Small Radio Telescope 2020

REU Summer 2020 Report

Blaine Huey

Rensselaer Polytechnic Institute
blainemhuey@gmail.com

August 14, 2020

Abstract

The Small Radio Telescope (SRT) is a education-oriented radio telescope developed by Haystack Observatory, which, since its inception in 1998, has progressed through many upgrades taking advantage of the new technologies that have matured in recent years. Since the code for operating the SRT was first written, there have been many advancements and changes in methodology in the world of software development. While the original had custom implementations of every aspect of communicating with motors, tracking celestial objects, signal processing, and displaying those results to the user, incorporating modern libraries could serve to vastly simplify the code and improve the robustness of SRT. Herein we describe the design process, features and operations, and validating experiments on a complete rewrite of the SRT control code in Python, aimed at making its usage by wide audiences easier in a way not previously possible.

I. Introduction

The Small Radio Telescope is an design for radio telescopes aimed at schools and universities for educating students about radio astronomy. Several hundred SRTs exist across the world doing just that. The design has slowly changed over the years to reflect changing materials and technologies, and it most recently received a refresh back in 2013, when it made the switch to use Software Defined Radio as it became increasingly low cost. While the software for the SRT has moved through a couple of different versions and been ported to C from its original Java version, it has not taken advantage of many of the powerful new libraries and languages that have emerged in software development in recent years.

II. Objectives

This project aims to create a complete rewrite of the SRT software from the ground up, using Python for its popularity, simplicity, and the multitude of libraries available to it. As part of rewriting the software, we aimed to keep as much of the previous functionality as possible while simplifying the code. The existing SRT software was highly configurable for different hardware setups and use cases, handled all motor movements and radio processing, and had a wide range of different operations it could perform. However, due to the nature of software at the time, it is also primarily aimed at Linux and controlling it is restricted to just the computer directly plugged into the SRT. Because of these restrictions, we also aimed to both make it work regardless of the base OS and easier to use in a classroom or lab setting, with multiple students able to see what it is doing simultaneously.

III. Design Process

Since one of the goals of this project was to incorporate existing, well-maintained projects to decrease the amount of custom code, the design process started with researching into the different Python libraries the would be able to take over certain functions of the SRT software. For astronomical object tracking calculations, *PyEphem*, *Skyfield*, and *AstroPy* were researched,

but ultimately *AstroPy* was chosen due to its large number of other included features. For creating the user interface, *Plotly Dash* was chosen since it had good support for showing interactive, live-updating graphs via the user's browser, which would allow for many users to see the status of the SRT simultaneously.

Since another one of the aims was to be able to work with a similarly wide variety of SRT hardware configurations as the previous iteration, *SoapySDR*, a vendor-neutral interface library for Software Defined Radios, was chosen to handle the connection to radio itself. The SRT C code supported different hardware by having specific files that would have to be included when the program was compiled for each type of hardware. While this did work well for the specific pieces of hardware it intended to support, each and every type of similar device would need a block written in this way. *SoapySDR*, while unfortunately not compatible with some of the past hardware, such as the computer ADC card iteration, works a wide enough swath of increasingly low-cost SDRs on multiple operating systems to justify its usage. Similarly, *GNU Radio* was used as the signal processing library both because of its ease of creating flowgraphs graphically and its support for SoapySDR, among many other SDR libraries.

The different functions of the program were grouped up into separate logical blocks, which would communicate with each other using *ZeroMQ*, a library which allows for efficient network sockets to be established and run with ease. Together, these were organized into a diagram depicting which parts would communicate with one another, in what direction, and with what information. This diagram went through several iterations once the new code had started to be actually written, but the final version is shown in Figure 1.

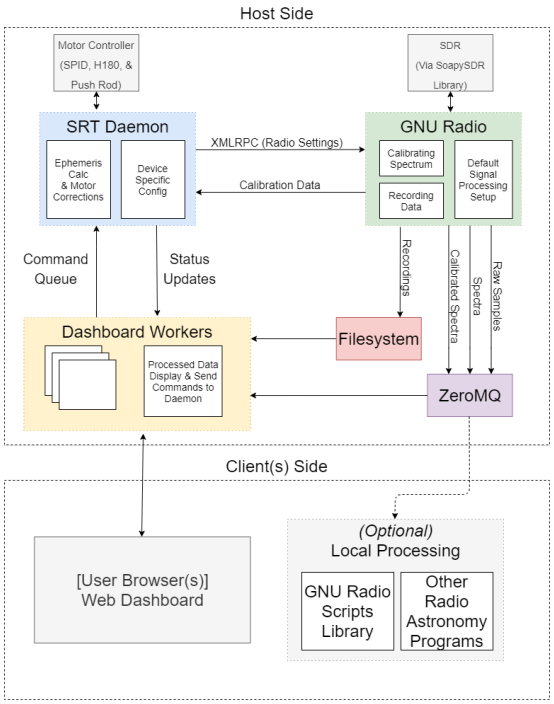

Figure 1: *The most up-to-date SRT Block Diagram*

IV. Program Details

i. Configuration

The settings for the SRT software, including the ranges of the motor, the location of the telescope, and many more details, are loaded using a YAML (Yet Another Markup Language) file from the configuration folder specified by the user. This is meant to take the place of the 'srt.cat' file which the previous software used. The transition to YAML allows us to take advantage of modern libraries, namely *yamale* and *PyYAML*, to validate and ingest the settings without having to parse them manually. The full list of necessary settings can be found in the documentation for the software.

In addition to the YAML containing settings, the SRT will also attempt to ingest a comma-separated values file called 'sky_coords.csv'. Such a file would contain all of the celestial objects that should be trackable by name. Each row would specify the coordinate system, both of its coordinates in

Table 1: *Example SkyCoords Table Structure*

coordinate_system	coordinate_a	coordinate_b	name
fk4	05 32 48	-5 27 00	Orion
galactic	00	0	G00

their typical order, and finally the name of the object to use in tracking. An example of such a structure is given in Table 1.

ii. Signal Processing

The signal processing for the SRT, which is represented by the green 'GNU Radio' block on the earlier Figure 1, is done using GNU Radio flowgraphs. The different operations that the SRT has to perform with regard to radio data, such as turning the raw samples into spectra, determining calibration values, and saving out files, are each handled with their own script. This was set up in order to make it as easy as possible for later users to make changes, or entirely rewrite, any one or all of the flowgraphs. Renders of all the flowgraphs used in this project are available in Appendix A, Radio Processing Flowcharts.

The GNU Radio *radio_process* script handles turning raw samples into calibrated spectra. This acts as the central portion of the signal processing, taking samples from their source, adding metadata tags to the stream, removing any DC offset, performing a weighed overlap add, taking its Fourier Transform, averaging over some number of spectra, and re-scaling the resultant value of each bin based on its calibration value, based on Equation 1. The calibrated scaling of each value is dependent on knowing the system temperature, t_{sys}, and the noise temperature of a known source, t_{cal}, which are assumed to already have been derived separately. Throughout this process, different ZeroMQ Publish blocks are placed to give any number of recipients access to the live data stream at specific steps in the processing, such as at the raw sample level, the spectra, and the calibrated spectra. The radio process script also contains a XMLRPC block, which allows for other processes to change many of the variables during runtime.

$$spectra(i) = raw_{spec}(i)\frac{t_{sys} + t_{cal}}{cal_{spec}(i) \times cal_{pwr}} \quad (1)$$

The scripts handling the other functions of the radio are comparatively much simpler. The *radio_save_raw*, *radio_save_spec_fits*, and *radio_save_spec* scripts take live data over a ZeroMQ port and store it in the Digital RF, FITS, or the '.rad' format of the previous SRT software, respectively. Finally, the *radio_calibrate* script determines the necessary values for scaling the uncalibrated output of the FFT into meaningful values for radio astronomy. Calibration values are calculated by performing a high-order polynomial fit on the averaged, uncalibrated spectra in order to determine the approximate shape of the radio's band-pass filter. The values of the polynomial function at each point are taken to be the new effective uncalibrated values, which aims to smooth out any noise in the calibration's sample spectra. Finally, the average of these spectra are calculated, cal_{pwr}, and the spectra are divided by their average, resulting in $cal_{spec}(i)$.

iii. User Interface

The user interface of the new SRT software is split into two separate pages, one of which is dedicated to monitoring and/or controlling the SRT (the 'Monitor Page'), and the other contains additional information about the system itself (the 'System Page'). Both of the two pages of the dashboard have a collapsible sidebar on the left side, which contains two different sections. One section gives a brief overview of the current status of the SRT, including whether it's running a command and its position. The other contains the links for switching between the Monitor and System pages. Screenshots of the most recent version of the user interface are available in Appendix B, Dashboard UI Diagrams.

Taking inspiration from the previous SRT interfaces, the monitoring page contains a series of buttons at its top for running operations

on the SRT, such as stowing the antenna or running calibration. Additionally, there are four different interactive graphs displayed on this screen. The 'Power vs Time' graph displays the received power over a certain range of time into the past. The first of the two spectrum graphs, 'Raw Spectrum', shows the processed and integrated radio FFT data, whose values don't necessarily have any real world units and have a shape that is influenced by the band-pass filter. The other, 'Calibrated Spectrum' shows the values after dividing out the calibration values taken when the 'calibrate' command was last run on a test source of known temperature (such as a clump of trees or a noise diode). Finally, there is the Azimuth-Elevation graph, which shows the current position of all objects specified to be tracked in the sky_coords.csv configuration file, as well as the reachable limits of the motor and the horizon. Clicking on a point allows you to send a command to track that object, perform an n-point scan about the object, or repeatedly move the antenna across it.

The system page contains many displays of information not necessary for actively controlling the SRT. In case of a serious problem occurring when operating the SRT, there is a section for emergency contact information. There is similarly a 'Message Logs' scrolling area for logs sent from the SRT, in order to assist in debugging or just determining what it has done recently. In the middle is a more verbose status blurb about the status of the SRT's command queue, including the number of commands queued up and what the SRT is currently trying to run. Finally, there is also a list of the files and folders in the SRT's specified recording save directory, from which users can directly download files from via the dashboard if the proper setting in the configuration YAML is set.

iv. Commands

Since the previous SRT software utilized a custom command language for automating tasks, the SRT 2020 software uses the same syntax for instructing the SRT to perform tasks. The SRT daemon keeps all commands received over the network in a FIFO (First-In First-Out) queue in order to run each command, synchronously, in the order they were received. This is the mechanism behind the scenes by which the independent dashboard workers can control and coordinate what the SRT is doing. Because of this, it is also possible to run the SRT entirely headless. The SRT 2020 software is packaged with a Python script for sending the SRT commands and text files listing commands to be executed in order, as well as printing its current status. A list of all currently valid commands is given in Appendix C, Commands.

v. Saving Data

The SRT 2020 software currently supports saving data into 3 different file types: Digital RF, FITS, and .rad. While Digital RF saves the raw I/Q samples from the radio, both FITS and .rad store the calibrated spectrum after averaging. Each format has different techniques for encoding metadata about the status of the SRT when it was taking the data, and so each has nearly the same metadata but accessible in a different manner.

The Digital RF library saves its samples and metadata in Hierarchical Data Format (.h5) files, which can later be read out by either its pre-built GNU Radio blocks or its *DigitalRFReader* and *DigitalMetadataReader* classes[4]. For saving into FITS files, each spectra is recorded into its own new Header Data Unit (HDU) extending the file, so each spectra sample also has its own header metadata. In addition to the normal metadata that the FITS format allows, an additional "METADATA" field was added that contains a JSON-encoded string containing all other relevant metadata. Finally, the .rad file format saves its spectra and metadata together in an ordered, labeled text file, following the scheme format established by the previous SRT software.

Table 2: *SRT Port Usage Chart*

Purpose[1]	Port	Type	Source
Status Updates	5555	PUB/SUB	Daemon
Command Queue	5556	PUSH/PULL	Dashboard
Radio Settings	5557	XMLRPC	Daemon
Raw I/Q Data	5558	PUB/SUB	Radio
Raw I/Q Data NT	5559	PUB/SUB	Radio
Raw Spectra	5560	PUB/SUB	Radio
Raw Spectra NT	5561	PUB/SUB	Radio
Cal. Spectra	5562	PUB/SUB	Radio
Cal. Spectra NT	5563	PUB/SUB	Radio

[1] Data originating from GNU Radio can either come in a stream containing only the content or a tagged stream which requires GNU Radio to decode effectively. All radio data is provided with and without tags, where NT indicated the ports without tags.

vi. Communication

Communication between the different components of the SRT software is accomplished primarily through ZeroMQ sockets. The only exception to this are radio commands, which are transmitted using the XMLRPC protocol. A list of all sockets used by the SRT software is shown in Table 2. With few exceptions, the sockets are using the 'PUB/SUB' architecture, meaning that any number of recipients are allowed to subscribe to that stream of data. The command queue, analogously, uses the PUSH/PULL architecture, which allows for many senders to send commands to one recipient, in order. It also will not drop any packets should the receiver be busy and stop listening, unlike the PUB/SUB architecture. Overall, these give the user the ability to be able to remotely control or access data (depending on the purpose of the port), just by opening the specific port on the host system. For instance, if the user were to open up port 5558, anyone could subscribe to the ZMQ publisher socket and receive a live stream copy of the raw I/Q samples from the SDR along with GNU Radio metadata data tags. Similarly, opening up port 5559 would let anyone subscribe for a copy of the raw I/Q samples that don't contain any GNU Radio metadata, which means the received bytes can just be cast into their appropriate complex float datatype without the need for GNU Radio to parse it.

vii. Extensibility

Since the components of the software are designed to be quasi-independent and only communicate with each other in unidirectional, well-defined ways, it is easy to extend the functionality of the software to other hardware and use cases not thought through in the initial creation of this program. Any GNU Radio script with the correct inputs and outputs could fill a role in the software, such as an improvement or entirely different method for calculating calibration values, and the rest of the system would not have to be adjusted. Additionally, even the user interface provided doesn't necessarily have to be the one used, since any program listening on the correct ports would have access to identical information. Finally, due to the usage of ZeroMQ, any number of subscribers could have live access to raw I/Q samples or spectra and process it how they wish on their own device.

This flexibility in the program was inspired after seeing projects such as *gr-radio_astro*, which had developed a different signal processing method and display for observations in radio astronomy. While that project lacked a way to handle the physical control of radio dish, which was a necessity for the SRT 2020 project, its existence made allowing the possibility of users simultaneously taking advantage of the best of multiple programs a consideration in the design process.

V. Using the Software

The SRT 2020 software was versioned controlled using git and is available on GitHub at `https://github.mit.edu/SmallRadioTelescope/srt-py`. While included in the repository are Markdown documents describing in great detail the procedure to set up the SRT software, a summary is included here.

Because many of the dependencies for the SRT are available through the *conda* package

manager, building and installing the software is handled through it as well. After downloading the srt-py source repository, open up a command prompt or terminal with conda installed (such as Anaconda or Miniconda) and navigate to the folder containing the srt-py directory. Additionally, ensure that you have conda-build and conda-verify installed:

```
conda install conda-build
conda install conda-verify
```

Building and installing the package from source can then be accomplished:

```
conda-build srt-py
conda install -c file://${CONDA_PREFIX
    }/conda-bld/ srt-py
```

Once installed, you can start the SRT Daemon and SRT Dashboard by running by executing the below command (for the default runtime options), where PATH_TO_CONFIG_DIR is the path to the configuration directory (as explained briefly in the 'Configuration' section). This will allow you to start controlling the SRT through your preferred method:

```
srt_runner.py --config_dir=
    PATH_TO_CONFIG_DIR
```

VI. Validating Experiment

Previous Research Experience for Undergraduates students at Haystack Observatory who were also tasked with making an upgrade to the Small Radio Telescope validated their changes each time by performing the Galactic Rotation Curve experiment and comparing the results against the most recent student before them. The experiment involves observing different galactic longitudes, from 0 to 90 degrees, at the hydrogen line frequency. The difference between the known frequency that neutral hydrogen gives off and the received frequency gives on the relative speed difference between Earth and that part of the galaxy. This can then be extended to plot a curve of the speed of the galaxy's rotation at different radial differences. The exact equations used are Equations 2 and 3 from SRT Memo 6[3]. This has significance in astronomy as it allows one to notice the discrepancy between the mass distribution from the what is observed in the galaxy and what would be expected based on its velocity distribution, in the a classroom or lab setting.

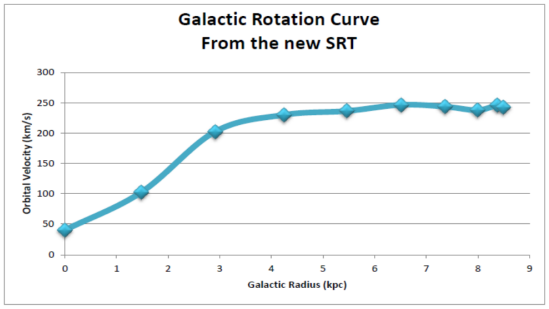

Figure 2: *REU 2012 Galactic Rotation Curve[1]*

$$V = \frac{(1420.406 - f)Vc}{(1420.406 - Vlsr)} \tag{2}$$

$$V_{max} = \omega R = V_{max}(R) + \omega_0 R \tag{3}$$

The most recent REU students to upgrade the SRT, who worked primarily on designing a new SRT structure when the original parts became unavailable [1] and making the Small Radio Telescope work with low-cost software-defined radios [2] produced the galactic rotation curve shown in Figures 2 and 3, respectively. Following through with the same process and technique of gathering observing the hydrogen line with the SRT, we obtain the curve shown in Figure 4. While there are some slight distinctions between each likely due to external factors, like noise and the approximation assumptions underlying the calculation itself, the newest curve bears a remarkable resemblance to the prior one, providing a sanity check on the new SRT software as a whole.

VII. Challenges Encountered

Rewriting the software for the SRT from the ground up in a different language presented many challenges. The first challenge, which took up the first few weeks of work on this

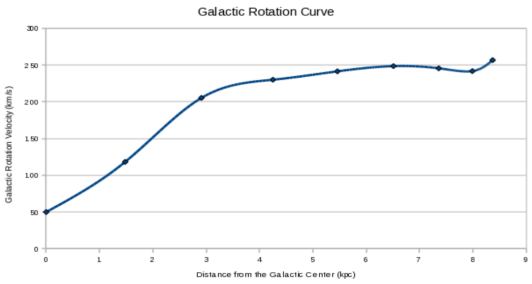

Figure 3: *REU 2013 Galactic Rotation Curve[2]*

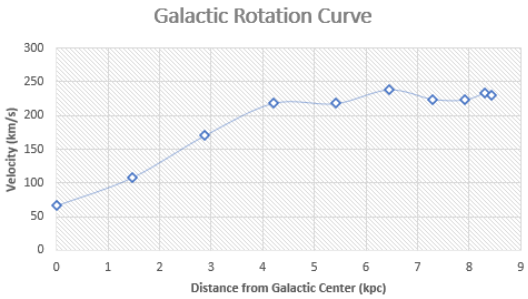

Figure 4: *REU 2020 Galactic Rotation Curve*

REU project, was researching and getting oriented with all of the different libraries that were available to handle different tasks. While the wealth of open-source libraries available for Python certainly made writing the code for the SRT easier overall, reading through the documentation for each to understand its features and weaknesses took a considerable amount of time before the real code development begin.

Another challenge that presented itself quite early was in coordinating the timing of the different parallel tasks that would have to be done to operate the SRT. For instance, sending a command 'Sun' to the SRT should make it track the sun. At any point in time, the Sun is at a particular azimuth and elevation, which has to be updated quite often. Simultaneously, the motor must be sent the most recent value, but its serial connection was only 600 baud, and so sending locations took time as well as asking for and receiving updated positions. Also, the SRT had to be listening for new commands throughout this time, like for starting recording data, and reporting its changing status. This was managed by making the daemon contain several threads whose purpose it was to update values, who were connected primarily through Queue objects of their own, although this did take a decent amount of time and testing to accomplish.

Some challenges also existed for designing the signal processing chain. Since most of the previous C code did all of the signal processing manually, from collecting samples, to calibrating, and graphing. it was necessary to step through all of the code and reverse engineer the logical steps that were occurring. This led to a couple of bugs, where the most recent one fixed involved the GNU Radio source block not setting the radio gain to its maximum value of 49.6 dB, which was done in the beginning of the previous code but went unnoticed since it was not visible from the signal processing code. The GNU Radio block equivalent has a default value of 20dB for this same setting, which led to much confusion as quiet signals became invisible. Additionally, while GNU Radio supports doing many operations through its blocks, sometimes it is necessary to build your own block to do a custom task. This poses something a weak point in the GNU Radio documentation, so a lot of time was spent trying to reverse engineer how message passing works in embedded python blocks.

The remote nature of the REU due to the pandemic also created some challenges of its own. While the people at Haystack went to incredible lengths to make it easy to remotely access the computer attached to the SRT and run real tests, most tests of the code were run locally with simulated components. This was due in part to the worry that if the SRT dish were to get stuck out of its stow position or another, similar physical problem were to occur, it would be difficult for someone to go out to the observatory and fix it while it was closed. While the impact of this may not have too much significance to the final software and was unavoidable given the circumstances, it did mean some bugs that could have been easily caught earlier if one were there in person lasted a bit longer than they should have.

VIII. Future Work

i. Additional Features & Fixes

There are several key areas where the SRT software can continue to improve. Firstly, there are still many features and settings that were present in one form or another in previous versions of the code that would increase the range of tasks that the SRT can discover and teach about. These issues themselves range from some small missing details like handling a vane calibration motor or a Python analog to the 'pswriter.c' script, to interesting projects in their own right, like the ability to do Very Long Baseline Interferometry (VLBI). Also, the calibration routine, while currently functional, has demonstrated some behaviors that could be worked out in future versions, notably that the polynomial fit that smooths the spectra tends to warp around spurs and outliers because of their significance to the least-squares value. It would also benefit the user interface to be more exposed to other people outside of the project, especially those active in radio astronomy or education, or students themselves in order to make it more straightforward and useful.

ii. Expanding Outreach

Since there exist so many SRTs around the world, it is important to coordinate with other institutions that also use SRTs to take advantage of the recent upgrades. This would also likely involve trying to expand the number of motors that the software supports and has been thoroughly tested on, which is somewhat lacking in its current form. Finally, increasing the number of educational experiments that have been worked out on the Small Radio Telescope would also greatly increase the utility of the software and the SRT to schools, universities, and the general public.

IX. Conclusion

While the Small Radio Telescope has undergone many design changes throughout the years to reflect the evolving nature of technology, its code had not fully reflected the changes that had since happened the world of software development. By rewriting the software from the ground up in Python and taking advantage of community-driven open-source libraries, we aimed to make a simultaneously more robust and feature-rich SRT control program. Experiments performed on the new code derive similar results as those that had been used to verify that past alterations to the design, and the new web interface and flowgraph structure will hopefully make it easier than ever to use the SRT platform to its full potential.

X. Acknowledgments

I would like to first thank my three mentors on this project, Ryan Volz, John Swoboda, and Alan Rogers. Their support and advice throughout this project is what allowed for the project to even get through to this point. I would also like to thank Dahlia Dry for helping with the astronomy and collecting the data that made verifying this project possible while at the same time working on her own projects. Finally, I would like to thank Timothy Morin and Jason SooHoo for their excellent IT help, and Heidi Johnson, Diane Tonellia, Nancy Kotary, Phil Erickson, Vincent Fish, and everyone else at Haystack Observatory, as well as the NSF, for making this REU program the fun and fascinating experience that it was.

References

[1] Johnson, Dustin and Dr. Alan E.E. Rogers. *Development of a New Generation Small Radio Telescope*. MIT Haystack Observatory (2012). Retrieved from: `https://www.haystack.mit.edu/wp-content/uploads/2020/07/srt_FinalReport.pdf`

[2] Higginson-Rollins, Marc and Dr. Alan E.E. Rogers. *Development of a Low Cost Spectrometer for Small Radio Telescope (SRT), Very Small Radio Telescope (VSRT),*

and Ozone Spectrometer MIT Haystack Observatory (2013). Retrieved from: https://www.haystack.mit.edu/wp-content/uploads/2020/07/srt_2013_HigginsonRollinsPaper.pdf

[3] Dr. Alan E.E. Rogers. *Measurement of Galactic rotation curve using 21-cm hydrogen emission* (1999). Retrieved from: https://www.haystack.mit.edu/wp-content/uploads/2020/07/memo_SRT_006.pdf

[4] Volz, R., Rideout, W. C., Swoboda, J., Vierinen, J. P., & Lind, F. D. *Digital RF (Version 2.6.5)*. MIT Haystack Observatory. (2020). Retrieved from https://github.com/MITHaystack/digital_rf

Appendices

A. Radio Processing Flowcharts

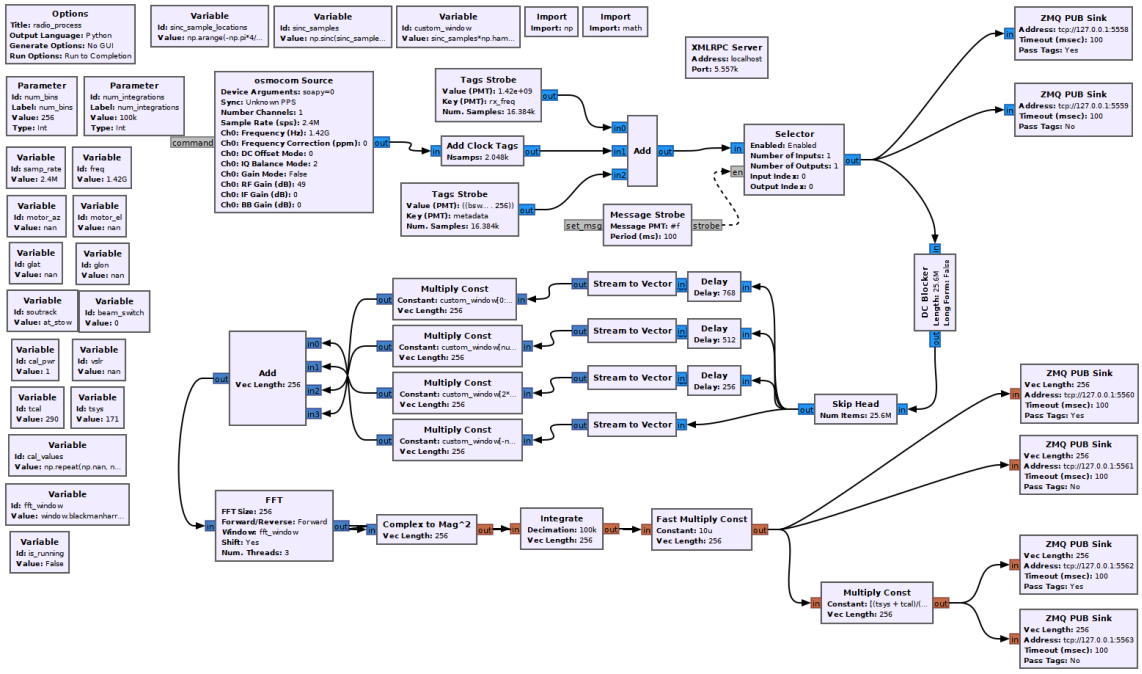

Figure 5: *The SRT GNU Radio Processing Flowgraph*

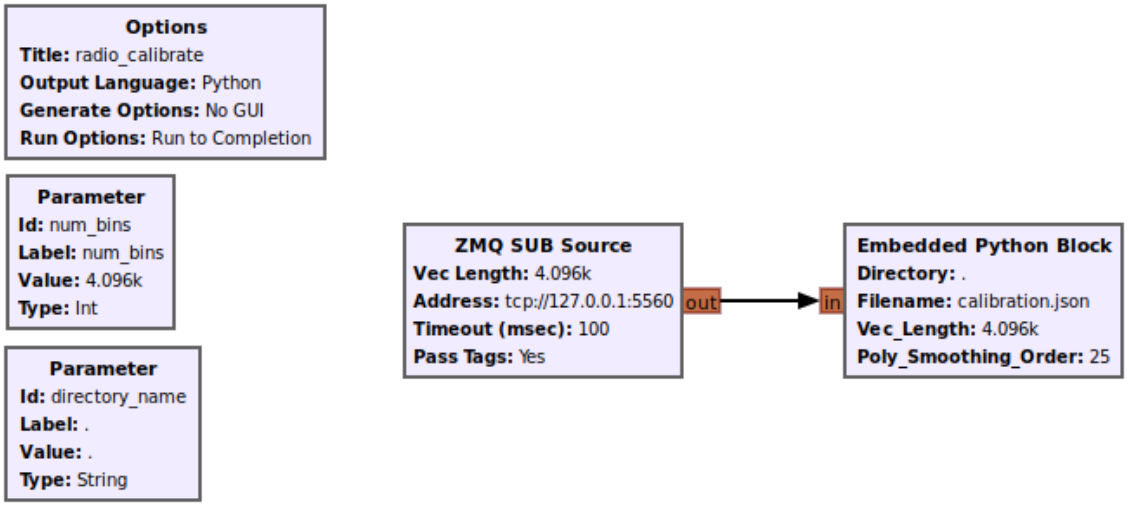

Figure 6: *The SRT GNU Radio Calibration Flowgraph*

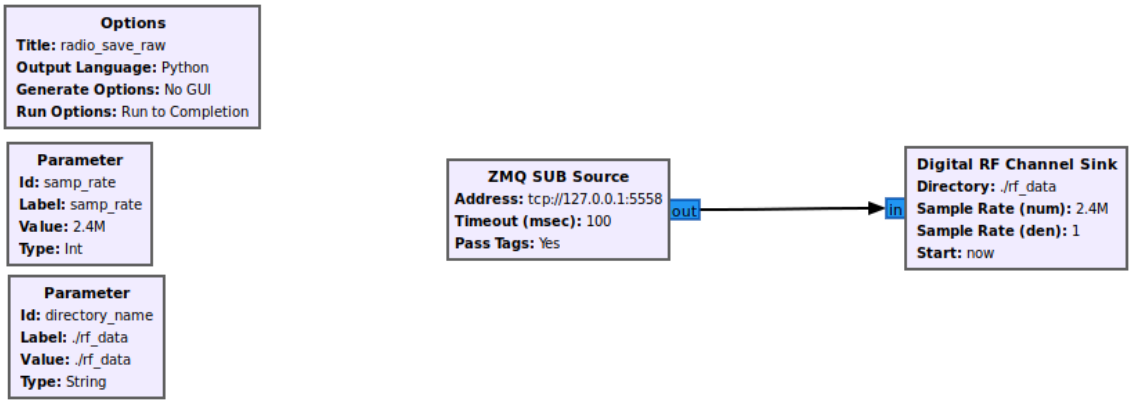

Figure 7: *The SRT GNU Radio Raw Sample Recording Flowgraph*

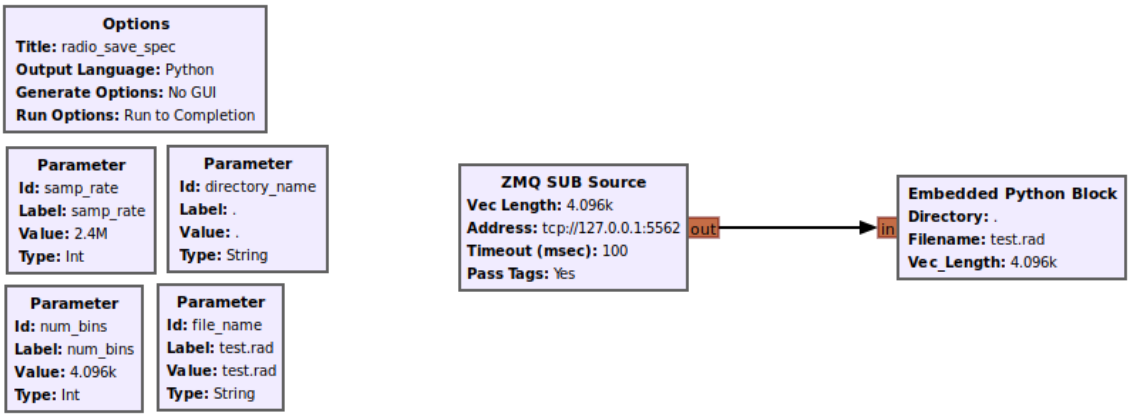

Figure 8: *The SRT GNU Radio Spectra Recording Flowgraph*

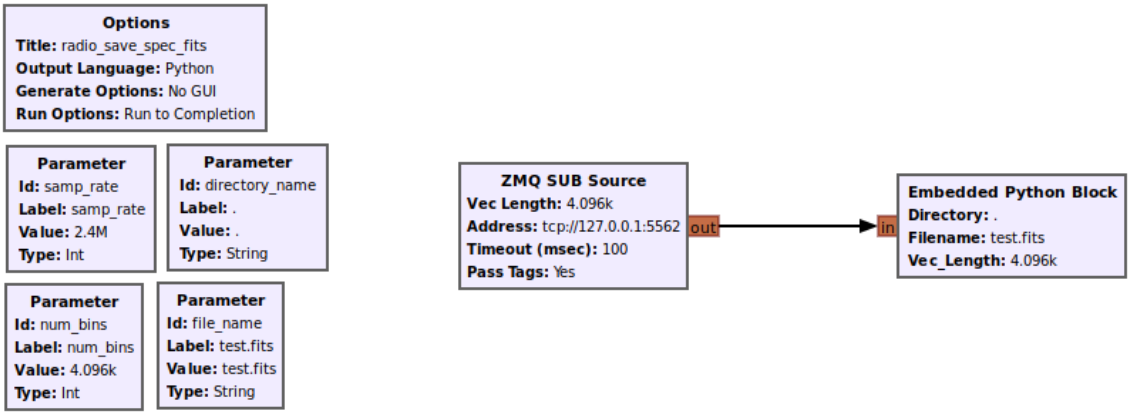

Figure 9: *The SRT GNU Radio FITS Spectra Recording Flowgraph*

B. Dashboard UI Diagrams

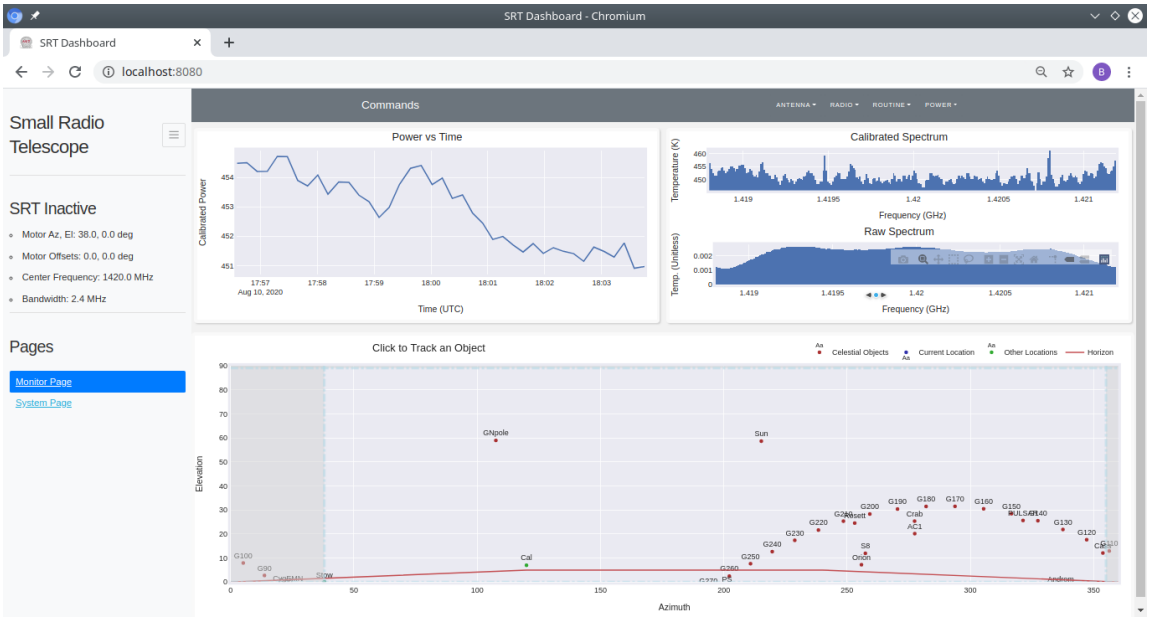

Figure 10: *The SRT UI Monitor Page*

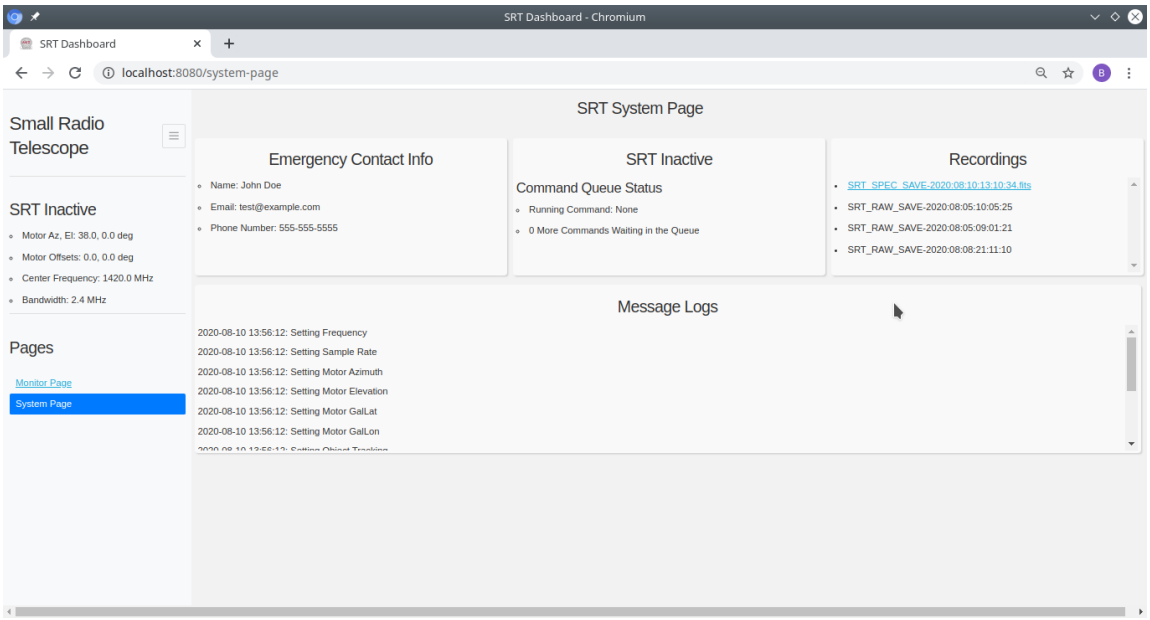

Figure 11: *The SRT UI System Page*

C. Commands

Table 3: *SRT Valid Commands*

Command	Parameters	Info
* [1]	Any text	Makes Line a Comment
stow	None	Sends the Antenna to Stow
cal	None	Sends the Antenna to Calibration Position
calibrate [2]	None	Saves Current Spec as Calibration Data
quit [3]	None	Stows and Gracefully Closes Daemon
record [4]	[filename]	Starts Saving into File of Name 'filename'
roff	None	Ends Current Recording if Applicable
azel	[az] [el]	Points at Azimuth 'az', Elevation 'el'
offset	[az] [el]	Offsets from Current Object by 'az', 'el'
freq	[cf]	Sets Center Frequency in MHz to 'cf'
samp	[sf]	Sets Sampling Frequency in MHz to 'sf'
wait	[time]	Stops Execution and Waits for 'time' Secs.
[time]	None	Waits for 'time'
LST:hh:mm:ss	None	Waits Until Next Time hh:mm:ss in UTC
Y:D:H:M:S	None	Waits Until Year:DayOfYear:Hour:Minute:Sec
[name] [5]	[n or b]	Points Antenna at Object named 'name't

[1] Only is considered a comment if the line starts with '*'.

[2] Calibration takes samples at its current location (which should typically be 'cal') and uses those to determine the shape of the band-pass filter and eliminate it from the calibrated spectra. Calibration should therefore by done against a non-source object of known noise temperature.

[3] Unlike the previous SRT software, 'quit' both stows and quits (ends the daemon process). After this is done, it will be necessary to restart the daemon process from command line or via the Dashboard 'Start Daemon' button.

[4] Currently, three different file types are supported, and the type used is determined by the file extension of the name given. FITS (Calibrated Spectra) is used when the file ends in '.fits', rad (Calibrated Spectra) is used when it ends in '.rad', and Digital RF (Raw I/Q Samples) is used when the name lacks a file ending. If no filename is provided, Digital RF will be used with an auto-generated name. In order to use rad or FITS with an autogenerated name, use "*.rad" or "*.fits" respectively.

[5] The names used for pointing at objects are set by the 'sky_coords.csv' file, which is further documented in the config folder portion of the docs. By default, 'Sun' and 'Moon' are already loaded.