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 2024 by the
Graduate Supervisory Committee:

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

ARIZONA STATE UNIVERSITY

August 2024

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

Here is a dummy dedication. The dedication can be vertically centered like this text is. See the LaTeX code for this dedication to see how to vertically center the text of your dedication.

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

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 module 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.

1.1 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's buffer size and enable SPI on boot. 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 for the PMCC. 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. After rebooting and verifying the SPI has been setup, we recommend connecting the CM4 to the internet through a wireless hotspot so we can install the necessary packages. The first package we need to install is the pigpio library. This is a library that allows us to control the GPIO pins on the CM4. 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 module.

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 **??**.

1.2 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 module 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.

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

$$\mathrm{tx}[16] = \mathrm{addr}[9] << 1 + \mathrm{rw}[1] \tag{1.1}$$

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

$$\mathrm{byte}[8][2] = [\mathrm{word}[16] >> 8,\ \mathrm{word}[16]\ \&\ \mathrm{0xFF}] \tag{1.2}$$

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. 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 **??**.

$$\text{maxValue} = (1 << (\text{MSB} - \text{LSB} + 1)) - 1 \qquad 0 \leq \text{LSB} \leq \text{MSB} \leq 15 \quad (1.3)$$

$$\text{mask} = \text{maxValue} << \text{LSB} \quad (1.4)$$

$$\text{buffer} = (\text{buffer} \;\&\; \sim \text{mask}) \mid (\text{data} << \text{LSB}) \qquad 0 \leq \text{data} \leq \text{maxValue} \quad (1.5)$$

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.

1.2.2 `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`	string	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 1. Global Variables 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 ??.

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 2. Wiring Configuration in the PMCC Configuration File

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 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 concate-

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

Table 3. Resolution Mode Configuration for PMCC

nated 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.

1.2.3 `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.

1.2.4 `driver`

The `driver` module is the highest level of the PyMCC module 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 setup a `PMCC_SPI` object to communicate with the PMCC. Following SPI setup, the user must call the `initalize_gpio()` method to setup 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 agruments to this method but are set to 500 Hz and 2000 retries by default. If we don't recieve 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 setup 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.

$$\text{Swing} = \sigma * 2\sqrt{2} \tag{1.6}$$

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.

1.2.5 `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 paramater 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.

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 4. Attributes of the `data` dataset in the HDF5 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 `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.

Chapter 2

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

2.1 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

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.

2.2 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 figs/3/scan.png

Figure 1. 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**.

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.

2.3 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 figs/3/system.png

Figure 2. 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.

on the same computer, control of each of these subsystems is handled by separate processes that communicate with the Central Central process through RMQ. This allows any 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**.

2.4 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.

figs/3/Anom1.png figs/3/anom2.png figs/3/anom3.png

Figure 3. 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.

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

2.4.1 Variational AutoEncoders

VAEs are a class of AutoEncoders that learn the mean and variance of the latent space of the data **kingma2022autoencodingvariationalbayes**. 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.

2.4.2 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.

2.5 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

`figs/3/HASP_Elbow.png` `figs/3/elbow.png`

Figure 4. 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.

figs/3/hasp1.png

figs/3/hasp2.png

Figure 5. 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.

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 baseline for what we expect to see during flight. Despite being 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 of the scope of this work but will be discussed in Section **??**.

2.6 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 the anomaly score metric tracks with manual monitoring of the time series of spectra

`figs/3/asthros_timeseries.png`

Figure 6. 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.

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 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 figs/3/pca1.png

figs/3/pca2.png

Figure 7. 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.

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 the their predecessors. Of the tested methods, PCA has shown the most promise for implementation into the ASTHROS Readout System.

2.7 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 3

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

3.1 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.

3.2 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 figs/5/Examples_fixed.png

Figure 8. 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 **??**.

3.3 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.

Step 1: Locating Anomalies	**Step 2: Classifying Anomalies**
Statistical Thresholding	Heuristic Rules**cillis2018snowballs**
Principal Component Analysis (PCA)**wold1987principal**	Convolutional Neural Network (CNN)**g**

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 determines 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 preprocessed. For more information about this and the dataset selection, see Section **??**.

3.3.1 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.

3.3.1.1 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.

$$\Delta x_i > \max(\mu_i + 50\sigma_i, 5000) \tag{3.1}$$

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.

3.3.1.2 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 **??**.

With the reconstructed values, we are able to calculate the residual ($Reconstructed_image - Original_image$) 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**.

3.3.2 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.

3.3.2.1 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.

3.3.2.2 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.

3.3.3 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.

3.4 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 **2016jdox.rept...... 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 **??**.

3.5 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.

3.5.1 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 to the curve as shown in 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.

3.5.2 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.

3.5.3 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.

3.6 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.

`figs/5/Threshold.png`

Figure 9. 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.

figs/5/PCA_Good.png

figs/5/PCA_Bad.png

Figure 10. 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.

figs/5/elbow.png

Figure 11. 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.

figs/5/heuristic_confusion.png

Figure 12. Confusion Matrix for the Heuristic Rules Method with Located Data from Statistical Thresholding.

NOTES

REFERENCES

APPENDIX A

THIS IS A CHAPTER-LEVEL HEADING FOR AN APPENDIX

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 bibendum. 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.

A.1 This is the first section-level heading in the appendix

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 bibendum. 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.

A.1.1 This is the first sub-section-level heading in the appendix

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 bibendum. 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.

A.1.1.1 This is the first sub-sub-section-level heading in the appendix

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 bibendum. 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.

A.1.1.1.1 This is the first paragraph-level heading in the appendix

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 bibendum. 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.

A.1.1.1.1.1 This is the first sub-paragraph-level heading in the appendix

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 bibendum. 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.