
| -e | --end | Ending frequency of the frequency sweep. Defaultly measured in Hz but units can be added to specify a different unit prefix. Available unit prefixes: Y, Z, E, P, T, G, M, k, h, D, d, c, m, u, n, p, f, a, z, y | 0 |
| -p | --period | Amount of time in the cycle of the pulse, measured in milliseconds | 1000ms |

| | | | |
|---|---|---|---|
| -o | --offset | Phase offset | 0.0 |
| -f | --file | Name for the output file | spit_pulse |
| -k | --peak | Second value of where the peak of the pulse occurs in the period | 500ms |
| -w | --width | The width of the pulse, measured in seconds | 100ms |

**Table 2: Command Line Flags used in pss.py**

| | | | |
|---|---|---|---|
| -r | --frequency | Central frequency of the pulse | 1400 |
| -b | --bandwidth | Bandwidth of the pulse | 400 |
| -p | --period | Amount of time the cycle of the pulse will last, measured in seconds | 200ms |
| -f | --file | Name for the output file | pss_complex_pulse |

## Using File with GNU Radio

Firstly, GNU Radio is the SDR program we decided to use, there are other that may work just as well if not better.

- Install GNU radio from this wiki: http://www.gcndevelopment.com/gnuradio/downloads.htm
- Install VcXsrv: https://sourceforge.net/projects/vcxsrv/
- Open GNU radio and wait for GUI to appear
    - If you only want to display a GUI of the pulsar data:
        - Drag a “File Source” block into the diagram window
        - Drag a “Throttle” block into the diagram widow
        - Drag a “QT GUI Sink” block into the diagram window
        - Attach the “File source” to the “Throttle” to the “QT GUI Sink”

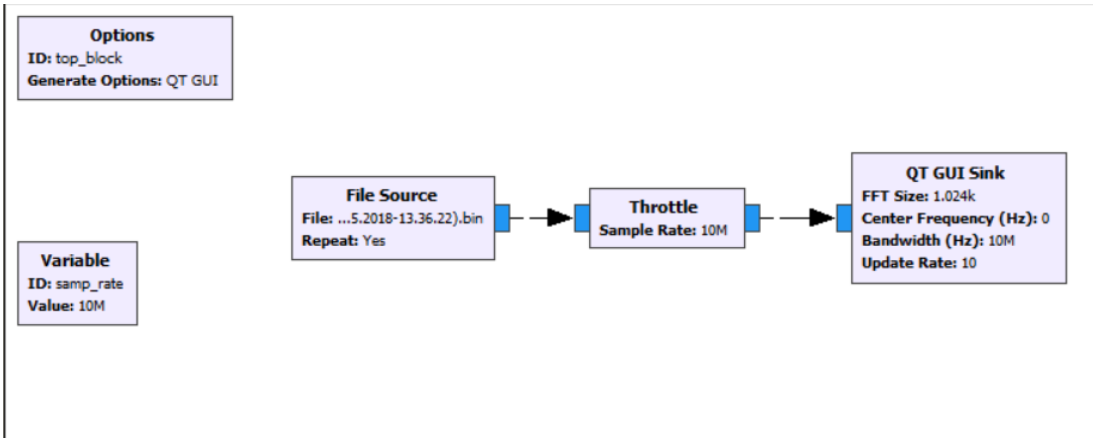


**Fig. 3: GNU radio flow diagram to QT GUI sink**

- Make sure you have a variable for your sample rate and that the “Generate Options” are “QT”
- If you are wanting to transmit the signal via a DAC:
    - Connect the DAC to the computer
    - Drag and “Osmocom sink” block into the diagram window
        - Make sure the drivers in the “Osmocom sink” match those of your DAC
        - Make sure the sample rate variable is not going to exceed the capability of your DAC
    - Connect the “File Source” to the “Osmocom Sink”
- Click the run arrow in the toolbar to run the flow diagrams

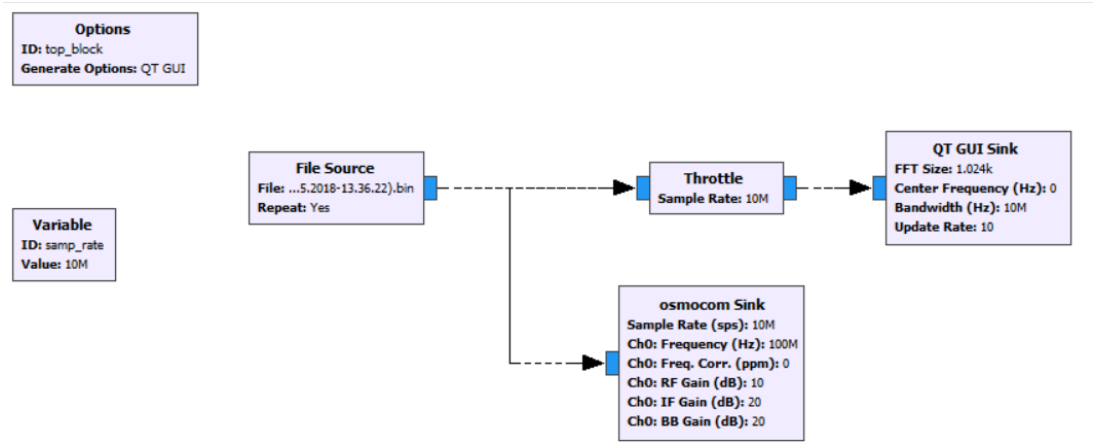


**Fig. 4: GNU radio flow diagram to Osmocom Sink**

# Appendix 2 - Maintenance Manual

## Creating Signals

### PSS

Creating a signal via the PSS library is significantly simpler in that we are mostly using pre-made functionality within PSS. Creation of the signal with PSS involves creating a PSS Signal object with input frequency, bandwidth, period of the signal, signal type (which is always voltage in our case), and the data type (of which we are always using int16). From this signal object we can then simply call a function inside the PSS Pulsar module, providing to it the signal created beforehand, that will then create a pulse. This data is then manipulated to create the interleaved complex values that are later written to file.

### SPIT

The creation of a signal with our non-PSS module is significantly more complex. To begin, a frequency swept sinusoid is generated between a given starting frequency and ending frequency over a specific time period, offset from a zero frequency with an input value as well. This sinusoid is generated as a complex signal, where real and imaginary values are sin and cosine values of a single signal.. This sinusoid is then multiplied by a gaussian distribution to create the shape of the pulse that is expected. This multiplication results in the single repeatable pulse that we expect.

## Writing Files

To make the binary files compatible with GNU radio we wrote them in an interleaving real and imaginary complex64 format. This means the first value is a real float64 then the second is a imaginary float64 and repeat.

### PSS

Currently from our understanding PSS is in no state to produce a signal with enough samples to be resemble a pulsar pulse. Github contributor paulthebaker stated that they were looking to redo PSS in the future. As their refactor progresses PSS may become useful for SPIT. Any future developers may want to keep tabs on PSS to see how it progresses. To create and write a file using pss use the pss.py file.

## SPIT

The main feature of this repository is to use spit.py to create and write pulsar files. This file allows for custom number of samples, start frequency, end frequency, and many other parameters to allow for a custom pulse. Please reference the command line flags section of the user manual for more details.

# Testing

We tested in three main ways: matplotlib, GNU radio, and transmission with a DAC

## Matplotlib

With this python library we were able to graph the signals we produced. MAtplotlib allows you to graph arrays of information easily. With this simple first step of testing we were able to see the visual structure of our signal before we wrote them to a file.

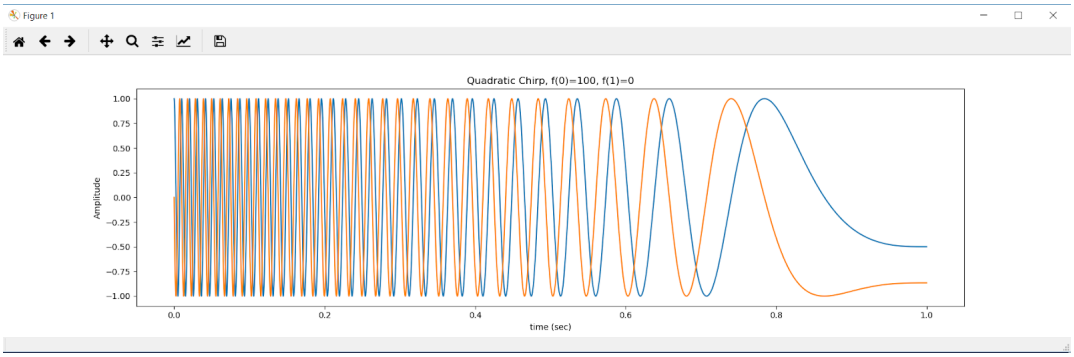


**Fig. 5: A quadrature chirp from 100 Hz downto 0, over 1 second, sampled 8192 times**

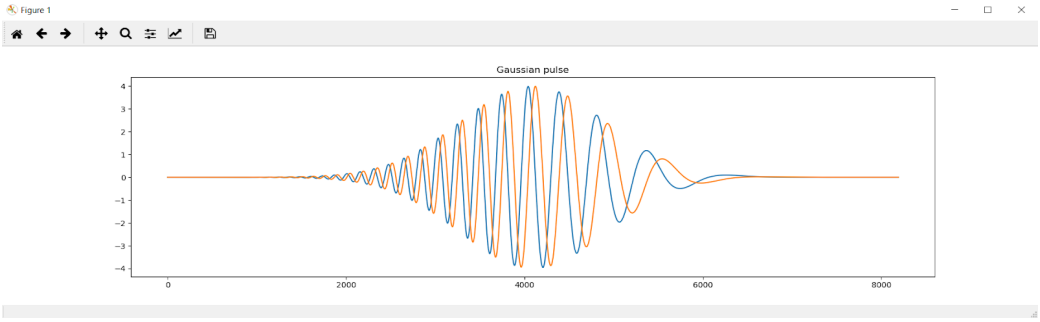


**Fig. 6: A gaussian pulse with a peak at the middle of the sample**

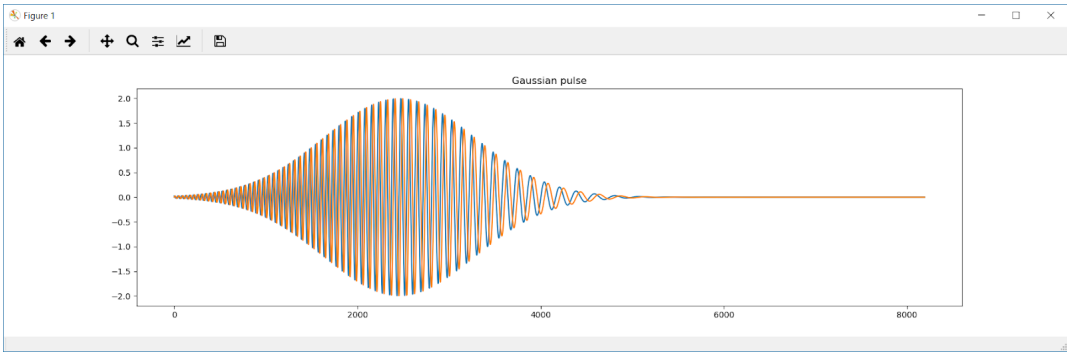


**Fig. 7: A gaussian pulse over 2 seconds with a peak at 0.3 seconds and width of 0.2 seconds**

## GNU Radio

GNU radio is a free application that allows you to work with radio waves. It also aids as a mediator between software and hardware. With GNU radio we were able to read the files we created and display them in various graphical formats.

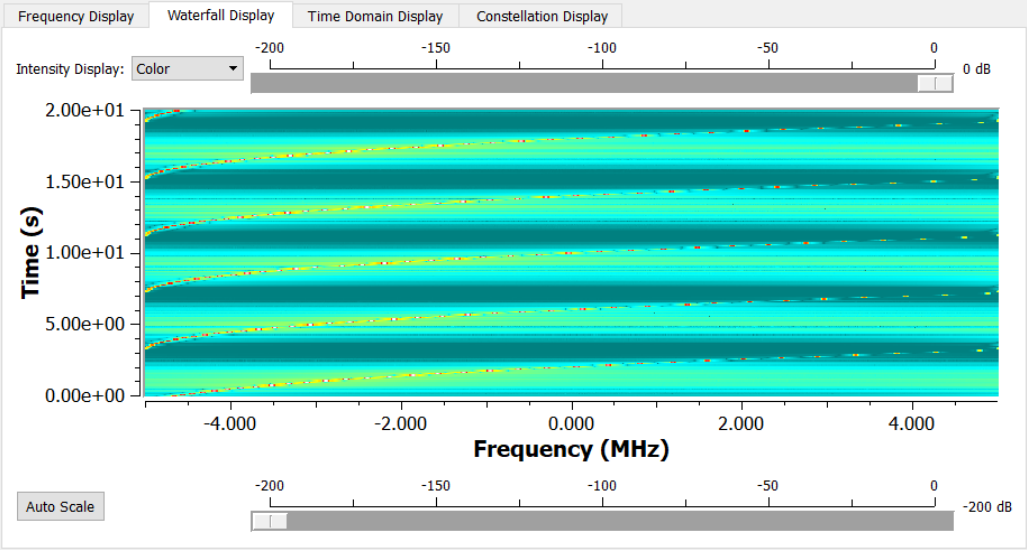


**Fig. 8: Waterfall display of pulse created by SPIT**

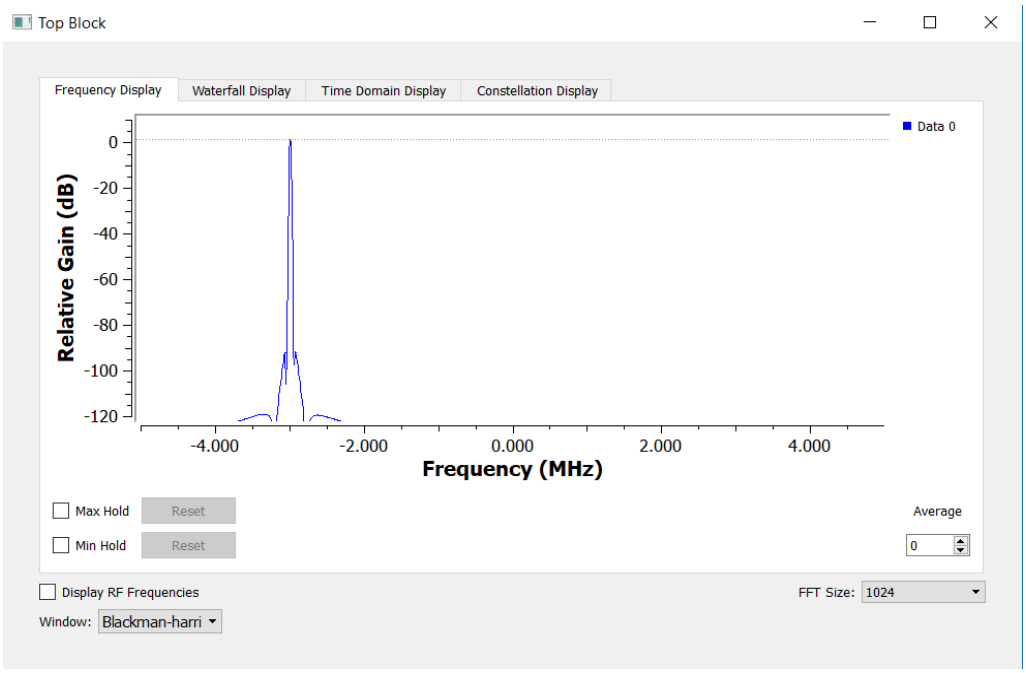


**Fig. 9: A Frequency display of pulse created by SPIT**

### DAC

A DAC was our last form a testing this piece of hardware allowed us to test how a receiver would handle our signal. With the help GNU radio we are able to transmit the signal through a DAC and broadcast it. After all three forms of testing we can compare all the various forms of graphs and get a better understanding of our signal.

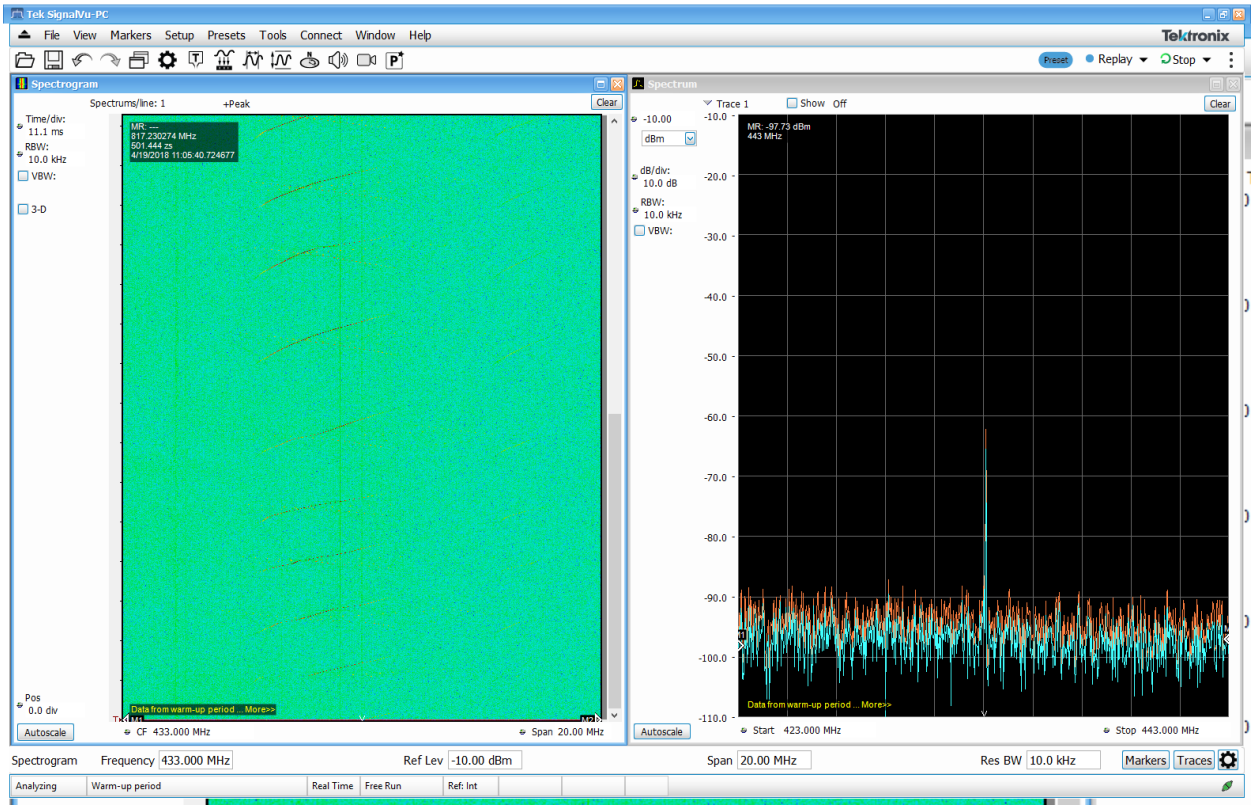


**Fig. 10: Receiving the signal transmitted by the DAC**

## Future Development

The file format may have to be changed based on any future specifications given by the Green Bank Observatory. Creating separate modules to write any various formats might be useful as well for future development for not only Green Bank but also any future users of this code.

With our current implementation of pulsar creation we have the dispersion locked in with the period. This is obviously not ideal for created a wide variety of pulses. A very beneficial feature that should be added to this repository is allowing the user to input a dispersion measure and created a pulse with that value factored in.