Development, Implementation, and Impacts of Novelty Detection Systems for Mission Operations

by

Paul Alexander Horton

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved June 2025 by the
Graduate Supervisory Committee:

Chris Groppi, Co-Chair
Jim Bell, Co-Chair
Hannah Kerner
Philip Mauskopf
Nargess Memarsadeghi

ARIZONA STATE UNIVERSITY

August 2025

ABSTRACT

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices biben- dum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

DEDICATION

To my cat, who had to put up with me every time I needed a break.

ACKNOWLEDGMENTS

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices biben- dum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

TABLE OF CONTENTS

Page

Chapter 1

INTRODUCTION

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices biben- dum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Chapter 2

DATA DISCOVERY SYSTEMS FOR PLANETARY SCIENCE

2.1 Abstract

While innovations in scientific instrumentation have pushed the boundaries of Mars rover mission capabilities, the increase in data complexity has pressured Mars Science Laboratory (MSL) and future Mars rover operations staff to quickly analyze complex data sets to meet progressively shorter tactical and strategic planning timelines. MSLWEB is an internal data tracking tool used by operations staff to perform first pass analysis on MSL image sequences, a series of products taken by the Mast camera, Mastcam. Mastcam consists of a pair of 400-1000 nm wavelength cameras on MSL's Remote Sensing Mast that, among other functions, uses a filter wheel to produce multispectral images by creating a sequence of products at different wavelengths. Mastcam's multiband multispectral image sequences require more complex analysis compared to standard 3-band RGB images. Typically, these are analyzed by the inspection of false color images created to aid visualization, such as band ratios between different spectral indices that can highlight specific potential mineralogic differences among iron-bearing phases, and decorrelation stretches to enhance the color differences between multiple filters.

Given the short time frame of tactical planning in which downlinked images might need to be analyzed (within 5-10 hours before the next uplink), there exists a need to triage analysis time to focus on the most important sequences and parts of a sequence. We address this need by creating products for MSLWEB that use novelty detection to

help operations staff identify unusual data that might be diagnostic of new or atypical compositions or mineralogies detected within an imaging scene. This was achieved in two ways: 1) by creating products for each sequence to identify novel regions in the image, and 2) by assigning multispectral sequences a sortable novelty score. These new products provide colorized heat maps of inferred novelty that operations staff can use to rapidly review downlinked data and focus their efforts on analyzing potentially new kinds of diagnostic multispectral signatures. This approach has the potential to guide scientists to new discoveries by quickly drawing their attention to often subtle variations not detectable with simple color composites. The products developed in this work have shown promising benefits for integration into mission operations by potentially decreasing tactical operations planning time through guided triage.

2.2 Introduction

While discovering the unexpected is one of the most exciting parts of research, the process of making a discovery often involves countless hours of sifting through otherwise mundane data. Advances in novelty detection systems can help to alleviate this arduous task by enabling researchers to focus their attention on the most interesting parts of their data. Novelty is defined as an unexpected occurrence in a sequence based on a data set of typical sequences. Given what is known to be "normal," the novelty detection algorithm highlights the most unusual features in an image. Novelty detection techniques work by analyzing the commonalities in a data set in order to generalize the structure of the data and pick out anomalies (**japkowicz1995novelty**). When a new observation is presented to a novelty detection system, the system identifies features that differ from the commonalities learned during training (**markou2003novelty**). This is different from traditional classification systems as what constitutes as *novel* is not predefined. Instead of classifying samples as *typical* and *novel*, novelty detection systems build a model from *typical* examples that can be used to discover anomalies. This makes novelty detection ideal for scenarios where *novel* examples are infrequent or difficult to obtain and *typical* examples are abundant (**japkowicz1995novelty**). Identifying anomalous examples is useful in many application domains such as structural fault detection for aerospace

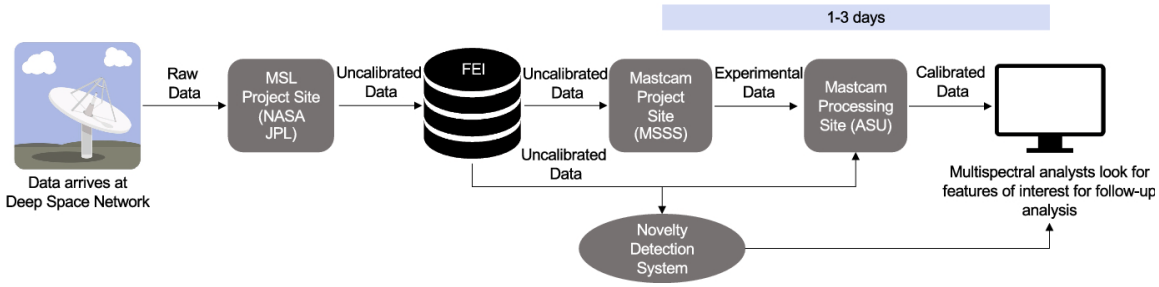


Figure 1. Illustration of the Mastcam image processing pipeline (**kerner2020comparison**)

systems by analyzing ambient vibrations (**worden1997structural**) or identifying brain tumors using MRI images (**wang2020brain**).

Planetary science is another domain where novelty detection proves useful. Due to the rapid turn-around required for tactical planning for landed Mars missions, efficient data analysis methods need to be employed to analyze data from scientific instruments (**bell2019tactical**). With high volumes of downlinked data, tactical operations planning teams have to quickly perform ground analysis to meet the ten hour turn-around time for MSL operations to uplink commands (**samuels2013preparation**). The timeline is even shorter for the upcoming Mars 2020 mission which has a desired five hour turn-around time (**wilson2017nasa**). During tactical planning, the MSL Curiosity rover sends compressed observations through one of three Mars orbiters to the Deep Space Network (DSN) on Earth so that operations team members can use these observations to make plans for the next sol (**kerner_multispec**). Rapid analysis is required for tactical planning as discoveries made outside of the time frame are subject to increased mission resource cost as the rover may need to reverse course to re-visit the target (**kerner2020comparison**). This need for rapid analysis makes systems that can quickly extract information and identify regions of interest in scientific data vital to efficient tactical planning.

This work primarily focuses on the Mastcam imaging system on the MSL Curiosity rover. Mastcam consists of a pair of multispectral Charge Coupled Device (CCD) imagers on MSL that are each capable of using an eight-position filter wheel to take red, green, and blue (RGB) images as well as multispectral images in nine bands from 400 to 1100 nm (visible to near-infrared) (**bell_mastcam**). A set of image products from Mastcam is called a sequence. Seven of the eight landed missions on Mars have employed camera systems capable of multispectral imaging (**bell2019tactical**). Future missions, such as Mars 2020 and Psyche, will also be carrying similar multispectral cameras (**bell2016mastcam**) (**bell_psyche**). Given that multispectral cameras are so prevalent in planetary exploration, the ability to rapidly detect novel features in multispectral images is beneficial across many missions.

The goal of this work is to operationalize previous work developed for novelty detection for Mastcam. **kerner2020comparison** quantitatively compared different novelty detection methods for analysis of multispectral images on Mars. With a typical data set, machine learning models including Reed-Xiaoli (RX) detectors, Principal Component Analysis (PCA), Generative Adversarial Networks (GANs), and Convolutional Autoencoders (CAE) were used to produce a measure of how atypical each image or multispectral pixel is in a new sequence. Using pixelwise analysis allows these methods to highlight difficult to identify novel regions within an image that may otherwise go undetected by operations staff. These methods[1] and the data set[2] used to evaluate them were made publicly available at publication. Multiple models were evaluated to show that certain models perform better for certain novelties than others – e.g., PCA was better suited for detecting spectral (compositional)

1. https://github.com/JPLMLIA/mastcam-noveltydet

2. https://zenodo.org/record/3732485

novelties than for morphological (shape) novelties. Their work provides an in-depth qualitative evaluation between these methods which was used to guide decisions about which methods to use. While their work demonstrated the capabilities of the algorithms, it did not evaluate the methods based on their effectiveness in the tactical planning pipeline. In order to be most useful to operations, these methods need to be automatically run on new data and integrated into tactical analysis workflows to accelerate tactical planning (**donahoe2020new**). New advances in this work include further development of the algorithms and the creation of an implementation strategy for operations. Additionally, we analyzed the outputs from the system to determine the usefulness of these methods in comparison to existing tactical planning procedures.

2.3 Related Work

Novelty detection is a form of anomaly detection that focuses on detecting novel examples in a data set (**domingues2019comparative**). Given what is known to be typical in a data set, these algorithms find novel examples that may be unknown to the user. Unlike a classifier, novelty detection systems detect whether an input is similar to examples in the typical training set or if the input is novel (**markou2003novelty**). Novelty detection can be seen as a one-class classification task where a model is trained to describe a data set of typical examples (**pimentel2014review**). For novelty detection, the training data represent a set of examples that an end user would identify as typical examples. When the model is used to infer the novelty of new data, the system calculates how different the new examples are compared to the training set. As abnormal examples are not well represented in the training set, novelty detection systems are not able to model abnormalities and thus highlight them as novel. This allows novelty

detection systems to highlight abnormal data by evaluating how well (or how poorly) it can model the examples. These methods can be applied to various domains such as fraud detection based on card activity (**oosterlinck2020one**), human verification for websites from mouse and keyboard usage (**kim2018keystroke**), fault detection for aerospace systems by analyzing ambient vibrations (**worden1997structural**) and brain tumor identification using MRI images (**wang2020brain**).

This work is based on previous work that developed novelty detection for multispectral images on Mars (**kerner2020comparison**). Figure **??** shows the current pipeline for multispectral image processing for operational analysis and where we propose to augment the pipeline with novelty detection systems. Calibrated data are often not available during the tight tactical analysis schedule, so uncalibrated data will be used for novelty analysis. Current methods for quick analysis involve generating quick look products, such as decorrelation stretches and filter ratios, which help to identify spectral changes in the sequence (**gillespie1986color**). For example, comparing the relative reflectance spectra of two different drill tailings can inform analysts of the similarities and differences in mineral composition (**wellington2017visible**). These quick look products are generated using calibrated data making them unavailable during the tactical analysis time frame.

Kerner et al. demonstrated four types of novelty detection systems for multispectral novelty detection: CAEs, GANs, PCA, and RX detectors (**kerner2020comparison**). All of these methods, except RX, are reconstruction based methods that attempt to "compress" and "reconstruct" an input to recreate the original image. When trained on typical examples, these methods are able to reconstruct normal examples well but are unable to represent novel examples. The novelty detection system is able to identify novel regions based on the reconstruction error, which is the difference

Figure 2. Examples from the eight categories of novel geology in the Mastcam multispectral image data set (**kerner_data**)

between the input and recreated image. For Mastcam multispectral sequences, these images have six bands instead of the traditional one (grayscale) or three (RGB). It is important to note that Kerner et al. evaluated these methods to identify novel geology in multispectral sequences, not create a visualization for tactical analysis that maximizes their ability spot hard-to-find features. Our goal is to create novelty detection products that could be integrated into actual tactical operations pipelines and evaluate their effectiveness for triaging analyst time in tactical operations.

2.4 Data Set

In this work, we created a data set of all multispectral Mastcam sequences based on the pre-processing methods outlined in the labeled Mars multispectral novelty detection data set (**kerner_data**). This previous data set provided images with both *typical* and *novel* labels cropped from the M-100 right eye of Mastcam on MSL between sols 1 to 1666. Each example in the data set is a 64 by 64 tile with 6 bands corresponding to 6 different filters. The tiles were sub-sampled from larger thumbnail images of around 140 by 100 pixels in size. Thumbnails were used rather than full resolution sequences as they are among the first available products during tactical planning. Additionally, uncalibrated images were used as these data are the most readily available in the tactical analysis timeline. To generate the data set, these

thumbnails were loaded as single band images in OpenCV before combining them to produce a single, six band, tile (**opencv_library**).

While we did not explicitly use the labeled Mars multispectral novelty detection data set in this work, we utilized the novelty classes to verify the system. The data set provided a set of novel tiles divided into 8 sub-classes: meteorite, float rock, bedrock, vein, broken rock, dump pile, drill hole, and DRT spot, as shown in Figure **??**. Multispectral analysts from the Mastcam operations team identified novel geology in images using bounding boxes based on operations experience and past publications of high interest targets (e.g., (**wellington2017visible**)). Sub-sampled tiles that intersected with these bounding boxes were included in the novel data set.

The goal of this work is to triage tactical analysis time by prioritizing interesting images and highlighting features that are otherwise difficult to identify in multispectral sequences. While our system should be able to identify novelties such as large veins and broken rocks, which are relatively easy for analysts to identify in RGB or single-band images, highlighting these examples will not reduce tactical analysis time as much as other features that are difficult or impossible to see without detailed analysis of multispectral sequences. The **kerner2020comparison** data set did not account for these operational priorities when evaluating novelty detection performance. In order to fully evaluate the benefit of our novelty detection system in tactical operations, we created a new data set using the same pre-processing as the Kerner et al. data set. Instead of separating the images into typical and novel data sets, we included all multispectral sequences from the left and right eyes of Mastcam in an unsupervised manner. The images in this data set were pre-processed in the same way as the original data set but kept as full size thumbnails instead of tiles in order to provide detections at the same resolution as they are viewed by operations staff. Starting with

every thumbnail taken by Mastcam, the data set was filtered by sequences containing 7 different filters: RGB (no filter) and 6 narrow-band spectral filters. Additionally, sequences containing the calibration tool were omitted. This filtering resulted in about 900 total sequences split between the left and right eye. The six narrow-band sequences were used as inputs into the algorithm and the RGB is was used for reference.

2.5 Methods

In this work, we employed a the pixelwise RX method for novelty scoring. Pixelwise RX is a distribution-based method that calculates the distance between each pixel in a test image and a background distribution, which can be visualized at the image level as a heatmap of novelty scores. In comparison with other methods, pixelwise RX is out-performed when identifying images with certain types of novelties such as float rocks, veins, bedrock, and broken rock, but performs better for other novel categories. While this may seem like RX is a poor choice for novelty detection, the novelty scores in **kerner2020comparison** were aggregated to image-level scores and did not compare algorithms on the basis of their ability to highlight novel pixels in an image in a way that is useful for tactical planners. Pixelwise RX may not have the highest performance for all types of novelty, but its scores and their associated visualizations are relatively simple and interpretable compared to other methods. This is an important characteristic in method selection for analysts using novelty detection methods in a tactical operations setting. Thus, we chose to use pixelwise RX for developing novelty-based tactical planning products.

In this study, we chose to focus on spectral novelties that are difficult for analysts to spot because they require analysis across multiple filters to find correlations difficult

to identify when looking at each image separately. While methods not based on novelty detection exist that help identify spectral novelties, they require calibrated data which are often unavailable during the tactical time frame.

Unlike the other algorithms considered, RX is not a reconstruction based method for detecting novelties. Instead, RX computes a background distribution from typical examples and compares this distribution to infer the novelty of pixels in new images (**reed1990adaptive**). For novelty detection in Mastcam sequences, the background is computed using the spectrum from each pixel in all sequences in order to generalize the entirety of the data set. For a multispectral image with n filters, the background distribution is defined by the $n \times 1$ mean spectrum vector, $\boldsymbol{\mu}$, and an $n \times n$ covariance matrix, $\boldsymbol{\Sigma}$, of all pixels in the training set (**guo2016anomaly**). This provides a mean for each multispectral band and a covariance matrix for the band pairings. To infer the novelty in a new image, $\boldsymbol{X}$, an RX score is calculated for each $n \times 1$ pixel vector, $\boldsymbol{x}_i \in \boldsymbol{X}$, using the Mahalanobis distance between the background and the filter response values as shown in Equation **??**.

$$\mathrm{RX}(\boldsymbol{x}_i) = (\boldsymbol{x}_i - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x}_i - \boldsymbol{\mu}) \tag{2.1}$$

This score will be referred to as the pixel novelty score as it is a measure of how novel a pixel is relative to the background distribution. As this method is pixel based, the dimension size of the sequence does not matter so inferences can be run on sequences of any resolution. This is particularly useful for Mastcam as the height of each sequence of images varies.

After computing the RX background distribution using typical sequences, Equation **??** can be applied to new sequences to identify novel features. Each sequence can be analyzed individually or relative to other sequences based on their pixel novelty scores. To reduce the effect of brightness in the novelty scoring, separate models

were created using a data set where the RGB image of the sequence is loaded as a gray scale image and used to divide the other six images in the input. Individual sequence analysis is accomplished by visualizing the pixel novelty scores as a heat map. Analysts can quickly review these heat maps to identify the most novel regions within a sequence relative to all prior sequences. To compare the novelty of different sequences, statistics can be calculated for each sequence and sorted. This can help prioritize which multispectral sequence to review first.

2.6 Results

To assess the system's performance for the novel features in the **kerner_data** data set, we located the full-size thumbnails of a selection of the novel tiles from each category and verified that the novelties highlighted in the new images were self consistent with previous results in the tiles. The novelty detection system using pixelwise RX performs well for highlighting easy-to-spot novelties in the data set. Figure **??** shows examples of some of the easy to catch novelties that the system is able to detect. We claim that these examples are relatively easy for analysts to identify because the novelties they highlight can be found through careful analysis of a single RGB image. In the first example, a broken rock is shown near the center of the image. The novelty heat map highlights the broken rock well as well other broken rocks in the area as shown by the yellow regions. The second example shows veins which are highlighted well in its corresponding heat map. By just looking at the RGB thumbnail, an untrained analyst could likely spot something novel in the image. Finally, the last image shows a dump pile. This example is clearly visible in the RGB, but would also be easy to identify because the dump pile was created by the

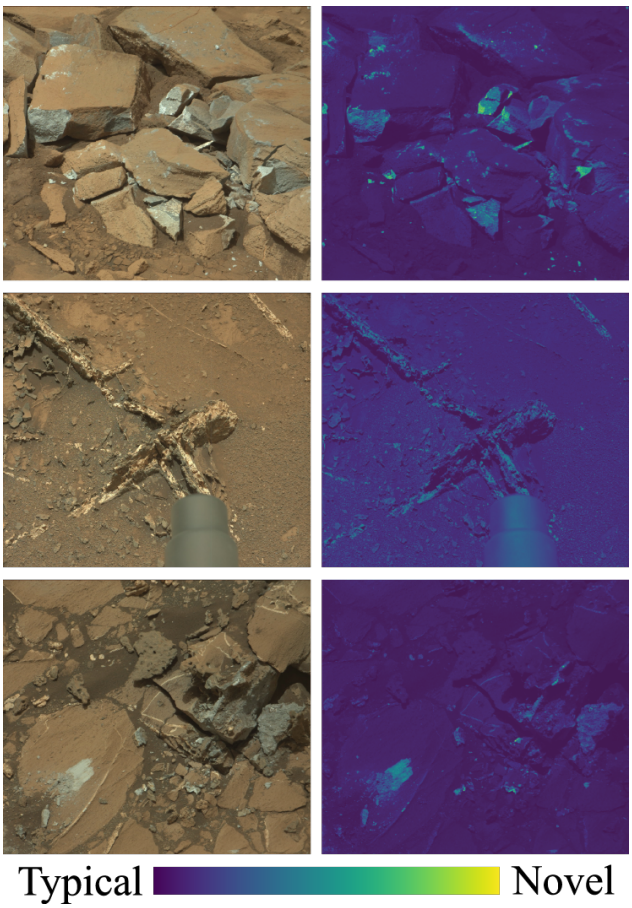


Figure 3. Examples of pixel novelty scores using RX for easy to spot novelties. The left column shows the RGB image from the sequence and the right column shows the output from RX with a color bar ranging from typical to novel. From top to bottom: Broken Rock (mcam05168), Veins (mcam04817), Dump Pile (mcam04892)

rover, so operations staff will undoubtedly be aware of its presence in the sequence. While these examples are novel and interesting, they are not the type of novelty that a tactical operations staff member is likely to miss, though highlighting these examples may help to pick them out of a large set of downlinked images.

To assist tactical operations planning, the novelty detection system is most useful when it is able to highlight features of interest that are not otherwise easily identifiable. Figure **??** shows an example of such a sequence. In this sequence, there is a ring of

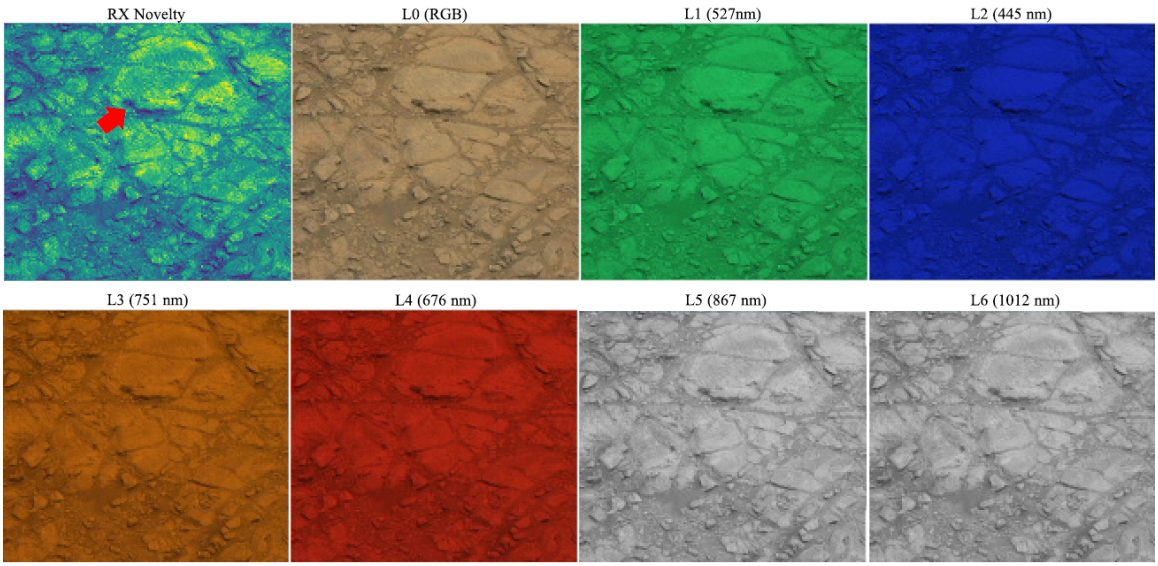


Figure 4. An example of difficult to notice novel features from mcam12276. In this example, there is a subtle ring of brighter dust on the edge of one of the rocks (shown by the red arrow). This is not noticeable in L0 (RGB) and faint in L5 (867 nm) and L6 (1012nm).

novel material around one of the rocks near the top of the thumbnail. This ring is almost unidentifiable in the RGB image and faint in filters L5 and L6. This ring is suspected by science team members to be due to a thin layer of brighter regolith deposited around the edge of the rock and not part of the rock composition. By taking every filter into account, RX is able to identify the spectral anomaly around this rock and provide guidance for follow up analysis.

In order to assess which sequences to analyze first, statistics about the pixel novelty scores can be calculated to rank multispectral sequences. Sorting by the mean pixel novelty score in each sequence orders the sequences based on mean novelty across the image. However, sorting by the mean score of all pixels in the sequence does not effectively prioritize images with obvious or localized novel features because the mean score can suppress high scores is small, localized features that are the only novel feature in the image. In contrast, images with high mean scores that have

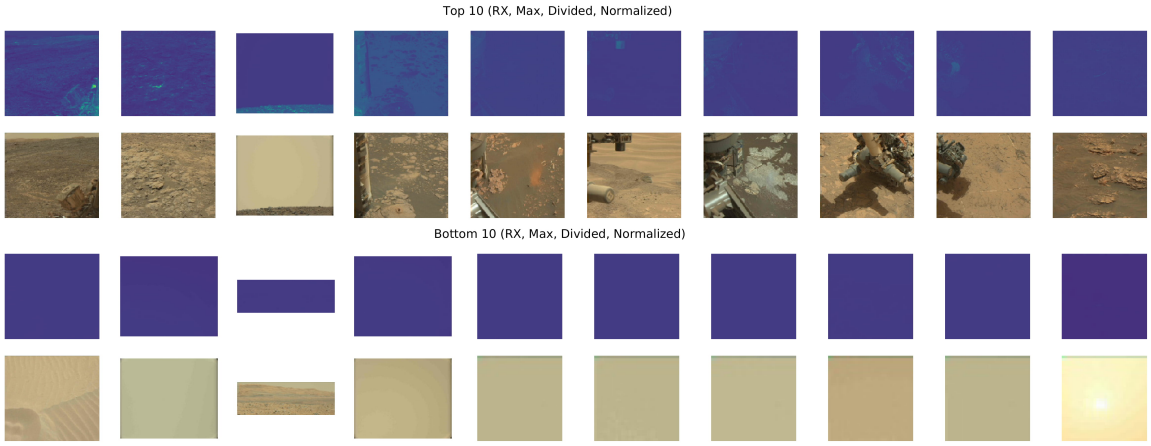


Figure 5. The most and least novel sequences in the Mastcam left eye data set. Sequences are sorted by maximum novelty value obtained from a brightness corrected background distribution. The pixel novelty scores are normalized across the entire data set.

uniformly high scores across the entire image will not show contrast in the heat map visualization and will be difficult to interpret using this visualization. To prioritize high novelty scores corresponding to localized features, we used the maximum pixel novelty score instead to sort novel sequences as it is better at prioritizing images with high novelty features. Examples of the top and bottom ten images sorted by maximum RX score are shown in Figure **??**. The top-scoring sequences show images of high variability when compared with the low-scoring images. The highest scoring pixel in these sequences are typically rover parts which, while being spectrally novel, are not necessarily useful for operations. Additionally, sequences containing shadows that move over time tend to have higher scores, as the shadow fans across the different channels and creates a rainbow. The low-scoring sequences mostly consist of sky and sand images which are relatively low interest from a tactical perspective. When comparing multiple images, we normalized the pixel novelty scores across the data set in order to display relative novelty. The scores are visualized without normalization when an analyst selects an individual sequence to analyze.

2.7 Conclusions

While initial results from reviewing the novelty detection output seem promising, it is difficult to evaluate the effectiveness of the system at assisting in operations without a longer term study in which many sequences would be analyzed for previously undiscovered features. This is necessary as generating a test set of undiscovered novelties is not feasible. The purpose of this study is to assist tactical planning so the key value is in the system's ability to highlight unique sequences and novel features within those sequences. The brightly colored heat map of novel regions for each sequence appears to provide a fast way of identifying spots for follow up analysis. The next steps for this work are to gather more feedback from the tactical planning team to determine the productivity increase while using novelty detection products. To quantitatively evaluate this, operations staff members can be given the task of analyzing Mastcam sequence with and without the novelty-based products. Analysis time can be recorded to identify if staff members are able to identify novel features quicker when the novelty detection system is used for triaging. Instances where the system highlights something that the staff member does not find novel can also help to understand the limitations of the method and improve future method development. This feedback will help to validate the usability of the novelty detection system and fine tune the features of the system, such as dividing by the brightness, global normalization, and limiting the training set to recent data. Additionally, the novelty detection products may improve masking rover parts and finding ways to reduce the rainbowing effect from shadows. Another approach may be to not use a background distribution based on the entire Mastcam multispectral history and

instead calculating new models for each image to find the most novel feature on a local scale.

Future work is also needed to improve ways of ordering the sequences based on their novelty scores. While sorting sequences by the maximum pixel RX score provides more useful outputs than sorting by the mean pixel RX score, there may be better ways of prioritizing sequences (e.g., sorting by the variance of scores in the image). To validate that the ranking's priorities align with the triage an operations staff member may perform, an experiment can also be created to have both humans and the novelty detection system generate rankings. The rankings can be compared using Spearman's rank correlation coefficient to compare their overlap. Such an experiment could also help to find better ways of sorting the sequences. Finding better ways to prioritize novel sequences is also beneficial for on-board autonomy applications **wagstaffnovelty**.

MSLWEB, an Arizona State University-based Mastcam tracking application, is the perfect platform for integrating novelty detection products into tactical planning workflows. Currently, MSLWEB supports the automatic generation of simple quick look products such as decorrelation stretches and filter ratios. Augmenting this system to support novelty detection would involve adding an inference endpoint to the pipeline and automatically passing new sequences through it. As this system is currently in use by operations staff, a strong justification is necessary to perform this integration. At the time, the novelty detection system has shown promising examples of novelty highlighting and sorting. Further analysis is needed to demonstrate how these examples would fit into the tactical planning pipeline and the time saving benefits it might provide.

Chapter 3

ON-BOARD SCIENCE DATA QUALITY ANALYSIS USING ANOMALY DETECTION FOR ASTHROS

3.1 Abstract

ASTHROS (Astrophysics Stratospheric Telescope for High Spectral Resolution Observations at Submillimeter-wavelengths) is a high-altitude balloon mission utilizing an array of sixteen spectrometers to create high spatial resolution 3D maps of ionized nitrogen gas in galactic and extragalactic star-forming regions. During data collection, we utilize on-the-fly mapping, where the instrument continuously collects spectra while scanning over a target area. After a sweep across the target, we take a calibration spectra to correct our science data. These calibration spectra provide a baseline for how the instrument is operating at a given moment. As we collect new calibration spectra, we can compare the current calibration with a series of past calibrations to determine if our system is producing anomalous spectra. Some examples of anomalous spectra are changes in RFI spike frequency, location, or amplitudes, changes in the overall readout level, and changes in the shape of the spectra. We compare statistical and data-driven methods for detecting these anomalies and evaluate their performance to determine the best fit for the ASTHROS readout system. For data-driven methods, we compare the latent space representation of our calibration spectra with past calibrations using models like Variational AutoEncoders (VAE) and Principal Component Analysis (PCA). By comparing with a rolling window of past calibrations, we allow our system to change gradually while identifying sudden irregularities. When spectra are labeled

Figure 6. An example scanning pattern for OTF mapping. ASTHROS will continuously produce ON spectra while scanning across the target and OFF spectra when observing a calibration source from (**mangum2007fly**).

as anomalous, they are prioritized for review so that the ground operations team can analyze and address the issue. On-board analysis is enabled by the readout system architecture which utilizes the RabbitMQ (RMQ) messaging networking. RMQ allows us to modularly build our readout system and create additional functionality, such as on-board analysis, without making modifications to the operation pipeline.

3.2 Introduction

ASTHROS (Astrophysics Stratospheric Telescope for High Spectral Resolution Observations at Submillimeter-wavelengths) is a high-altitude ballooning mission utilizing an array of sixteen spectrometers to create high spatial resolution 3D maps of ionized nitrogen gas in galactic and extragalactic star-forming regions (**siles2020asthros**). On-the-fly (OTF) mapping will be employed during data collection where each spectrometer will continuously produce ON and OFF spectra as we scan across our target as show in in Figure **??** (**mangum2007fly**).

Once data collection has started, our spectrometer array will continuously produce

and timestamp spectra and save the raw time stream to itself and a separate storage device. The clocks of all the computers on board ASTHROS will be synchronized using a local NTP server such that the timestamps can be harmonized with pointing data from the gondola to label ON and OFF spectra (**mills1991internet**). After a sweep across the target, we will find spectra taken during the OFF segments and use them to evaluate our system's performance. By generating an anomaly score for these calibrations, we can determine if our system is behaving differently than expected when compared to a series of past calibrations.

Currently, the methods for evaluating system performance are done on the ground by checking spectra manually as they come down. This is an error-prone process as it is difficult to manually check each calibration spectrum for anomalies. By implementing on-board anomaly detection, we can automate this process and prioritize analysis of anomalous spectra for review by the ground operations team. This will allow us to quickly identify and address any issues with the system during flight to catch errors while we're still able to correct them instead of waiting until the end of the mission to find out that our data is unusable. Even if the anomaly detection system is not perfect, any form of quantifiable measure of science data quality will be an improvement over the current manual process.

This paper will focus on the on-board analysis of our calibration spectra to detect anomalies in the system. Section **??** will go over the system architecture for ASTHROS and how we can utilize RabbitMQ (RMQ) to enable on-board analysis. After that, Section **??** will discuss the methods we used to detect anomalies in our calibration spectra. Then, in Section **??**, we will discuss the data we used to evaluate our methods and how we collected the data. Section **??** will present the results of our analysis and

conclude which method is the best fit for the ASTHROS readout system. Finally, Section **??** will discuss future work for implementation on ASTHROS.

3.3 System Architecture

The ASTHROS readout system is composed of three main computers that respectively handle commanding the instrument, storing science data, and analyzing our telemetry and science data. Our goal with this readout system is to be able to collect and analyze data in real time so that any changes in detector behavior can be quickly identified and addressed during flight. To achieve this, we utilize RabbitMQ (RMQ) to create a modular system that can be easily expanded to include additional functionality by creating a standard protocol for Remote Procedure Calls (RPC) and Publish/Subscribe (Pub/Sub) messaging (**dobbelaere2017kafkaversusrabbitmq**). RMQ is a lightweight messaging broker based communications to seamlessly create a network of computers that are able to share status and commands between multiple concurrent systems (**thompson2024architecture**).

The core of ASTHROS's readout are three systems: the Command, Storage, and Analysis computers as shown in Figure **??** (**horton2024readout**). The Command computer is responsible for sending commands to other processes and receiving status updates from the various publishers on the RMQ network. This computer also manages telemetry through a connection to the gondola. Key system devices, such as the power supplies and the antenna, are connected directly to this computer via serial connections to allow for real-time monitoring and control. Despite being physically on the same computer, control of each of these subsystems is handled by separate processes that communicate with the Central process through RMQ. This allows any

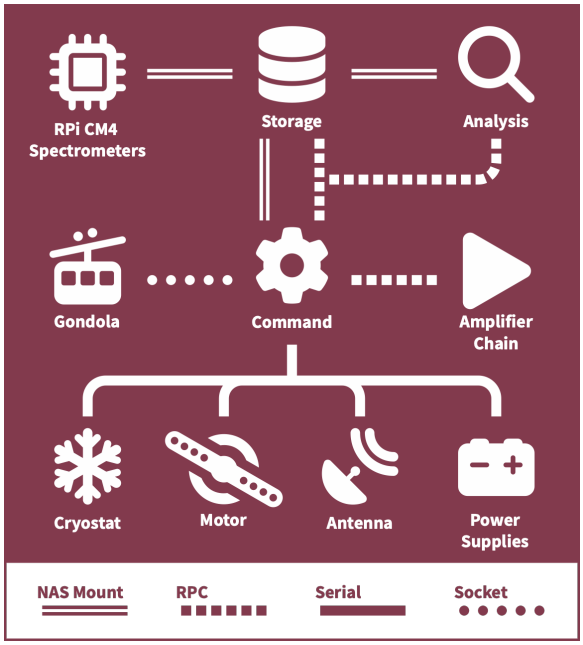


Figure 7. A high level overview of the ASTHROS Readout Network System Architecture showing the Command, Storage, and Analysis computers. The Command computer is responsible for sending commands to other processes and receiving status updates from the various publishers on the RMQ network. The Storage computer is responsible for storing all science data and telemetry data collected during the flight. The Analysis computer is responsible for analyzing the science data collected during the flight. The overview also shows some of the various subsystems and how they're connected in the network.

of these processes to be moved to a different computer or replaced with a new process without changing the overall system architecture. As most of the systems are located on the Command computer, we host the RMQ broker on this computer to minimize latency and reduce the number of network hops between the Command computer and the other systems.

The Storage computer is responsible for storing all science data and telemetry data collected during the flight. This computer is a custom-built rugged Network Attached Storage (NAS) that is able to store all the data collected during the flight. This NAS houses 4 Solid State Drives in a RAID 10 configuration (**chen1994raid**).

Other computers on the network, such as the Command computer and the Analysis computer, are able to mount the storage computer as a network drive to read and write data. Additionally, this storage computer has two managed network switches that allow for networking to every system on ASTHROS. Our spectrometers, controlled by Raspberry Pi CM4s, are connected to this computer via Ethernet and are able to store their data locally on their own Solid State Drives and send a copy to the storage computer. We do this for redundancy in case the storage computer fails during flight.

Finally, we have the Analysis computer which is responsible for analyzing the science data collected during the flight. This computer has the critical task of analyzing the calibration spectra to determine if the system is behaving as expected by measuring data quality. The methods described in this paper are run on this computer and are able to communicate with the Command computer to prioritize analysis of anomalous spectra via telemetry. Like other computers on the network, the Analysis computer is able to mount the storage computer as a network drive to read and write data. Using RMQ, the analysis computer is able to listen to the status exchange for pointing updates from the gondola. If the pointing data indicates that the telescope is observing a calibration source, the Analysis computer will pull the most recent calibration spectra from the storage computer and compare it to a series of past calibrations to determine if the system is behaving as expected. In addition to it's primary function of analyzing science data, the Analysis computer is also able to run additional processes to prepare data for the science team. It does this by time harmonizing the science data with the pointing data from the gondola to label each spectra with the correct pointing information so that it may be used for mapping.

Apart from the main three computers, the ASTHROS readout system also consists of a set of Raspberry Pi Compute Module 4s (CM4s) that control the spectrometers

and a set of Arduino Nano Every microcontrollers that control the amplifier chains. Both of these are configured to communicate over the RMQ network for Command and Control and produce status updates over the Status Exchange. The CM4s are responsible for controlling the spectrometers and collecting the raw time stream data from the spectrometers (**mohammed2024digital**). The Arduinos are responsible for controlling the amplifier chains to adjust attenuation values for our variable attenuators (**Ricardo**).

3.4 Methods

Anomaly detection, also known as outlier detection, is a way of finding out-of-distribution data points in a dataset (**kerner2022domain**). This works by comparing a data point to typical examples from the dataset and determining if the data point is significantly different from the rest of the data. Because of the nature of anomalies, we can't use traditional supervised learning methods to detect them as we don't have labeled examples of what an anomaly looks like (**horton2021integrating**). In other words, we don't fully know what we don't know and, instead of training our system on known issues, we need to train our system to recognize when something is different from what it has seen before. To do this, we need to use unsupervised learning methods that learn the underlying structure of our data and determine if a new data point is significantly different from the rest of the data. Figure **??** shows examples of the types of anomalies we have seen in past missions. Typically, these anomalies present themselves as changes in RFI spikes, the shape of the readout spectra, or the overall readout bias level.

For this implementation, we focus on two methods for determining if a calibration

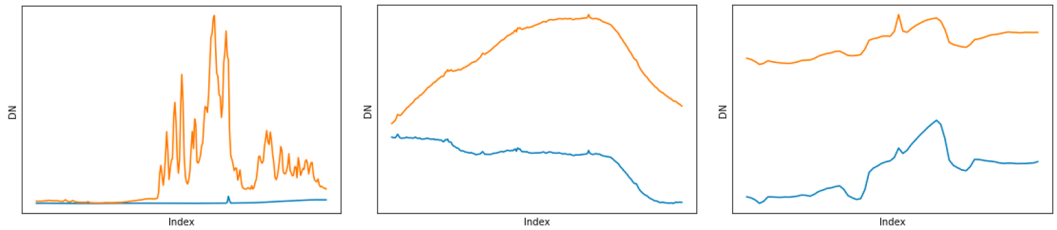


Figure 8. Examples of anomalous spectra from past missions. From left to right, we see changes in RFI spike shape and location, changes in the shape of the spectrum, and changes in the readout bias level.

spectrum is anomalous: Variational AutoEncoders (VAE) and Principal Component Analysis (PCA). Both have their strengths and weaknesses and we will compare their performance on the ASTHROS calibration spectra to determine which is the best fit for our system.

3.4.1 Variational AutoEncoders

VAEs are a class of AutoEncoders that learn the mean and variance of the latent space of the data (**kingma2013auto**). This is done by training the model to encode the input data into a lower dimensional space and then decode it back to the original input. We selected VAEs for this task because they are able to learn the underlying structure of the data and generate new examples of the data by sampling from the latent space. This is useful for anomaly detection as we can compare a new spectrum to the reconstructed spectrum to determine if it is significantly different from the rest of the data. When a new spectrum is input into the VAE, the encoded and then decoded spectrum will be different from the original spectrum if the new spectrum is anomalous.

The architecture for our VAE is a simple feedforward neural network with multiple layers. Our encoder consists of an input layer, a hidden layer, and two output layers

for the mean and variance of the latent space. The input layer has 8192 nodes, one for each channel in our calibration spectra. The hidden layer has 256 nodes which feeds into our output layer which has 64 nodes of mean and variance. We re-parameterize the mean and variance to sample from the latent space and feed this into the decoder which has the same architecture hidden layer and input layers as the encoder but in reverse. The output layer of the decoder has 8192 nodes to reconstruct the original spectrum.

To train the VAE, we have to normalize our spectra to values between 0 and 1. We then split our data into a training and testing set and train the VAE on the training set. We then use the VAE to encode the testing set to it's latent space representation and decode it to produce a reconstructed version.

To generate an anomaly score from this method, we take a test spectrum, transform it using the VAE to it's latent space representation, and then decode the spectrum to produce a reconstructed version. This reconstruction is imperfect as the VAE model is fit to our typical spectra. We compare the reconstruction to our original spectrum to calculate error from the difference between the two specra. This score can be used as a time series of anomaly scores to determine if a new spectrum is significantly different from the rest of the data. For flight, we will have to retrain the VAE on a subset of the data to account for changes in the system over time making more challenging for implementation.

3.4.2 Principal Component Analysis

PCA is a method that reduces the dimensionality of a dataset by finding the directions of maximum variance in the data (**wold1987principal**). This is done by

finding the eigenvectors, called principal components, of the covariance matrix of the data and projecting the data onto these eigenvectors. The first principal component is the direction of maximum variance in the data, the second principal component is the direction of maximum variance orthogonal to the first principal component, and so on. Because our calibration spectra are high dimensional with 8192 channels, we use PCA to reduce the dimensionality of our data to a lower dimensional space where we more easily compare the spectra.

To use PCA for anomaly detection, we first fit the PCA model to a series of past calibration spectra. This can be done in two ways, fitting the PCA model to a subset of the spectra in an entire dataset and using a rolling window of past spectra to create unique fits for each additional spectrum. The first method can be used to determine if PCA is a good method for fitting to the dataset by seeing how many principal components are necessary to cover the variance of a typical spectrum. The second method can then be used in a real time setting to determine how a new spectrum compares to the recent history of previous spectra.

To generate an anomaly score from this method, we take a test spectrum, transform it using the fit PCA model to it's latent space representation, and then inverse transform the spectrum to produce a reconstructed version. This reconstruction is imperfect as the PCA model is fit to our typical spectra. We compare the reconstruction to our original spectrum to calculate the mean squared error from the difference between the two spectra. We use this value as our anomaly score as it provides a measure of how different a given spectrum is from previous spectra.

3.5 Data

The data for these tests are taken from two separate data collection instances. The first was a High-Altitude Student Platform (HASP) flight (**guzik2008development**) that tested one of our Pacific MicroCHIP Corp (PMCC) Spectrometers (**mohammed2024digital**). A team of students at Arizona State University test flew a spare PMCC spectrometer to measure the deuterium to hydrogen ratio in the atmosphere (**HADHR**). Unfortunately, the spectrometer was not calibrated properly so many of the spectra were unusable for science. Additionally, due to the HASP payload's suite of student projects, we had many instances of changing RFI due to other instruments on board. Ironically, with all these problems, this dataset is an ideal candidate for testing the anomaly detection system as we are able to use it to evaluate good and bad spectra. This dataset consists of about 7,000 spectra of varied levels of noise.

The second dataset was taken from one of our 100GHz integration tests with a flight calibrated spectrometer. This data is extremely stable and provides us an ideal

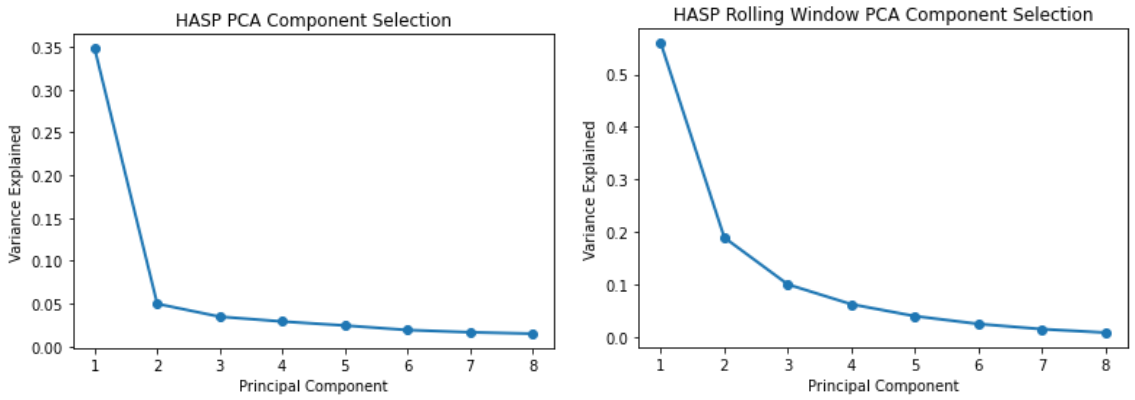


Figure 9. Elbow plots for the HASP dataset. Left shows the amount of variance each component accounts for in the entire dataset and the right shows the average variance amount when using a rolling window of 10 spectra.

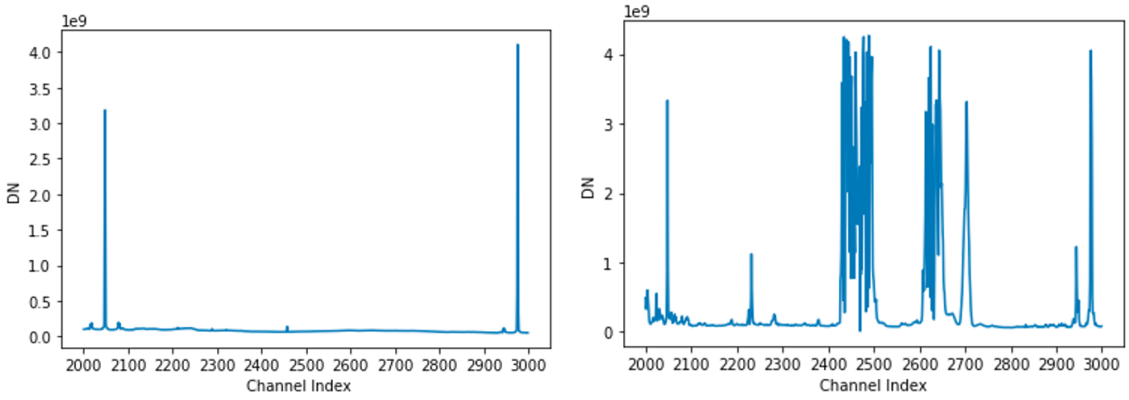


Figure 10. Examples of spectra with the lowest (left) and highest (right) anomaly scores from the HASP dataset when using the PCA method to fit the entire HASP dataset.

baseline for what we expect to see during flight. Despite being an extremely clean dataset of over 20,000 spectra, there are instances of RFI in the data that can be used to demonstrate the real time data quality assessment of the spectra.

One limitation of this dataset is that there aren't examples of changes in the baseline shape or bias readout level. These changes are rare and typically caused by physical changes in the instrument's hardware. In order to account for these, we would need to inject simulated anomalies in the data. This is outside the scope of this work but will be discussed in Section **??**.

3.6 Results and Conclusions

Both datasets are unlabeled, which makes measuring accuracy difficult so we have to use more qualitative methods to compare the two. The end use case of this is to provide a trackable metric that the operations team can use to monitor data quality. While accuracy is certainly important, any method that is able to enhance the manual

monitoring process is sufficient. As such, we compare the methods based on their ability to be implemented into the ASTHROS System Architecture and how well the anomaly score metric tracks with manual monitoring of the time series of spectra over time. Additionally, we can compare the least and most anomalous spectra form the system to what sorts of anomalies produce high anomaly scores.

With both datasets, the large number of features in the 8192 channels, made it difficult for the VAE to learn the latent space for the spectra. Even after we managed to get it to behave properly, it became apparent that this method wouldn't adapt well for a rolling window of previous spectra. We would have to retrain the system on the fly to ensure that it adapted to gradual changes in instrument behavior. With more refining and tests with domain adaptation, we may be able to revisit this method but, due to the overhead of having to retrain and the overall complexity of the system, we opted to focus on more adaptable methods.

PCA was an ideal method for this use case. We began with the HASP dataset to see how the method would be able to rank spectra from least to most anomalous. We first fit all of the 7000 spectra with PCA to determine the number of components that would be optimal for comparison. Most of the variances was captured with the

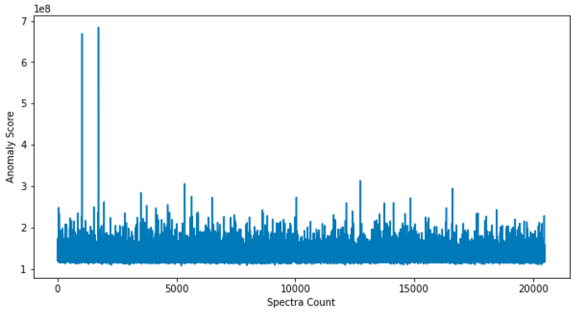


Figure 11. A time series of anomaly scores using the PCA method for the 100GHz integration test data. Two large spikes are visible in the data indicating large changes between a spectrum and the previous spectra in the rolling window.

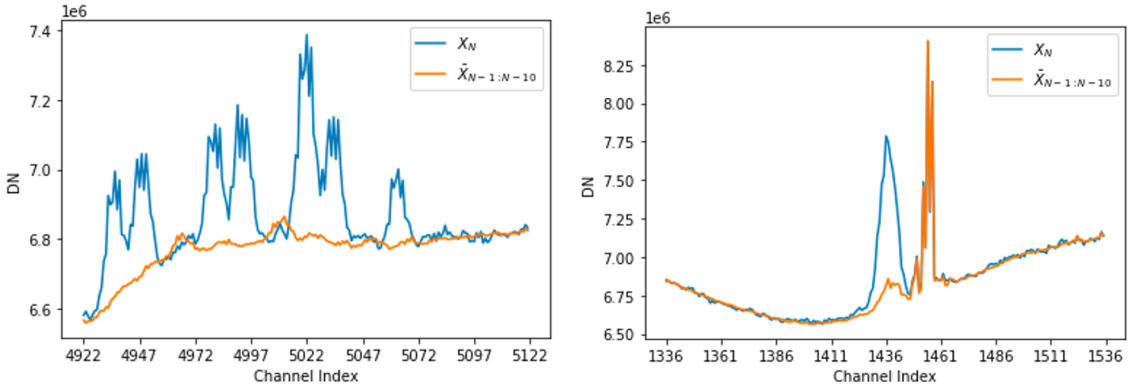


Figure 12. Examples from the 100GHz integration test data showing the spectra at the time of the two spikes in Figure **??**. The blue line shows the spectra that has the high anomaly score and the orange line shows the average spectrum from the past 10 spectra. These spectra were highlighted due to the PCA method's inability to recreate the features shown in the new spectra when trained on the rolling window of previous spectra.

first two principal components but, we opted to use five components as the variance increases when we're only looking at a portion of the dataset as shown in Figure **??**. Using these number of components, we calculate the anomaly score for all of the spectra in the dataset to review the least and most anomalous spectra. See Figure **??** for the least and most anomalous spectra in the dataset. We've zoomed into a range of indices that were heavily affected by RFI, likely from another instrument. As expected, the least anomalous spectra look relatively clean and the most anomalous spectra are all very noisy.

On the cleaner 100GHz dataset, the method truly shined. Using the same rolling window of 10 spectra and 5 principal components, we were able to produce a time series of anomaly scores for all 20,000 spectra as shown in Figure **??**. These values on their own don't provide much information but, when compared with other values across the dataset, we are able to pick out instances where our anomaly scores spiked. In the time series, there are two major spikes in the time series and a handful of

smaller spikes throughout the test. We looked into these larger spikes in anomaly score and, they corresponded with spectrum that had changes in their RFI spikes. Figure **??** shows the affected channels of these anomalous spectra by overlapping them with the average of the previous 10 spectra. As shown in both examples, the new spectra has an increase in RFI that is different from the most recent spectra. On the other hand, the least anomalous spectra all closely match their predecessors. Of the tested methods, PCA has shown the most promise for implementation into the ASTHROS Readout System.

3.7 Future Work

The PCA method tested in this paper is currently being implemented onto the ASTHROS Readout Systems to test it in operations during ASTHROS's flight in 2024. The method will run on the analysis computer and integrate with the rest of the ASTHROS network to provide real-time data quality status messages over the RMQ status exchange. During integration testing, the PCA method will be tested to ensure that it can handle the data rates and data sizes that will be produced during the flight.

While the VAE method was less suitable for our use case, with more fine-tuning and optimization for the rolling window of past spectra, it could be a viable method for detecting anomalies in the future. There are other methods, such as a Gaussian Mixture Model (GMM) or a Hidden Markov Model (HMM), that could be tested in the future to see if they are more suitable for our use case as well.

A limitation of our datasets are the lack of changes in shape of the spectrum or changes in bias level. In the future, we will make a simulator to inject these

anomalies into our datasets to further test these methods. Additionally, with the flight of ASTHROS, we will have real calibration data to test everything on and reassess how we can improve our methods after using them for operations.

Chapter 4

ANOMALY DETECTION FOR THE ROMAN SPACE TELESCOPE WIDE FIELD INSTRUMENT'S SCIENCE DATA PROCESSING PIPELINE

4.1 Abstract

The Roman Space Telescope (RST) Wide Field Instrument (WFI) will be utilizing a preliminary Science Data Processing (SDP) pipeline during its Integration and Test, and to some extent during Operations, to track basic statistics and identify known features such as cosmic rays, snowballs as well as possible anomalies in raw detector data. In our detectors, these anomalies appear as jumps in the ramp of a readout and are classified as cosmic rays if they appear as a streak or snowballs if they're more circular. The WFI employs an array of 18 H4RG-10 detectors that collect image samples. Each set of raw frames within a non-destructive exposure is packaged by the SDP pipeline into image cubes for each detector. Each cube is a time series of 4096×4096 accumulating pixel frames. The preliminary analysis pipeline is used to locate anomalies in these time-series accumulation frames and identify the type of anomaly, either natural phenomena or detector characteristic. To compare different methods, we've implemented both heuristic-based and data-driven methods to identify anomalies. For the heuristic-based approach, we identify snowballs and cosmic rays by the size and shape of outlier pixel clusters between consecutive frames. For data driven methods, we evaluated a Convolutional Neural Network (CNN) model, and more traditional methods like Principal Component Analysis (PCA). CNN is a supervised learning/classification method. Thus, we used a labeled dataset of

anomalies to perform segmentation of the image and identify anomalies. We used previously identified cosmic rays and snowballs to measure the accuracy and efficiency of the mentioned approaches. In evaluating these methods, we aim to pick the best fit for the SDP pipeline's anomaly detection in terms of both performance and runtime.

4.2 Introduction

The Roman Space Telescope (RST) Wide Field Instrument (WFI) employs an array of 18 H4RG-10 detectors to collect image samples. Each detector on the WFI utilizes a up-the-ramp readout scheme that produces 4096×4096 pixel images at each frame along the exposure. For a given exposure, each frame gives us information on the amount of light collected over time and allows us to identify both the location and time of anomalies within a ramp. During an exposure, natural or detector-related events may occur that affect groups (or sets) of pixels across the detector. Because we're collecting time-series information, we can see the exact frame that the event occurs and observe how the event affects subsequent frames. There are many different sources that might cause errors in our detectors' data, such as read noise patterns, thermal noise, compression errors, and software errors, but in this paper, we are particularly interested in external and natural sources of errors (**cillis2018snowballs**). Both cosmic rays and snowballs are transient events that result in sudden increases in charges and pixels' Data Number (DN) values, compared to their typical neighboring pixels. These events are also rare. Therefore, we can use anomaly detection to identity these outliers. Figure **??** shows examples of these two events.

An anomaly is defined as an unexpected occurrence in a sequence based on a data set of typical sequences (**horton2021integrating**). Given our image is mostly

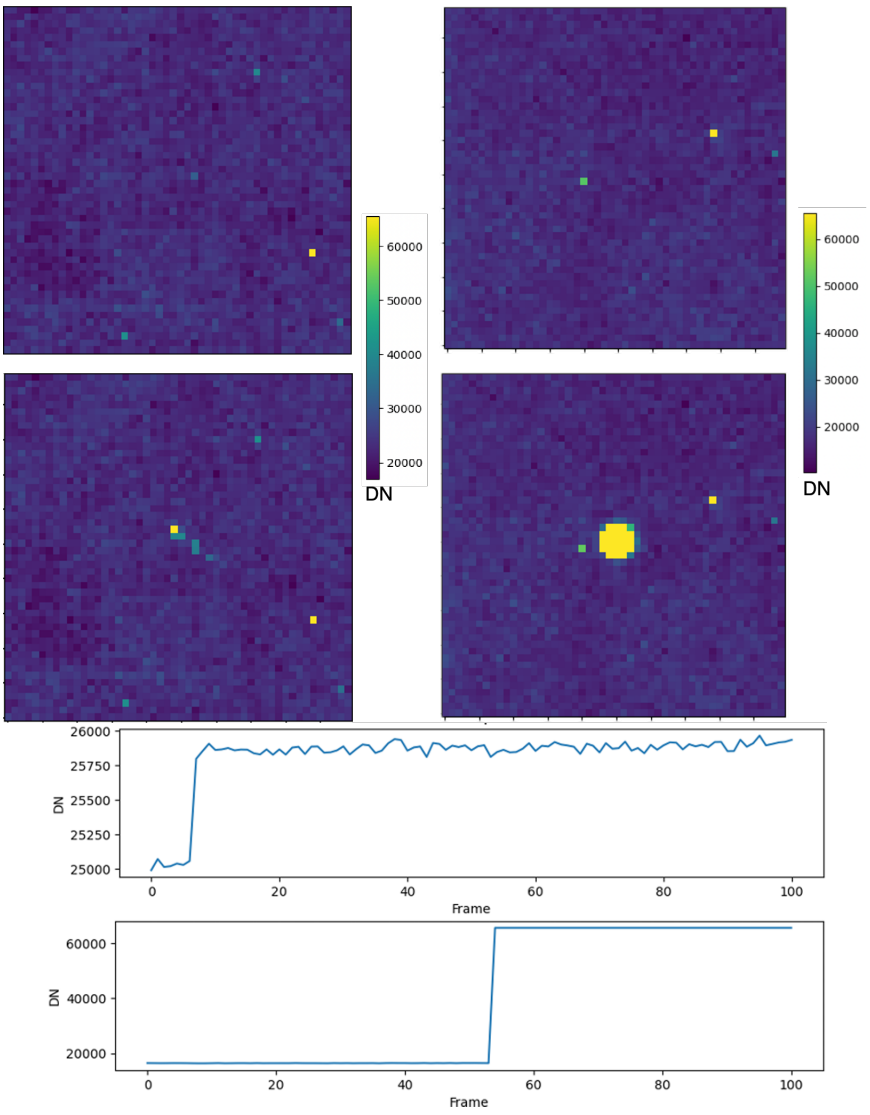


Figure 13. An example of a cosmic ray (left) and snowball (right) in RST's detector data. The top frames show the frame prior to the event and the bottom frames show the area after the event occurred. The bottom graphs show the overall DN value across the entirety of the exposure, showing large jumps at the time of the event for the cosmic ray (top) and snowball (bottom).

typical pixels, we can use this feature to identify out of distribution pixels that may be caused by snowballs or cosmic rays. While snowballs are a relatively newly recognized detector phenomenon, detection and rejection of snowballs are common practices for astronomical images (**van2001cosmic**). These cosmic rays appear as streaks across multiple pixels within an exposure as the incident cosmic ray imparts energy as a point spread function across neighboring pixels (**pych2003fast**).

Both snowballs and cosmic rays will appear as jumps in DN value in a ramp but the shape of the affected neighboring pixels helps us differentiate between the two. Snowballs will be circular in shape and affect 8 or more pixels (**cillis2018snowballs**). On the other hand, Cosmic Rays will be any grouping larger than 2 pixels that shows a linear streak.

The goal of our anomaly detection is to highlight these features as part of the RST WFI SDP's pipeline. As such, SDP's processing time should keep up with data generation time so that we can identify issues within the detectors as we collect new data. This poses a challenge due to the massive data size of the data products from the array of 18 detectors. To address this challenge, raw data is automatically processed into interpretable products as part of the SDP pipeline. As this anomaly detection system will be part of the SDP pipeline as an additional data product, the runtime of any method must be taken into account along with the method's accuracy.

The rest of this paper is organized as follows. First we go over the methods for identifying and classifying snowballs and cosmic rays in Section **??**. Then, in Section **??** we explain the dataset chosen and how labels for the dataset were generated. After that, we discuss the results of the different methods and draw conclusions about which methods to implement in the SDP pipeline in Section **??**. Finally, we go over future works in Section **??**.

Step 1: Locating Anomalies	**Step 2: Classifying Anomalies**
Statistical Thresholding	Heuristic Rules (**cillis2018snowballs**)
Principal Component Analysis (PCA) (**cillis2018snowballs**)	Convolutional Neural Network (CNN) (**gu2018recent**)

Table 1. Methods for Locating and Classifying Anomalies

4.3 Methods

To identify the anomalies in our data, we split our problem into two distinct tasks: 1) locating and grouping pixels that have irregular ramps and 2) classifying those pixels as cosmic rays, snowballs, or something else entirely. By splitting our anomaly detection into two tasks, we can use the first step to identify regions of interest and reduce our search area for snowballs and cosmic rays to that of anomalous pixels. This can drastically reduce the amount of pixels we have to classify as we also identify the frame that the anomaly occurred in, reducing our focus to a smaller window around the event. The following are potential candidate methods for accomplishing these two tasks.

There are other viable methods for accomplishing these tasks, such as Reed-Xiaoli (RX), Localized Reed-Xiaoli (LRX), and James Webb Space Telescope's (JWST) Bayesian Generalized Least Squares (GLS) for locating anomalies, but the scope of this work will focus on the methods in Table **??**. RX is an unsupervised learning methods that uses a window around a test pixel to compare with the local background (**reed1990adaptive**). LRX is similar to RX but instead uses a double concentric window to compare the test pixel with a guarded local background (**molero2013analysis**). JWST's GLS estimator is also an unsupervised method that calculates the slope, or ramp, of each pixel along the time domain and deter-

mines the probability that discontinuities in the ramp occurred due to cosmic rays (**robberto2015cr**). Please note that the JWST data was constructed similarly to the RST's, resulting in non-destructive accumulating frames.

Before any of the data is used for these methods, they must be loaded and pre-processed. For more information about this and the dataset selection, see Section **??**.

4.3.1 Locating Anomalies

To locate the anomalies, we looked for methods that would be able to not only identify where a potential anomaly is located but when the event occurred that caused the anomaly.

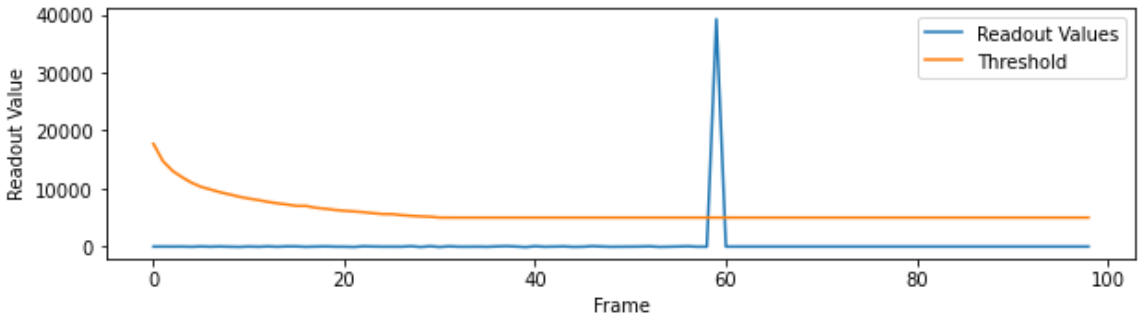


Figure 14. The readout values for a flagged pixel. The read out values are the amount of DN gained in a given frame. We compare this value to our calculated threshold and, if surpassed, that specific location and frame is flagged for classification.

4.3.1.1 Statistical Thresholding

Our initial approach is readout thresholding, where we take two subsequent frames, x_i and $x_{(i-1)}$ where n is the number of frames in an exposure. For each readout frame Δx_i, we calculate the mean μ_i and standard deviation σ_i across its pixels. We use these values to calculate the threshold for jumps in the ramp with a minimum threshold value of 5000 DNs.

$$\Delta x_i > \max(\mu_i + 50\sigma_i, 5000) \tag{4.1}$$

These values were chosen for dark frames as they allow us to easily identify large jumps traditionally associated with snowballs or cosmic rays. From here, we create a pixel mask for each frame of all readout values that exceed this threshold. We then preform a series topological transformations to bridge and fill in incomplete holes through dilation, binary hole fill, and erosion procedures using SciPy's ndimage library (**2020SciPy-NMeth**). Finally, we remove any grouping of pixels of two pixels or less to ensure any jumps in the ramp we discover affect multiple pixels and then use scikit-image's measure library to locate the central pixel for each group (**scikit-image**). We are then left with groupings of pixels that could potentially be either a cosmic ray or snowball.

4.3.1.2 Principal Component Analysis (PCA)

Given the majority of pixels within our image aren not affected by anomalies, we can use a random subset of pixels within an exposure and fit PCA to the ramps of these samples. If we limit our principal components to two, we can create a fit for the majority of simple ramps. Using our PCA fit, we can reduce all of the pixels in our image to our latent space representation and then inverse transform them to compare the original and reconstructed result (**wold1987principal**) Examples of this are shown in Figure **??**.

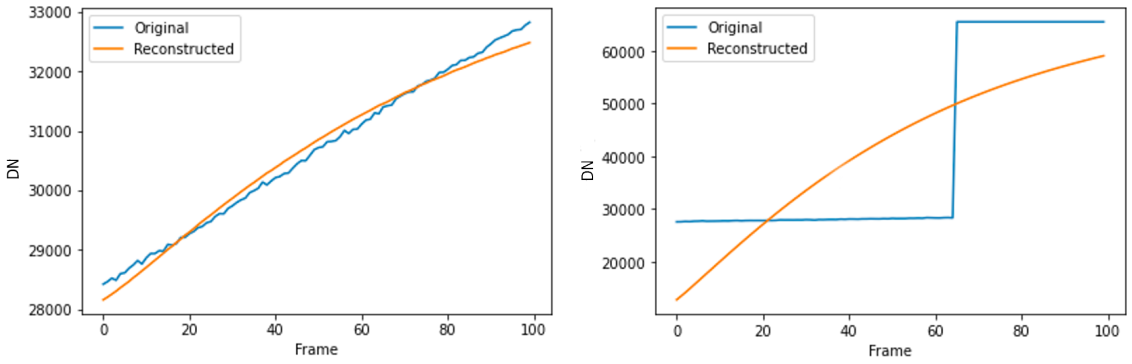


Figure 15. Examples of reconstruction of ramps using PCA. The DN values for the reconstructed image are calculated by transforming the original values and then inverse transforming our original ramp using only a subset of the principal components (i.e. two) to reproduce the shape. By limiting the number of principal components to two, we can have the reconstructed image fit the shape fairly well for normal ramps and not fit ramps with anomalies.

With the reconstructed values, we are able to calculate the residual ($Image_{Reconstructed} - Image_{Original}$) of the errors and identify where anomalies occur based on spikes in the ramp of the residual. To identify these spikes, we can employ a similar method to the Statistical Threshold method to calculate the change in residuals between each frame and find out of distribution values. This results in identifying the frames for each pixel that have large jumps in the residual. By limiting the number of PCA components to two, our reconstructed ramps will be relatively smooth, causing original ramps that have large jumps in them to have a large change in residual at the frame of the event.

Finally, like the statistical method, we create the pixel mask by highlighting any pixel whose change in residual is above the threshold. We then identify regions using the same topological transformations and scikit-image's measure library for finding blobs (**scikit-image**).

4.3.2 Classifying Anomalies

From the locating anomalies methods, we obtain a frame and pixel mask matrix containing information about potential anomalies with a ramp for each pixel in an exposure. We also obtain a table listing groupings of pixels to classify. We call this table the Event Table as it describes the time and location of potential anomalies. These groupings are the input into our classification methods as they help limit the search to specific pixels and frames across the exposure.

4.3.2.1 Heuristic Rules

For heuristic rules, these groupings are labeled and measured by the number of pixels affected and the major and minor axes lengths of the affected area. We use these values to determine the type of anomaly. Snowballs are large circular anomalies that cover 9 or more pixels. Cosmic rays are oblong anomalies that cover more than 2 pixels. We determine the circularity of the anomaly be comparing the minor and major axis using the following criteria.

$$\text{Circular: minor\_axis} \geq \text{major\_axis}/2$$

$$\text{Large: area} \geq 9$$

From this we are able to produce two output products: a new data cube where each frame is a mask identifying pixels affected by anomalies, and an updated Event Table with information about each anomaly.

4.3.2.2 Convolutional Neural Network

For each listing in the Event Table, we take a 32 square pixel sample region around the central pixel spatially and the three frames around the event frame for a sample of $32 \times 32 \times 3$ pixels. This is the input into our CNN which is trained using hand labeled data from the DCL dataset. The CNN architecture processes input images through two convolutional layers followed by ReLU activations and max pooling, which flattens the output. Finally we pass the output through two fully connected layers to produce class scores for None, Cosmic Rays, Snowballs, and Potential Anomalies classes based on the labeled dataset. We preformed two tests with the CNN by training and testing on our all of the labeled dataset and just the data from the same detector to see how well the method generalizes. Both tests were preformed with an 80/20 test/train split with balanced classes. The output classes from the CNN are then used to label the pixels in the mask with their associated class and update the Event Table with anomaly labels.

4.3.3 Output Products

To align with the rest of the products from the SDP pipeline, we package the outputs from the anomaly detection pipeline as Hierarchical Data Format 5 (HDF5) files (**The_HDF_Group_Hierarchical_Data_Format**). HDF5 is a file format that is designed to store large amounts of data and is perfect for the types of products we need to produce for the SDP. Because of HDF5's efficient read/write procedures, the SDP is able to keep up with the data generation rate while producing analysis products. For each exposure, we produce three binary mask arrays for events labeled

as Cosmic Rays, Snowballs, and Potential Anomalies for each pixel within a ramp. These binary masks are the same shape as the exposure and allow us to quickly identify and flag problem pixel/frame combinations in an exposure. We also produce a single image for each anomaly type where each pixel is the frame number of an event. This allows us to visualize anomalies and see how they may change throughout an exposure. Finally, the Event Table is formatted so that each row in the HDF5 array is an anomaly that has central pixel, size, shape (semi-major and semi-minor axis), and classification as columns. These products are produced during SDP pipeline and added as part of the automated report.

4.4 Data

The H4RG detectors used in the WFI array are able to perform non-destructive reads while producing an exposure resulting in measurements through time, also referred to as up-the-ramp measurements. This allows us to take the up-the-ramp measurements taken during an exposure and order them in a series to create frames within the exposure. This results in a time series of frames for each integration from a detector's pixel values being reset at the beginning of an exposure to the final accumulated pixel values at the end of an exposure (**casertano2022determining**). For testing the effectiveness of the anomaly detection pipeline, we will be utilizing real exposures taken during the selection phase of the flight detectors for the WFI detector array. During the selection phase of flight detectors, over 70 detectors went through numerous experiments with different types of light exposures, both bright and dark. The dark tests consisted of a two-hour long exposure where detectors accumulated across 100 frames. These tests are ideal to identify cosmic rays and snowballs due to

their long exposure time, resulting in more opportunities for anomalies to appear in the frames. For the purposes of testing these methods, this paper focuses on using just these dark exposures.

The data from these tests are provided in the form of Flexible Image Transport System (FITS) files that consist of 101 frames of 4096×4096 pixel images (**wells1979fits**). The first frame in an exposure is the reset frame and can be disregarded. The FITS data is loaded into a NumPy array of unsigned integers by iterating over the array (**harris2020array**). To correct the read direction of FITS data, the data is processed by subtracting every pixel values from the maximum possible DN value for each pixel, $2^{16} - 1$, across all frames. This leaves us with a data cube of 100 frames, each with 4096×4096 pixels.

The data provided by the DCL lab contains labeled information about known snowballs during the exposure. These snowballs are our preliminary ground truth for identifying the effectiveness of our methods. In addition to these snowballs, the outputs for each method are reviewed to identify snowballs and cosmic rays that are not in the original labels Because the heuristic method is overly sensitive, each anomaly highlighted by the method was hand labeled as potential anomaly, cosmic ray, snowball, or non-anomaly. These hand labels are used to measure the accuracy of each method and train methods such as the CNN in Section **??**.

4.5 Results and Conclusions

As we are able to split our system into two different subsystems, the results here will discuss which of the methods are best for accomplishing each individual task. Then we will go more in depth with the best pairing of methods integration into the

RST WFI SDP. For consistency all of the tests were preformed on a 2021 M1 Macbook Pro Max with 64GB of unified memory. Much of the development of this work was done on NASA Center for Climate Simulation PRISM GPU cluster.

4.5.1 Locating Anomalies

The two methods we tested for locating anomalies were statistical thresholding and PCA. Both methods look for large jumps in a time series but each method is looking for a different type of jump. The statistical thresholding method is purely looking for jumps in the DN accumulated each frame by identifying large changes in DN compared to the rest of the accumulated DN in the frame. The PCA method trains on a subset of pixels across the exposure and is looking for large jumps in the residual error from transforming and inverse transforming each pixel. For performance, both methods require calculating the difference between values, a threshold, and then identifying groups of flagged pixels. On average, this takes about 2 minutes and 10 seconds for each dark image with 100 frames. Because the PCA methods needs to fit, transform, and inverse transform the entire exposure, running it adds on an average of 42 seconds to locating anomalies. This makes the Statistical Threshold method faster than PCA in every experiment due to PCA requiring pre-processing.

As for performance, both methods are over-sensitive to false positives. The PCA method found less groupings than the statistical thresholding method but many of the omitted locations are false negatives. We chose to use two principal components as any higher numbers of components would have fit to jumps in the ramp and caused lower errors. Two components also gives us an overall shape that matches a typical ramp and explains most of the variance of the exposure's ramps without over fitting

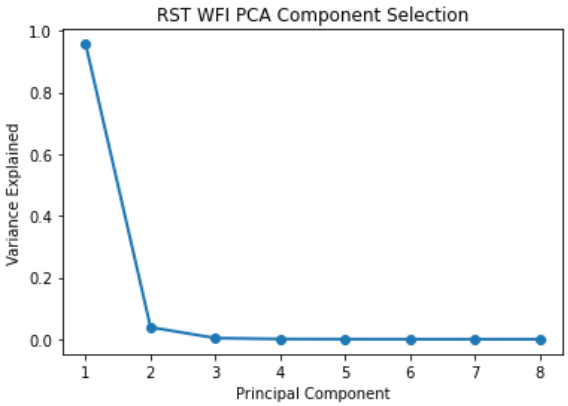


Figure 16. Elbow plot for the PCA method on typical ramps. A limit of 2 components was selected to ensure we aren't over fitting to the curve and missing anomalies.

to the curve as shown in Figure **??**. This false negative rate varies from exposure to exposure but ranged from 65 to 80% missed anomalies. Despite being overly sensitive to any jump in the ramp, the statistical thresholding method is preferred here due to it's higher accuracy and lower runtime. This method had a false negative rate of 10% across the entire labeled dataset of over 5000 hand labeled events. A method that filters most of the image without removing real anomalies is preferable to a method that filters more of the images including anomalies. The second step, Classifying Anomalies, can then label these false positives as None or Potential Anomaly.

4.5.2 Classifying Anomalies

For classifying anomalies, we have the traditional method using heuristic rules around the shape of the grouping and a CNN trained on samples from the labeled anomalies. We utilized the output of the statistical thresholding method as the input into both of the classification methods. Neither method were spectacular at

accomplishing the task but the heuristic rules method did outperform the CNN by a wide margin.

We experimented with the CNN by training it on a subset of events in the Events Table that were hand labeled by humans and then testing it on a separate set of events. We tested training using a subset of all exposures in the labeled dataset as well as just exposures from a specific detector. In both cases, we were unable to have the CNN generalize to different exposures. Because of the class imbalance and hard to differentiate nature of the dataset, the CNN would misclassify cosmic rays as potential anomalies and almost never correctly identify them as cosmic rays unless the streak was large enough to differentiate. The CNN was able to correctly classify snowballs about 50% of the time, labeling them as potential anomalies in other cases. In the future, this method may be better suited than heuristic rules but further refining of the labeled dataset is necessary. As it stands, with the overlap between cosmic rays and potential anomalies, there is little room for improvement without further defining how we want to differentiate these classes.

Heuristic rules outperformed our CNN method in both accuracy and runtime. The outputs from Event Table in locating anomalies include measurements of the size and shape of the anomaly. This makes it very efficient to run through each entry of the table and pre-define a look up table to classify each anomaly. The time the heuristic rules pipeline takes to generate a report depends on the number of anomalies in the Event Table but, on average, it is able to classify 100 events from the Event Table a second.

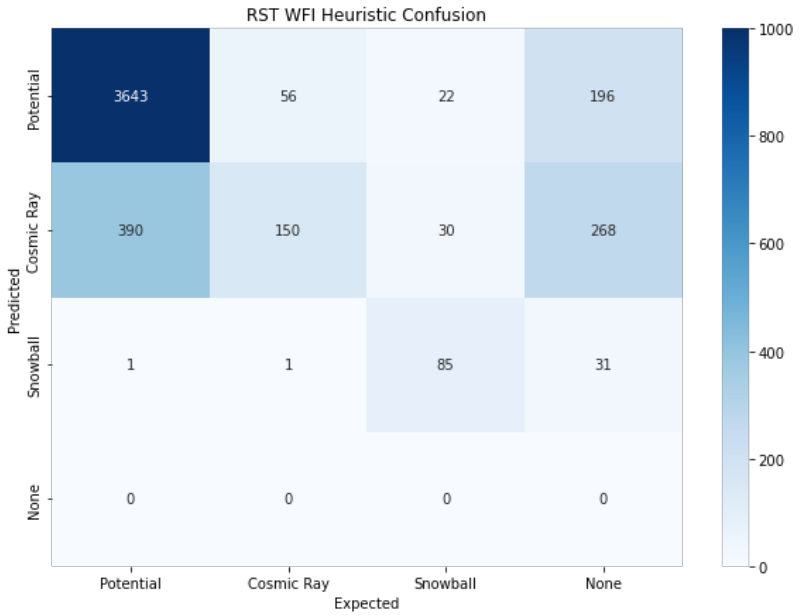


Figure 17. Confusion Matrix for the Heuristic Rules Method with Located Data from Statistical Thresholding.

4.5.3 SDP Pipeline Integration

Of the methods tested, the best combination of methods for the SDP pipeline would be the Statistical Thresholding method for locating anomalies and the Heuristic Rules method for classifying anomalies. This combination method was able to find every pre-labeled snowball from the original dataset and correctly identify 85 out of 137 (62%) of hand labeled snowballs. The method also over classified cosmic rays by identifying many potential anomalies and non-anomalies as cosmic rays. This is likely due to the less rigid criteria for identifying cosmic rays compared to snowballs. Figure **??** shows the confusion matrix for the Heuristic Rules method with the located data from the Statistical Thresholding method. This method is currently being used in the SDP pipeline to identify anomalies in the WFI detector data.

4.6 Future Work

This work is a preliminary analysis of the anomaly detection methods for the RST WFI SDP pipeline. There are many areas for improvement and future work to be done. First and foremost, more methods will need to be evaluated in order to determine a better alternative to the Heuristic Rules method for classification. The CNN method is a good candidate but needs more fine tuning to generalize to different exposures. In addition to this, more labeled data will be needed to train the CNN and other methods, such as Mask RCNN or transformer-based architectures, particularly on illuminated data. This labeled data could come from other tests if we were to generalize beyond just the dark exposures. With a more well defined dataset of labeled data, we will be able to better assess the accuracy of these methods too.

Another area for improvement may be finding better ways to locate anomalies. We may revisit PCA to see if we can adjust the parameters (number of components and threshold for error) to see if we can have it not discard real anomalies. Additionally, other methods like RX, LRX and GLS, as described at the beginning of Section **??**, could be better suited for filtering the initial exposure.

The current implementation of the SDP pipeline uses a bad pixel mask to mask regions of the image that are problematic. Because we didn't have a bad pixel mask for the data we were using, we had to analyze the entire image with our anomaly detection methods. This led to a large number of false positives from known bad pixels that appeared as anomalies in the dataset due to their sharp ramp in DN values at the beginning of the exposure. In the future, we will need to integrate the bad pixel mask into the anomaly detection pipeline to filter out these false positives and use that to better assess the accuracy of the system.

Finally, we will need to test the system on more than just dark exposures. The current system is designed to work with dark exposures but we will need to adjust the system to work with other types of exposures that might have more difficult to detect anomalies. There may not be a one size fits all solution to this problem but testing different types of exposures will help us determine the best methods for each type of exposure.

Chapter 5

ELECTRON DENSITY MAPPING IN CARINA NEBULA

5.1 Introduction

5.2 Data

Our data comes from two main sources, the Stratospheric Observatory for Infrared Astronomy (SOFIA) and the Deep Space Network (DSN). Additionally, for our calculations, we need a radio continuum brightness temperature, which we take from the Atacama Large Millimeter/submillimeter Array (ALMA) observations of the Carina Nebula (**Rebolledo_2021**).

5.2.1 SOFIA — Stratospheric Observatory for Infrared Astronomy

SOFIA was a joint project of NASA and the German Aerospace Center (DLR), operating from a Boeing 747 aircraft equipped with a 2.5-meter telescope designed for infrared observations. On SOFIA was the German REceiver for Astronomy at Terahertz frequencies (GREAT) which observed a wide range of spectral lines between 1.25 and 2.5 THz using a heterodyne receiver (**heyminck2012great**). Using the L1 channel of GREAT, we observed the [NII] 205 μm line in the Carina Nebula, providing a high-resolution spectral cube of the region. This spectral cube was generated using the SOFIA Data Processing System (DPS) which processes raw data from the instruments, calibrates it, and combines multiple exposures to create a final spectral

cube (**shuping2014overview**). The cube itself is a 36 by 36 pixel map spanning 11b̈y 11ẅith a 202 spectral channels covering a velocity range of -110 to 90 km/s with a velocity resolution of 1 km/s. The data has a beam size of 20.77änd is tuned to a frequency of 1.45 MHz for measuring the [NII] 205 μm line. Due to the calibration, the values are already in Main Beam Brightness Temperature (K_{mb}) units.

5.2.2 DSN — Deep Space Network

The DSN is a global network of antennas that supports interplanetary spacecraft missions and radio astronomy observations. For radio astronomy, the DSN Deep Space Station 43 (DSS-43) in Canberra, Australia, is equipped with a 70-meter dish that can observe radio recombination lines (RRLs) at frequencies within the K-band (17 to 27 GHz) (**virkler2020broadband**). Our RRL data for Carina Nebula was collected using the DSS-43 antenna as an average of the RRL observations from H70α (18.768 GHz) to H62α (26.939 GHz). This average was taken by scaling all the components to the central value of H67α (21.385 GHz) and gridding the data to match the lowest frequency, H70α. The resultant map is much larger than the original SOFIA map, spanning 150b̈y 76(̈343 x 335 pixels) with a beam size of 56änd a spectral resolution of 2.4 km/s across 165 channels (-300 to 98 km/s). Unlike the SOFIA data, the RRL data is in units of antenna temperature (K_{Ta*}), which is the raw output of the radio telescope before calibration. Using the efficiency of the antenna (0.5 for DSS-43), the data will need to be converted to the Main Beam Brightness Temperature (K_{mb}) units for comparison with the SOFIA data.

5.2.3 ALMA — Atacama Large Millimeter/submillimeter Array

Our radio continuum brightness temperature data comes from the ALMA observations of the Carina Nebula. ALMA is an array of radio telescopes located in the Atacama Desert in Chile, designed to observe the universe at millimeter and submillimeter wavelengths. We utilize data from **Rebolledo_2021**, which provides a continuum brightness temperature map of the Carina Nebula between 1-3 GHz with a beam of 24s̈panning 613b̈y 297(̈3574 x 3415 pixels). This data is in units of Jy/beam, which is a common unit for radio continuum observations but needs to be converted to brightness temperature (K) for our calculations.

5.3 Methodology

Our analysis follows the methodology outlined in **pineda2019electron**, which describes how to calculate electron density from the ratio of [NII] 205 μm line intensity to the intensity of Hydrogen Radio Recombination Lines (RRLs). This analysis was done using the `spectral-cube` package in Python, which allows for efficient manipulation of spectral data cubes (**robitaille2016spectral**). `spectral-cube` wraps many of the functionalities of the `astropy` package, which is a core package for astronomy in Python (**astropy:2013**; **astropy:2018**; **astropy:2022**). Our methodology for analysis consists of preprocessing the data cubes to ensure they are on the same grid with the same spatial and spectral resolutions, convolving the [NII] 205 μm data with the beam of the RRL data, and then performing calculations to derive the electron density from the ratio of the two lines. In addition to the electron density, we are also able to estimate the [NII] 122 μm line intensity, which whose ratio with the [NII] 205

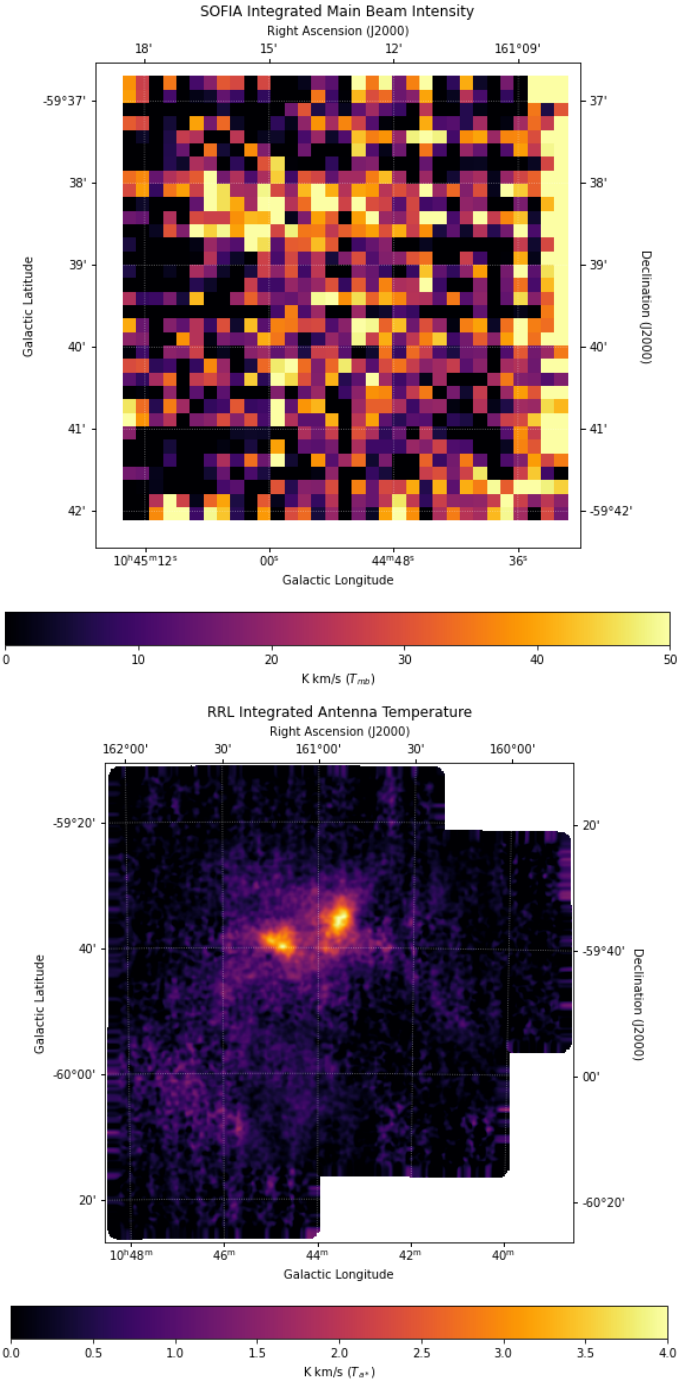


Figure 18. The raw data cubes for the [NII] 205 μm line (top) and the RRL data (bottom). Both lines are integrated intensities across the entire spectral range of the dataset. The [NII] 205 μm line is shown using Main Beam Temperatures, while the RRL data is shown in Antenna Temperature.

μm line can be used to estimate electron density as well (**goldsmith2015herschel**). Figure **??** shows the raw data cubes for both the [NII] 205 μm line and the RRL data.

5.3.1 Preprocessing the Data Cubes

In order to properly compare the two data cubes, we first need to pre-process everything to ensure our data points match both spatially and spectrally. First, we need our data to be in the same units, so we will convert the RRL data from antenna temperature (K_{Ta*}) to Main Beam Brightness Temperature (K_{mb}) using the efficiency of the antenna as shown in Equation **??**, where T_{Ta*} is the antenna temperature and η_{mb} is the main beam efficiency of the antenna.

$$T_{mb} = \frac{T_{Ta*}}{\eta_{mb}} \tag{5.1}$$

Next we need to ensure both datasets have a common beam size. As the SOFIA data has a smaller beam than the RRL data, we will convolve the [NII] 205 μm data with a Gaussian kernel that matches the beam size of the RRL data. This is handled by the `spectral-cube` package, which uses the `astropy.convolution` module to convolve the data with a Gaussian kernel that matches the beam size of the RRL data. We perform this step first to ensure that the [NII] 205 μm data is properly smoothed to match the RRL data before interpolating the data to a common grid, where we may lose some spatial resolution.

Following this, we spectrally interpolate the [NII] 205 μm data to match the larger spectral resolution of the RRL data. We also crop both data cubes to the same spectral range. This is accomplished by convolving the [NII] 205 μm data with a Gaussian kernel with a standard deviation with that matches the difference in spectral resolution between the two datasets as shown in Equation **??**, where σ is the standard

deviation of the Gaussian kernel, $\Delta\nu_{[NII]}$ is the spectral resolution of the [NII] 205 μm data, and $\Delta\nu_{RRL}$ is the spectral resolution of the RRL data.

$$\sigma_{FWHM} = \frac{\Delta\nu^2_{RRL} - \Delta\nu^2_{[NII]}}{\sqrt{8\ln(2)}} \tag{5.2}$$

After spectral interpolation, both spectral cubes are sent through a Savitzky-Golay filter to smooth the data and improve our signal-to-noise ratio (SNR). We use a polynomial order of 3 and a window length of 10 spectral channels, which is sufficient to smooth the data without losing too much spectral resolution. This process is outlined by **pineda2019electron** and is implemented using the `savgol_filter` function from `scipy` (**2020SciPy-NMeth**). Figure **??** shows the mean spectra of both the [NII] 205 μm line and the RRL data at each step of spectral interpolation and smoothing.

After the spectral interpolation and smoothing, we can crop both data cubes to the same spatial region and reproject them to a common grid. The RRL map covers a much larger area than the SOFIA map, so we will need to crop the RRL data. Using the world extrema of the SOFIA data, we can create a subcube of the RRL data whose bounds match the SOFIA data. The spatial resolution of the SOFIA data is smaller than the RRL data, so we use the `reproject` function from `spectral-cube` to interpolate the SOFIA data to the RRL data's grid. This matches the WCS of both data cubes, allowing us to compare the two datasets directly. After this step, we have two data cubes that are on the same grid, with the [NII] 205 μm data convolved to match the beam size of the RRL data and both datasets having the same spatial and spectral resolution as shown in Figure **??**.

In addition to the above preprocessing steps, we also need to perform some additional steps to prepare the ALMA continuum brightness temperature data. First, we need to convert the ALMA data from Jy/beam to K to get our brightness temperature.

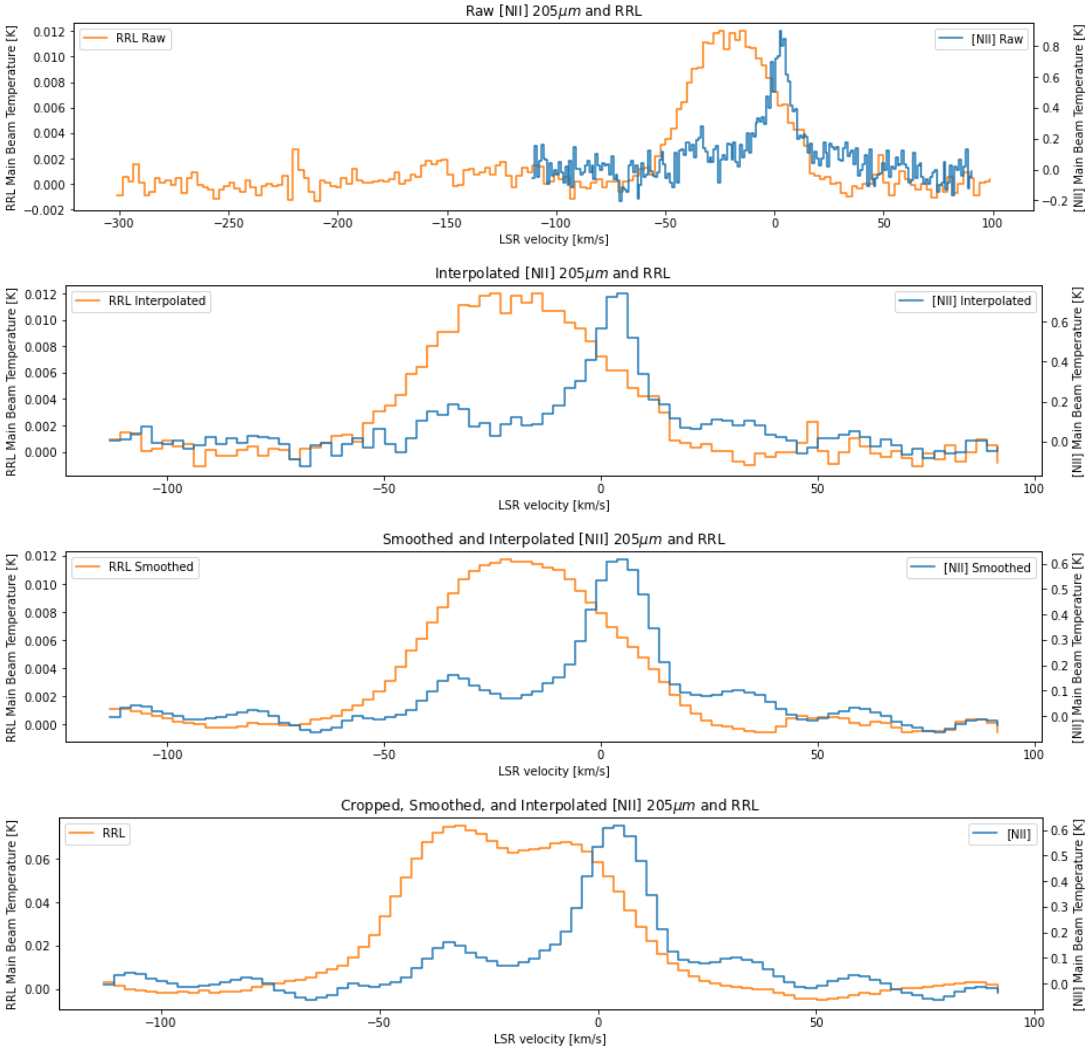


Figure 19. The mean spectra of the [NII] 205 μm line (blue) and the RRL line (orange) at each step of preprocessing. Note that the first three panels show the mean spectra of the entire RRL region which includes more area than the SOFIA data. The top panel shows the raw spectra prior to any spectral interpolation or smoothing. The second panel shows the spectra after spectral interpolation to the RRL data's spectral resolution and cropping to the same common range. The third panel shows the spectra after applying a Savitzky-Golay filter to smooth the data. The final panel shows the final spectra after spatial cropping and interpolation.

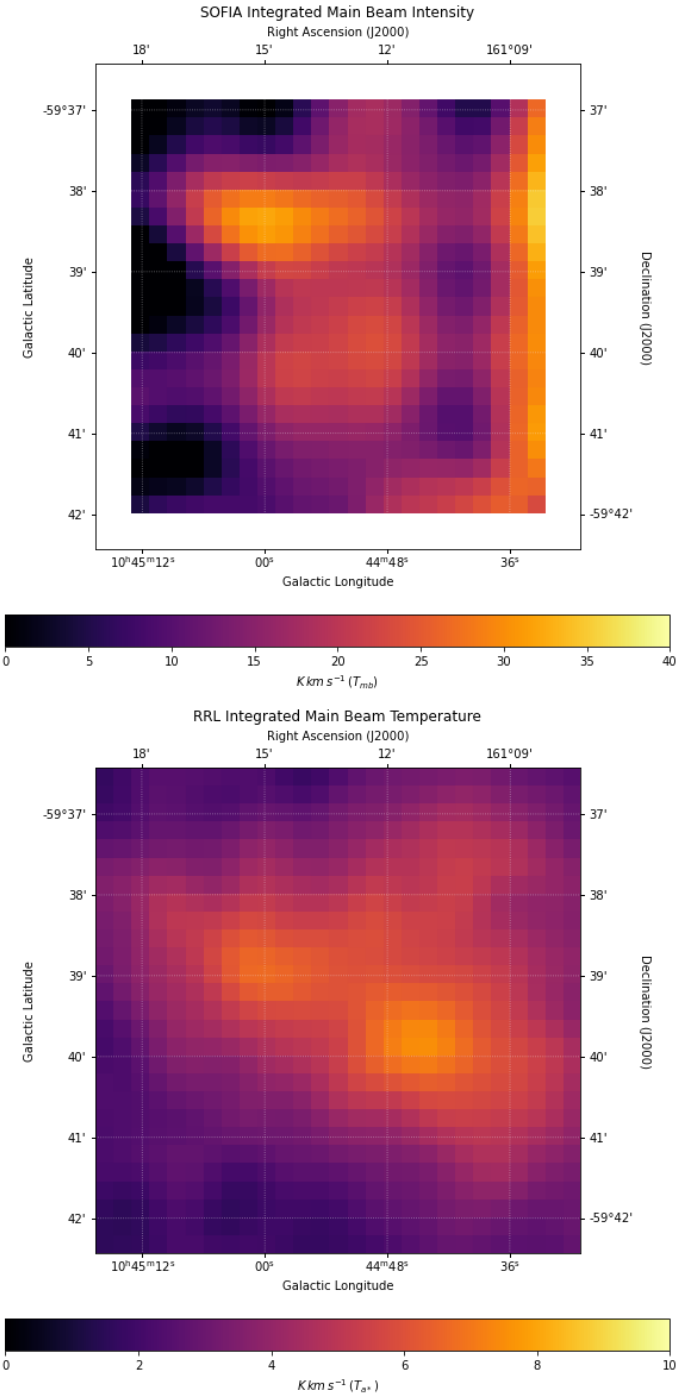


Figure 20. The processed data cubes for the [NII] 205 μm line (top) and the RRL data (bottom). Both lines are the integrated main beam intensities across the entire spectral range of the dataset.

This is done using Equation **??**, where T_b is the brightness temperature in K, S is the flux density in Jy/beam, λ is the wavelength in centimeters, and θ is the beam size in arcminutes (**rohlfs2013tools**).

$$T_C = \frac{S\ \lambda^2}{2.65\ \theta^2} \tag{5.3}$$

The also needs to be corrected to match the frequency of the RRL data. This is accomplished by using a power law to scale the brightness temperature to the frequency of the RRL data as shown in Equation **??**, where T_{Bf} is the final brightness temperature in K, T_{Bi} is the initial brightness temperature in K, ν_f is the final frequency in GHz, ν_i is the initial frequency in GHz, and α is the spectral index.

$$T_{Bf} = T_{Bi} \left(\frac{\nu_f}{\nu_i} \right)^{\alpha} \tag{5.4}$$

For Carina Nebula, we use a spectral index of -0.1, which is used for optically thin thermal bremsstrahlung emission (**salatino2012spectral**). As our RRL data is scaled to the H67α line, we will use the frequency of this line (21.385 GHz) as our final frequency, extrapolating from the ALMA data which is centered around 2.1 GHz. Finally, while ALMA data is not a spectral cube, `spectral-cube` can handle spatial interpolation of 2D data as well. We use the WCS of the final [NII] 205 μm data cube to reproject the ALMA data to the same grid.

5.3.2 Calculating Electron Density

The first step to calculating electron density is to see how the [NII] 205 μm line intensity and the RRL intensity are related. We begin with the following equation for the integrated intensity of the [NII] line (**pineda2019electron**):

$$\int T_{mb}^{[NII]} d\nu = \frac{A_{ul} h c^3 N_u}{8\pi k_b \nu_{ul}^2} \tag{5.5}$$

In this equation, A_{ul} is the Einstein A coefficient for spontaneous decay rate, h is Planck's constant, c is the speed of light, N_u is the column density of the upper level, k_b is Boltzmann's constant, and ν_{ul} is the frequency of the transition. We can simply this further by noting that the upper level column density, N_u, is related to the fractional population of the upper level, $f(^uP_l)$, and the total column density of ionized nitrogen, $N(N^+)$, as follows:

$$N_u = f(^3P_u)N(N^+) \tag{5.6}$$

There are two fine structure transitions for [NII] at 122 μm ($^3P_2$ - $^3P_1$) and 205 μm ($^3P_1$ - $^3P_0$). Combining Equations **??** with **??** results in the following equation for the integrated intensity of the [NII] 205 μm line:

$$\int T_{mb}^{[NII]} d\nu = \frac{A_{ul} h c^3 f(^3P_u) N(N^+)}{8\pi k_b \nu_{ul}^2} \tag{5.7}$$

$$= 5.145 \times 10^{14}\ A_{ul} \nu_{ul}^{-2} f(^3P_u) N(N^+)\ \text{[K km/s]} \tag{5.8}$$

In the simplified version, A_{ul} is in s^{-1}, $f(^3P_u)$ is dimensionless, $N(N^+)$ is in cm^{-2}, and ν_{ul} is in Hz. We utilize PyNeb, a Python package for calculating emission line intensities, to obtain our Einstein A coefficient, A_{ul} and the fractional population of the upper level, $f(^3P_u)$ (**luridiana2015pyneb**; **froese2004breit**; **7288EL**; **tayal2011electron**).

Now we can look at the integrated intensity of the RRL line using the following equation (**pineda2019electron**):

$$\int T_{mb}^{RRL} d\nu = 1.87 \times 10^{-7}\ \frac{n_e N(H^+)}{\nu_{RRL} T_e^{3/2}}\ \text{[K km/s]} \tag{5.9}$$

For the RRL, n_e is the electron density in cm^{-3}, $N(H^+)$ is the column density of ionized hydrogen in cm^{-2}, ν_{RRL} is the frequency of the RRL in Hz, and T_e is the electron temperature in K. Because the hydrogen recombination lines can be affected

by non-local thermal equilibrium (NLTE) effects, we also need to account for the deviation from LTE in the RRL emission (**gordon2002radio**).

$$G_{NLTE}(n_e, T_C) = \frac{T^{RRL}}{T^{RRL}_{NLTE}} = b_n \left[1 - \frac{1}{2}\tau_c \beta_n\right] \tag{5.10}$$

In this equation, b_n and β_n are the departure coefficient and amplification factor for a specific principal quantum number n, and τ_c is the continuum opacity. τ_c can be calculated using the continuum brightness temperature, T_C, and the electron temperature , T_e, as follows **goldsmith2024electron**:

$$\tau_c = \frac{T_C}{T_e} \tag{5.11}$$

The GNLTE coefficients are obtained using a Fortran program in the appendix of **gordon2002radio**, which was later wrapped in Python by **2017ascl.soft07001W**. These values depend on electron density, n_e, and electron temperature, T_e, for a given principal quantum number n. A precomputed table of these values is used to interpolate the values for our specific electron density at $T_e = 8000$ K. Adding on this factor, we can rewrite Equation **??** line as follows:

$$\int T^{RRL}_{mb} d\nu = 1.87 \times 10^{-7} \; \frac{N(H^+)}{\nu_{RRL} T_e^{3/2}} \; n_e \; G_{NLTE}(n_e, T_c) \text{ [K km/s]} \tag{5.12}$$

Finally, we can compute the ratio of Equation **??** and **??** to get the following equation:

$$R^{[NII]}_{RRL} = \frac{\int T^{[NII]}_{mb} d\nu}{\int T^{RRL}_{mb} d\nu} = 2.75 \times 10^{21} \; \frac{A_{ul} \nu_{RRL} T_e^{3/2}}{\nu_{ul}^2} \; \frac{N(N^+)}{N(H^+)} \; \frac{f(^3P_u)}{n_e G_{NLTE}(n_e, T_c)} \tag{5.13}$$

Most of the terms in this equation are constants that do not depend on the electron density, n_e, except for the last term, RHS $= f(^3P_u)/n_e G_{NLTE}(n_e, T_c)$. The remaining unknown terms T_e the electron temperature in K, and $N(N^+)/N(H^+)$, the

abundance ratio of ionized nitrogen with respect to ionized hydrogen, can be estimated using the distance to the Galactic center, R_{gal}, in kpc (**pineda2019electron**; **balser2015azimuthal**; **esteban2018revisiting**):

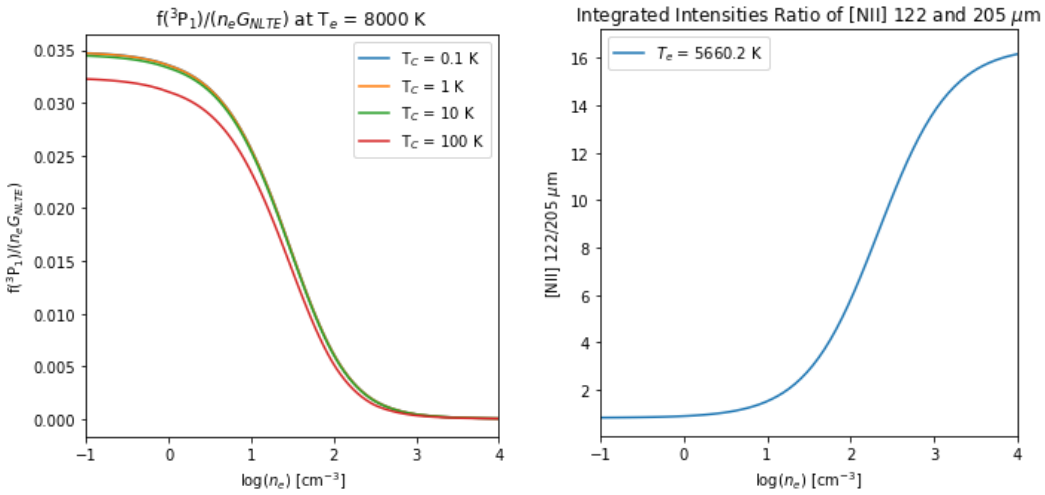
$$T_e = (4446 \pm 301) + (467 \pm 34) R_{gal} \tag{5.14}$$

$$12 + \log(N/H) = (8.21 \pm 0.09) - (0.059 \pm 0.009) R_{gal} \tag{5.15}$$

Rearranging Equation **??** to isolate the term that depends on electron density allows us to look at the relationship between those terms and electron density in Figure **??**. The curves are generated using fractional population data from **luridiana2015pyneb** and GNLTE coefficients from **gordon2002radio** to write a function that calculates RHS for a given electron density, n_e, and continuum brightness temperature, T_c. Because these curves are related to the ratio of the [NII] 205 μm line intensity to the RRL intensity, we can use them to calculate the electron density for a given ratio of intensities. This is accomplished by calculating the ratio of the integrated intensities and dividing by the constant terms. Then we can use a root solving algorithm, such as `scipy.optimize.fsolve`, find the value of electron density that whose RHS matches the calculated ratio **2020SciPy-NMeth**. This process is repeated for each pixel in the [NII] 205 μm data cube, resulting in a map of electron density across the Carina Nebula.

In addition to electron density, we can also estimate the [NII] 122 μm line intensity using the relationship between the two [NII] lines (**goldsmith2015herschel**). The [NII] 205 μm and 122 μm lines correspond with the $^3P_2$ - $^3P_1$ and $^3P_1$ - $^3P_0$ transitions, respectively. We can take the ratio of these two lines to see their dependency on electron density by using Equation **??**:

$$\frac{\int T_{mb}^{[NII]\ 122} d\nu}{\int T_{mb}^{[NII]\ 205} d\nu} = \frac{A_{21} f(^3P_2) \nu_{205}^2}{A_{10} f(^3P_1) \nu_{122}^2} \tag{5.16}$$



(a) The fractional population of the 205 μm level of ionized nitrogen, $f(^3P_1)$, and the GNLTE coefficients, $G_{NLTE}(n_e, T_c)$, as a function of electron density for various continuum brightness temperatures.

(b) Ratio of the integrated intensities of the two lines of [NII] as a function of electron density at an electron temperature of 5662 K.

Figure 21. Ratios of Electron Density with Fractional Population and G_{NLTE} Coefficients for Calculating Electron Density and [NII] 122 μm

This ratio relies on electron temperature and density in the form of the fractional populations $f(^3P_1)$ and $f(^3P_2)$. Figure **??** shows the ratio of the two lines at the calculated electron temperature for Carina Nebula, which is approximately 5660 $\pm$ 313.7 K.

5.3.3 Uncertainties

The primary sources of uncertainty in our electron density calculations come from the actual measurements of the [NII] 205 μm line and the RRL data, as well as the usage of R_{gal} to estimate the electron temperature and nitrogen abundance. As we are taking the integrated intensities of the [NII] 205 μm line and the RRL data, we can calculate the uncertainties in these measurements using the RMS noise of the data

cubes. We calculate this by taking the standard deviation of the data cube in a region where we expect no signal, such as the edges of the spectral range for each pixel, far away from the center of the spectral line where Carina is located. This produces a per channel noise map for each pixel in the data cube which we can use to calculate the uncertainties in our integrated intensities as follows:

$$\sigma_{int} = \text{RMS} \times \sqrt{N_{chan}} \times \Delta\nu \tag{5.17}$$

This uncertainty, along with the uncertainties in T_e and $N(N^+)/N(H^+)$, can be propagated through the calculations to estimate the uncertainty of the value used to calculate the electron density in the root solving algorithm.

To calculate electron density, we use a Monte Carlo approach, where we randomly sample from a normal distribution centered around the measured value with a standard deviation equal to the uncertainty. We then solve for the electron density using the sampled values and repeat this process many times (e.g., 1000 iterations) to generate a distribution of electron density values. This distribution can then be used to calculate the mean and standard deviation of the electron density for each pixel in the map, providing us with an uncertainty estimate for our electron density calculations. This method was chosen over direct propagation of uncertainties because it allows us to account for the non-linear relationship between the electron density and the ratio of intensities.

5.4 Results

In observing the spectra of the [NII] 205 μm line and the RRL data, we can see two distinct regions of interest in the Carina Nebula. The first region is the Keyhole Nebula which is a prominent feature in the Carina Nebula and is known for its high

Location	v_{min} km/s	v_{max} km/s
Keyhole Nebula	-40	-20
Carina Extended	-20	20

Table 2. Distinct Spectral Regions within the Carina Nebula Data from SOFIA

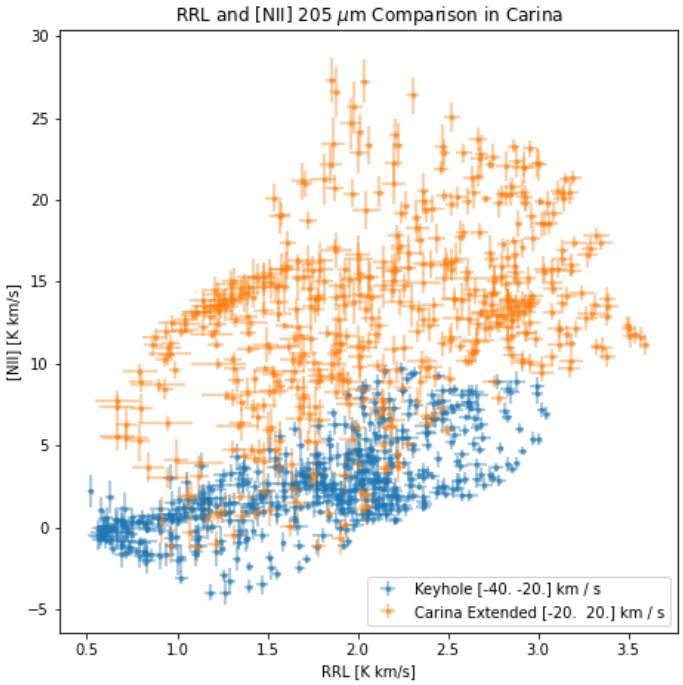


Figure 22. The integrated intensities of the [NII] 205 μm line and the RRL data for each pixel in the Carina Nebula. The Keyhole Nebula is shown in blue, while the Carina Extended region is shown in orange.

electron density **brooks2000unlocking**. Behind the Keyhole Nebula is the Carina Extended region, which is a more diffuse region with lower electron density. Table **??** shows the spectral slabs used to calculate the electron density for both regions.

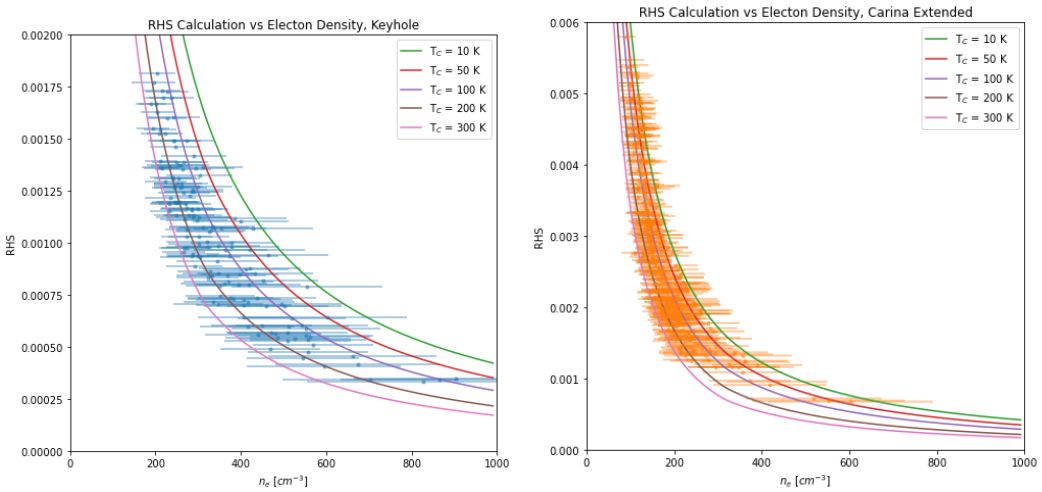
Another way to visualize this separation is to look at the separation between the two regions when plotting the integrated intensities of the [NII] 205 μm line and the RRL data. Figure **??** shows a scatter plot of the two lines, showing an overall trend of less [NII] in the Keyhole Nebula compared to the Carina Extended region.

Now that we have the integrated intensities of the [NII] 205 μm line and the RRL data, we can calculate the electron density for each pixel in the map. Using the Monte Carlo approach, we perform 500 iterations to generate a distribution of electron density values for each pixel. Many of the pixels in our maps have a low signal-to-noise ratio (SNR), which results in a large uncertainty in the electron density. In some cases, the root finding algorithm fails to converge, resulting in an undefined value for the electron density. We omit these values as well as those with an SNR less than 3. Figure **??** shows the relationship between the density dependent variables of Equation **??** and the electron density for the Keyhole Nebula and Carina Extended region as well as a measure of the average uncertainty of the calculation during the Monte Carlo sampling.

Figure **??** and **??** show maps for the [NII] 205 μm and RRL inputs as well as the calculated electron density for the Carina Nebula and the Keyhole Nebula, respectively.

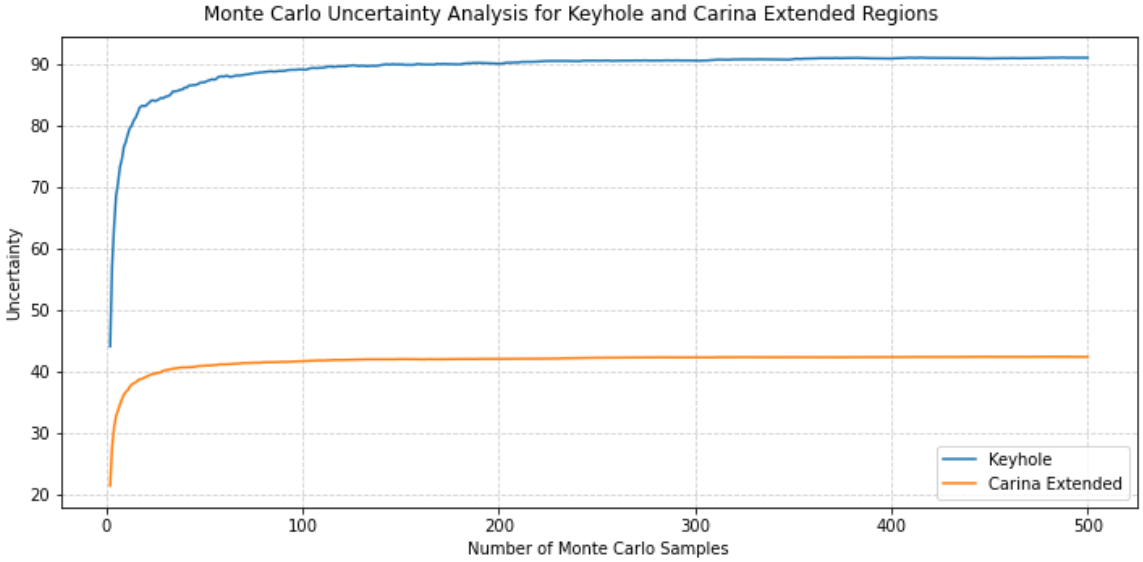
We use a non beam corrected version of the [NII] 205 μm line to calculate the integrated intensity of the [NII] 122 μm line as we are interested in what our measurements would be on SOFIA if we had observed the [NII] 122 μm line as well.

5.5 Conclusions



(a) Density Dependent Variables for Keyhole Nebula vs Electron Density

(b) Density Dependent Variables for Carina Extended vs Electron Density



(c) Measure of Uncertainty in Electron Density Calculations for Both Regions

Figure 23. The density dependent variables of Equation **??** for the Keyhole Nebula (left) and Carina Extended region (right). The blue line shows the calculated value of the ratio of integrated intensities, while the orange line shows the average uncertainty in the electron density calculations. The red line shows the electron density calculated from the ratio of integrated intensities.

Chapter 6

ASTHROS PAYLOAD READOUT SYSTEM DESIGN

ASTHROS, the Astrophysics Stratospheric Telescope for High Spectral Resolution Observations at Submillimeter-wavelengths, is a balloon-borne observatory designed to study the universe in the submillimeter wavelength range. The readout system is responsible for controlling the detectors, reading out the data, and storing the data on a solid state drive. The readout system is designed to be modular and scalable, allowing for easy integration of new detectors and readout systems. Each package is designed to be self-contained, focusing on a single device in the readout system, so that changes to the hardware can be made without affecting the rest of the system.

The primary function of this chapter is to describe the readout system in excruciating detail so future users can understand how every part of ASTHROS works as well as the design decisions made during the development of the system. While this is not a traditional chapter in the dissertation sense, it is an important part of the work done during my research and documenting the system is vital to the success of the project. This chapter starts with a section on the overall network architecture of TODO:

6.1 Network Architecture

TODO:

6.1.1 Command

TODO:

6.1.2 Storage

TODO:

6.1.3 Analysis

TODO:

6.1.4 Raspberry Pi Compute Module 4

From a fresh install of Raspberry Pi OS, we need to install the necessary packages to run the readout system. The first step is to enable Secure Shell (SSH) so that we can remotely access the Raspberry Pi. This can be done in the Raspberry Pi Configuration tool by enabling SSH in the Interfaces tab. While we're in the interfaces tool, we can also enable SPI and Remote GPIO. These are necessary to interface with the PMCC.

Next, we need to configure the networking interfaces. `eth0` is the wired interface to the rest of the readout network. We need a static IP for the CM4 to ensure that other devices on the network can easily find it. In the network settings at the top right of the screen, we can select "Edit Connections" under Advanced Options. From there, you will see "Wired connection 1" which is the default name for the wired interface.

Select the interface and navigate to the IPv4 Settings tab. Chance the method from "Automatic (DHCP)" to "Manual" and add the IP address, Netmask, and Gateway. For ASTHROS, we have the IP addresses of all CM4s set to `192.168.1.13X` where `X` is uniquely assigned to each CM4. The Netmask is set to `23` so we can access all devices on the `192.168.1.X` readout network as well as the `192.168.0.X` gondola network. Finally, the Gateway is set to `192.168.1.1` when attached to the gondola and `192.168.1.101` when attached to the readout network. This is because, when connected to a test bench, we utilize the NAS as our router to access other devices on the network. When connected in flight configuration, we disable the NAS's router functionality and use the gondola's router instead.

Next, we need to configure the SPI interface to work with the PMCCs. This process is different for Compute Module 5s (CM5s) as they have a different SPI driver and interfaces for the GPIO pins. While we are in the process of upgrading to CM5s, the current version of the readout, and thus this documentation, is designed for CM4s. The first step is to increase the SPI buffer size. This is done by appending the following to the end of the `/boot/cmdline.txt` file:

```
spidev.bufsiz=65536
```

This sets the SPI buffer size to 64KB which is the maximum size to support the burst readout of the PMCCs. For loading on boot, we need to add the SPI device to the `/etc/modules` file. This is done by adding the following line to the file:

```
spi_bcm2835
```

Each of the CM4s will control up to four PMCCs, so we need to enable unique SPI busses for each PMCC. This is done on the `/boot/config.txt` file by adding the following lines:

```
dtoverlay=spi0-1cs
dtoverlay=spi3-1cs
dtoverlay=spi4-1cs
dtoverlay=spi5-1cs
```

This enables the SPI0, SPI3, SPI4, and SPI5 busses on the CM4. While we could, in theory, only use two SPI busses and use the chip select lines to control two PMCCs on each bus, we decided to use a single chip select line for each PMCC to simplify the wiring harness and ensure our bandwidth is not saturated.

At this point, it is a good idea to reboot the CM4 to ensure that all changes have taken effect. To verify the SPI busses are enabled, we check the `/dev` directory for the SPI devices which should be `/dev/spidevX.0` where `X` is the SPI bus number (0, 3, 4, or 5). After rebooting and verifying the SPI has been set up, we recommend connecting the CM4 to the internet through a wireless hotspot in order to install the necessary libraries. The first library we need to install is the pigpio. This is a library that allows us to control the GPIO pins on the CM4 without needing root access. To install the pigpio library, we need to run the following commands:

```
sudo apt-get update
sudo apt-get install pigpio
```

After installing the pigpio library, we need to enable the pigpio daemon to run on boot. This is done by running the following command:

```
sudo systemctl enable pigpiod
```

Finally, we need to install the actual Python packages that we will use to control the PMCC. First we need to clone the `PyMCC` repository from GitHub. This is done by running the following command:

```
git clone https://github.com/asthros/pymcc.git
```

Because this is a private repository, you will need to enter your GitHub username and personal access token. You will need to generate a personal access token on GitHub and use that as your password when prompted. After cloning the repository, we need to install the Python packages. This is done by running the following command:

```
pip3 install -r pymcc/requirements.txt
```

This will install all the necessary packages to run the `PyMCC` package.

Finally, we need to edit the hosts file on the CM4 to ensure that we can access the other devices on the readout network by name. This is done by updating the `/etc/hosts` file with the IP addresses and hostnames in Appendix **??**.

6.2 RabbitMQ

The ASTHROS readout system relies on RabbitMQ to communicate between different devices and systems on the readout network. Past ballooning missions, such as BLAST-TNG, utilize sockets to communicate between systems (**gordon2019highly**). The resulting network of sockets across multiple devices becomes difficult to manage and debug as many of the definitions for the sockets are hard coded into the software. Devices in such a system often use User Datagram Protocol (UDP) to broadcast messages across the network. This approach requires each device to listen for messages on a specific port and parse the messages to determine if they are intended for that device. For systems that require multiple devices to communicate with each other, this can become cumbersome as each device must be aware of the other devices on the network and the ports they are listening on. Additionally, UDP is a connectionless

protocol, meaning that there is no guarantee that a message will be received by the intended recipient. To work around this, some ballooning readout systems would be built as a single monolithic program that orchestrates all the devices on the network (**gordon2019highly**). This approach is not scalable and makes it difficult to debug and maintain the system.

For ASTHROS, we wanted to build a system that was modular, consisting of microservices that can easily be integrated into the rest of the system using a shared protocol. To accomplish this, we needed to decouple the devices in a way that data production and consumption are separated. Services that produce housekeeping data are not concerned with who needs the data, and services that consume housekeeping data are not concerned with where the data comes from, simply how the data is formatted. This is where a message broker like RabbitMQ comes in. RabbitMQ is an open-source Advances Messaging Queue Protocol (AMQP) implementation that serves as a broker for incoming and outgoing messages by routing messages to bound destination queues located on an exchange (**dunne2018comparison**). We chose RabbitMQ because of the centralized message broker architecture that allows for easy integration of new devices and services. All of our devices are networked together, making a centralized message broker the ideal solution for our readout system.

The core of RabbitMQ is explained on the documentation as well as **toshev2015learning**. We like to use a post office analogy to explain how RabbitMQ works. Within RabbitMQ are three main components: producers, queues, and consumers. Producers are the senders of a message, which is analogous to a person sending a letter. In our case a message is any binary of blob of data, but we typically use JSON formatted strings. When a producer sends a message, it addresses the message with a routing key. A queue is similar to a mailbox where letters are stored

until they are picked up by the recipient. The recipient of the message is the consumer, who retrieves the message from the queue. Just like a normal mailbox, a queue can have multiple producers sending data to it and multiple consumers who can try and read the data.

In all of this, the message broker is the post office and postal workers. When a producer makes a message, it hands it to the post office to be delivered to all who need it. This is done using the routing key that specifies the queue the message should be delivered to. The post office then takes the message and delivers it to the correct mailbox. When a consumer is ready to read a message, it checks its mailbox for any messages and reads them. If the consumer is not ready to read the message, the message stays in the mailbox until the consumer is ready.

Behind the scenes, there is a fourth concept important to RabbitMQ: exchanges. Exchanges are the post office's sorting facility. Typically, producers are not typically sending messages directly to queues. Instead, they publish messages to an exchange with a specific routing key. When a queue is made, it can be bound to an exchange with routing keys. If the routing key of the messages matches one of the routing keys bound to the queue, the message is delivered to that queue. Routing keys can be wildcards, allowing queues to subscribe to specific types of messages. For ASTHROS, we follow the format of `<type>.<device>.<subtype>` for our routing keys. For example, a housekeeping value from temperature sensors on Power Supply Unit 4 would be `housekeeping.temp.4`. This allows a queue to be bound broadly to a specific type of message, such as `housekeeping.*.*` to receive all housekeeping messages or `*.temp.*` to receive all temperature messages. For systems that have multiple devices, this allows us to easily ingest all data from those devices without having to know the specific routing keys for each device.

6.2.1 RMQTools

While RabbitMQ serves as a powerful backbone for our networking infrastructure, we needed to develop a way to wrap our devices in ways that will communicate with the RabbitMQ server. To accomplish this, we developed a Python package called `RMQTools` that provides a simple interface for creating both publisher to subscriber systems as well as Remote Procedure Call (RPC) frameworks. At its core is the `RmqConnection` class that wraps the `pika` library to add additional functionality and simplify the process of creating a RabbitMQ connection (**pika**). Pika is a single threaded RabbitMQ client library in python that provides an implementation of the AMQP protocol. Many of our modules will require threading as we will often have a main thread running processes on the hardware while a second thread is running the RabbitMQ connection making it difficult to use the `pika` library directly. `RmqConnection` handles this threading by allowing us to create a connection to the RabbitMQ server and run the connection in a separate thread from the main thread.

As a simple example of publisher to subscriber systems, we instantiate the `RmqConnection` class as an `rmq` object with the RabbitMQ server address and port as well as the username and password for the server. We can then use `@rmq.set_status_exchange(name)` to specify which status exchange we will be using. These are done for on both the publisher and subscriber side, ensuring that both are using the same server and exchange. Now that we have configured the setup, we can implement a publisher. This is done using the `@rmq.publish_status(interval, routing_key)` decorator around a function that returns a dictionary. This creates a new threading object that will run the function at the specified interval and publish the status to the RabbitMQ server. This is accomplished by creating a new thread

that creates a connection to the RabbitMQ server, sets the exchange, and creates a `Publisher` with the specified routing key. Then the function is run at the specified interval and the resulting dictionary is published to the RabbitMQ server.

The subscriber is also implemented using a decorator, `@rmq.subscribe_status( queue_name, routing_key(s))`. The wrapped function must take four arguments; the channel, the method, the properties, and the body. For the most part, the channel, method, and properties are not used by the callback function but are useful when debugging received messages or differentiating between multiple routing keys. Body is the actual method that is received from the RabbitMQ server in the form of a JSON string. The decorator wrapped callback function can then process the message in whatever way it needs to.

For RPC executions, we've implemented a similar system using the `RmqConnection` class. On the server side of the RPQ call, we create a new `RmqConnection` object and set the command exchange using `rmq.set_command_exchange(name)`. After this is done, we can use the decorator `@rmq.handle_commnd(queue_name)` to wrap a function that we want to be called when a command is received. This command receives the positional and keyword arguments from the RPC execution and returns a `ResponseObject` with the result of the function. The `ResponseObject` is a simple class that contains args and kwargs attributes that are used to send the response back to the caller. The client of the RPC execution uses the decorators `@rmq.send_command(command_id, queue_name)` and `@rmq.handle_response(command_id)`. This works well for threaded applications as the decorated function will run in a separate thread from the main thread once all threads are started. When the thread is started and the command is sent and executed, the decorated `handle_response()` corresponding to the `command_id` will

be called to handle the response. For most of our use cases, we would like to be able to control when these commands are sent so we often times use the back end `_send_command()` helper function to manually send the command to the RabbitMQ server. This works similar to the decorator's use of `@rmq.handle_response()` decorated functions to handle responses but instead of starting a thread to send the command, we simply call the `rmq._send_command()` function with a lambda function that creates the `ResponseObject` for the specific command we want to send. This is useful for sending commands across the server from a centralized application that interfaces with the ground control system. In addition to `handle_response`, we also can implement `handle_error()` and `handle_timeout()` to handle errors and timeouts from the RPC execution. These are useful for creating workflows that require an error or timeout response to be handled differently by the system, such as sending a warning to the ground control system that the command failed or timed out.

6.2.2 Hardware Locks

While not specific to RabbitMQ, as a result of the threading model we use, we need to implement a way to lockout commands from being sent to hardware while the hardware is being used by another thread. This can occur when a publisher thread is collecting data to send a status update at the same time a command is being sent through an RPC execution. This is to prevent conflicts on the serial or SPI bus when two threads are trying to send bytes to the same device at the same time. In addition to protecting the device, we also need to ensure that high prioity commands, like user executions, are not blocked by low priority commands, such as sending a status update. To accomplish this, we created a simple priority lock system that allows us

to create a lock object that we can then use to decorate functions that may be called at the same time.

As we could not find a suitable implementation of a priority lock in Python, we created our own. **prioritylock** of StackOverflow proposed the theory behind this lock and we implemented and modified it to fit our needs. To start, we create a `PriorityLock` object that takes a default timeout length and a number of priority levels. In initialization, we create a boolean, `data_free` to determine if the data is free, a lock object using the `threading` library, and an array of `Queue` objects whose length is equal to the number of priority levels. This lock object is not used to protect the hardware but instead used to protect modification of the Queues and the `data_free` boolean. When a function is decorated with `PriorityLock.with_lock()`, the function will not execute until the `PriorityLock` is acquired. The decorator takes in the priority level of the function, a timeout length, and optional arguments for return values in case of a timeout or reject. When a decorated function is called, the decorator calls `request_access(priority)` with the priority level of the function. `request_access()` first checks if the requesting priority level is less than the minimum priority level of the `PriorityLock`. `set_min_priority()` can adjust the minimum priority level of the `PriorityLock` to stop lower priority functions from executing. This is useful during critical operations when we want to stop functions like status calls from executing. If the rejecting priority level is less than the current minimum priority level, we return `False`, which will result in the decorated function returning the reject value. We now acquire the queue lock and check `data_free` to see if there is another thread using the resource. If the resource is free, we set `data_free` to `False` and return `True` to the decorator. If the resource is not free, we create a new `Event` object and put it to the queue of the requesting priority level. We then return the

`Event` object to the decorator. Regardless of what we return, we release the queue lock at the end of the function.

Once the result of `request_access()` is returned to the decorator, perform any necessary waiting before executing the function. If the result was an `Event` object, we wait for the event to be set, using either the default timeout or the timeout specified in the decorator. If the timeout is reached, we run `remove_event()` to remove the event from the queue and return the timeout value as the result of the decorated function. `remove_event()` needs to be a bit more complex than simply removing the event from the queue in order to ensure that the queue doesn't change while we are removing the event. With the queue lock acquired, we create a new queue object and iterate through the queue of events in the requesting priority level. We put all events that are not the one we are removing into the new queue, skipping the event we are removing. Once we are done populating the new queue, we replace the old queue in the array with the new one and release the queue lock.

After we're done waiting for the event to be set, we can finally execute the decorated function to obtain our result. As a final step, we run `release_lock()` to release the `PriorityLock` before returning the result. This is vital as it accounts for any additional calls that may have been made while we were waiting for the function to execute. `release_lock()` obtains the queue lock and iterates through the array of queues in reverse order to check the high priority queues first. If there are any events in the queue, we get the most recent event, set it, and return. If there are no events in the queue, we set `data_free` to `True` and return. This allows us to have multiple threads waiting for the same resource and ensures that the highest priority thread is executed first. Because we run `release_lock()` at the end of each decorated function, we can be sure all function calls are executed.

6.3 PMCC

For ASTHROS, we utilize an array of 4GHz spectrometers called the PMCC ASIC P19800B ASIC RF Spectrometer, henceforth referred to as the PMCC. These PMCCs are interfaced with via SPI for control, diagnostics, and readout (**PMCCP19800B**). To communicate with the PMCCs, we utilize Raspberry Pi Compute Module 4s (CM4s) with custom harnesses. The CM4 was chosen because it can be configured to operate at the 1.8V logic level necessary for PMCC by moving a diode on the CM4 IO board (**cm4io**). Additionally, the CM4 has 4 SPI buses, allowing us to control up to 4 PMCCs per device (**cm4**). The PMCCs are also connected to a GPIO pin on the CM4 to allow us to send a reset signal to the PMCCs. The custom harness used to connect the PMCCs to the CM4 mounts onto the CM4 IO board's GPIO pins and converts the 40 ribbon cable to four sets of connections for the PMCC's SPI and reset pins. Additionally, the CM4 has an SSD mounted to the side of it's IO board enclosure for raw spectra storage and easier local debugging when the device is not connected to the rest of the readout network. Finally, two CM4s and eight PMCCs are mounted in a custom enclosure that is designed to be mounted on the back of the ASTHROS primary mirror.

`PyMCC` is the Python package developed to interface with the PMCCs and communicate with the rest of the readout system. Originally, the PMCCs were controlled by a C program that was designed to provide a simple CLI for manually controlling the PMCCs. As we needed to control multiple PMCCs and have them communicate with the rest of the readout system, we decided to rewrite the control software and drivers in Python. The core of `PyMCC` is a Python driver for the PMCCs that provides an interface for controlling the PMCCs and reading out the data. Built on top of the

driver are Python programs that allow for manual control of the PMCCs, as well as a server that allows for control of the PMCCs over the RabbitMQ network.

6.3.1 `spi_utils`

At the lowest level of the `PyMCC` driver is the `spi_utils` module. This module provides an interface for communicating with the PMCCs over SPI using the `spidev` Python library. The `spidev` library provides an interface to the Linux kernel's SPI device driver (**spidev**). Additionally, the PMCC has 16-bit registers that require us to send and receive 16-bit words instead of the typical 8-bit bytes that `spidev` expect. This was the primary reason for the development of the `spi_utils` module as it handles the conversion between 16-bit words and 8-bit bytes and provides an easier interface for configuring the PMCCs registers without having to worry about the low-level details of the SPI communication.

The `spi_utils` module provides a `PMCC_SPI` class that is used to communicate with the PMCCs. The `PMCC_SPI` class is initialized with the bus, device, SPI mode, bits per word, and clock speed for the PMCC with which we are communicating. The bus and device are specific to the PMCC we are communicating with and are based on the wiring harness used to connect the PMCC to the CM4. The SPI mode, bits per word, and clock speed are all set to the values specified in the PMCC manual.

To simplify addresses, the `PMCC_SPI` object has a `make_addr()` method that takes the address of the register we want to write to and the read/write bit. Valid addresses for the PMCC are 0-511, and the read/write bit is 0 for a write and 1 for a read. When sending a command to the PMCC, the first word of the command is the address of the

register we want to write to shifted left by 1 bit to make room for the read/write bit.

$$\text{tx}[16] = \text{addr}[9] << 1 + \text{rw}[1] \tag{6.1}$$

Because `spidev` uses 8-bit communication, we need to split the 16-bit word into two 8-bit bytes.

$$\text{byte}[8][2] = [\text{word}[16] >> 8,\ \text{word}[16]\ \&\ 0\text{xFF}] \tag{6.2}$$

The helper method returns these two bytes as an array that can be used in other methods to convert an address and command into a format that can be sent over SPI.

For reading and writing to the PMCC, the `PMCC_SPI` object has an `xfer()` method that takes the address of the register, a read or write flag, and optional data to write and length of data to read. By default, the length of data to read is 1, and the data to write is None. The `xfer()` method first obtains the TX bytes from the `make_addr()` method. For both read and write commands, we utilize the `spidev` library's `xfer3()` function as it allows us to send and receive data of arbitrary length in a single SPI transaction (**spidev**). `spidev`'s `xfer2()` and `xfer()` will fail at list values longer than the maximum SPI buffer size. On the other hand, `xfer3()` will automatically split the data into multiple SPI transactions if the data is longer than the maximum SPI buffer size. This is vital for burst reads on the PMCC as our data can be much longer than the maximum SPI buffer size. For writes, the `xfer()` method sends the TX bytes and the data to write to the PMCC. The data is split into two 8-bit bytes using Equation **??**. During this transaction, the PMCC does not send any data back, so the `xfer3()` function returns an array of zeros. If our transaction is unsuccessful, instead of returning zeroes, we will receive an empty array that we can check for. For single register reads, the `xfer()` method sends the TX bytes followed by a dummy word to the PMCC. While we are sending the dummy word over the MOSI line, the PMCC is sending the data we requested over the MISO line that is returned by the `xfer3()`

function along with the original TX bytes. After checking that we have received data from the PMCC, we return the data as an array of 16-bit words. This is done by utilizing NumPy to cast the output as a `np.uint8` array and then returning a view of that array with big-endian 16-bit unsigned integer data type (**numpy**). For reads that are longer than a single register, we send the TX bytes followed by a dummy word for each word we want to read and follow the same process as a single register read.

For simple reads, `PMCC_SPI` has a `read()` method that takes the address of the register we want to read from and optionally the number of words we want to read. By default, this method reads a single word from the PMCC. The `read()` method calls the `xfer()` method with the read flag set to 1 and the number of words to read. If we are only reading a single word, we return the first word and only word in the array of words returned by the `xfer()` method. Otherwise, for burst reads, we return the entire array of words.

Often times, we want to read a specific value over and over until that value is True. Many status bits on the PMCC operate in this way to indicate when a specific operation has completed. To accomplish this, the `PMCC_SPI` object has a `poll()` method that takes the address of the register we want to read from, the bit within the register we want to check, the amount of time to wait between reads, and the maximum number of reads. The `poll()` method then issues a `read()` of that register and checks if the bit is set by shifting the read value to the right by the bit number and checking if the least significant bit is set. If the value is not set, we wait for the specified amount of time and read the register again. This process is repeated until the value is set or the maximum number of reads is reached. We then return if the value was set or not instead of raising an exception if the value is not set.

The implementation of raising exceptions is left to the user of the `poll()` method depending on the use case.

For simple writes, `PMCC_SPI` has a `write()` method that takes the address of the register we want to write to and the data we want to write. The `write()` method calls the `xfer()` method with the read flag set to 0. From there, the `xfer()` method sends the data to the PMCC and returns None as the PMCC does not send any data back. If there is an issue with the transaction, the `xfer()` method will raise an exception indicating that it received null from the transfer to the specific address.

The documentation for the PMCC specifies specific bits and ranges of bits within a register address to set different configurations on the device. We often only want to change a specific value at an address and not the entire register. To accomplish this, the `PMCC_SPI` object has a `mask_data()` method that takes the most significant bit (MSB), the least significant bit (LSB), the value we want to write, and the original buffer we are overwriting. This closely matches the way the PMCC documents the use of each register with either a single bit or an inclusive range of bits. First we check if the MSB and LSB are valid values, and if they are not, we raise an exception. Valid values for addressing the 16-bit registers are 0 to 15 for the LSB and LSB to 15 for the MSB. Next, we check if the value provided will fit within the length specific by the MSB and LSB. We then use the MSB and LSB to calculate the maximum value that will fit in the mask. We use this maximum value to determine if the provided value is too large, in order to raise an exception if it is. Finally, we create a mask using the maximum value and shifting it to the left by the LSB. We then take the original buffer and do a bitwise AND with the inverse of the mask to clear the bits between the LSB and the MSB. Finally, we shift our data to the left by the LSB and do a bitwise OR with the original buffer to set the bits between the LSB and MSB to

the new value. This process is shown in Equation **??**.

$$\text{maxValue} = (1 << (\text{MSB} - \text{LSB} + 1)) - 1 \qquad 0 \leq \text{LSB} \leq \text{MSB} \leq 15 \quad (6.3)$$

$$\text{mask} = \text{maxValue} << \text{LSB} \quad (6.4)$$

$$\text{buffer} = (\text{buffer} \ \& \ \sim \text{mask}) \mid (\text{data} << \text{LSB}) \qquad 0 \leq \text{data} \leq \text{maxValue} \quad (6.5)$$

To further simplify the process of setting specific bits in a register, the `PMCC_SPI` object has a `read_write()` that first reads from the address we want to write to, modifies the data we want to change, and then writes the modified data back to the PMCC. The `read_write()` method takes the address of the register we want to read from and one of the following formats for the data we want to write:

- A tuple of MSB, LSB, and value to write to the register
  - e.g. `(15, 8, 0xAA)` would set the register to `0b1010 1010 XXXX XXXX`
- A tuple of a single bit and value to write to the registers
  - e.g. `(2, 0x1)` would set the register to `0bXXXX XXXX XXXX X1XX`
- An array containing combinations of the above two formats
  - e.g. `[(15, 8, 0xAA), (2, 0x1)]` would set the register to `0b1010 1010 XXXX X1XX`

The `read_write()` method first reads the data from the PMCC using the `read()` method and stores it in a buffer. Then we check if the changes provided are a tuple or an array of tuples. If it's a tuple, we just wrap it in an array in order to iterate over it. For each change in the array, we unpack the tuple and call the `mask_data()` method to modify the data we read from the PMCC, updating the buffer each time. If we are only changing a single bit, MSB and LSB are set to the same value. To complete

the transaction, we write the modified buffer back to the PMCC using the `write()` method.

Finally, we provide a `close()` method that simply calls the `close()` method on the `spidev` object to close the SPI connection.

6.3.2 `config`

There are a number of device specific configurations that need to be set for each PMCC in order to operate correctly. To simplify the process of writing code to configure the PMCCs, we utilize the YAML configuration file format to store the configuration for each PMCC (**yaml**). This YAML file has information about the RMQ configuration as well as spectrometer configuration. For now, we will focus on the spectrometer configuration and discuss the RMQ configuration in Section **??**. The spectrometer config section, `spec`, is split into two main sections for the PMCCs, global variables used for every spectrometer, and spectrometer specific variables. For each experiment, we would like to have a single configuration file that can be used on every CM4 to configure multiple PMCCs. To accomplish this, each CM4 is given a unique name that we use to differentiate between each device. Each PMCC connected to a CM4 is then indexed, so we can individually address each one by specifying the CM4 name and the PMCC index.

The format for the global configurations is shown in Table **??**. These configurations are used to set values we don't expect to individually change for each PMCC. While we may create different configurations for different experiments, such as integration time and magnitude or power mode, we will likely make these changes to all PMCCs at once and not individually.

Key	Type	Description
`spec_file`	path	Path to the spectrometer hardware file
`int_time`	int	Integration time in milliseconds
`clock_freq`	int	Reference clock frequency in MHz
`resolution`	int	16 or 32 for 16-bit or 32-bit readout resolution
`shift`	int	0 or 4 for 0 or 4 bit shift
`magnitude`	bool	True for magnitude mode, False for power mode
`window_bypass`	bool	True to enable rectangular window bypass
`window_bit_growth`	bool	True to enable window div 2 bypass
`butterfly_shift`	bool	True to enable butterfly shift for improved noise measurements
`wiring`	array	See Table **??** for wiring configuration
`groups`	dict	Dictionary of CM4 names and an array of PMCC chip IDs

Table 3. Global Variables in the PMCC Configuration File

Key	Type	Description
`dev`	string	Path to device address (e.g. `/dev/spidev0.0`)
`gpio`	int	GPIO pin number for the reset signal
`speed`	int	SPI device speed in Hz, typically 5000000 unless changes for stability reasons and debugging

Table 4. Wiring Configuration in the PMCC Configuration File

After the global configurations, an array of four spectrometer wiring configurations is provided. In full operation, we will have four PMCCs connected to each CM4, so we need a way of specifying the wiring for each PMCC's SPI bus and GPIO reset pin. Because the harness is identical for each CM4, we can specify the wiring for each PMCC along the harness, and it will be the same for every CM4. The only exception to this is the lone 100GHz PMCC connected to its own CM4. The wiring for this PMCC is simply the first index in the wiring array will still work with the rest of the system. Each item in the wiring configuration is as follows in Table **??**.

Finally, we have the group configuration dictionary. Each PMCC comes with a chip ID specified by the manufacturer used to set pre-calibrated values, such as the

		Resolution	
		16-Bit	32-Bit
LSB Shift	0	0x080	0x0C0
	4	0x180	0x1C0

Table 5. Resolution Mode Configuration for PMCC

ADC time skew. The configuration dictionary consists of a CM4 name as the key and an array of PMCC chip IDs as the value. This allows us to specify which PMCCs are connected to each CM4 and configure them accordingly. For the 100GHz PMCC and CM4, the array will only contain a single chip ID.

When loading in the configuration file, we create a `PMCC_Config` object that all configuration information necessary for the CM4. We initialize this object with the `spec` part of the YAML file, the group name of the CM4, and an array of PMCC indexes to configure (e.g. `[1, 2, 3, 4]` for all spectrometers). The `PMCC_Config` object then creates properties for each of the global configurations that can be accessed by the `PMCC_SPI` object. We set the resolution mode of the PMCCs using the `resolution` and `shift` properties and a lookup table for the proper register values as shown in Table **??**. We could, theoretically, set higher values for LSB shift but for the purposes of ASTHROS, we only need 0 and 4 bit shifts.

After the global variables are loaded, we create a dictionary of device configurations for individual PMCCs. This dictionary is indexed by the CM4 group name concatenated with the PMCC index. Each value in the dictionary is a `PMCC_Device_Config` object that is initialized with the chip ID, the wiring configuration at the PMCC index, and the `spec_file` for configuration. The wiring information is paired with the chip ID to create a `PMCC_Device_Config` object that can later be used to configure the PMCC. The provided `dev` path for the SPI configuration is split and stored into

the bus and device number for the `PMCC_SPI` object to use. Finally, the pre-calibrated values are loaded from the `spec_file` by searching for the chip ID in the file and storing the associated values in the `PMCC_Device_Config` object. If the chip is not found in the file, we raise an exception that the configuration provided was not valid. The final product is a `PMCC_Config` object that contains all the necessary information to configure both the CM4 and any number of PMCCs connected to it.

6.3.3 `consts`

The `consts` module is simply a collection of constants used throughout the `PyMCC` module. Many of these are register addresses so that we can easily reference them in the code without having to go back and forth between the PMCC manual and the code. Additionally, we have some large arrays that are used in configuration that we don't want to hard code into the code. For example, `WINDOW_COEFFS` is a vector of 513 values used to configure the symmetrical 1024 point FFT on the DSP. In addition to addresses and coefficients, we keep an array of default values for the PMCC registers so that we can easily identify issues with the device after reset.

6.3.4 `driver`

The `driver` module is the highest level of the `PyMCC` package and is responsible for providing all functionality for the PMCCs. Each PMCC is controlled by a `PMCC_Driver` object that is initialized with a `PMCC_Config` object and the index of the spectrometer in the config that we want to control. After initializing the object, the user must call the `initialize_interface()` method to set up a `PMCC_SPI` object to communicate

with the PMCC. Following SPI set up, the user must call the `initialize_gpio()` method to set up the GPIO pin for the reset signal. Both of these methods check the wiring configuration in the `PMCC_Config` object to ensure that the correct wiring is provided. With both of these methods called, the `PMCC_Driver` object is ready to start the PMCC configuration.

The first thing done before any configuration is toggling the reset signal on the PMCC. This is done by calling the `reset()` method with a boolean value to set the reset signal high followed by low. `reset()` is often called multiple times in the configuration process to ensure that the PMCC is in a known state before configuring it.

> **Note:** The PMCC documentation specifies individual bit fields for each register. When referring to a specific register in the documentation, we will use the format `reg_name` with lowercase letters. These will be identical to the register names in the documentation. During the development of `PyMCC`, we often had to refer to SPI addresses that contain multiple PMCC registers. When referring to these addresses, we will use the format `REG_NAME` with all uppercase letters. These are not documented in the PMCC manual but are used to reference addresses defined in the `consts` module.

After resetting the PMCC, we need to calibrate the Phase Lock Loop (PLL) on the PMCC using `initialize_pll()`. The PLL is used to synthesize all required clocks for the PMCC with the use of an external reference clock. Initializing the PLL is done in three steps, resetting the PLL with `reset_pll()`, calibrating the PLL with `calibrate_pll()`, and finally loading the ADC with the PLL values using `load_adc()`. `reset_pll()` resets the PLL and sets the ADC gain, offset, and time skew configurations. This is done by the following sequence of commands:

1. Write to the `CHIP_CONF` to reset the DSP.

2. Read and write the `PLL_LOCK_CONF` to set the `lock_desired_count` to 3, `lock_tune_off` to 2, and `lock_tune_on` to 4. These set the lock detector control that will later be used to determine if we have locked the PLL.
3. Read and write the `PLL_FVCO_CAL_CONF` to set the `fvco_cal_settletime` to 4. This is used by the PLL's Voltage Controlled Oscillator (VCO) to determine the settling time for the VCO. By setting this to 4, we are setting our settle time to $2^4 = 16$ times the reference frequency.
4. Read and write the `ADC_GAIN_ACCUM` and set the `adc_gain_cal_accum` to 3 which sets the on-chip gain calibration accumulator length to 8192.
5. Read and write the `ADC_GAIN_CONF` and set the `adc_gain_cal_settle` to 3 which adjusts the delay during the gain calibration to 63 clock cycles.
6. Read and write the `ADC_OFFS_CONF` and set the `adc_offs_cal_accum` to 3 and the `adc_offs_plr` to 1. This sets the on-chip offset calibration accumulator length to 8192 and the polarity of the offset calibration to fine (comparator) adjustment mode.
7. Read and write the `ADC_TIME_SKEW_COEF` to set the `time_skew_select` to 0xF. This enables the use of manual time skew codes for the ADC.
8. Finally, write to the `DEMUX_DEL_ADJ_A` to `DEMUX_DEL_ADJ_D` to adjust the delay in the input interleaver clock for the four ADC groups. This value is set to 11 for all four registers, setting the delay to $18.8 * 11 = 206.8$ ps. Currently, this step is hard coded to 206.8 ps but, in the future, we may want to adjust this value based on the chip ID as these values are pre-calibrated for each chip.

After resetting the PLL, we calibrate the PLL using `calibrate_pll()`. This takes many of the values we set in `reset_pll()` to execute the calibration process. The calibration process is as follows:

1. Check if the clock frequency set in the configuration is a multiple of 2000. Clocks must be a factor of 2GHz to ensure that the PLL can lock to the reference clock.
2. Read and write to the `PLL_CONF` and set the `pll_freq_adjust` and the `pll_ndiv`. The `pll_freq_adjust` is used to set the sampling rate of the ADC. We set this value to 2 which indicates a 4GHz clock for the ADC. The `pll_ndiv` is used to set the divider ratio for the PLL feedback. This value is calculated using $N_{div} = 2000\text{MHz}/F_{freq}$ and set in the `pll_ndiv` register. For our 100 MHz reference clock, we set the `pll_ndiv` to 20.
3. Now we begin the calibration process by reading and writing to the `PLL_FVCO_CAL_CONF` to set the `fvc_cal_start` to 1 and resetting the `fvco_cal_settletime` to 4. This starts the calibration process and sets the settling time to 16 times the reference frequency.
4. Finally, we poll the `fvco_cal_cal_done` bit until the band selection is completed. If the calibration is not completed after a default of 10 retries, we raise an exception indicating that the PLL calibration failed.

After the PLL is calibrated, we load the ADC with the PLL values using `load_adc()`. Many of the commands sent during this set are writes instead of reads and writes. This is because we actually want to override the registers are not setting to 0. The process for loading the ADC is as follows:

1. Write to the `VGA_CURRENT_CONF` to set the `vga_current_out_adjust` and `vga_offset_rng`. The `vga_current_out_adjust` is used to adjust the reference current at the VGA output buffer from 2.1 to 6.3 mA in steps of .6 mA. We set this to $2.1 + 6 * .6 = 5.7$ mA by setting the value to 6. The `vga_offset_rng` is used to adjust the offset compensation reference current from 250 to 600 uA in

steps of 50 uA. We set this value to $250 + 7 * 50 = 600$ uA by setting the value to 7.

2. Write to the `VGA_GAIN_CONF` to set the `vga_peak_cntrl` and `vga_gain_adjust`. The `vga_peak_cntrl` is used to reduce the inductive AC peak of the VGA by increasing the capacitance. We set this value to a code of 5. The `vga_gain_adjust` is used to adjust the VGA gain from 0 to 10.4 dB in steps of approximately .53 dB. We set this value to $8 * .53 = 4.24$ dB by setting the value to 8.
3. Write to the `VGA_CONFIG` to enable the common-mode compensation (`vga_cm_comp`), the VGA offset compensation (`vga_offset`), and the VGA enable (`vga_en`). These values are all defaulted to enabled, but it is good practice to set them to ensure that the VGA is properly configured.
4. Write the four `adc_time_skew_adjust1` to `adc_time_skew_adjust4` registers to set the time skew for the ADC. These values are pre-calibrated for each chip and are set in the `PMCC_Device_Config` object.
5. Write to the `ADC_TIME_SKEW_CONF` to set the `time_skew_mode`, `time_skew_select` and `time_skew_polarity`. The `time_skew_mode` is set to 1 to enable calibration for a configured time instead of continuous calibration. The `time_skew_select` is set to 0b1111 to enable each of the four time skew adjustments. The `time_skew_polarity` is set to 1 to enable inverse polarity of the time skew code adjustment direction.
6. Write a 0 to `adc_rst_n` to register a reset to the ADC.
7. Write a 0 to `adc_sub_clk_gen_rst_n` to reset the clock generators for all four ADC groups.
8. Sleep for 100 ms to allow the ADC to reset.
9. Write a 1 to `adc_rst_n` to enable the ADC.

10. Write a 1 to `adc_sub_clk_gen_rst_n` to enable the clock generators for all four ADC groups.

After those three steps, the PLL is calibrated and the ADC is ready for configuration. To verify this, the `PMCC_Driver` object has a `check_connection()` method that checks if we can read from the PMCC. We first read the `CHIP_ID` register to ensure that we can communicate with the PMCC. Despite having the same name as the chip ID we use to differentiate between PMCCs, the `CHIP_ID` register is a fixed value that is set by the manufacturer and will always be 0x6 for the second generation 4GHz PMCCs. Reading the `CHIP_ID` register is a good way to verify that SPI connection is working. We then read the `PLL_LOCK_LOL` register to determine if we have a Loss of Lock (LOL) on the PLL. If the PLL is locked, the `PLL_LOCK_LOL` register will be 0. If we make it past both of these checks, we return True to indicate that the PMCC is connected and the PLL is locked.

The usual next step in the configuration process is to calibrate the gain and offset of the ADC. This is done by calling the `calibrate_adc()` method. This method is a wrapper for the `calibrate_adc_offset()` and `calibrate_adc_gain()` methods and runs both of them twice. The calibrations run in interactive mode so running them twice allows the PMCC to iterate on the calibration values and get a more accurate result. To run the offset calibration, we set the following values within the `ADC_OFFS_CONF` register:

- `adc_offs_cal_en` to 1 to enable the offset calibration
- `adc_offs_cal_mode` to 1, setting the operation mode to a zero offset calibration.
- `adc_offs_cal_interactive` to 1 to start a new calibration using the previous adjustment codes.

After setting these values, we poll the `adc_offs_cal_ack` bit until the calibration is complete. For the gain calibration, we simply have to set the `adc_gain_cal_en` to 1 to enable the gain calibration. We then poll the `adc_gain_cal_ack` bit until the calibration is complete. After both calibrations are run twice, we are done calibrating the ADC.

In the instance we already know the values we want to set for the gain and offset, we provide a `calibrate_adc_preset()` method that takes arrays of the 23 gain and 23 offset values to set the calibration values. These values are loaded into the 23 `ADC_REF_ADJUST_XX` and 23 `ADC_OFFS_ADJUST_COMP_XX` registers respectively. The `adc_gain_cal_mode` and `adc_offs_cal_mode` registers also need to bet set to 0x1 and 0x2 to enable the use of the preset values.

After calibrating the ADC, we are able to configure the DSP. This is highly subjective to the experiment being run but, for ASTHROS, we have a specific configuration that we use that works for the integration time and resolution we are using. In future version of the code, we will likely pull out some of the hard coded values and make them configurable in the YAML file. The configuration process is as follows:

1. Reset the DSP by writing a 1 to the `CHIP_CONF` register's `dsp_reset`.
2. Load the window coefficients into the DSP by writing the 513 values in the `WINDOW_COEFFS` array to the `WINDOW_REGISTER` register one at a time followed by a pulse to the `dsp_coeff_dest_wind` bit to shift store the value and shift the register.
3. Write to the `dsp_skip_fft_stage` register to 0 to use all four stages of the FFT. This results in 512 frequency bins per sub-band.
4. Write to the `DSP_READOUT_CONF` with the mode from the configuration file as shown in Table **??**. This sets the resolution and shift for the readout.

5. Write to the `CHIP_CONF` register to set the `dsp_enable` bit to 1 to enable the DSP and `dsp_acc_mode` to 1 to enable continuous FFT operation.
6. Read and write to the `FFT_CONFIG` to set the `dsp_magn_bypass`, `dsp_wind_bypass`, `dsp_div_red_wind`. These values are set in the configuration file and are used to enable the magnitude or power mode, the windowing function, and window bit growth.
7. Calculate the number of integrations to run based on the integration time. We do this by diving our desired integration time by the length of time it takes to run a single accumulation on the DSP. For the ADC running at 8 GS/s (Gigasamples per second) reading 16384 time domain bins, this number is 2.048 us. The number of accumulations we collect per integration is a 24-bit value split across two registers. The 16 most significant bits are stored in the `dsp_acc_num_msb` register and the 8 least significant bits are stored in the `dsp_acc_num_lsb` register.
8. If we are in power accumulation mode, we need to set the data shift for the DSP. The maximum output resolution is $<$40 bits so we need to shift the data to the right depending on the integration time. We calculate this by taking the binary logarithm of the number of accumulations per integration and subtracting 8. We get eight because there is a maximum output resolution of 40 bits and the most we can shift the value is 32. By subtracting 8 from the binary logarithm, we get the number of bits we need to shift the data to the right that would maximize the output resolution without overflowing the readout. This value is then stored in the `dsp_data_shift` register.
9. If we are using a butterfly shift, we set the data divide by 2 blocks to the following values:

- `dsp_bfly_shift_pfb` to `0b10101` to alternate between enabled and disabled IFFT processor stages.
- `dsp_bfly_shift_fft` to `0b1010101010` to alternate between enabled and disabled FFT processor stages.

The combination of these essentially skips every other stage of the IFFT and FFT processor, increasing performance for band-limited noise measurements. It accomplishes this by skipping the stages that would divide the data by 2 to reduce the risk of overflow.

10. Write to the `VGA_CURRENT_CONF` to set the `vga_current_out_adjust` and `vga_offset_rng`. We set these values to the same values as we did in the ADC calibration.
11. Finally, we perform a `check_connection()` to ensure everything is in order and return the status.

At this point, the PLL has been locked, the ADC has been calibrated and the DSP has been configured. The most common next step is to start the DSP and begin acquiring data. This is done in two parts, pulsing the accumulation bit and then reading the data. Pulsing the accumulation bit is done in the `pulse_acquisition()` method by writing a 1 to the `dsp_start_acc` register. This starts the DSP's operation and begins accumulating data. After starting accumulation, we immediately perform a single `retrieve_data()` to clear any garbage data that may be present in the DSP's output buffer.

The `retrieve_data()` method follows the following process to read a spectra from the DSP:

1. Read the `dsp_data_ready` register to determine if the DSP has finished accumulating data. We poll this register at a rate of 500 Hz for a maximum of 2000

retries. Both of these values are adjustable arguments to this method but are set to 500 Hz and 2000 retries by default. If we don't receive data after the maximum number of retries, we raise an exception indicating that the DSP did not finish accumulating data. Otherwise, we continue to the next step.

2. Write a pulse to the `dsp_start_readout` register to start the readout process.
3. Take a timestamp to record the time the readout started.
4. Start the readout process by performing a burst read of the `DSP_READOUT_ADDRESS` at `0x4000`. If we are performing a 32-bit readout, we read the address 8193 times to get the first half of our data. Otherwise, for a 16-bit readout, we read the address 8192 times.
5. If we are performing a 32-bit readout, we read the second half of the data by reading the `DSP_READOUT_ADDRESS` at `0x8000` 8191 times. This strange number of reads is due to a bug that causes issues if reading two 8192 blocks of data. Without following this specific sequence, we will be missing one value in the second half of the data, shifting the entire readout.
6. Regardless of the resolution, we write a pulse to the `dsp_reset_ready` register to begin the next accumulation. This discards any data that may be present in the readout buffer so that new data can be stored.
7. Finally, we return the data and the timestamp to the user. If the resolution is 32-bit, we concatenate the two arrays and use NumPy to return a view of the data as an array of 32-bit unsigned integers. Otherwise, for 16-bit readouts, we use NumPy to return a view of the data as an array of 16-bit unsigned integers.

Another common operation after set up is to read raw data from the ADC. This can be useful to ensure that the ADC is working correctly by measuring the mean

and swing of the digitized signal. The `retrieve_adc()` method does just this using the following procedure.

1. Write a 0 to the `dsp_reset` register and a 1 to the `dsp_enable` to enable the DSP.
2. Write a 0 to the `dsp_proto_en` to disable prototype mode on the ADC.
3. Write a 0 to the `DSP_DEBUG_MODES` to clear any other debug modes that may be enabled.
4. Write a 1 to the `debug_wr_from_adc` register at the `DSP_DEBUG_MODES` address to begin writing ADC samples into the debug buffer.
5. Begin polling the `debug_wr_from_adc_done` register to determine when the ADC samples have been written to the debug buffer. If we reach the maximum number of retries, we raise an exception that the ADC data was not ready after the maximum number of retries.
6. Write a 0 to the `DSP_DEBUG_MODES` to disable writing the ADC samples to the debug buffer.
7. Write a 1 to the `debug_wr_by_spi` register at the `DSP_DEBUG_MODES` address to begin moving the debug buffer to the SPI readout buffer.
8. Finally, we perform single reads of the `ADC_READOUT_ADDR` at `0x2000` until we have read all 16384 samples in sets of 8 words across the 2048 lines in the ADC. To convert these values into the actual readout from the 20 ADC cores, we have to do quite a bit of bit manipulation. This is because each of the 20 core's readout is a 6-bit value split across the 8 words in the line. Accomplishing this is done by the following steps:

    a) Start with an array of 8 16-bit words that contain the 20 6-bit for cores A to T.

RRRRSSSSSSTTTTTT OOPPPPPPQQQQQQRR MMMMMMNNNNNNOOOO

JJJJKKKKKKLLLLLL GGHHHHHHIIIIIIJJ EEEEEEFFFFFFGGGG

BBBBCCCCCCDDDDDD XXXXXXXXAAAAAABB

b) Take the array of 8 words and reverse the order. The first word from the readout contains the least significant bits of the line.

XXXXXXXXAAAAAABB BBBBCCCCCCDDDDDD EEEEEEFFFFFFGGGG

GGHHHHHHIIIIIIJJ JJJJKKKKKKLLLLLL MMMMMMNNNNNNOOOO

OOPPPPPPQQQQQQRR RRRRSSSSSSTTTTTT

c) Convert the array of 8 16-bit words into an array of 16 8-bit bytes.

XXXXXXXX AAAAAABB BBBBCCCC CCDDDDDD EEEEEEFF FFFFGGGG

GGHHHHHH IIIIIIJJ JJJJKKKK KKLLLLLL MMMMMMNN NNNNOOOO

OOPPPPPP QQQQQQRR RRRRSSSS SSTTTTTT

d) Unpack the 8-bit bytes into their binary bits using NumPy's `unpackbits()` method.

XXXXXXXXAAAAAABBBBBBCCCCCCDDDDDDEEEEEEFFFFFFGGGGGGHHHHHH-

IIIIIIJJJJJJKKKKKKLLLLLLMMMMMMNNNNNNOOOOOOPPPPPPQQQQQQRR-

RRRRSSSSSSTTTTTT

e) Throw away the first 8 bits of the output as these are used for padding the data.

AAAAAABBBBBBCCCCCCDDDDDDEEEEEEFFFFFFGGGGGGHHHHHHIIIIIIJJJJJJ-

KKKKKKLLLLLLMMMMMMNNNNNNOOOOOOPPPPPPQQQQQQRRRRRRSSSSSSTTTTTT

f) Reshape the array of bits into an array of 20, 6-bit values.

AAAAAA BBBBBB CCCCCC DDDDDD EEEEEE FFFFFF GGGGGG HHHHHH

IIIIII JJJJJJ KKKKKK LLLLLL MMMMMM NNNNNN OOOOOO PPPPPP

QQQQQQ RRRRRR SSSSSS TTTTTT

g) Pack the 6-bit values back into 8-bit bytes using NumPy's `packbits()` method. This method adds zero padding to the end of the array if the length is not 8.

```
AAAAAAXX BBBBBBXX CCCCCCXX DDDDDDXX EEEEEEXX FFFFFFXX
GGGGGGXX HHHHHHXX IIIIIIXX JJJJJJXX KKKKKKXX LLLLLLXX
MMMMMMXX NNNNNNXX OOOOOOXX PPPPPPXX QQQQQQXX RRRRRRXX
SSSSSSXX TTTTTTXX
```

h) Right shift the 8-bit bytes by 2 to remove the padding added by the previous step.

```
AAAAAA BBBBBB CCCCCC DDDDDD EEEEEE FFFFFF GGGGGG HHHHHH
IIIIII JJJJJJ KKKKKK LLLLLL MMMMMM NNNNNN OOOOOO PPPPPP
QQQQQQ RRRRRR SSSSSS TTTTTT
```

9. Perform this process for all 2048 lines in the ADC readout.
10. Compute the mean and standard deviation of the ADC lines.
11. Compute the swing of the ADC using the mean and standard deviation.

$$\text{Swing} = \sigma * 2\sqrt{2} \tag{6.6}$$

12. Return the buffer of ADC readouts, the mean, the standard deviation, and the swing.

Finally, we provide two methods to read and write to all of the PMCC registers. These methods are `fetch_registers()` and `write_registers()`. There are 512 different SPI addresses that can be read and written to on the PMCC. This method allows us to dump the current state of the PMCC or set the PMCC to a known state. These are mostly used for debugging and are not used in the normal operation of the PMCC.

6.3.5 `data_utils`

`data_utils` provides a few handy classes to handle storing data from the PMCC. The simplest of these is the `PMCC_Register_Writer` which, as the name suggests, is able to write the PMCC registers to a file. This is useful for debugging and for storing the state of the PMCC for later use. This class is initialized as an object with a path to the directory where the data will be stored. Using this object, we can call the `write_registers()` method with an array of register values to write the data to a CSV. The data in the CSV will include the register address, the integer value, the hex value, and the default hex value for each register. The file is saved with a timestamp in the filename, so we can identify when the file was written. The `PMCC_Register_Writer` object also has a `read_registers()` method that reads the data from the CSV and returns an array of register values that could be used to set the PMCC to the state it was at when the snapshot was taken. Finally, there is a simple `get_files()` method that returns all the files in the directory where the data is stored.

We also provide a `PMCC_ADC_Writer` class that is used to write the raw data from an ADC test. This class is initialized with a path to the directory where the data will be stored. The `write_adc()` method is called with the data from the `retrieve_adc()` method from `PMCC_Driver` and writes the data to a CSV. This results in 2048 lines of data with the 20 values for each core in each line. The file is saved with a timestamp in the filename, so we can identify when the test was done.

Finally, we have our spectra writers. We need to support writing in both HDF5 and CSV formats. HDF5 will be used during the flight to store the data in a more efficient format, whereas CSV is used for debugging and testing. For writing to HDF5,

we provide the `PMCC_H5_Spectra_Writer` class. This class takes in a path to the directory where the data will be stored, a reference to the spectrometer's driver, the maximum number of writes before the file is closed, and the data type of the spectra. In initialization, we create the directory if it does not exist. We also start a counter at 0 to keep track of the total number of spectra written by the writer.

Before writing the data, we need to create the HDF5 file using the `new_file()` method. If a file is already linked to the writer, we closed that file and open a new one. Next, we create a new file with `spec_<prefix>_<timestamp>.h5` as the filename. `prefix` is an optional parameter that can be used to identify the file and `timestamp` is the current time. The file is created with two datasets. The first dataset is called `stamps` and is used to store the timestamps of the spectra. The second dataset is called `data` and is used to store the spectra. Both datasets are created using the `gzip` compression filter with a compression level of 4. This is done to reduce the size of the file and speed up the writing process. The shape of these datasets is determined by the `max_writes` parameter and the length of the spectra. The `stamps` dataset is a 1D array of 64-bit doubles with a length of `max_writes`. The `data` dataset is a 2D array with a shape of `max_writes` by the length of the spectra and a data type matching the specified data type during initialization. A header is added to the `data` dataset and contains the attributes shown in Table **??**. After creating the file, we start a counter at 0 to keep track of the number of spectra in the file.

When writing data to the file, we use the `write_spectra()` method. This method takes in a timestamp and the spectra to write to the file. If the file does not exist, or we have reached a maximum number of writes, we rerun the `new_file()` method to create a new file. We then write the timestamp and spectra to the `stamps` and

Attribute	Description
id	Name of the the spectrometer (<CM4_name><PMCC_index>)
chip	PMCC Chip ID from manufacturer
int_time	Integration time (ms)
resolution	32 or 16-bit data resolution
shift	0 or 4 for 0 or 4 bit shift
magnitude	True if magnitude mode is enabled, False if power mode is enabled
window_bypass	True if window bypass is enabled
window_bit_growth	True if window bit growth is enabled
butterfly_shift	True if butterfly shift is enabled

Table 6. Attributes of the data dataset in the HDF5 file

data datasets at the current index. After writing the data, we increment the count of spectra in the current file and the total count of spectra written by the writer.

The PMCC_CSV_Spectra_Writer class is used to write the data to a CSV file. It is almost identical to the PMCC_H5_Spectra_Writer class but writes the data to a CSV file instead of an HDF5 file. The PMCC_CSV_Spectra_Writer also implements write_spectra() and new_file() methods so that they it be used interchangeably with the PMCC_H5_Writer class. For new_file(), the file is created with the name spec_<prefix>_<timestamp>.csv and no header is written to the file. Just like the PMCC_H5_Spectra_Writer, the PMCC_CSV_Spectra_Writer will close the previous file if one exists and start a new counter for the number of spectra in a file when a new file is created. The write_spectra() method concatenates the timestamp and the spectra and writes them to the file as a single line. This method also increments the count of the number of spectra in the file, the total number of spectra written by the writer, and will create a new file if one doesn't exist or we reach the maximum number of spectra in a file.

Finally, we provide a wrapper class called PMCC_Spectra_Writers that takes in an array of PMCC_H5_Spectra_Writer and PMCC_CSV_Spectra_Writer objects. This

class is used to write data to multiple files at once. This is useful for writing data to both HDF5 and CSV files at the same time as well as data to multiple locations, such as locally on the CM4 and remotely on the NAS. `new_file()` and `write_spectra()` simply call the same methods on all of the writers in the array. This wrapper also keeps track of the total number of spectra written which is useful for housekeeping and telemetry.

6.3.6 RabbitMQ Control Loop

Of the control loops on ASTHROS, the PMCC control loop is the most complex. This is because controlling the PMCC is an asynchronous driven process, where we need to be able to start and stop taking spectra while listening to commands and sending telemetry. Additionally, one RMQ Control Loop can control multiple PMCCs at once, routing commands to the correct PMCC based on the routing keys. This complexity has led us to develop a threaded approach to controlling the PMCCs that interacts with the threads from the RabbitMQ control loop.

When calling the `rmq_main.py` script, we pass three arguments: `config`, the configuration file, `group`: the group name, and an optional list of indexed `ids` to control. If no `ids` are passed, we will control all of the PMCCs in the group. The `config` has all of the information necessary for `PMCC_Config` as well as the RabbitMQ connection parameters. Additionally, it contains a mapping of group names to the PMCC IDs that are in that group. For example, the `1A` group contains PMCC `91` at the zeroth index and PMCC `119` at the first index. The 100 GHz channel, named `100G`, only contains one PMCC as there is only one spectrometer on that channel. After creating the RabbitMQ connection, we create a dictionary of `PMCC_Thread`

objects for each spectrometer in the group. These objects are created by wrapping the `PMCC_Driver` object, pairing it with `PMCC_Spectra_Writers` and `PMCC_ADC_Writer` objects, and adding threading capabilities for collecting spectra via a collecting thread and event flag.

After creating this dictionary, we manually create `threading` threads for the `rmq._publish_status()` decoration, passing `PMCC_Thread.send_status()` as the target function at a 1 second interval. When the thread is started, it will call the `send_status()` method on each of the PMCC threads in the group and return the number of spectra collected, if we are currently collecting spectra, any errors that have occurred, and a timestamp of when the status was sent. These threads are manually added to the threads `rmqtools` handles by appending them to `rmq.threads`.

Finally, we create the RPC execution handler to wrap with `rmq.handle_command()` This handler first determines which spectrometers the command is for and then matches the string command to their corresponding `PMCC_Thread` methods. For most atomic methods, such as `pll` to initialize the PLL, or `reset` to reset the PMCC, we simply run call the command and then return the result. In the case of collecting spectra, a continually running process, our command starts the process and returns a status message indicating that the process has started rather than any data from the process. We do this in order to avoid hanging the RabbitMQ connection while waiting for data to be collected. This is accomplished with three functions, `take_spectra()`, `start_spectra()`, and `stop_spectra()`. `take_spectra()` is a while loop that runs while the collecting falg is set. While running, it simply uses the driver's `retrieve_data()` method to collect spectra and pass that data to the `PMCC_Spectra_Writers` object to write the data to the file. If an error occurs during the flag, the error message is logged and sent to the status exchange during

`send_status()`. `start_spectra()` creates a new file for the `PMCC_Spectra_Writers` object, sets the collecting flag, and starts the collecting thread. If the flag is already set or the thread is alive, we return false and set an error stating that the spectra is already being taken. Finally, `stop_spectra()` handles stopping the thread. If the thread is alive, we clear the collecting flag and join the thread to wait for it to finish. Once joined, we create a new thread so that we can start collecting spectra again if the user calls `start_spectra()`. In the case that the thread is not alive, we first check if the collecting flag is set. If it is, we know that there was an error during the collection process that stopped the thread. In this case, we add the error to the error buffer and return False, indicating that the spectra was not stopped correctly. If the flag is also not set, we know that `stop_spectra()` was called before `start_spectra()` and we return False, indicating that no spectra was being taken.

In addition to taking spectra, we also provide commands to handle the usual functionality of the PMCC. These commands include the following:

- `test` - Test PMCC Connection using the `check_connection()` method.
- `pll` - Initialize the PLL using the `init_pll()` method.
- `adc` - Calibrate the ADC using the `init_adc()` method.
- `dsp` - Configure the DSP using the `init_dsp()` method.
- `stats` - Run the `retrieve_adc()` method and get the ADC statistics. This method also writes the data to a CSV file using the `PMCC_ADC_Writer` object.
- `reset` - Reset the PMCC using the `reset()` method.
- `read` - Reads the PMCC registers using the `fetch_registers()` method.
- `write` - Writes to the PMCC registers using the `write_registers()` method using a dictionary of register-value key-value pairs.

- `ack` - Toggles the PMCC acquisition bit using the `pulse_acquisition()` method.
- `rmqtest` - An echo test to ensure that the RabbitMQ connection is working. We include this as the multiple PMCCs could result in individual connections failing.

Regardless of the command, the results are returned using a `ResponseObject` that contains a dictionary of the results using the spectrometer name as the keys. These spectrometer names are created by combining the group name and the index of the PMCC in the group. Combining the group name with the index allows us to harmonize the data down the line into one response when calling multiple CM4s at once.

6.4 Cryocool

ASTHROS uses a closed loop cryocooler to cool the receiver system to 4K during flight. The cooler is a low-power, 4-stage, pulse tube refrigerator built by Lockheed Martin Corp. through the NASA Advanced Cryocooler Technology Development Program (ACTDP) for the James Webb Space Telescope (JWST) (**olson2005lockheed**) (**coulter2003nasa**). The specific cooler we are using was, fortuitously, a leftover prototype from the ACTDP that was not selected for the JWST mission and can be repurposed for ASTHROS (**kawamurateraherts**).

To interface with the cryocooler we have three main components: the cryocooler controller, the temperature sensors, and the pressure sensors. All three of these devices are connected to our central command computer. The cryocooler controller uses an RS-422 interface to read and write parameters to the device. The temperature sensors are connected to a Lakeshore 240 temperature controller which is connected via USB

Key	Type	Description
CONN	dict	Connection parameters for the RS-422 connection specifying PORT, BAUD, and READ_TIMEOUT
TELEMETRY	list	List of registers read when issuing a telemetry command
TEMP_RAMP	path	Path to the temperature ramp file
REGS	path	Path to the register configuration file

Table 7. Cryocooler Configuration Parameters

to the command computer. The pressure sensor is connected to the central command computer via an RS-485 interface.

These three systems are split into different modules in the `cryocool` package.

6.4.1 `cryocool`

Within the `cryocool` package, we have the `cryocool` module that is used to control the cryocooler. The `Cryocool` class is instantiated with a configuration YAML that contains the parameters for the cryocooler. The necessary fields for configuration are shown in Table **??**. We first set up the serial communication with the cryocooler using the `CONN` parameters and the Python serial library. We then make note of the status registers that are read out when we issue a status command to the cryocooler. After this, we read in the temperature ramp as a pandas dataframe from the `TEMP_RAMP` file (**reback2020pandas**). This file contains points for a calibration curve to convert voltage from the internal temperature sensors to temperature.

Finally, we read in the register configuration file from the `REGS` path. This file is a CSV table that contains information for each register on the cryocooler and how to convert read values to their intended units. Table **??** shows the fields in the register configuration file. Each row of the table is read in as a `Reg` object that allows us to more

Key	Type	Description
`Name`	str	Full name of the register
`Label`	str	Parameterized name of the register
`Address`	int	Address of the register
`Write`	bool	True if the register is writable
`Units`	str	Units of the register
`Inverse`	int	Blank if the value is not inverted, value to multiply by if the value is to be inverted
`MinVal`	float	Minimum value of the register
`MaxVal`	float	Maximum value of the register
`MinCount`	int	Minimum digital value of the register
`MaxCount`	int	Maximum digital value of the register
`Housekeeping`	bool	True if the register is a housekeeping value
`Temperature`	bool	True if the register is a temperature value

Table 8. Cryocooler Register Configuration Parameters

easily convert between the raw register values and the intended units. The `Reg` object takes in the configuration row and saves the values as attributes. If a register specifies an inversion or min/max values, we label the register as convertible. If convertible, the `Reg` object has `convert_to_count()` and `convert_to_val()` methods that convert between the raw register value and the intended units. This is accomplished by a simple linear conversion as shown in Equation **??**.

$$y = \max\left(\min\left(\frac{Y_{max} - Y_{min}}{X_{max} - X_{min}}\left(x - X_{min}\right) + Y_{min}, Y_{max}\right), Y_{min}\right) \tag{6.7}$$

Additionally, if the value is converting to a count, we round the result to the nearest integer. Note that the conversion is clamped to the min and max values of the register. This is to ensure that the conversion does not exceed the bounds of the register.

It is also worth noting that the `Reg` object, despite knowing if a register is a temperature value or not, does not convert the temperature values to Kelvin. This is handled by the `Cryocool` object as there are multiple ways of providing and displaying temperature values. When converting between count and value, the `Reg` object will return a voltage value as per the documentation of the cryocooler. The `Cryocool`

object will then convert this voltage to a temperature using the temperature ramp as explained later.

Once the `Cryocool` object is initialized and the `Reg` objects are created and stored in a dictionary for easy access. Because we may refer to registers by their label or their address, we have implemented a `get_reg()` method that takes in either the label or the address and returns the corresponding `Reg` object. If the label or address is not found, the method will raise an exception indicating that the register was not found.

6.4.1.1 Serial Communication

There are three main commands that we can issue to the cryocooler, `get()`, `set()`, and `status()`. All three of these commands are issued by writing to the cryocooler and reading the response. The sent command needs to have a checksum appended to the end of the command to ensure that the cryocooler receives the command correctly. We implemented a `calc_checksum()` method that calculates the checksum for a given byte array. This method of calculating the checksum is as follows:

1. Initialize the checksum to 0.
2. For each byte in the byte array, add the byte to the checksum.
3. If the checksum is greater than 255, subtract 256 from the checksum and add 1 to the checksum.
4. Take the two's complement of the checksum and mask it with 0xFF.
5. Return the checksum.

To encode the checksum we have also implemented an `encode_checksum()` method that takes in a byte array and appends the checksum to the end of the array.

In addition to the checksum, the cryocooler control software requires byte stuffing to ensure a start frame delimiter is not present in the data. For our cryocooler, we use the byte sequence `0x01 0x02` as the start frame delimiter. To ensure that this sequence is not present in the data, we insert a `0x01` byte after every `0x01` byte in the data. This occurs after the checksum is calculated and before the data is sent to the cryocooler. To accommodate this, we have implemented a `stuff_bytes()` and `unstuff_bytes()` method that will add and remove the extra `0x01` bytes respectively.

Commands sent to the croycooler are done using the serial library in Python. To support the serial communication, we have implemented `serial_write()` and `serial_read()` methods to act as the bare-bones for the `get()`, `set()`, and `status()` methods. The `serial_write()` command simply takes in a byte array and writes that on the serial buffer. In most cases, we want to flush the input and output on a write to ensure that the command is sent and received correctly. This is done using an optional `flush` argument in the `serial_write()` method.

The `serial_read()` method is a bit more complicated. This method takes no arguments and simply reads the serial buffer until it hits a predetermined timeout or we stop receiving bytes. We loop through the serial buffer checking how many bytes are waiting to be read. Once the number of bytes waiting to be read stops increasing, we perform a read of the buffer and store the data. After that, we unstuff the bytes and check the checksum. If the checksum is correct, we return the data.

The `get()` method is the next level of abstraction above the `serial_read()` method. This method creates a command to read a register from the cryocooler, sends the command, and then returns the response. The returned value will be the raw digital value from the register and is converted to the intended units in another method. The method builds the byte array command as follows.

1. Start with the start frame delimiter `0x01 0x02`.
2. Add a `0x02` to specify the command will be 2 bytes (a command and a register address).
3. Add the command byte `0x02` to specify that we are reading a register.
4. Add the register address to the byte array in big endian format.
5. Calculate, encode, and add the checksum to the byte array.
6. Byte stuff everything after the start frame delimiter.

After sending the command, we read the response from the serial buffer and then convert the value from bytes in an array to an integer.

The `set()` method is structurally similar to the `get()` method but is used to write to a register on the cryocooler. The method builds the byte array command as follows.

1. Start with the start frame delimiter `0x01 0x02`.
2. Add a `0x04` to specify the command will be 4 bytes (one for the command, one for the register address, and two for the value)
3. Add the command byte `0x01` to specify that we are writing to a register.
4. Add the register address to the byte array in big endian format.
5. Add the value to the byte array in big endian format.
6. Calculate, encode, and add the checksum to the byte array.
7. Byte stuff everything after the start frame delimiter.

The set method has an optional `skip` parameter to read the response from the cryocooler. If `skip` is set to True, the method will return None and not read the response. If `skip` is set to False, the method will read the response from the cryocooler and return the value of the register after writing.

Finally, we have the `status()` method which is a special case of the `get()` method. The cryocooler has implemented a status command that reads 14 status registers at once. These registers are regularly read out to monitor the health of the cryocooler. The `status()` method builds the byte array command as follows.

1. Start with the start frame delimiter `0x01 0x02`.
2. Add a `0x01` to specify the command will be 1 byte (one command).
3. Add the command byte `0x05` to specify that are reading the status.
4. Calculate, encode, and add the checksum to the byte array.
5. Byte stuff everything after the start frame delimiter.

The `status()` method then reads the response from the cryocooler and splits the 14 values into an integer array.

6.4.1.2 User Facing Methods

With all of the serial communications methods in place, we can now build off of those to create the user-facing methods. The first of these is the `reset()` method. This method uses the `set()` method to write a 1 to the reset register on the cryocooler. It skips the response from the cryocooler as the reset command does not return a value.

Next we have a `get_data()` method that reads the value for a specific register on the cryocooler. This method can take either a `Reg` object, a label, or an address as an argument. The method will then call the `get()` method with the register address to obtain the digital count from the register. Optional parameters for `convert` and `in_kelvin` are available to convert the digital count to the intended units. The result after conversion of temperature registers will be in Voltage. If both `convert` and

`in_kelvin` are set to True, the method will convert the voltage to temperature using the temperature ramp. This level of control over the conversion allows for more flexibility in how the data is displayed to the user.

The `set_data()` method is the next user-facing method. Like the `get_data()` method, this method can take a `Reg` object, a label, or an address as an argument. The method also takes in a value to write to the register and parameters specifying any preprocessing that needs to be done on the value. The `in_kelvin` and `convert` parameters make their return here to specify if the value needs to be converted from temperature to voltage and if the value needs to be converted to a digital count. When providing a temperature in Kelvin, both `in_kelvin` and `convert` should be set to True. The method will convert the value into a digital count and then use the `set()` method to write the value to the register. The response from the cryocooler is then read and reconverted to the original units provided by the user.

`get_status()` is similar to the `get_data()` method but does not take any arguments for the register. Instead, it simply takes the optional parameters for `convert` and `in_kelvin` to convert the status registers to the intended units. Get status calls the `status()` method to read the status registers and then iterates over the `TELEMETRY` registers specified in the configuration file to match the read values to their `Reg` objects. For each register, the method will convert the value to the intended units and return a dictionary of the register labels and values.

Finally, we have a `get_all()` method which gets every register specified in the configuration file. This method is used to get a snapshot of the cryocooler's state and is used for debugging and monitoring the health of the cryocooler during the flight. We first read the status registers using the `get_status()` method. We then iterate over all registers in the configuration file and read the value of each register that wasn't

Key	Type	Description
ID	str	Manufacturer ID for the device
NAME	str	Full name of the device
LABEL	str	Parameterized name of the device
TYPE	str	Type of readout for the device (Sensor, Kelvin, Celsius, or Fahrenheit)
CHANNELS	list	List of channels on the device with the NAME and CHANNEL number of the channel

Table 9. Lakeshore Configuration Parameters

read in the status command. This method accepts keyword arguments for `convert` and `in_kelvin` to convert the values to the intended units in the `get_status()` and `get_data()` methods.

6.4.2 `cryotemp`

The `cryotemp` module wraps the lakeshore package with a `Lakeshore` class that can be used to read values from a Lakeshore 240. The configuration for the Lakeshore devices is stored in the YAML file that contains the connection parameters for the device. On ASTHROS, we have two Lakeshore 240s that are connected to the central command computer via USB. The configuration file for the Lakeshore devices is shown in Table **??**.

The `Lakeshore` class is initialized with one of the Lakeshore configurations from the array of configurations in the YAML file. The class then implements three simple methods to read the values from the Lakeshore device. `read_channel()` and `read_index()` are used to read data from the Lakeshore device for a specific channel number or index in the configuration file. The `read_all()` method reads all channels

on the Lakeshore device and stores them in a dictionary with the channel name as the key and the value as the value.

6.4.3 `pressure`

The `pressure` module is used to read data from the pressure sensor on ASTHROS. This method is relatively simple as the pressure sensor is a single device connected to the central command computer via RS-485. The only configuration needed for the pressure sensor is the connection parameters for the RS-485 connection (`PORT`, `BAUD`, and `READ_TIMEOUT`). The `Pressure` class is initialized with the connection parameters and implements a `get_pressure()` method that reads the pressure from the sensor. This method has helper methods to `write()` and `readall()` data from the device. As we only have one command to read the pressure from the sensor, we hard code the command in the `get_pressure()` method. This command is `@253PR1?;FF` and is sent to the pressure sensor using the `write()` method. We then read the response from the sensor and find the pressure value in the response between the `ACK` and `;` characters.

6.4.4 RabbitMQ Control Loop

Using the drivers in the `cryocool`, `cryotemp`, and `pressure` modules and the `rmqtools` package, we can create a control loop for the cryocooler to interact with ther est of the system. To start, we instantiate `Cryocool` and `Pressure` objects using the configuration file. For temperatures, we create a dictionary of `Lakeshore` devices using the configuration file. `LAKESHORE` in the configuration file is an array

of configurations for each Lakeshore device so we need to iterate over the array and create separate `Lakeshore` objects for each device, mapping them to the dictionary using `LABEL` as the key. Once all of the hardware is initialized, we create a RabbitMQ connection using parameters under the `RMQ` key in the configuration file.

With all of our pieces in place, we can now begin connecting them together to work with the `rmqtools` package. First, we initialize a `PriorityLock` so that we don't end up with multiple threads trying to access the same hardware at the same time. In theory, we could have multiple locks for each device, but for simplicity we have a single lock for any hardware access under the cryocool umbrella. For status publishers, we define three functions: `cryocool_status()`, `pressure_status()`, and `lakeshore_status()`. These functions run their respective status commands and wrap the response as a dictionary under the `data` key. A `status` and `time` key are added to the dictionary and returned. In the case of an error, the `status` key is set to 1 and the `data` key is replaced with `error` and the error message.

These functions are wrapped with `rmq.publish_status()` at intervals of 20 seconds, offsetting by 5 seconds for each function to stagger the responses. These functions are also wrapped with our `PriorityLock` at the lowest priority level with a `timeout_return` set to a dictionary with the timeout error message. Each status buffer is set with a different queue name as follows: `housekeeping.cryo`, `housekeeping.pressure`, and `housekeeping.lakeshore.<label>`.

Finally, we create a `cryocool_command()` function to handle any command sent to the cryocooler. This function is decorated with `rmq.handle_command()` and a priority lock of 1 to ensure that it is handled before any routine status commands. As described in **??**, this decorated function will receive a string for the command name

as well as any positional arguments or keyword arguments. The following commands are supported:

- `read_all` - Read all of the values from the `Cryocool` object.
- `read` - Read from a specific register of the `Cryocool` object.
- `write` - Write to a specific register of the `Cryocool` object.
- `pressure` - Read the pressure from the `Pressure` object.
- `lakeshore` - read all values from the `cal` and/or the `temp` `Lakeshore` objects.

Finally, after the command is executed, the response payload is updated with the status of the command and the data returned from the command. This is returned in the rmqtools wrapper to be packaged and sent back to the sender.

After defining all of these functions, we simply run `rmq.start()` to start the threads and begin listening for commands.

6.5 Motor

Behind the antenna is a small stepper motor used to chop between the sky and the internal calibration source. This motor is controlled using the intelligent motion systems MDrive Motion Control System (**mdrive**). The motor is connected to the command computer via a serial RS-485 connection. The MDrive system is a command language used to program and configure the motor. While we could use the MDrive software to control the motor, we have implemented a Python interface to control the motor to better integrate with the rest of our architecture.

The `motor` package consists of a single `motor` module that contains the `Motor` class. The `Motor` class is initialized with a configuration as outlined in Table **??**.

Key	Type	Description
CONN	dict	Connection parameters for the RS-485 connection specifying PORT, BAUD, and READ_TIMEOUT
HWVARS	dict	Dictionary of hardware variables for the motor, described in set_hwvars()
FUNCTIONS	dict	Dictionary of functions for the motor, described in set_functions()

Table 10. Motor Configuration Parameters

After initializing the serial connection with the motor, an optional parameter `reset` can be passed to the `Motor` class to reset the motor and initialize the hardware variables and program functions. This is done by running `reset()` followed by `set_hwvars()` and `set_functions()`. Both of these methods will be explained after discussing the bare-bones methods for serial communication with the motor.

Like most serial connections thus far, we have implemented `write()` and `readall()` methods to write commands and read responses from the motor. These are nearly identical to the `write()` and `readall()` methods in the `pressure` module with slight changes to the command and response formats. The `write()` method takes in a command as a string and writes it to the serial buffer. It appends a `CR` character to the end of the command to signify the end of the command before encoding the string to bytes and writing it to the buffer. If an optional `flush` parameter is set to True, the method will flush the input and output buffers before writing the command. Also optionally, the `write()` method can take a `readwait` parameter that will read the response from the motor after writing the command using the `readall()` method. The `readall()` method reads the serial buffer until it hits a timeout or stops receiving bytes. It then takes the byte array from the read and decodes it to a string.

To reset the motor, we have implemented a `reset()` method that sends a reset command to the motor. This command is simply the string `\x03` which is the ASCII

Key	Name	Description
HC	Hold Current	Current in the motor used to hold it in place after a movement. Entered as an integer percentage of the maximum current.
RC	Run Current	Current in the motor used to run the motor. Entered as an integer percentage of the maximum current.
VM	Velocity Maximum	Maximum velocity of the motor in steps per second.
A	Acceleration	Acceleration of the motor in steps per second squared.
D	Deceleration	Deceleration of the motor in steps per second squared.

Table 11. Motor Hardware Variables

character indicating the end of text. This command resets the motor and clears the buffer.

The MDrive system has a set of hardware variables that can be set and read to control the motor. `set_hwvars()` is a method that initializes the hardware variables for the motor. During initialization, we take the `HWVARS` dictionary from the configuration file and pass that as a paramter to the `set_hwvars()` method. `set_hwvars()` iterates over the dictionary provided and calls `set_hwvar()` for each key-value pair. `set_hwvar()` takes in a key and value and writes the command to set the hardware variable to the motor as simply `<key> <value>`. The response for this command is then read and returned. The values we define in the `HWVARS` configuration are shown in Table **??**.

The MDrive system supports programming functions that can be called to move the motor. In the `FUNCTIONS` configuration, we define a dictionary of functions that can be called to move the motor. The name of each function is the key in the dictionary and the value is a dictionary defining the function. Each function has a `LB` label and `ADDR` address that maps to a memory label and address location on the motor. Finally, the configuration has an array of `STEPS` that define the commands we would like the

motor to execute. These steps can be anything from the manual but we primarily use the following for motor movement.

- `P <position>` - Position Counter, used to mark the current position of the motor for future movements as a frame of reference
- `MA <position>` - Move Absolute, used to move the motor to a certain position based on the position counter
- `MR <steps>` - Move Relative, used to move the motor a certain number of steps relative to its current position
- `H` - Hold Program Execution, used to wait for the motor to finish moving

The `set_functions()` method initializes the functions for the motor. It iterates over the `FUNCTIONS` dictionary from the configuration file and calls `set_function()` for each key-value pair. To program a new function we first write to the motor with a command to enter programming mode at a specific address in memory (`PG <ADDR>`). From there we write a command to label the current address in memory for future reference (`LB <LB>`). We then iterate over the steps written in the configuration file and write each step to the motor. After all steps are written, we write an `E` command to indicate to the runtime to End Program Execution. Finally, we write a `PG` to exit the programming mode. For ASTHROS, we have a few predefined functions we use to interact with the motor. These are shown in Table **??**.

To run a function on the motor, we have implemented a `run_function()` method that takes in the name of the function to run. This name needs to match the key in the `FUNCTIONS` configuration. The method then uses `write()` to write an execute command with the label of the function to the motor (`EX <LB>`). This echos the function back after writing. After running the function, the method runs a print

Key	Description
INIT	Initializes the motor by moving it as far in the positive direction as possible until it hits the physical stop, programs that position as 0, and then moves back to the open position.
FLIP	Flips the motor close then open for debugging
CAL	Closes the motor so that our detectors are pointed at the calibration source
SKY	Opens the motor so that our detectors are pointed at the sky
TOGGLE	Uses conditional branches to toggle between the CAL and SKY functions using our position. If the position is outside of the expected range, the motor will run the INIT function to reset the position.

Table 12. User Programmed Motor Functions

command to obtain the current position of the motor (`PR P`). The method then returns the name of the command sent, the response from running the function, and the current position of the motor.

Finally, we have a `status()` method that writes a print all command to the motor to get the status of the motor (`PR AL`). This prints out every register in memory and returns a dictionary of the labels and values of the registers. This is useful when debugging functions to ensure that they have been programmed properly. Additionally, the values of the registers can be used to monitor the health of the motor during flight.

6.5.1 RabbitMQ Control Loop

For the motor's main control loop we utilize `rmqtools` to expose methods defined in the `Motor` class to publish housekeeping data and handle commands. To begin, we load in the configuration for the motor and RabbitMQ from the YAML file. In starting this script, we have the option to pass a `-quiet` flag that disables the motor reset on startup. This is useful when testing the motor to avoid having it move to the

reset position every time the script is run. We create our `Motor` and `RmqConnection` objects and then begin the main loop.

In this main loop, we create a `PriorityLock` to avoid collisions when writing commands to the motor. We then define two functions to publish status over the status exchange and handle RPC executions. The first is simply `motor_status()` which is decorated with both the `rmq.publish_status()` and `lock.with_lock()` decorators outlining a 15 second interval for publishing and lowest level of priority. The function itself simply calls `motor.status()` and packages the response as a dictionary with the `status` and `time` keys. Any errors are caught and placed in the return dictionary with the `status` key set to 1 and the `data` key replaced with the error message. When the function is called by the status thread, it will obtain this dictionary and format it so that it can be published to the status exchange with the queue name `housekeeping.motor`.

The second function is `motor_command()` which is decorated with `rmq.handle_command()` with the `motor.command` key and a priority lock of 1 to ensure that it is handled before any routine status commands. The command parses the command string from the RPC execution and matches it to one of the following commands:

- `execute` - Executes a function on the motor based on the label in the configuration file.
- `write` - Writes a specific hardware value to the motor based on the passed variable and value.
- `status` - Receives the status of the motor from `motor.status()`

The response from the command is packaged as a `ResponseObject` and returned from the function to be handled by the `rmqtools` package to return to the sender.

Key	Type	Description
INFO	str	Description of the register
FIELDS	list	List of the fields contained in the register
STRUCT	str	Struct format string for the register to split into fields
R	int	Register read address (optional for write only addresses)
W	int	Register write address (optional for read only addresses)

Table 13. Antenna Register Configuration Parameters for a Single Register

6.6 Antenna

ASTHROS flys with a 2.5m diameter antenna to collect far-infrared light. The antenna and the structure surrounding it have built in heaters and temperature sensors to ensure the antenna is at the correct temperature during flight. Additionally the secondary hexapod mirror is affixed above the antenna and can be moved to adjust the focus of the telescope. The antenna and hexapod are controlled by an on board computer embedded in the structure of the telescope. This computer is connected directly to the central command computer via RS-485 as well as broadly to the rest of the system via Ethernet through the readout network.

The `antenna` package includes the `antenna` module that contains the `Antenna` class which is used for all communication with the computer controlling the antenna. The `Antenna` class is initialized with a configuration file that contains the connection parameters for the RS-485 connection as well as a path to a YAML file that contains the register configuration for the antenna. This YAML file outlines the registers on the antenna as shown in Table **??**.

At the bottom of the YAML file are arrays for `ALARMS` and `NACK` errors. In the case that we recieve an error from the antenna, we can check the error code against the `ALARMS` and `NACK` arrays to determine the error.

For communication with the antenna, our most bare-bones method is the `decode_read()` method which reads whatever is in the serial buffer and returns the byte array. It accomplishes this by reading the first four bytes of the buffer to determine the length of the message in the header. It then reads the message from the buffer plus the checksum and returns the header plus the message as a byte array. This method is used primarily in `command()` which is used to send read and write commands to the antenna. This method takes a `message_id`, `payload` and `encoding` for the payload. The payload is a dictionary matching the `FIELDS` in the register configuration file for writes. The method then uses `encode_frame()` to encode the message and add the checksum. If any of the data in the payload is in the wrong format, we cast it to the correct format based on the encoding using `fix_payload()`. For the checksum, the antenna utilizes a CRC-16 CCITT checksum with a precomputed lookup table computed table (**koopman2004cyclic**). The process for calculating the checksum is as follows:

1. Initialize the checksum to 0xFFFF.
2. For each `byte` in the message, use the following equation:

$$\texttt{crc = ((crc < < 8) XOR table[(crc > > 8) XOR byte])} \tag{6.8}$$

3. To keep the checksum as a 16 bit integer, mask the checksum with `0xFFFF`.
4. After iterating through all the bytes in the message, the checksum is the final value.

After a request byte array is created, the method writes the request using the serial buffer and then uses `decode_read()` to read the response from the antenna. Similar to `encode_frame()` we have a decode frame that takes in a byte array and the encoding for the payload. During decoding, we check the checksum of the message and return

the payload if the checksum is correct. In the case of an error from the antenna, we unpack the message as bytes and return the error message.

With `command()` in place, we have implemented the user facing methods for the `Antenna` class. These methods are `read_reg()`, `read_all()`, `write_reg()` and `read_write_reg()`. `read_reg()` takes in a register label and reads the register from the antenna. It checks if the register has a read address and then calls `command()` to read the register. We then map the fields in the register to the values in the response and return 0 and a dictionary of the fields and values. If the message is an error, we check the error against the `NACK` table and return 1 and the error message. `read_all()` simply iterates over all of the registers in the configuration file, checks if the register has a read address, and then runs `read_reg()`.

`write_reg()` is used to write to a register on the antenna. The method takes in a register label and fields as keyword arguments (`kwargs`) to write to the register. After checking that the label exists and the register has a write address, the method creates a payload dictionary. First, we check if the `default` was passed to the method, giving us a default value for any fields not passed. Then we iterate through all of the fields in the register, checking if they're in the fields passed to the method and adding them to the payload. If a field is missing from the `kwargs` it will be set to the default value if provided or raise an exception. Once the payload is created, we call the `command()` method to write the register and return the response or error message if necessary.

Finally, `read_write_reg()` is used to read and write to a register on the antenna. This is primarily used to make changes to a specific field in a register without changing the other fields. The method takes in a register label and `kwargs` for any fields to write to the register. The method first reads the register using `read_reg()` and then updates the fields in the register with the fields passed to the method. After updating

the fields, the method calls `write_reg()` to write the register back to the antenna, returning the response.

6.6.1 RabbitMQ Control Loop

For the antenna's main control loop, we utilize `rmqtools` to manage sending housekeeping at regular intervals and handling commands from other parts of the network. We load the configuration file and separate the antenna configuration from the RabbitMQ configuration. We then create and configure `Antenna` and `RmqConnection` objects and start the main loop. Much like other control loops, we define a `PriorityLock` and will be utilizing this to prevent collisions when communicating with the antenna.

For status publishing, we define a `antenna_status()` function that is decorated with `rmq.publish_status()` at a 15 second interval and the lowest level of priority lock. `antenna.read_all()` is called and the response is wrapped with the status of the command as well as the time of the response for time harmonization. If any error occurs when reading the values, the status is set to 1 and the error message is returned in place of the data. These values are sent to `housekeeping.antenna` on the status exchange.

Unlike other systems, we also have a `antenna_position()` function that provides a higher temporal resolution status of just the antenna position. This is decorated with `rmq.publish_status()` at a 1 second interval and the lowest level of priority lock. This function calls `antenna.read_reg()` twice with `ELEVATION_MEAS` and `HEXA_POS_REAL` to obtain the elevation and position of the hexapod. These values are packaged handled exactly like the `antenna_status()` function and published to the status exchange with the queue name `pointing.antenna` on the status exchange.

Finally, we define a `antenna_command()` function that is decorated with `rmq.handle_command()` with the `antenna.command` key and a priority lock of 1 to prioritize it over status and pointing publishing. To expose the methods in the `Antenna` class, we define a set of commands that can be sent to the antenna:

- `read` - Reads a specific register based on an ID label in the configuration file.
- `write` - Writes to a specific register using the ID label in the configuration file and keyword arguments for the fields to write.
- `read_write` - Performs a read and write on a specific register using the ID label in the configuration file and keyword arguments for the fields to write, falling back to the read values if not provided.
- `specsheet` - Returns a listing of all of the registers in the configuration file along with their descriptions and ability to read and write. Useful when you forget the label of a register.

The response from these commands is packed as a `ResponseObject` and sent back to the sender as a response.

6.7 Power Supply Unit

On ASTHROS, we use custom Smart Power Supply Units (PSU) to provide power to almost every component on the readout system. A Smart PSU allows us to set the voltage and current for each channel independently and monitor the current and voltage on each channel. This level of granularity is vital for components on ASTHROS like the mixer and amplifiers that require precise voltages and currents to operate properly. The Smart PSUs are controlled over SPI using command line tools provided by the manufacturer, with six addressable Smart PSUs connected to two

RS-485 ports on the command computer. These tools handle most of the low level details of packetizing and parsing the data from the Smart PSU so that end users can easily interact with the Smart PSU. This method of operating the Smart PSU is not ideal for in flight operations as the command line tools are not designed with real time operations in mind. As such, we have developed and implemented a Python package that essentially wraps the command line tools as a driver for the Smart PSU so that it can be integrated into the rest of the readout system.

The core of the design for the ASTHROS Smart PSU drivers is the `SmartPsu` class that contains a set of `channels` through witch the Smart PSU can be controlled. The `SmartPsu` is instantiated using a pointer to an `Environment` object that contains information about the execution environment for the Smart PSU. Because our only method for communicating with the Smart PSU is through compiled command line tools, our `Environment` object is necessary to provide paths to the tools so that the rest of the system can function properly. The `Environment` object is a simple class that contains a set of paths to the tools as well as setting up a logging system that other classes can use to log messages, warnings, and errors.

`Channel` is a class that defines a single channel on the Smart PSU and contains the base functions to read and write to that channel. These channels are defined by the following parameters:

- `PORT` - The device port for the channel (e.g. `/dev/ttyS7`).
- `ADDRESS` - The address of the channel on that port (e.g. `0`).
- `BANK` - The bank of the channel within that address (e.g. `A`, `B`, ...).
- `CHANNEL` - The channel number within that bank (e.g. `0`, `1`, ...).

In addition to these parameters, during initialization within the `SmartPsu` class, we store a host of other default parameters for the channel that will be explained later

in this section. The `Channel` object has methods to `read`, `write`, and `safe_write()` the defined channel using the tools outlined in the `Environment` object. The `read()` method simply reads the channel using the `read_voltage_current_tool` defined in the `Environment` object and returns the voltage, current and enabled status of the channel. Likewise, the `write()` method takes in a voltage and enable integer and writes to the channel using the `configure_supply_tool`. Additionally, a delay parameter is typically 0 but can be set to a non-zero value to specify a ramping time for the voltage to reach the requested value. For most use cases, we use `safe_write()` which takes an additional maximum current parameter to check after writing to the channel. When `Channel` is setup within the `SmartPsu` class, minimum and maximum allowed voltages are defined for the channel such that any `safe_write()` will validate the requested voltage is within the allowed range and raise an exception if it is not. After verifying the input, the method will write to the channel using `write()` and then read the channel using `read()`. If the current is above the maximum current, the method will raise an exception. We also check if the read voltage and enable status is within a certain tolerance of the requested voltage as well as ensuring the channel was correctly enabled, raising exceptions if not.

`SmartPsu` is initialized using an `Environment` object and a configuration file that contains the channel configurations. The configuration file can be either a CSV or INI file, CSV being the preferred format but INI files are supported for backwards compatibility with the original implementation. Regardless of the format, the configuration file specifies the following information for each channel as shown in Table **??**. Each line of the file is read and a `Channel` object is created with these attributes for usage within `SmartPsu`. When calling `safe_write()` within the `Channel` class, we use these parameters to verify the voltage and current are within the allowed

Key	Type	Description
ADDRESS	int	The address of the Smart PSU on the port.
PORT	str	The port of the Smart PSU (e.g. /dev/ttyS7).
BANK	str	The bank of the channel within that address (e.g. A, B, ...).
CHANNEL	int	The channel number within that bank (e.g. 0, 1, ...).
V_INI	V	The initial voltage of the channel.
V_NOM	V	The default voltage of the channel when enabled.
RAMP	steps	How many steps to ramp the voltage to the nominal voltage when enabled.
I_RFOFF	mA	The RF off current of the channel.
I_TOFF	%	The tolerance for the RF off current.
I_RFON	mA	The RF on current of the channel.
I_TON	%	The tolerance for the RF on current.
V_MIN	V	The minimum allowed voltage of the channel.
V_MAX	V	The maximum allowed voltage of the channel.
I_MAX	mA	The maximum allowed current of the channel.
ORDER	int	The priority order of the channel during a sequenced startup.
ENABLE	int	Whether or not the channel should be enabled during a sequenced startup or a secret third thing
DELAY	s	The delay value when writing to the channel.
SYSTEM_ID	str	The high level system name for the channel (e.g. LO_1A, RFE_1A, RBE_1CTRL, ...).
TYPE	str	The type of channel within the system (e.g. MULT, SYNTH, SPECT, ...).
SUBTYPE	str	The specific subtype of the channel within the system (e.g. REF, CM4, PMCC, ...).
DESCRIPTION	str	The full name of the channel.

Table 14. Power Supply Unit Configuration Parameters

ranges. These objects are stored in a dictionary with the key being a combination of the `ADDRESS`, `BANK` and `CHANNEL` values. `SmartPsu` has a `get_channels()` method that allows a user to specify an attribute of a channel and return a dictionary of channels that match that attribute. This is useful when wanting to get all of the channels for a specific `SYSTEM_D`, `TYPE` and/or `SUBTYPE`. Once a user collects the channels they want to modify, they can use `initialize()` to initialize the channels to their initial voltages, `read_all()` to read all of the channels, `check_current()` to check if all currents are within the allowed ranges, `change_voltage()` to change the voltage to a specific value, and `shutdown()` to shutdown all of the channels.

In addition to setting voltages, we are able to run a diode check on channels using `check_diodes()`. This method sets a range of voltages to the channels and returns the voltage and current over the interval. Optionally, when in command line mode, this method will plot the curve of the diode check. When these voltages are set, we use a special enable value of 2 to indicate that the channel is in diode check mode. In order to closely match the original drivers, we have also implemented a `check_mixers()` method which runs a check on the mixers with minor modifications. There is also a `check_amps()` method that validates the performances of the power amplifiers by gradually increasing or decreasing the drain voltage by a pinch voltage and measuring the current.

Apart from being a power supply, the Smart PSU also controls the synthesizer and some of the temperature sensors on the readout system. We utilize the Micro Lambda (MLVS) Voltage Controlled Oscillator (VCO) based synthesizer in our readout pipeline (**mlvs**). This synthesizer is built into the Smart PSU and is controlled over SPI using the `spi_readwrite_tool` provided by the manufacturer. We have wrapped the `spi_readwrite_tool` in a `SPIReadWrite` class to handle most of these additional

interactions. A general purpose `read()` method is provided to utilize the tool with the necessary parameters, such as the port and address of the Smart PSU. Higher level methods are provided for the following interactions:

- `get_ID()` - Returns the ID of the synthesizer.
- `get_freq()` - Returns the frequency of the synthesizer.
- `set_freq()` - Sets the frequency of the synthesizer between 500 and 20000 MHz.
- `get_status()` - Returns the status of the synthesizer.
- `get_temperature()` - Returns the temperature of the synthesizer.
- `get_highest_temperature()` - Returns the highest recorded temperature of the synthesizer.
- `run_self_test()` - Returns the results of a self test on the synthesizer.
- `get_ref_source()` - Returns the reference source of the synthesizer (e.g. `INT`, `EXT`).
- `set_ref_source()` - Sets the reference source of the synthesizer (e.g. `INT`, `EXT`).

For the temperature sensors, we utilize the `temperature_tool` tool provided by the manufacturer to read all of the temperature sensors for a given Smart PSU. When running the tool, we specify the resolution of the temperature sensor in bits and the number of milliseconds to wait while reading. If a resolution is not specified, it defaults to the previously set resolution. When changing resolutions, we specify a 4000 ms delay as the command takes longer to execute. The delay value for the `temperature_tool` needs to be long enough to receive values from each sensor but not too long that it causes the system to hang unnecessarily. The result from the tool is a list of temperature sensors by their ID and their associated temperature. The `Temperature` class is initialized with the `port` and `address` of a Smart PSU and uses the `Environment` object to load a configuration file. This configuration file

contains a mapping of temperature sensor IDs to unique labels for each sensor. The `read()` wraps the `temperature_tool` and returns a dictionary of the sensors keyed by their ID and matched with their associated readout. For human legibility, we also have a `get_temps()` method that maps the sensor IDs to their labels and returns a dictionary of the labels and their associated readouts.

6.7.1 Command Line Interface

Often times during testing, we need to run the Smart PSU without the overhead of having the RabbitMQ control loop running. This is especially true when we don't have the complete network setup or are testing specific components on the bench, detached from the physical payload. In previous a previous iteration of the Smart PSU control system, we had a command line interface that allowed us to issue commands to the Smart PSU and get responses back. To ensure that the scientists and engineers on the team can continue their work uninterrupted, we have implemented a one to one command line interface that allows us to run the Smart PSU using an identical control scheme as the previous iteration. This command line interface can be executed as one off commands or as an open control loop to allow for interactive testing. Table **??** shows the menu of commands a user may select when running the interactive testing loop.

6.7.2 RabbitMQ Control Loop

Unlike the driver implementation and the Command Line Interface, which were developed to function exactly like the original implementation for backwards compati-

Command	Description
`start`	Start a subsystem on the PSU.
`checkdiodes`	Run a diode check on a specific channel.
`checkamp`	Check the amplifiers on a specific channel.
`readinifile`	Read the initialization file.
`readall`	Read the voltage and current of all channels.
`checkpart_RFoff`	Check the status of a part connected to an output absed on the specified current nominal values and tolerances with RFoff.
`checkpart_RFon`	Check the status of a part connected to an output absed on the specified current nominal values and tolerances with RFon.
`stop`	Stop a subsystem on the PSU.
`changevoltage`	Safely change the voltage of the given channel.
`safesynth`	Safely turn on the synthesizer.
`forcestop`	Turn off PSU without specifying config file (best effort guess of priorities based on current output settings).
`checkmixers`	Check HEB mixers I-V reading the current from the isense ADC inputs.

Table 15. Power Supply Unit Command Line Interface

bility, the RabbitMQ Control Loop for the Smart PSU wraps the driver in a more modern interface that allows for easier integration with the rest of the system. In other implementations on the system, we have initialized our drivers with a YAML configuration file that contains all necessary parameters to setup the driver. This is not the case for `SmartPsu` as the original implementation was designed to have every parameter hard coded into the driver. For our RabbitMQ Control Loop, we pass a configuration file that we will use to override any default values in the driver when the function is called. This allows us to maintain exact functionality with the original implementation while also allowing us to modify the parameters of the driver when necessary. This configuration file has each of the potential commands that can be sent to the driver and provides configurable default values for each of the arguments.

The actual control loop is setup using `rmqtools` and is similar to the other control loops in the system but has an added complexity of multiple Smart PSUs. We load

in the configuration file and setup the `RmqConnection` object as we would for any other control loop. Within the main loop we implement a `PriorityLock` to prevent collisions when writing to the Smart PSU. This is especially important given we have multiple Smart PSUs that can be controlled at the same time, resulting in multiple threads who may potentially be trying to write to the same Smart PSU. For RPC commands, we setup two threads, one for the `SmartPsu` object and one for the `SPIReadWrite` object which primarily controls the synthesizer. The commands handled by the `SmartPsu` object are identical to the command line interface but are wrapped using `rmq.handle_command()` to allow for RPC execution. There is one additional command, `temperature`, which is used to read a temperature sensor on the Smart PSU using a `Temperature` object. The `safesynth` command is moved from `SmartPsu`'s `handle_command()` to `SPIReadWrite`'s `handle_synth_command()` as it is specific to the synthesizer and not the Smart PSU. The rest of the `SPIReadWrite` commands are accessed through the handler, providing access to the synthesizer through RPC executions.

For housekeeping, we define `psu_housekeeping()` and `psu_temperatures` methods that takes in a `SmartPsu` object and `psu_id`. We do this because we have six Smart PSUs that each need their own status threads for power and temperature readings. Using `partial` from `functools`, we're able to iterate over the Smart PSUs and create a thread for each one, wrapping the partial function with the `rmq.publish_status()` decorator with `housekeeping.psu.<psu_id>` and `housekeeping.temp.<psu_id>` as the queue names. These methods will be setup with a staggered delay to prevent all of the threads from executing at the same time when `rmq.run()` is called.

Finally, all of these decorated methods are setup with the same `PriorityLock` to prevent collisions when reading and writing to the Smart PSU. In theory, we could

use two separate locks for each of the ports but this would require more complex logic to handle the locks and would not be as clean as using a single lock. Additionally, there is little benefit to using the two locks as the cadence in which we expect to send status messages allows us to easily handle housekeeping one at a time without needing two parallel processes.

6.8 Variable Attenuator

ASTHROS utilizes a custom built five-stage Intermediate Frequency (IF) Low Noise Amplifier (LNA) in the readout chain to amplify the signal after the mixer in the cryocooler (**mathewson2024customizable**). Each one of these amplifier stages is housed with a variable attenuator to control the gain of the amplifier (**Ricardo**). Specifically, we utilize the F1956NBGI8 Attenuator from Renesas Electronics for adjusting our attenuation levels (**renesas**). As this is a standalone device on the system, it is controlled by it's own Arduino Nano Every microcontroller and interacts with the rest of system as a networked device using MQTT (Message Queuing Telemetry Transport) (**mqtt**). Each Arduino Nano Every is configured to control up to 8 variable attenuators. Each IF slice has a separate variable attenuator to control the gain of the amplifier. The attenuator is set using eight digital channels that can be used to set the attenuation level. A PCA9571 is used as an I^2C I/O expander to control the digital channels on the attenuator (**pca9571**). These expanders all have the same address of `0b0100101X` where `X` is the read/write bit. Because of this, we utilize an LTC4305 to multiplex the I^2C bus to provide each expander with a unique address (**ltc4305**). Each multiplexer is set with a unique address utilizing hardware

dip switches on the board. For the sake of simplicity, these multiplexers are given indexed addresses starting at 0 based on their position in the stack of slices.

6.8.1 Amplifier Control Library

To control the variable attenuator, we developed a C++ library, `if_amp` that can be used in our Arduino sketches. The `if_amp` library is a simple library that utilizes the `Wire` library to provide intuitive functions to control multiple variable attenuators. The most bare-bones function in `if_amp` is the `rw_pca()` function. This function takes in the address of a multiplexer, a pointer to an attenuation value, and a read/write bit. The sequence of commands to write to the expander is as follows:

1. Start a transmission to the multiplexer's address.
2. Write a value to the multiplexer indicating to act as a repeater for the next transmission.
3. End transmission to the multiplexer.
4. If we are writing to the expander,
    a) Start a transmission to the expander's address.
    b) Write the value at the pointer to the expander.
    c) End transmission to the expander.
5. If we are reading from the expander,
    a) Send a request for data to the expander.
    b) Read the data from the expander.
    c) Save the data to the pointer.
    d) End transmission to the expander.

6. Start a transmission to the multiplexer's address.
7. Write a value to the multiplexer indicating to stop repeating transmissions.
8. End transmission to the multiplexer.

At multiple points during this sequence, end transmission may produce an error. If that is the case, we return the status of the error. If the sequence completes successfully, we return 0. These errors are handled by the user-facing functions in the `if_amp` library.

To build off of the `rw_pca` function, we have implemented a few helper functions that can be used in the final, user-facing functions. `convert_addr()` and `convert_attn()` are used to convert the addresses and attenuation values to the correct format for the `rw_pca()` function. `convert_addr()` takes in the index for an IF slice and converts it to the address for the multiplexer by formatting it as `0b11XXXXX` where `XXXXX` is the index in binary. `convert_attn()` takes in an attenuation value between 0 and 31.75 and converts it to a value between 0 and 255. The attenuator can only be set in 0.25 dB steps so we multiply the input by 4 and round to the nearest integer. Conversely, `convert_attn_back()` is a function that takes in the value from the expander and converts it back to the attenuation value by dividing the value by 4. Finally, `check_addr()` and `check_attn()` are helper functions to verify a given address or attenuation value is valid.

The user-facing functions in the `if_amp` library are `set_attn()` and `get_attn()` functions. In both cases, the function takes in the index of the IF slice and either the attenuation value to set or a pointer to store the read attenuation value. The address and, if necessary, the attenuation value are checked and converted to the correct format. Both functions then call the `rw_pca()` function to write or read the value from the expander, specifying the read/write bit as necessary. Finally, both

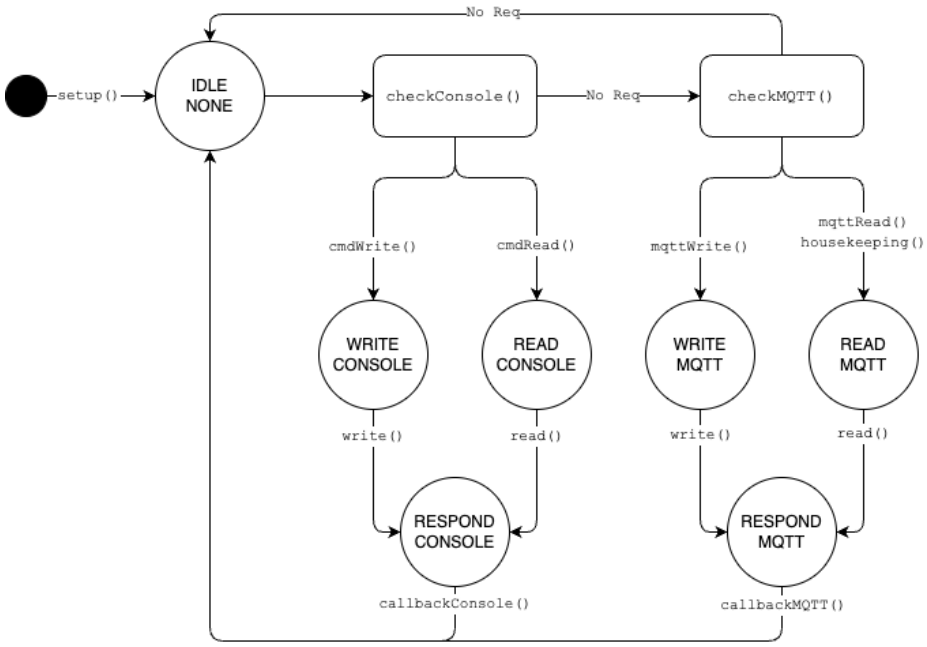


Figure 24. Finite State Machine for the IF Amplifier Control Loop. Each state shows the current step in the process followed by the system that requested the action.

functions return the status of the function call, which is nonzero for any error that may occur from invalid input or a failed transmission. A listing of all the possible errors can be found in Appendix **??**.

6.8.2 Arduino Control Loop

For the Arduino control loop, we have implemented multiple ways to interact with the device using the `if_amp` library. For flight, we will be using the MQTT protocol to send messages to the Arduino to set the attenuation values and read the current attenuation values. For testing, we have implemented both a simple serial interface. A web interface was also developed to control the attenuators using a web browser but

was later removed to save on memory. Including the web interface adds additional static memory usage to the Arduino Nano Every in the form of large HTML strings.

The structure for the Arduino control loop is a finite state machine that keeps track of the current request and which system requested it. Figure **??** shows the state diagram for the finite state machine.

The system starts with `setup()` which initializes all of the hardware interfaces and sets our initial state to `IDLE`. Within `setup()`, we start our serial interface and initialize the I^2C bus. Because of a hardware issue on the Printed Circuit Board (PCB), the `Wire` library needs to be patched to include timeouts on the I^2C bus. This is because of a hardware issue where the Arduino has power but the rest of the system is off. In this scenario, there is a floating voltage on the I^2C bus that causes the `Wire` library to hang indefinitely. The patched `Wire` library has a new `setTimoout()` method that sets a timeout for the I^2C bus to prevent this from happening.

After setting up serial and I^2C, the system sets up our Ethernet and MQTT connections. For Ethernet, we are using the W5500 Ethernet shield from Wiznet (**w5500**). This shield is connected to the Arduino using SPI and enables us to connect the Arduino to the rest of the readout network. To setup the networking, we give the Arduino a unique MAC address and a static IP address. On ASTHROS, we have two of these systems so we provide them with addresses based on their hard coded ID. In the case that the Arduino is not connected to the network, the system will continue to run but will only accept messages from the serial interface. After setting up the Ethernet connection, we setup a client to connect to the MQTT broker. This client takes the IP address of the broker and the port to connect to and attaches a callback function to handle incoming messages. We then attempt a connection to the broker through the client using the name of our device and an account on our

RabbitMQ server setup for handling MQTT requests. Once connected, we subscribe to the `mqtt/comm/<ID>` topic where `<ID>` is the ID of the Arduino. While AMQP and MQTT are different protocols, RabbitMQ has a plugin to handle MQTT messages and route them to the correct AMQP exchange. By default messages published via MQTT are sent to the `amq.topic` exchange with the MQTT topic as the routing key, replacing the forward slashes with periods. At the end of `setup()`, we initialize all of the attenuators to 31.75 dB and set the state to `IDLE`.

The `loop()` function is the main function that runs the finite state machine. On top of the state and system variables, we have array buffers for `addr`, `attn`, and `status` to store the address, attenuation value, and status of the current function call. There is also a flag for `single` that is used to indicate if the current call is for a single channel or all channels.

In the `IDLE` state, we begin by checking if there is a message available on the serial interface. Using the `StaticSerialCommands` library, we can parse messages and call the appropriate callback function. Available commands are `w <addr> <attn>`, `wa <attn>`, `r <addr>`, and `ra` for writing and reading a single channel or all channels. These commands do not actually execute the function but instead set the state to `WRITE` or `READ` with the appropriate address, attenuation value, and single flag. We also set the system to `CONSOLE` to indicate that the serial interface requested the action. In the case of a single channel operation, the first index of `addr` and `attn` are used. If the command is not recognized, an error message is sent back to the serial interface and we stay in the `IDLE` state.

If there are no commands waiting in the buffer and we are connected to the MQTT broker, we check for incoming messages. If there is a message, we parse the message and set the state to `WRITE` or `READ` with the appropriate address, attenuation value,

and single flag. Messages from the MQTT broker are JSON strings with a `cmd` key that specifies the command and `addr` and `attn` keys that specify the address and attenuation value. Additionally, we record the `routing_ID` of the message in order to respond directly to the sender. MQTT does not support direct addressing so we workaround this by sending a message to the topic exchange with the `routing_ID` as the route. Whomever sends the request will need to subscribe to the `mqtt/comm/<routing_ID>` topic to receive the response.

In the `READ` state, we call the `read()` command which begins a read operation on the Arduino. `read()` iterates over either all of the channels or just the first index and calls `single_read()`. The `single_read()` function calls the `set_attn()` function from the `if_amp` library with the address and a pointer to the attenuation value. We then store the status of the `set_attn()` call in the corresponding index of `status`. There is an error offset parameter that is used to add an offset to the status in case of errors. This is used later in `single_write()` to differentiate between an error in writing and reading back the value. After reading all of the specified channels, we set the state to `RESPOND`.

In the `WRITE` state, we call the `write()` command which begins a write operation on the Arduino. `write()` iterates over either all of the channels or just the first index and calls `single_write()`. `single_write()` calls the `set_attn()` function from the `if_amp` library with the address and the attenuation value and saves the status. If the status is nonzero, we set the corresponding index in the attenuation buffer to -1 to indicate an error. If not, we perform a `single_read()` with an offset of 100 to indicate any errors in reading back the value came from the read operation.

After either a read or write operation, the system ends up in the `RESPOND` state. In this state, we check the system variable to see who requested the operation. If

the system is `CONSOLE`, we print out the status, attenuation, and address of either the single channel or all channels. If the system is `MQTT`, we package the status, attenuation, and address into a JSON string and publish to the MQTT broker with the `mqtt/comm/<routing_ID>` topic. In either case, after sending the response, we clear the three buffers and set our state back to `IDLE`.

The firmware also has a few extra features to help configure the system. The console also accepts an `mqtt` command that enables or disables the MQTT interface. This is useful as, if the system is not connected to the network, the MQTT interface will continually try to connect and fail. At a predefined interval, the system will publish a message to the `mqtt/comm/<ID>` topic with the status of the system. This is integrated into our state machine while checking for MQTT message by comparing the last time we published a status message with the current clock time and setting the state to `READ` and system to `MQTT` if the time has elapsed. This read will read all channels and have an additional flag, `mqttStatus` that is checked in the `RESPOND` state to publish the status message to the MQTT broker with the `mqtt/comm/<ID>` topic instead of the routing ID.

There are also some quirks to the system that we attempt to mitigate in the firmware. In addition to the I^2C bus issue, there is a hardware issue where the Ethernet shield will not initialize properly if the IF slices are powered on before the Arduino. In this case, we have a `pulseReset()` function that pulses the reset pin on the Ethernet shield to reset the shield and attempt to initialize the connection. This occurs because power will leak through the SPI bus and partially power the shield. When the system is fully on, the shield has an incorrect reference voltage and will not initialize properly during `setup()`.

6.9 Housekeeping

In our initial implementation of housekeeping, we intended to conform to the structure necessary for downstream processing using GSEOS (Ground Support Equipment Operating System) (**hauck2003use**). GSEOS is the ground station software previously used by the ASTHROS team to digest and visualize telemetry data from the gondola. Due to changes on the gondola's software side, the housekeeping system will need to be modified to conform a different framework. For now, this section will outline the current implementation of the housekeeping system as the connections to the rest of our readout system are still valid. Steps for implementing the new system will be outlined in a subsection below.

At it's core, the housekeeping system is the culmination of every other system in the readout pipeline. Every system previously defined produces some sort of status message that needs to be captured, analyzed, and forwarded to the operations team. Thanks to all of the groundwork done by `rmqtools`, we are able to very easily implement a system that can receive messages from every other system on the network and publish them to the GSEOS.

For the readout system, our route to sending data to GSEOS is a socket open on one of the gondola computers that expects our data in a very specific format such that it can packetize the data and downlink it to the ground station. Once at the ground station, we are able to take the formatted data and monitor it using the GSEOS software. Because we have a single pipeline for all of our data, we need to segment and packetize the data into different blocks such that we do not exceed our downlink bandwidth. To packetize the data properly, we have implemented a `Housekeeping`

library that handles harmonizing data from various subsystems, formatting it for GSEOS, and sending it to the socket at regular intervals.

At its core, the `Housekeeping` library is a set of data queues in a fixed sequence, where each data queue is checked in a predefined order on a set interval. A good analogy for this is a clock, where each position on the clock is a data queue. When the clock ticks and points to a new position, the data queue at that position is checked for new data. If there is new data, it is formatted and sent to the socket. We call these data queues `blocks`, with each block being unique to the type of data it is processing.

Regardless of the data that is being processed, each block has the following characteristics and behaviors. Blocks are setup with a `msg_id` signifying the type of data being processed. This is used on GSEOS to identify where the data needs to be loaded in the system after it is received. Following that is `route` which is a list of all of the RabbitMQ routing keys that this block needs to subscribe to. Next we have a `data` class which is a custom `QueueTTL` object that acts as a dictionary with a time to live (TTL) for each key. This allows us to consolidate data from multiple sources into a single block without having to worry about sending stale data if one of the sources goes down. Finally, we have a `data_queue` object which is simply a Queue that contains encoded data that is ready to be sent to the socket.

Each block then has a set of methods that are used to handle acquiring and processing the data. First, the block has a `subscriber()` method that is structured to be decorated with `rmqtools` to handle the incoming messages. As each subsystem may need a different method to handle incoming messages, each block has their own `subscriber()` method unique to the block's datatype. Within this method, we retrieve the data, place it in the `data` object, and then call the `encode()` method to encode the data for GSEOS before placing it in the `data_queue`. The `encode()`

method is highly dependent on the type of data being processed and is unique to each subsystem, which we will further discuss in subsections for each block. At the start of each encoded message is a stamp from `get_stamp()`, a utility method that creates a header for the data with a timestamp and the `msg_id` of the block. The end result of the `encode()` method is a byte array that contains the predefined encoding format for GSEOS. Finally, we have a `simulate_data()` method that is used to simulate data for testing purposes. This is a counterpart to the `subscriber()` method as instead of receiving data from RabbitMQ, we generate random data, place it in our `data` object, and call the `encode()` method to encode the data for GSEOS.

These blocks handle the logic behind acquiring and encoding data but need to be implemented in the main housekeeping loop. Before we get to the main loop where our sequenced clock sends data at regular intervals, we need to setup the blocks to receive data. This is done by wrapping the `block.subscriber()` method with `subscribe_status` and passing the `block.route` list to the decorator. We give these queues the unique name `housekeeping_<seq_id>` where `<seq_id>` is the index of the block in the sequence. When looking at the RabbitMQ console, we are able to easily see which queues are receiving data and which ones are not. Once all of the blocks are attached to the RabbitMQ server, we can start the listeners and begin the housekeeping loop. Each listener is on a separate thread and will asynchronously process messages as they come in. When the housekeeping loop gets to the block's index in the sequence, it will check the `data_queue` for any data that is ready to be sent. In order to get the most recent data, the queue is read until it is empty and the data is sent to the socket. If there is no data in the queue by the time the clock is ready to move, we skip the block and the loop continues, checking the next block in the sequence for data. If there is data, we encode it along with the status of the system and send it to the socket. Afterwards, we sleep until the next tick of the clock and repeat the process. In practice, this looks like the following sequence, where the clock has a 12 second cycle and each block is checked every second:

Timing	Message ID	Block
1	4	PSU Block for Smart PSU 4
2	7	PSU Block for Smart PSU 7
3	5	PSU Block for Smart PSU 5
4	8	PSU Block for Smart PSU 8
5	6	PSU Block for Smart PSU 6
6	10	PSU Block for Smart PSU 10
8	2	Antenna Block
9	3	Motor Block
10	1	Temperature Block
11	11	Cryocooler Block
12	12	Intermediate Frequency Block

Table 16. Housekeeping Sequence

6.9.1 Antenna Block

The antenna system is fairly straightforward for implementation. There is a single `housekeeping.antenna` routing key to subscribe to and the data is a simple dictionary packed with various readout values from the antenna. Below is a list of the values that are packed into the data dictionary.

- `M1_HW_VER` - The hardware version of the M1 antenna system.
- `M1_SW_VER` - The software version of the M1 antenna system.
- `M2_HW_VER` - The hardware version of the M2 antenna system.
- `M2_SW_VER` - The software version of the M2 antenna system.
- `M1_TEMP` - The temperatures of M1's 9 Replicas and their average from -327.68 to 327.67 C.
- `M2_TEMP` - The temperatures of M2's 3 Legs, the Leg Average, and the M2 Mirror and Board from -327.68 to 327.67 C.
- `M2_TEMP_THRESH` - The lower and upper limits for the thermostat on the M2 from -327.68 to 327.67 C.

- `GPS_POS` - The Latitude, Longitude, and Altitude of the GPS system from -90 to 90 deg , -180 to 180 deg, and 0 to 428496.7296 m respectively.
- `ELEVATION_MEAS` - The elevation of the antenna from -90 to 90 deg.
- `HEXA_POS_SET` - The current desired position and rotation of the hexapod from -214,748.3648 to 214,748.3647 mm and -214,748.3648 to 214,748.3647 deg respectively.
- `HEXA_POS_CORR` - The correction position and rotation of the hexapod from -214,748.3648 to 214,748.3647 mm and -214,748.3648 to 214,748.3647 deg respectively.
- `HEXA_POS_REAL` - The actual position and rotation of the hexapod from -214,748.3648 to 214,748.3647 mm and -214,748.3648 to 214,748.3647 deg respectively.
- `FLAGS` - Status enabled flags from the M1 and M2 Heaters, the M2 thermostat, and the Hexapod correction.
- `ALARMS` - Status alarms indicating issues communicating with the M2 or Hexapod controllers, the legs being too cold from the M2 thermostat, or a bad elevation reading.

6.9.2 Cryocooler Block

The cryocooler block is similar to the antenna block in that it is a simple dictionary of values from the cryocooler system. Unlike the antenna block, the cryocooler block has three routes to listen to, `housekeeping.cryo`, `housekeeping.lakeshore.cal`, and `housekeeping.lakeshore.temp`. Within the `subscriber()` method, we check the routing key of the incoming message and handled each one accordingly. Because

our `QueueTTL` object handles the TTL of each key, we simply `put()` the data into the `data` object and the logic for handling stale data is handled when we call `get()`.

As for the data, the cryocooler provides these values as bytes so all we need to do is ensure the data is in the right order and then pass the bytes to the `data_queue`. This results in minimal processing for the data from `housekeeping.cryo` but we will still need to process the data from `housekeeping.lakeshore.cal` and `housekeeping.lakeshore.temp` as they are not in the correct format for GSEOS. Below is the complete list of values that are packed into the data dictionary for the cryocooler block.

- `status` - The current status byte of the cryocooler.
- `caseTemp` - The temperature of the case.
- `reserve` - Reserved value (typically not used).
- `piOut` - Voltage output of the PI Loop.
- `busVoltPeak` - Bus voltage at the motor voltage peak.
- `controlTemp` - The temperature of the control sensor.
- `auxTemp` - The temperature of the auxiliary sensor.
- `busCurr` - The current of the bus.
- `busVoltAvg` - The average voltage of the bus.
- `opMode` - The current operating mode.
- `compFreq` - The frequency of the compressor motor.
- `tempSet` - The desired temperature for Cold Head 1.
- `motorDrive` - The peak output of the voltage drive.
- `intGain` - The integral factor of the PI temperature and voltage controller.
- `propGain` - The proportional factor of the PI temperature and voltage controller.
- `peakDrive` - The voltage limit on the peak output of the bus.

- `motorPeakPos` - The peak positive voltage of the motor.
- `motorPeakNeg` - The peak negative voltage of the motor.
- `avcPeakPos` - The peak positive voltage of the AVC (Adaptive Vibration Control).
- `avcPeakNeg` - The peak negative voltage of the AVC.
- `voltRamp` - The maximum voltage ramp rate.
- `lsCal` - The calibration temperatures (2) of the Lakeshore, encoded as data from 0 to 65535 to be later converted to a float
- `lsCryo` - The cryocooler temperatures (8) of the Lakeshore from 0 to 65,535.99 K.

6.9.3 Intermediate Frequency Block

The IF block is one of the more simple blocks in the system. Unlike the other systems, the IF systems does not use the traditional `status` exchange as it operates using the MQTT protocol. Our RabbitMQ server has a plugin to handle MQTT messages and route them to the `amq.topic` exchange. As such, we have a separate `RmqConnection` object whose status exchange is set to `amq.topic` and use this object to wrap the block's `subscriber()` method. For routing, we subscribe to `mqtt.status.0`, `mqtt.status.1` and `mqtt.status.2` for the three IF slices. Each slice has up to 8 channels, except for slice 0 which only has 1 for the 100GHz receiver. This is a total of 17 channels that we need to monitor. The data from the IF block is a simple dictionary of attenuation values in dB as well as the status flag from each channel. We pack this data into a byte array, consisting of all of the attenuation values followed by the status flags.

6.9.4 Motor Block

The motor block is another relatively simple system. The motor only has one route, `housekeeping.motor`, which receives the mirror position from the motor controller. The following values are packed into the byte array for the motor block.

- `HC` - Hold Current as a percentage of the maximum current[3]
- `RC`- current as a percentage of the maximum current[??]
- `A` - The acceleration of the motor in steps per second squared[??].
- `D` - The deceleration in steps per second squared[??].
- `VM` - The maximum velocity of the motor in steps per second[??].
- `VI` - The initial velocity in steps per second[??].
- `P` - The current position of the motor in steps.
- `CB` - The current control bounds, a mode selection for torque performance[??].
- `ER` - The error code from the motor controller.

6.9.5 Smart PSU Block

The Smart PSU block is one of the more densely packed blocks in the system. Because we have so many Smart PSUs, we actually have multiple `PsuBlock` objects in the sequencer so that we can have separate threads for each PSU and spread out the bandwidth to the GSEOS socket. The route for each of these blocks is variable and set to `housekeeping.psu.<psu_id>` where `<psu_id>` is the ID of the PSU, which is also the `msg_id` for the block. The actual data from the Smart PSUs are the voltages,

3. This value is a user set value and should not change during operations. We downlink this value in case it does change due to an error in the system.

currents, and enabled status from each channel as well as some overhead data for the PSU itself. The data is packed into a byte array with the following values:

- `A_Voltage` - The voltage from bank A, channels 0 to 7, from -32.76 to 32.76 V.
- `B_Voltage` - The voltage from bank B, channels 0 to 3, from -32.76 to 32.76 V.
- `C_Voltage` - The voltage from bank C, channels 0 to 4, from -32.76 to 32.76 V.
- `D_Voltage` - The voltage from bank D, channels 0 to 3, from -32.76 to 32.76 V.
- `E_Voltage` - The voltage from bank E, channels 0 to 7, from -32.76 to 32.767 V.
- `S_Voltage` - The voltage from bank S (the synthesizer), channels 0 and 1, from -32.76 to 32.76 V.
- `A_Current` - The current from bank A, channels 0 to 7, from -99.99 to 99.99 A.
- `B_Current` - The current from bank B, channels 0 to 3, from -99.99 to 99.99 A.
- `C_Current` - The current from bank C, channels 0 to 4, from -99.99 to 99.99 A.
- `D_Current` - The current from bank D, channels 0 to 3, from -99.99 to 99.99 A. This value is actually read from the voltage of the X bank divided by a scaling factor.
- `E_Current` - The current from bank E, channels 0 to 7, from -99.99 to 99.99 A.
- `S_Current` - The current from bank S (the synthesizer), channels 0 and 1, from -99.99 to 99.99 A.
- `Enabled` - The status of each channel as a big endian bit array with each bit representing the status of a channel for banks A, B, C, D, E, and S.
- `Synth_ID` - The ID of the synthesizer, should be "MLVS-1520AS" and is downlinked for verification we have an SPI connection.
- `Synth_Freq` - The frequency of the synthesizer in MHz, from 0 to 99999.99 MHz.
- `Synth_Temp` - The internal temperature of the synthesizer in C, from -327.68 to 327.67 C.

- `Synth_Max_Temp` - The maximum temperature of the synthesizer in C, from -327.68 to 327.67 C.
- `Synth_Status` - The status code from the system, a bit array with each bit representing a different status flag. This value is actually 9 bits long so we zero pad the value to 16 bits.
- `Error_Codes` - 4 bit error codes for each channel indicating: OK, Voltage Low, Voltage High, Current Low, Current High, Out of Tolerance, and Channel enabled/disabled when expected to be disabled/enabled.

6.9.6 Temperature Block

The temperature block also interacts with the Smart PSU system but is focused on the temperature sensors attached to each PSU. We don't need to split the temperature block into multiple blocks as we're able to fit all of our combined data into a single block. For routing, we have 6 routing keys for the 6 power supplies on the gondola, `housekeeping.temp.<psu_id>` where `<psu_id>` is the ID of the PSU. Apart from that, the rest of the block is very simple. We load in a configuration file that contains a mapping of every temperature sensor's ID and a description of the sensor's location. We use the ordering of this configuration file to determine the order of the sensors in the data array. Other than that, we simply iterate over the sensors in the configuration file and use the ID as a key in the `data` object to order and encode each value from -327.68 to 327.67 C.

6.10 Central Command

The core of the entire readout network is the central command system. In developing the central command system, we had two goals in mind: develop something that was easily adaptable to whatever subsystems we needed to control and develop something that whose interface would be identical in both testing and in flight. This led us to looking towards `click`, a Python library for building command line applications (**click**). Click enabled us to create a custom command line interface (CLI) that consolidated all of our systems into a single interface. Additionally, with `click_repl`, we were able to create a read-eval-print loop (REPL) that would act as a terminal interface to the system, allowing us to chain commands together and interact with the system in a more natural way.

To accomplish the first of these two goals, we utilize click's ability to add groups and subcommands to the CLI. We call these `plugins` and simply developed a plugin for each subsystem that we needed to control. Separating each of the subsystems into their own plugins allow us to easily develop new plugins for new subsystems without having to modify the core of the system. At the core of each plugin is a unique `Controller`, which handles methods necessary for interacting with the subsystem in the plugin. Usually, this interaction with the subsystem is done through a RabbitMQ connection, sending RPC executions to the subsystem and waiting for a response. Normally, each command sent is atomic, where each command's execution exists in a vacuum, but we can work around this by storing a persistent `Controller` in within `click.get_current_context()`. This allows us to store the state of the system between commands, create global variables that are accessible to all commands, and,

most importantly, only need to create a single instance of the `Controller` for each plugin.

In terminal mode, we are able to accomplish our second goal of having a single interface for both testing and flight. Terminal mode is a special mode of the CLI that uses a REPL to allow the user to interact with the system in a more natural way, without having to type out `python central_command.py -config config.yml ...` every time. Within terminal mode, the first thing you type is the name of the plugin you want to interact with, which loads the plugin and gives you access to all of the commands in that plugin. It also has tab completion for all of the commands in the plugin, so you can easily see what commands are available. There is additionally a history feature that allows you to scroll through the previous commands you have entered, and use reverse search to find commands you have entered in the past. We primarily use this terminal interface for testing, as it allows us to easily interact with the system in an isolated environment. To match the flight system, we have implemented a version of Central Command that listens to a socket and receives strings from the ACE computer on the gondola. The string is then parsed exactly as it would be in the terminal interface, and the command is executed as if it were entered in the terminal, making it identical to the REPL system when in flight. The biggest challenge here is capturing the output of the command and sending it back to the ACE computer. This is done by exclusively using the `click.echo()` method to print to the terminal in REPL mode and then overriding the behavior of `click.echo()` to send the output to the socket instead of printing it to the terminal.

Table **??** shows a list of the plugins that are currently implemented or planned for the system. Two plugins do not control a specific subsystem but are instead used to provide broad functionality to ASTHROS. `seq` is a special scripting plugin that allows

Plugin	Description
`antenna`	Controls the Antenna Systems.
`cryo`	Controls the Cryocooler Systems.
`fabric`	An implementation of the `fabric` allowing us to execute commands on any computer on the network in case of emergency.
`ifsys`	Controls the Intermediate Frequency Systems.
`motor`	Controls the Mirror's Motor Systems.
`pressure`	Controls the Pressure Systems.
`psu`	Controls the Smart PSU Systems.
`seq`	A special system that allows us to execute commands in a sequence, similar to a script.
`spec`	Controls the Spectrometer Systems.
`synth`	Controls the Synthesizer Systems.

Table 17. Plugins for the Central Command System

us to write a sequence of commands to be executed in order as a text file. When the file is executed, it is parsed and each command is executed in order. This makes running a sequence of commands, such as starting up the spectrometer, or running a test sequence, much easier. For flight, we will develop sequences for most operations that need to be performed as it allows us to reduce our required bandwidth to the ground station when issuing multiple commands. Likewise, the `fabric` plugin is a special plugin that allows us to execute commands on any computer on the network. `fabric` is a Python library that allows us to execute commands on remote computers over SSH. While we would hope that our Central Command system is able to handle all required commands, there may be times when we need to execute a raw command on a remote computer. This is especially useful in flight, when SSHing into a computer may not be possible and we need a backup plan.

For the spectrometer system, we had to handle multiple instances of subsystems responding to the same command. This occurs as we have multiple CM4s on the gondola, each with their own spectrometer system. To handle this, we place a `PriorityLock` on the response handler for the command so that only one command is

executed at a time. Each command updates a response dictionary with the response from the command, which harmonizes the responses into a single dictionary. This allows us the end user to transparently communicate with multiple spectrometer systems at once without having to worry about the underlying implementation.

Chapter 7

CONCLUSIONS

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices biben- dum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

NOTES

REFERENCES

APPENDIX A

ASTHROS NETWORK ADDRESSES

Lorem those ipsums

APPENDIX B

AMPLIFIER CHAIN ERROR HANDLING

Lorem those ipsums