EXPERIMENTAL EVIDENCE OF THE PFOTZER MAXIMUM THROUGH AIRBORNE INSTRUMENTATION AND DATA COLLECTION

by

McKay Murphy

A senior thesis submitted to the faculty of

Brigham Young University - Idaho

in partial fulfillment of the requirements for the degree of

Bachelor of Science

Department of Physics

Brigham Young University - Idaho

April 2020

BRIGHAM YOUNG UNIVERSITY - IDAHO

DEPARTMENT APPROVAL

of a senior thesis submitted by

McKay Murphy

This thesis has been reviewed by the research committee, senior thesis coordinator, and department chair and has been found to be satisfactory.

Date	Ryan Nielson, Advisor
Date	David Oliphant, Senior Thesis Coordinator
Date	Ryan Nielson, Committee Member
Date	Kevin Kelley, Committee Member
Date	Todd Lines, Chair

ABSTRACT

EXPERIMENTAL EVIDENCE OF THE PFOTZER MAXIMUM THROUGH AIRBORNE INSTRUMENTATION AND DATA COLLECTION

McKay Murphy

Department of Physics

Bachelor of Science

Cosmic rays provide beneficial information on stellar events while also posing health and safety risks to the human race. To better understand and identify these particles, two different types of radiation detectors were constructed and tested. The first using a Geiger counter and the second using a scintillator photomultiplier. It was found that the Geiger counter device achieved better results in finding the Pfotzer maximum than the Muon detector because of the overwhelming sensitivity of the Muon detector.

ACKNOWLEDGMENTS

I would like to thank all the members and faculty working with the High Altitude Research Team and my work at BYU-Idaho. There are so many to list. My research would've not been possible without them. I would also like to thank my wife Alma who kept me sane and level during times of tumult and stress.

Contents

List of Figures

Chapter 1

Introduction

1.1 A Brief History of Cosmic Rays

The history of cosmic ray detection begins with the discovery of radioactivity. Henri Becquerel first discovered spontaneous radioactivity by placing a photographic plate next to uranium salt. The resultant exposure created a foggy pattern on the plate, which Becquerel hypothesized as being some form of waves. Becquerel had shown not only the first evidence of radioactivity but also its interactions with matter. Other famous scientists such as Marie and Pierre Curie continued Becquerel's work and discovered varying types of radioactivity.

Later, Julius Elster and Hans Geitel discovered a yet inexplainable phenomenon, showing that isolated and still air could still be spontaneously ionized. The source of this ionizing radiation was hypothesized to be terrestrial, i.e. sourced from the Earth and its decaying radioactive substances. If this was true, then it follows that increasing the distance from the Earth (say in a balloon) should therefore decrease the amount of ionizing radiation. Victor F. Hess, an Australian-American physicist, began testing this hypothesis from 1911 - 1912. After several balloon flights, he

Figure 1.1 Victor F. Hess embarking on a balloon-bourne experiment. Image from CERN

had demonstrated that ionizing radiation appeared to increase rapidly with altitude, contrary to what was expected. Thus, Hess concluded that these sources came from beyond the Earth. Cosmic rays had been discovered, opening a vast new world of scientific research. [9]

Georg Pfotzer extended these measurements by discovering the first variance in cosmic rays in the atmosphere. Pfotzer followed Hess' method of experimentation by sending balloons into the upper stratosphere. He discovered that, contrary to what was believed, the density of cosmic rays eventually plateaued around a specific altitude. This phenomenon later became known as the Pfotzer Maximum. We know it today as the altitude at which cosmic rays begin to interact less with the atmosphere.

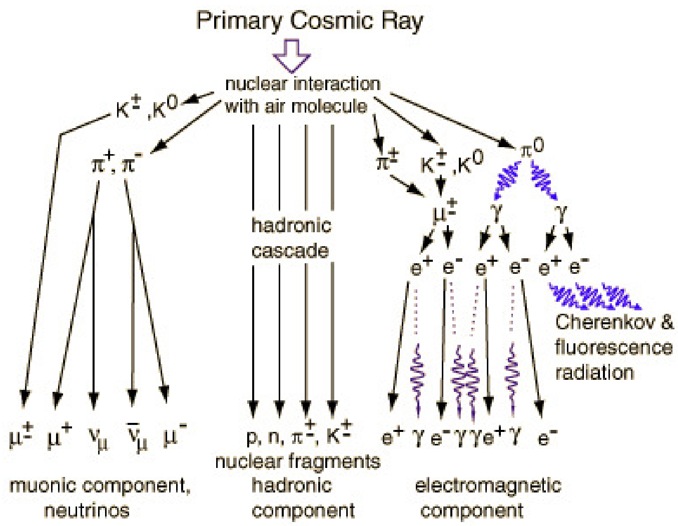

Figure 1.2 A decay chain of a proton interacting with the atmosphere.

1.2 Cosmic Rays Today

"Cosmic rays" is a broad term describing many types of particles. In a general sense, any relativistic particle entering the Earth's atmosphere is considered a cosmic ray. The majority of these particles are protons, with a small amount consisting of helium nuclei, and very slight traces of heavier elements or electrons. Slower cosmic rays often are deflected by the Earth's magnetic field, later being absorbed by the atmosphere and producing effects such as the Northern Lights. Only the faster of these incoming cosmic rays are able to resist the magnetic deflection and continue on their course. When reaching the atmosphere, whether quick or slow, these particles begin to collide with atmosphere molecules and produce secondary cosmic rays. Secondary particles can in turn create tertiary particles, and so forth as illustrated in figure 1.2.

This chain reaction of cascading particles poses health and safety risks for anything journeying above the Earth's surface. Commercial airline flights, for example, expose passengers to radiation doses many times greater than what is regularly encountered on the surface. Aside from the harmful effects, this shower of cosmic particles also carries critical information on what's going on beyond our planet and solar system. By identifying the fragments, we can deduce the primary particle, its origin, and properties.

This hunt for cosmic rays is one that has kept scientists busy for nearly a century. Applications branch far beyond public health and into areas of astrophysics, meteorology, and relativity. With its usefulness now laid out, we still encounter a crucial problem: How do we detect a subatomic particle traveling at more than half the speed of light?

1.3 Cosmic Ray Detection

A number of methods can be employed to measure ionizing radiation. Among this list are two used in this project, namely Geiger counters and scintillation detectors. Geiger counters remain one of the most iconic detection methods for ionizing radiation. Born from the work of Dr. Hans Geiger, a German physicist working on particle detection in the early 20th century, the Geiger tube was first designed as a way to measure the number of decaying particles in radioactive substances. The Geiger tube itself is a cylindrical shell encasing an inert gas and conducting wire held at a high voltage. When ionizing radiation strikes an atom of the inert gas, the gas molecule becomes ionized and an electron is expelled from it. This free electron knocks into other molecules, ionizing them as well. When these electrons reach the wire, the potential difference is noticed by the detector.This change is recorded as one 'count.' [5] This is the first and primary detection method used in this project and its specific application will be described later in greater detail.

Scintillation detectors differ from Geiger counters primarily in their detection materials used. Whereas Geiger counters rely on an inert gas, scintillation detectors depend upon a scintillator for their particle detection. Scintillators are special materials that emit photons when high energy particles pass through. Scintillator materials can take many forms and types. Plastic and synthetic scintillators were ideal for this project because of their sturdy and inexpensive design. When a scintillator is struck by a particle, the emitted light from the scintillator must be then collected by another device such as a photomultiplier. In the case of this project, a silicon photomultiplier or SiPM is a small circuit board that contains thousands of small semiconductors. The depletion region of each semiconductor is sufficiently "steep" to prevent electrons from flowing. An emitted photon from a scintillator will give an electron the needed momentum to flow over the depletion region. This electron can also collide with other electrons, thus creating a flowing current. The number of these semiconductors triggered in this process is collected based on the current produced, and thus the number of semiconductors on the SiPM surface is proportional to the energy of the incident particle. Therefore, a cosmic ray can not only be detected, but also measured. [10]

The object of this paper is to compare and contrast these two detection methods and determine their effectiveness in finding the Pfotzer Maximum. Each detector was operated independent of the other. The specifications of their layout and construction will be further discussed in the next section. The advantages and disadvantages of both detector types in application will be discussed further on.

Chapter 2

Method and Procedures

As previously discussed, cosmic rays take several different forms. Therefore, a variety of detection methods is needed to determine the characteristics of an interacting particle or wave. For the course of this research, two different detection methods were utilized; a Geiger-Müller (GM) tube board designed by Aware Electronics, and a CosmicWatch Muon detector designed by researchers at the Massachusetts Institute of Technology. Both detectors operate using an Arduino microcontroller at their core. Chronologically, the GM device was designed built first in the winter of 2017, with the Muon detector was built the following winter in 2018.

2.1 HARPI - The Airborne Geigercounter

HARPI, short for High Altitude Radiation Project Instrument, was a research project designed to investigate how total radiation changed as a function of altitude. The RM-60 Micro-Roentgen Radiation Monitor it carries was designed to be connected to a computer or external monitoring device via a telephone port on the side of the box. In order to gather and store data as part of a balloon payload, the RM-60

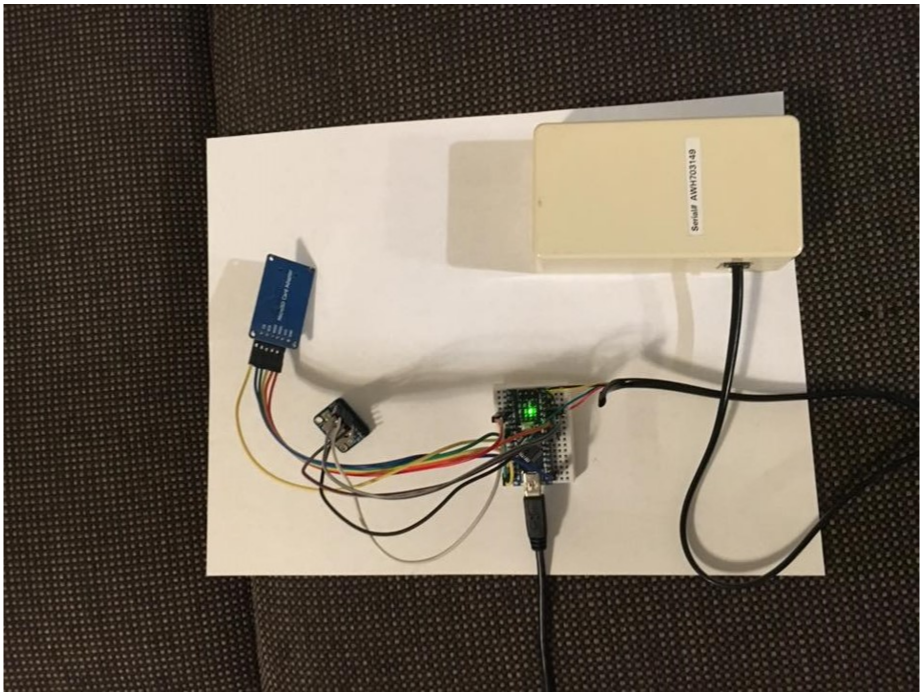

Figure 2.1 HARPI pictured with all of its components.

needed to remotely gather and store data, have its own power source, and endure the extremities encountered during a high altitude voyage. The latter point in that list bears its own section which will be touched on in greater detail later in this chapter.

It was discovered that the RM-60 phone jack used four wires in its data collection: one for the power supply (vin), one for the return voltage for the data, and two for ground. The device could be powered by an Arduino's 5 Volt and ground pins, and could collect data while connected to a digital pin. Several various codes were used as framework, including the built-in library samples included with the Arduino software. The device also included the addition of a DS3231 Real-Time Clock and a Micro-SD card adaptor to both keep track of the time of each event and to save it remotely.

With all of these components laid out, an Arduino Nano was chosen to power the data collection because of its compact size, sufficient memory, and cheap production cost. When first assembled for preliminary tests, the device certainly didn't look

Figure 2.2 Interior of the RM-60 Radiation Monitor from Aware Electronics. The Physical Geiger tube is the large metal cylinder in the bottom left.

attractive but it functioned as expected and operated as follows; When there was a voltage fluctuation in the Geiger tube, the RM-60 would send a dropping pulse through the data collection wire. The Arduino would measure this pulse as one count. After ten seconds, the Arduino would collect all the counts it received and convert it into a measure of counts per minute (cpm). This was recorded to the Micro-SD's text file, along with the corresponding timestamp (see Appendix A for complete code).

2.2 CosmicWatch Muon Detectors

Unlike HARPI, the CosmicWatch detector design and code were already complete (see Appendix B). However, this still left the job of assembling and calibrating the detector, which soon proved to be no simple task, and required the assistance of

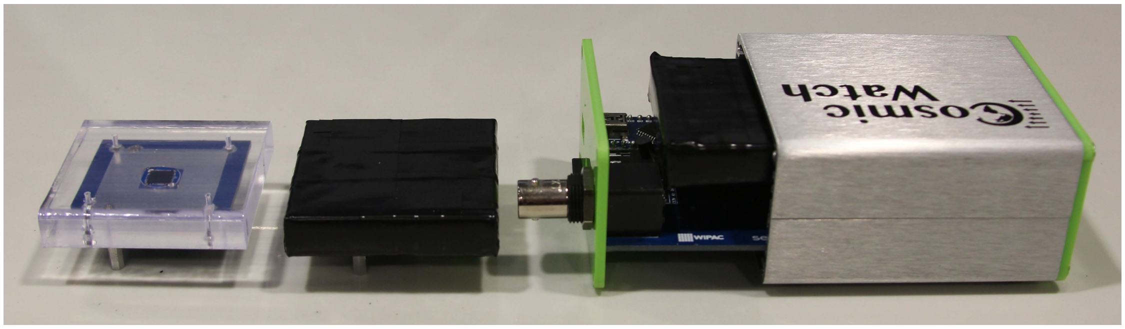

Figure 2.3 From left to right: The SiPM Photomultiplier detector, the light-sealed detector, and the light-sealed detector incorporated into the Arduino and metal casing. Image from CosmicWatch.

several electrical engineers. Surface mount technology (SMT) components only a few millimeters in size were soldered onto a custom printed breadboard and encased in an aluminum housing.

The Muon detector used a scintillator as its means of 'particle catching'. The scintillator is composed of " a polystyrene base (essentially just an inexpensive transparent plastic) mixed with a primary dopant of 1% by-weight of POP (2,5-diphenyloxazole) and 0.03% secondary dopant POPOP (1,4-bis[2-(5-phenyloxazolyl)]benzene)." [10] When exposed to ionizing radiation, it emits light of about 400-420 nm. Relativistic particles are therefore 'caught' and exchanged for photons.

An ON Semiconductor MicroFC 60035 C-Series photomultiplier is the SiPM for this system. It detects photons using thousands of small semiconductors on its surface. The resultant current produced from these semiconductors is proportional to the intensity of the initial particle. As discussed previously, the energy of an incident particle can be approximated based on the magnitude of the current, or the number of semiconductors triggered during the event. A simple particle passing through the scintillator can now be detected and its energy quantified.

2.3 High Altitude Data Collection - A Crash Course

Data collection in the upper stratosphere presents a number of hazards. With increasing altitude, temperatures begin to plummet, pressure approaches zero, jet streams create turbulence, airspace rules and regulations for unmanned aircraft need to be met, and the data needs to be retrievable. There are a number of different solutions to these difficulties. The following listed were chosen based on their effectiveness, cost-efficiency, and/or availability of supplies:

Payload items were attached by strong cords and quick-links in a long train-like fashion. Thus, the string of scientific equipment remained flexible against strong winds and turbulence. Each payload item was wrapped in thick foam, which is both easily replaceable, provides moderate insulation, and shock absorption upon landing. Power sources vary per sensor. HARPI can utilize both 9-volt Lithium batteries and rechargeable battery packs, while the muon detector uses just a rechargeable battery pack. It's important to note that although temperature doesn't affect the data collection of the detectors directly, it does affect the effectiveness of the power source used. Therefore, rechargeable battery packs were chosen that could withstand low temperatures.

As part of FCC regulations, a Notice to Airman (NOTAM) must be filed before the flight with detailed balloon launch locations, times, and flight directions. The Cambridge University Spaceflight Landing Predictor was used to predict the balloon's expected path for the NOTAM and the recovery teams. [3] The predictor makes several assumptions including constant upwards balloon velocity, little variance in predicted weather patterns, and a reasonable balloon burst altitude. All these assumptions have proven reasonable and have provided recovery teams with enough information to estimate its landing location.

Figure 2.4 A flight prediction produced from the CUSF Balloon Flight Predictor. This was taken on February 6, 2018.

When launching a balloon, all plans listed on the NOTAM must be followed to ensure a relatively predictable flight. Launch locations were chosen with low obstacles such as trees and power lines, large open areas, and minimal wind shielding. Launching a balloon is also ideal with low wind and rain, to ensure balloon durability during the flight.

The balloon and all payload items were tracked via a number of communication devices including several HAM radios, beacons, and trackers. Assuming all tracking and scientific equipment is functioning properly, the balloon flight itself is relatively simple: the balloon travels approximately 30,000 meters into the air where the eventual change in pressure bursts the balloon and sends the payload parachuting back down to Earth.

There are many more specifics to high altitude ballooning than were listed here. Only a rough outline suffices to detail the function of the payloads. For those seeking more detailed specifications exclusive to those of BYU-Idaho's High Altitude Research

Team, Aileen Godfrey's thesis is an excellent source of information. [8]

Chapter 3

Analysis and Results

3.1 Cleaning and Fitting Data

Raw data files returned from HARPI and the altimeters were massive, and often out of sync. While altitude data can pinpoint the exact moment of launch, radiation detection data doesn't see a significant change for several hundred meters. With payload items isolated from each other, and with additionally poor time synchronization between devices on our end, device start times had to be recorded for data analysis. The offset between one data set could then be correlated with another. Additionally, the time interval between data points varied per device, as the altimeter measurements returned every two minutes, while HARPI returned a data point every minute.

To better align HARPI event data with the proper timestamp and altitude, we first employed the method of polynomial function fitting. The process was to fit a high order polynomial function (up to sixth order) to the altitude vs. time data. Because acceleration cannot be neglected after the balloon burst, this was first done by separating ascent and descent data points. Once a polynomial equation was set, so that altitude became a function of time, it could be adjusted to the desired data

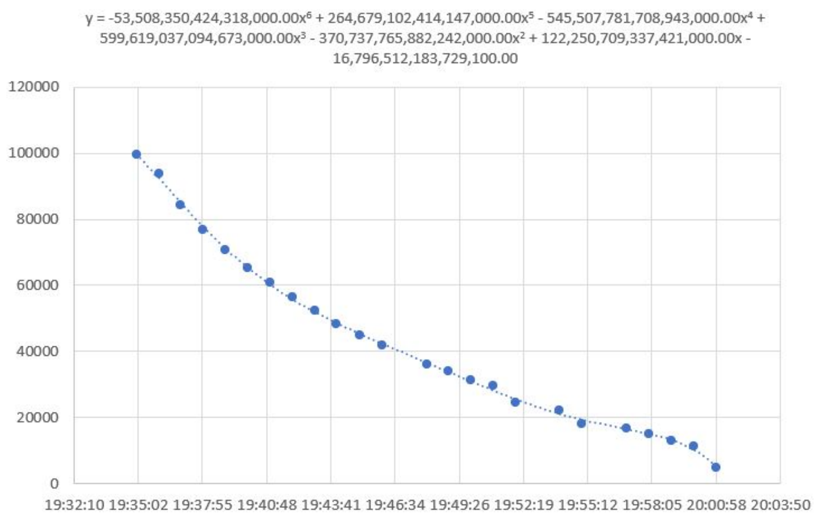

Figure 3.1 The first attempt at data fitting with a sixth-order polynomial function via Microsoft Excel. The process was very laborious and time consuming.

set to determine the altitude at a given time.

It was soon realized that this process was simply too complex and inefficient. Fitting a polynomial function to a sporadic object in freefall wasn't reasonable. It also presented the issue of potentially high uncertainties in altitude data. The function could vary by several hundred feet or more, which is significant for data points closer to the ground. To better combat this, interpolation of data points was decided upon as the next method of altitude approximation. The method here relied on expanding the data from two-minute chunks to smaller intervals. Using the assumption that the balloon's velocity remained constant between two points, basic linear interpolation could then tell us where the balloon was between these smaller intervals. Not only did this reduce the uncertainties in the altitude to justify them being negligible, but the whole process could be automated.

A simple program was assembled in Python to refine this process. The program not only was designed to handle a wide variety of data files, but also was capable of dealing with periods of acceleration such as balloon launch and burst. The program has proven effective with previous flight data files and awaits its use on an uncharted voyage (see appendix C).

On a few occasions, data recording would malfunction or a physical piece of hardware would break. Flights where HARPI suddenly stopped collecting data due to a power related problem or break in the Geiger tube itself had to be thrown out. Rarely, HARPI would return a cpm value tens of times greater than its surrounding points. The cause of this 'spike' could have been natural, such as a burst of cosmic rays, but was of no interest in this particular project. Null and extreme values were thus removed or estimated using data points from its nearest neighbors.

3.2 The Pfotzer Maximum - HARPI's results

The overall goal of HARPI was to replicate Georg Pfotzer's experimental result. As found in Pfotzer's works, the number of events would increase with altitude, until hitting a 'wall' at around 18,000 meters. Pfotzer's discovery was later named the Pfotzer Maximum, the altitude at which the ionizing radiation count is at its maximum.

As was discussed prior, cosmic rays traveling at relativistic speeds interact with the atmosphere and fracture violently into several secondary particles. These particles also collide or decay into tertiary particles, which decay and collide as well, and so on and so forth. Many of these particles never make it to the Earth's surface and instead transfer their energy to the atmosphere. As the atmosphere grows thicker towards the surface, the likelihood of a particle being absorbed increases. Inversely, radiation counts increase with altitude until it reaches the Pfotzer Maximum, which is the altitude at which showering radiation is at its peak. Above this point, cosmic rays

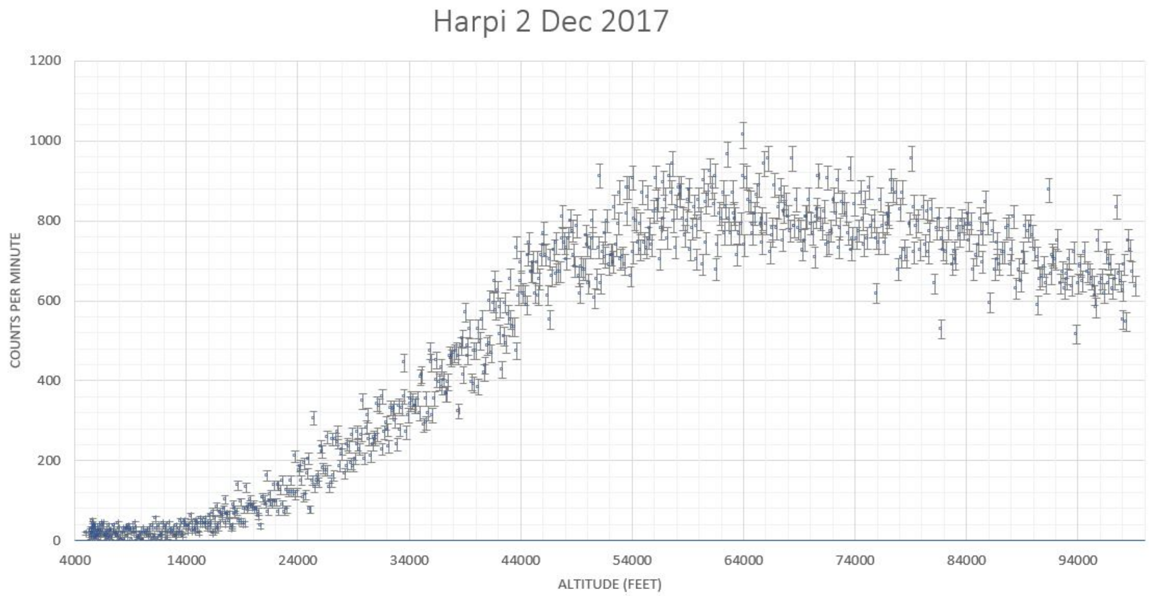

Figure 3.2 Data from the maiden voyage of HARPI on December 2, 2017. This graph shows the number of counts per minute as a function of altitude. The curve peaks around 18,000 meters, the location of the Pfotzer Maximum. Uncertainties in the measurements increase as counts increase.

are less likely to interact with a thinner atmosphere and thus the radiation density begins to decrease.

HARPI mirrors this result, as shown in Figure 3.2. On its maiden voyage, HARPI showed a sudden plateau in cpm around the expected altitude of 18,000 meters. Through several different flights, HARPI showed this bell curve consistently. Multiple flight scenarios were done with HARPI to further validate Pfotzer's theory, such as varying weather conditions, time of year, and location of launch.

HARPI presented several difficulties after repeated flights. The Geiger tubes used for HARPI were old and brittle. They were designed for desktop radiation detection, not high altitude data collection. Several Geiger tubes burst or broke after flights and had to be replaced. They were the only GM tubes we had on hand, and they still performed remarkably well. If future models of HARPI were to be constructed, a more durable GM tube would be the primary objective.

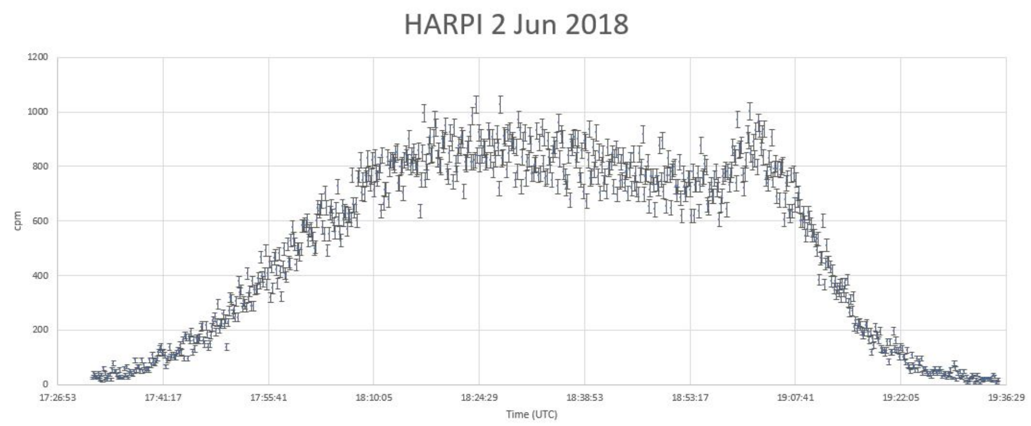

Figure 3.3 Another flight of HARPI on June 2, 2018. This graph is set with the cpm against the time, thus showing the physical balloon flight path. The peak can be seen clearly at 60,000 feet, again repeated after the burst of the balloon.

3.3 Extreme Background Radiation - Muon Detector Results

Although operating under a different detection method, the muon detector was theoretically able to show the existence of the Pfotzer Maximum: It behaved as a way to detect ionizing radiation. Preliminary tests and flights proved inconclusive, as the data was too uncertain to show any significant results.

There were several reasons as to why the CosmicWatch muon detectors failed to deliver the desired results. First and foremost, the scintillator and photomultiplier combination is more sensitive. Background radiation at BYU-Idaho, from radioactive sources both natural and man-made, cause hundreds of counts to be collected every minute (see Figure 3.3). When flown into the stratosphere, an increase in higher energy particles can be seen, but the data proved too 'noisy' to be considered useful. The number of events with respect to altitude increases with altitude but the Pfotzer Maximum is more difficult to pinpoint.

To more precisely narrow the search for ionizing radiation, a second detector was used to utilize the coincidence measurement function designed by CosmicWatch. With two detectors sitting on top of each other, the first detector would be free to record all events. The second detector would only record an event it experienced if it occurred very shortly after being detected by the detector above it. Thus, the background counts would be filtered out, and only radiation incoming from above (or wherever the detectors were angled) would be collected. While great in theory, several problems were encountered in powering these devices in tandem. Future experiments can be done to find solutions to the powering problem.

Another roadblock which was first encountered when working with the detectors was their design. Although compact and functioning, problems were encountered with spontaneous detector failure, power shortages, SMT component malfunctions, assembly failures on our end, possible current jumping between close parts, and crushed components during flights.

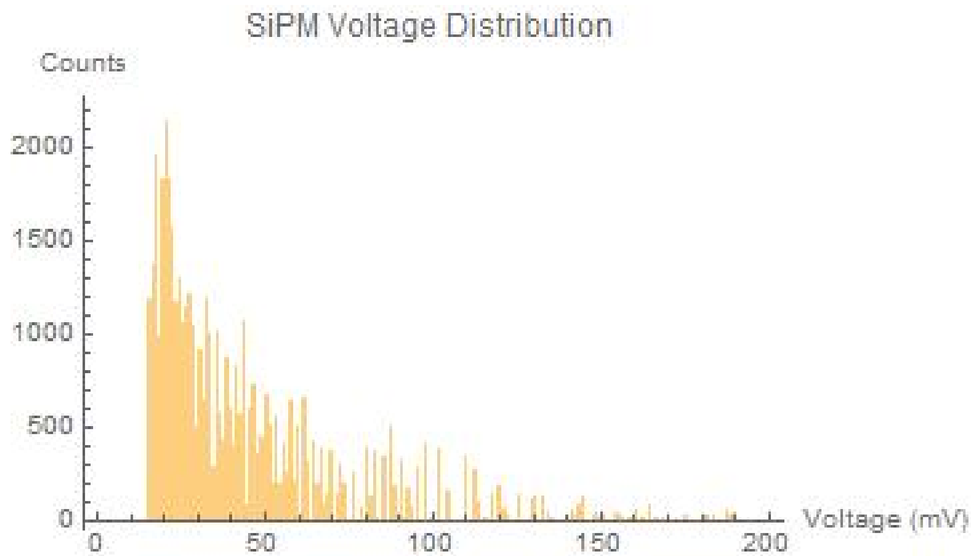

Figure 3.4 A background test of the Muon detector ran on May 20, 2019 at the BYU-Idaho Physics department. Pictured is a histogram of the SiPM voltage of each count, quantized by the number of semiconductors triggered on the SiPM. Extremely low energies below 15 mV are filtered out. The Gaussian-like curve peaks in lower 25 mV range. This is unknown if it is a problem with the detector itself or if the background of Southeastern Idaho truly is this high.

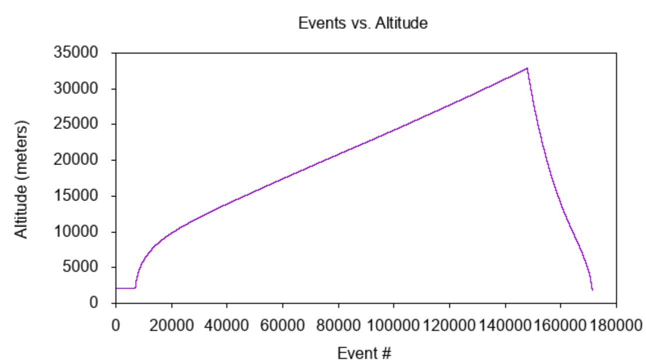

Figure 3.5 Events as a function of time for the duration of the flight on June 1, 2019. It is clear that total number of events increases rapidly below 10,000 feet and decreases during the descent. Of importance is the slope of this curve, with a steeper curve representing fewer counts around a specific altitude.

Chapter 4

Conclusion

HARPI performed remarkably well. Although not explicitly designed for low pressures and temperatures, the RM-60 detectors proved adequate in measuring radiation and detecting the Pfotzer Maximum. Future experiments with HARPI can be branched to a wide variety of radiation-based experiments, including radiation shielding for both terrestrial and extraterrestrial habitation, long-term solar and atmospheric health studies, and particle identification systems. Provided the devices are properly maintained, they can continue to collect data as long as desired.

The muon detectors failed to meet our expectations. One single detector proved too 'noisy' but two detectors set in tandem might have yielded the desired outcome. Battery packs that are better at withstanding the would be needed. Troubleshooting the detectors' spontaneous failure would also need to be completed. Still, the muon detectors have great potential for data collection. Their overall simplistic and compact design make them capable of being used in a wide variety of settings.

In addition, HARPI recently received some upgrades and improvements, including a reduced and simplified PCB with temperature sensor, real-time clock, and SD card reader. As previously mentioned, these devices can provide valuable research opportunities for undergraduate students for years to come. The world of cosmic ray detection shares its roots and therefore its future with radiation detection; the means of particle detection will only improve and the information gathered by both students and researchers will tell us much about the world inside and outside our planet.

Bibliography

[1] Aware Electronics Corp. "RM-60 Micro-Roentgen Radiation Monitor Computer Interface User Manual." (1989).

[2] B. R. Martin, "Nuclear and Particle Physics: an Introduction." 2009.

[3] Cambridge University. *CUSF Landing Predictor 2.5.* https://predict.habhub.org/. Accessed 2017-2019.

[4] Chesley Hannah, "Building a Pocket-Sized Muon Detector and Starting a Cosmic Ray Research Group." Brigham Young University - Idaho. 2018.

[5] Ernest Rutherford and H Geiger. An electrical method of counting the number of α-particles from radio-active substances. Proceedings of the Royal Society, Series A, Volume 81, p 141–161. 1908.

[6] Georg Pfotzer. "A Contribution to the Morphology of X-Ray Bursts in the Aururoal Zone," (1939)

[7] Georg Pfotzer. "Vertical Intensity of Cosmic Rays by Threefold Coincidences in the Stratosphere," (1939)

[8] Godfrey, Aileen. "Creation of the High Altitude Research Team to Research and Record the 2017 Total Solar Eclipse." Brigham Young University - Idaho. 2017.

[9] M. W. Friedlander, A thin cosmic rain: Particles from outer space. Harvard University Press, 2002.

[10] Spencer N. Axani, "The Physics Behind the CosmicWatch Desktop Muon Detectors," 20 December 2018.

[11] W.R. Leo. "Techniques for Nuclear and Particle Physics Experiments - A How-to Approach" (1987).

Appendix A

HARPI Arduino Program

```
/*****************************************************
 *               H. A. R. P. I.     2.0
 *     High Altitude Radiation Project Instrument
 *     Geigercounter, RTC, and Temperature Sensor
 *          by McKay Murphy     July 2019
 *****************************************************/

//IF THE CODE WON'T UPLOAD, MAKE SURE YOUR PROCESSOR IS USING THE
   OLD BOOTLOADER//

//RM-60 manual states that the deadtime for the detector is 90
   microseconds, or 9.0*10^-5
//using equation 5.14 in the deadtime pdf, and assuming it
   constitutes a non-extendable deadtime,
//we find that the true count rate m as compared to
//the actual count rate k over time interval T is insignificant,
   even at high count rates
//i.e. (1 - (k/T)t) << 1

```

```
// libraries used
#include <SPI.h>
#include <SD.h>
#include <Wire.h>
#include "RTClib.h"
RTC_DS3231 rtc;

const byte geigerPin = 2; //Geiger pin
const byte CSpin = 10; //SD pin, attached to the sd card's CS pin

int count; //count variable
int deadTime; //time the detector is not operating
float totalDeadTime;

File dataFile;  //dataFile varaible

byte tempPin = 0;  //tempurature pin
float tempRaw;
float tempC;





void setup() {
  Serial.begin(9600); //for serial monitor

  // initialize the SD card
  while (!SD.begin(CSpin)){
     Serial.println("Card initialization failed, or not present!");
     delay(2500);
```

```
}
Serial.println("Card initialized!");
Serial.println("");

while (!rtc.begin()){
  Serial.println("RTC initialization failed!");
  delay(2500);
}

while (analogRead(tempPin) == 0){
  Serial.println("Temperature initialization failed!");
  delay(2500);
}
Serial.println((String)"Temperature sensor initializaed! It is
 currently " + convertCelcius(analogRead(tempPin)) + " Celcius.");


DateTime startTime = rtc.now();
Serial.print("RTC initialized! Current time is ");
Serial.print(startTime.hour(),DEC); Serial.print(":");
Serial.print(startTime.minute(),DEC); Serial.print(":");
Serial.println(startTime.second());
Serial.println("");


pinMode(geigerPin, INPUT);  //setup our geiger pin
attachInterrupt(digitalPinToInterrupt(geigerPin), test, RISING);
 //if our counter voltage rises, run test()

byte testCount;
while (digitalRead(geigerPin) != 1){
```

```
    testCount++;
    Serial.println((String)"Testing Geigercounter..." + testCount +
   "/10");
    delay(1500);

    if (testCount == 10)
    {
      Serial.println("");
      Serial.println("Geigercounter hasn't registered a count!");
      Serial.println("Either you're unlucky, or the RM-60 isn't
   working! Check connections and try again.");
      Serial.println("Stopping here.");
      while(1);
    }

  }
  Serial.println("All modules initialized, we're ready to go!");
  Serial.println("");
}
```

```
void loop() {
  delay(60000); //wait 60 seconds before interrupting

  long deadTimeStart = millis();
  //serial monitor print
  DateTime startTime = rtc.now();


  Serial.print(startTime.hour(),DEC); Serial.print(":");
  Serial.print(startTime.minute(),DEC); Serial.print(":");
  Serial.print(startTime.second());
  Serial.print("  ");

  Serial.print(count); Serial.print("  ");

  Serial.print(totalDeadTime); Serial.print("  ");

  Serial.println(convertCelcius(analogRead(tempPin)));

  //dataFile print
  dataFile = SD.open("datafile.txt", FILE_WRITE);

  dataFile.print(startTime.hour(),DEC); dataFile.print(":");
  dataFile.print(startTime.minute(),DEC); dataFile.print(":");
  dataFile.println(startTime.second());
  dataFile.print("  ");

  dataFile.print(count);  dataFile.print("  ");

  dataFile.print(totalDeadTime); dataFile.print("  ");

```

```
  dataFile.println(convertCelcius(analogRead(tempPin)));

  dataFile.close();


  count = 0;  //reset our counts

  deadTime = millis() - deadTimeStart;
  totalDeadTime += deadTime;
}


//converts our analog reading into a temperature
float convertCelcius(float V){
  float C = ((V * 5.0 / 1024.0) - 0.5) * 100;
  return C;
}



//used to count a Geiger count
void test() {
  count++;
}
```

Appendix B

CosmicWatch Muon Detector Program

```
/*
  CosmicWatch Desktop Muon Detector Arduino Code

  This code is used to record data to the built in microSD card
   reader/writer.

  Questions?
  Spencer N. Axani
  saxani@mit.edu

  Requirements: Sketch->Include->Manage Libraries:
  SPI, EEPROM, SD, and Wire are probably already installed.
  1. Adafruit SSD1306      -- by Adafruit Version 1.0.1
  2. Adafruit GFX Library -- by Adafruit Version 1.0.2
  3. TimerOne              -- by Jesse Tane et al. Version 1.1.0
*/

```

```
#include <SPI.h>
#include <SD.h>
#include <EEPROM.h>

#define SDPIN 10
SdFile root;
Sd2Card card;
SdVolume volume;

File myFile;

const int SIGNAL_THRESHOLD    = 50;        // Min threshold to trigger on
const int RESET_THRESHOLD     = 50;

const int LED_BRIGHTNESS      = 255;         // Brightness of the LED [0,255]

//Calibration fit data for 10k,10k,249,10pf; 20nF,100k,100k, 0,0,57.6k,  1 point
const long double cal[] = {-9.085681659276021e-27, 4.6790804314609205e-23, -1.0317125207013292e-19,
  1.2741066484319192e-16, -9.684460759517656e-14, 4.6937937442284284e-11, -1.4553498837275352e-08,
   2.8216624998078298e-06, -0.00032303262067203 7, 0.019538631135788468, -0.3774384056850066, 12.324891083404246};

const int cal_max = 1023;

//initialize variables
char detector_name[40];
```

```

unsigned long time_stamp                  = 0L;
unsigned long measurement_deadtime        = 0L;
unsigned long time_measurement            = 0L;      // Time stamp
unsigned long interrupt_timer             = 0L;      // Time stamp
int           start_time                  = 0L;      // Start time
    reference variable
long int      total_deadtime              = 0L;      // total time
    between signals

unsigned long measurement_t1;
unsigned long measurement_t2;

float temperatureC;


long int      count                       = 0L;         // A tally
    of the number of muon counts observed
float         last_adc_value              = 0;
char          filename[]                  = "File_000.txt";
int           Mode                        = 1;

byte SLAVE;
byte MASTER;
byte keep_pulse;

float cpm;
int loopin;
float countPrev;
float timePrev;

```

```

void setup() {
  analogReference (EXTERNAL);
  ADCSRA &= ~(bit (ADPS0) | bit (ADPS1) | bit (ADPS2));    // clear prescaler bits
  //ADCSRA |= bit (ADPS1);                                  // Set prescaler to 4
  ADCSRA |= bit (ADPS0) | bit (ADPS1); // Set prescaler to 8

  get_detector_name(detector_name);
  pinMode(3, OUTPUT);
  pinMode(6, INPUT);

  Serial.begin(9600);
  Serial.setTimeout(3000);

  if (digitalRead(6) == HIGH){
     filename[4] = 'S';
     SLAVE = 1;
     MASTER = 0;
  }

  else{
     //delay(10);
     filename[4] = 'M';
     MASTER = 1;
     SLAVE = 0;
     pinMode(6, OUTPUT);
     digitalWrite(6,HIGH);
     //delay(2000);
    }
```

```

SD.begin(SDPIN);
/*
bool test = !SD.begin(SDPIN);
while (test) {
  Serial.println(F("SD initialization failed!"));
  Serial.println(F("Is there an SD card inserted?"));
  int n = 0;
  while (n < 3){
    digitalWrite(3,HIGH);
    delay(500);
    digitalWrite(3,LOW);
    delay(200);
    n += 1;
  }
  n = 0;
  test = !SD.begin(SDPIN);
}
*/

get_Mode();
if (Mode == 2) read_from_SD();
else if (Mode == 3) remove_all_SD();
else{setup_files();}

if (MASTER == 1){digitalWrite(6,LOW);}
analogRead(A0);


start_time = millis();
loopin = 1;
```

```
}

void loop() {
  if(Mode == 1){
  Serial.println(F
   ("##################################################################################
   );
  Serial.println(F("### CosmicWatch: The Desktop Muon Detector"));
  Serial.println(F("### Questions? saxani@mit.edu"));
  Serial.println(F("### Comp_date Comp_time Event Ardn_time[ms] ADC
   [0-1023] SiPM[mV] Deadtime[ms] Temp[C] Name CPM(s^-1)"));
  Serial.println(F
   ("##################################################################################
   );
  Serial.println("Device ID: " + (String)detector_name);

  myFile.println(F
   ("##################################################################################
   );
  myFile.println(F("### CosmicWatch: The Desktop Muon Detector"));
  myFile.println(F("### Questions? saxani@mit.edu"));
  myFile.println(F("### Comp_date Comp_time Event Ardn_time[ms] ADC
   [0-1023] SiPM[mV] Deadtime[ms] Temp[C] Name CPM(s^-1)"));
  myFile.println(F
   ("##################################################################################
   );
  myFile.println("Device ID: " + (String)detector_name);

  write_to_SD();
  }
}
```

```

void setup_files(){
  for (uint8_t i = 1; i < 201; i++) {
      int hundreds = (i-i/1000*1000)/100;
      int tens = (i-i/100*100)/10;
      int ones = i%10;
      filename[5] = hundreds + '0';
      filename[6] = tens + '0';
      filename[7] = ones + '0';
      if (! SD.exists(filename)) {
          Serial.println("Creating file: " + (String)filename);
          if (SLAVE ==1){
           digitalWrite(3,HIGH);
           delay(5000);
           digitalWrite(3,LOW);
          }
          delay(500);
          myFile = SD.open(filename, FILE_WRITE);
          break;
      }
   }
}

void write_to_SD(){
  while (1){
    bool appendCPM = false;
    if (analogRead(A0) > SIGNAL_THRESHOLD){
      int adc = analogRead(A0);

      if (MASTER == 1) {digitalWrite(6, HIGH);
          count++;
```

```
        keep_pulse = 1;}

    analogRead(A3);

    if (SLAVE == 1){
        if (digitalRead(6) == HIGH){
            keep_pulse = 1;
            count++;}}
    analogRead(A3);

    if (MASTER == 1){
          digitalWrite(6, LOW);}

    measurement_deadtime = total_deadtime;
    time_stamp = millis() - start_time;
                                     //timestamp of event
    measurement_t1 = micros();
                                     //start measurement time, to
calculate deadtime
    temperatureC = (((analogRead(A3)+analogRead(A3)+analogRead(A3)
)/3. * (3300./1024)) - 500)/10. ; //temperature outread

    //every 10 seconds, we'll check the count rate
    if (time_stamp > 60000. * loopin){
      countPrev = count - countPrev;
      cpm   = countPrev / ((time_stamp - timePrev) / 1000.) * 60.;
                                         //counts divided by runtime
      timePrev = time_stamp - timePrev;
      loopin += 1;
                                     //ensures we only run on
multiples of 10 seconds
```

```
      appendCPM = true;
                                         //add the extra print to our
   output
    }

    //for master detector
    if (MASTER == 1) {
        digitalWrite(6, LOW);
        analogWrite(3, LED_BRIGHTNESS);
        Serial.print((String)count + " " + time_stamp+ " " + adc+
   " " + get_sipm_voltage(adc)+ " " + measurement_deadtime+ " " +
   temperatureC);
        myFile.print((String)count + " " + time_stamp+ " " + adc+
   " " + get_sipm_voltage(adc)+ " " + measurement_deadtime+ " " +
   temperatureC);
        if (appendCPM){
          Serial.println((String)" " + cpm);
          myFile.println((String)" " + cpm);
        }
        else{
          Serial.println("");
          myFile.println("");
        }

        myFile.flush();
        last_adc_value = adc;}

    //for slave detector
    if (SLAVE == 1) {
        if (keep_pulse == 1){

```

```
            //every 10 seconds, we'll check the count rate
          if (time_stamp > 10000. * loopin){
            countPrev = count - countPrev;
            cpm   = countPrev / ((time_stamp - timePrev) / 1000.) * 60.;                                   //counts divided by runtime
            timePrev = time_stamp - timePrev;
            loopin += 1;                                                    //ensures we only run on multiples of 10 seconds
            appendCPM = true;                                               //add the extra print to our output
          }

          //if we triggered within the timeframe
            analogWrite(3, LED_BRIGHTNESS);
            Serial.print((String)count + " " + time_stamp+ " " + adc+ " " + get_sipm_voltage(adc)+ " " + measurement_deadtime+ " " + temperatureC);
            myFile.print((String)count + " " + time_stamp+ " " + adc+ " " + get_sipm_voltage(adc)+ " " + measurement_deadtime+ " " + temperatureC);
            if (appendCPM){
              Serial.println((String)" " + cpm);
              myFile.println((String)" " + cpm);

            }
            else{
              Serial.println("");
              myFile.println("");
```

```
            }
            myFile.flush();
            last_adc_value = adc;}}

      keep_pulse = 0; //reset again
      digitalWrite(3, LOW);
      while(analogRead(A0) > RESET_THRESHOLD){continue;}

      appendCPM = false; //triggers every 10 seconds
      total_deadtime += (micros() - measurement_t1) / 1000.;}
    }
}

void read_from_SD(){
    while(true){
    if(SD.exists("File_210.txt")){
      SD.remove("File_209.txt");
      SD.remove("File_208.txt");
      SD.remove("File_207.txt");
      SD.remove("File_206.txt");
      SD.remove("File_205.txt");
      SD.remove("File_204.txt");
      SD.remove("File_203.txt");
      SD.remove("File_202.txt");
      SD.remove("File_201.txt");
      SD.remove("File_200.txt");
      }

    for (uint8_t i = 1; i < 211; i++) {

      int hundreds = (i-i/1000*1000)/100;
```

```
      int tens = (i-i/100*100)/10;
      int ones = i%10;
      filename[5] = hundreds + '0';
      filename[6] = tens + '0';
      filename[7] = ones + '0';
      filename[4] = 'M';

      if (SD.exists(filename)) {
          delay(10);
          File dataFile = SD.open(filename);
          Serial.println("opening: " + (String)filename);
          while (dataFile.available()) {
              Serial.write(dataFile.read());
              }
          dataFile.close();
          Serial.println("EOF");
        }
      filename[4] = 'S';
      if (SD.exists(filename)) {
          delay(10);
          File dataFile = SD.open(filename);
          Serial.println("opening: " + (String)filename);
          while (dataFile.available()) {
              Serial.write(dataFile.read());
              }
          dataFile.close();
          Serial.println("EOF");
        }
      }

    Serial.println("Done...");
```

```
    break;
  }

}

void remove_all_SD() {
  while(true){
    for (uint8_t i = 1; i < 211; i++) {

      int hundreds = (i-i/1000*1000)/100;
      int tens = (i-i/100*100)/10;
      int ones = i%10;
      filename[5] = hundreds + '0';
      filename[6] = tens + '0';
      filename[7] = ones + '0';
      filename[4] = 'M';

      if (SD.exists(filename)) {
          delay(10);
          Serial.println("Deleting file: " + (String)filename);
          SD.remove(filename);
        }
      filename[4] = 'S';
      if (SD.exists(filename)) {
          delay(10);
          Serial.println("Deleting file: " + (String)filename);
          SD.remove(filename);
        }
    }
    Serial.println("Done...");
    break;
```

```
  }
  write_to_SD();
}

void get_Mode(){ //fuction for automatic port finding on PC
    Serial.println("CosmicWatchDetector");
    Serial.println(detector_name);
    String message = "";
    message = Serial.readString();
    if(message == "write"){
      delay(1000);
      Mode = 1;
    }
    else if(message == "read"){
      delay(1000);
      Mode =  2;
    }
    else if(message == "remove"){
      delay(1000);
      Mode = 3;
    }
}

float get_sipm_voltage(float adc_value){
  float voltage = 0;
  for (int i = 0; i < (sizeof(cal)/sizeof(float)); i++) {
    voltage += cal[i] * pow(adc_value,(sizeof(cal)/sizeof(float)-i
   -1));
    }
    return voltage;
    }
```

```
boolean get_detector_name(char* det_name)
{
    byte ch;                              // byte read from eeprom
    int bytesRead = 0;                    // number of bytes read so far
    ch = EEPROM.read(bytesRead);          // read next byte from eeprom
    det_name[bytesRead] = ch;               // store it into the user buffer
    bytesRead++;                          // increment byte counter

    while ( (ch != 0x00) && (bytesRead < 40) && ((bytesRead) <= 511) )
    {
        ch = EEPROM.read(bytesRead);
        det_name[bytesRead] = ch;           // store it into the user buffer
        bytesRead++;                      // increment byte counter
    }
    if ((ch != 0x00) && (bytesRead >= 1)) {det_name[bytesRead - 1] = 0;}
    return true;
}
```

Appendix C

Altitude Interpolation Program

```
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.dates as d
import numpy as np
import datetime as dt
import os

START = 20 #if the difference in altitude is greater than this, we
    can start adding to a list.

#calculates the points between flights, as well as the predicted
    burst altitude and time, under ideal circumstances.
class Flight():
    def __init__(self):
        self.data = self.getData()
        self.alt = self.data["altitude"] #grab altitude columns
        self.time = self.data["time"] #grab time columns

        self.time = self.time.str.split(' ', n = 1, expand = True) #
```

```
split date from time
        self.time = self.time[1].str.split(':', expand = True) # split time into [h,mm,ss]

        self.maxIndex = self.alt.loc[self.alt == self.alt.max()].index[0] #locate our max index

        #our lists we'll use
        self.altList = []
        self.timeList = []
        self.vList = []


    def getData(self):
        self.filePath = input("Please enter the file path: ")
        return pd.read_csv(self.filePath) #importing file


    def initCond(self):
        a = -9.7 #m/s at an altitude of about 30,000 meters
        self.v0 = self.vAve
        deltaT = 0

        #calculate the time between the unknown variables
        #if (int(self.time[0][self.maxIndex]) < int(self.time[0][self.maxIndex + 1])):
         #   deltaT = 60

        #how much time has passed between the last point and burst altitude
        deltaT += (int(self.time[1][self.maxIndex+1]) - int(self.
```

```
time[1][self.maxIndex]))

        #initial conditions -> {last point before burst, burst,
burst again, first point after}
        yi1 = self.alt[self.maxIndex]
        yf1 = yi1 + (self.v0 * (-np.sqrt((-4 * self.v0 * deltaT) / a
) + 2 * deltaT))
        yi2 = yf1
        yf2 = self.alt[self.maxIndex+1]

        #times -> {from [self.maxIndex] point to burst, from burst
to [self.maxIndex+1]}
        self.t1 = ((yi2 - yi1) / self.v0) * 60
        self.t2 = (deltaT * 60 - self.t1)

        burst = self.alt[self.maxIndex] + self.t1*self.v0

        print("Burst was at {} UTC.".format(str(self.time[0][self.
maxIndex]) + ":" + str(self.time[1][self.maxIndex]) + ":" + str(
int(self.t1))))
        print("Burst altitude was {} meters.".format(round(burst,1))
)
        print("Max height was {} meters.".format(burst + (self.v0
**2)/(2 * 9.7)))


    #assuming the velocity between two points is constant
    def calcPreBurst(self):
        altTot = 0
        timeTot = 0

```

```
        #averaging the last 5 points to predict average ascent rate towards burst
        for n in range(5):
            altTot += (self.alt[self.maxIndex-n] - self.alt[self.maxIndex-n-1])

        #adjust if the hour is different from the first and last index
        if (self.time[0][self.maxIndex] < self.time[0][self.maxIndex-n-1]):
            timeTot = 60

        timeTot += (int(self.time[1][self.maxIndex]) - int(self.time[1][self.maxIndex-n-1])) #total time over last five indexes
        self.vAve = (altTot / timeTot) / 60 #calculating our average velocity on those last 5 points


        print("Max alt reported: {}".format(self.alt[self.maxIndex]))
        print("Index: {}".format(self.maxIndex))
        print("")


       #deals with appending the minute of the burst event
    def appendPostBurst(self):
        t = 0
        dt = 1
        #while we're still ascending up
        while t < self.t1:
            self.altList.append(self.altList[-1] + self.v0)
```

```
            self.timeList.append(str(self.time[0][self.maxIndex]) + ":" + str(self.time[1][self.maxIndex]) + ":" + str(t))
            t += dt

        v = self.v0 #we're gonna start falling now, our velocity is changing
        while t != (self.t2 + self.t1):
            v -= -9.7 * dt #gravity is less higher up, but not by much
            self.altList.append(self.altList[-1] + self.v0)
            self.timeList.append(str(self.time[0][self.maxIndex]) + ":" + str(self.time[1][self.maxIndex]) + ":" + str(t))
            t += dt



        #with the assumption that the velocity is constant between two points, fill in the missing holes.
    def getAltitude(self,index1,index2):
        altDiff = int(self.alt[index1+1]) - int(self.alt[index1]) #difference in velocity
        timeDiff = int(self.time[1][index1+1]) - int(self.time[1][index1]) #difference in time
        if timeDiff == -59:
            timeDiff += 60
        velAve = (altDiff/(timeDiff*60)) #average velocity in m/s
        self.altList.append(self.alt[index1] + velAve * index2)
        self.vList.append(velAve)


        #adds approximated values to our output lists
```

```python
    def appendList(self,index1,index2):
        for a in range(index1,index2): #for the range of our given indecies
            if (abs(self.alt[a + 1] - self.alt[a]) > START) and (int(self.time[1][a+1]) - int(self.time[1][a])) < 20: #if our change in altitude is greater than a set amount...
                if (a == self.maxIndex):
                    self.appendPostBurst()
                else:
                    for b in range(0,60): #break up readings by second and append to the list
                        self.timeList.append(str(self.time[0][a]) + ":" + str(self.time[1][a]) + ":" + str(b))
                        self.getAltitude(a,b)
            print(a)


    def output(self):
        dates = []
        print("Got here!")
        print(self.timeList)
        #converting our list of strings into a list of times
        for p in range(len(self.altList)):
            dates.append(dt.datetime.strptime(self.timeList[p], "%H:%M:%S"))

        export = pd.DataFrame([self.timeList,self.altList]).transpose().\
        rename(columns={0:"Time (UTC)",1:"Altitude (m)"}) #turn our strings back into a dataframe
        print("")
```

```
        if not (os.path.exists(self.filePath[:-4])):
            print("Creating directory: " + self.filePath[:-4])
            os.mkdir(self.filePath[:-4])

        plt.plot(dates,self.altList) #plots our data, just to doublecheck
        export.to_csv(self.filePath[:-4] + "/" + self.filePath[:-4] + "_processed.csv", index = False) #output file


    def mergeData(self):
        print("")
        filePath2 = input("Please enter the file path of merging data: ")
        print("")
        timeDiff = input("Enter the starting time of the device in UTC format hh:mm:ss: ")
        print("")
        dataHeader = input("Enter header name for your data column: ")
        print("")
        timeHeader = input("Enter header name for your time column: ")
        print("")

        '''
        separateTime = int(input("Enter 1 if your time is formatted as hh:mm:ss.0000, or 2 if it's another format:"))
        print("")

```

```
        if separateTime == 2:
            timeFormat = []
            print("Customizable time formats not created yet...")
        '''

        self.data2 = pd.csv_read(filePath2)

        self.time2 = list(map(int,self.data2[timeHeader].str.split(':', expand = True))) #split time into [h,mm,ss]
        timeDiff =  list(map(int,timeDiff.str.split(":"))) #split our time into three sections
        self.dataImport = list(map(float,self.data2[dataHeader])) #data we'll be looking at


        self.timeOffset = []
        self.timeList2 = []
        self.dataList = []

        #there will be a time change (likely) in our data, let's see what that offset is.
        for n in range(len(timeDiff)):
            self.timeOffset[n] = timeDiff[n] - self.time2[0][n]

        for i in range(len(self.time2)):
            for j in range(len(self.timeList)):
                if self.time2[0][i] == self.timeList[0][i] and\
                self.time2[1][i] == self.timeList[1][i] and\
                self.time2[2][i] == self.timeList[2][i]:
                    self.timeList2.append([self.timeList[all][i]])

```

```
def main():
    print("")
    print("Interpolation HART File Processor")
    print("written by McKay Murphy")
    print("July 2019")
    print("")

    n = Flight()

    n.calcPreBurst()
    n.initCond()
    n.appendList(0,len(n.alt) - 1)
    n.output()

    #n.mergeData()
```

```python






if __name__ == '__main__':
    main()
```

Appendix D

Additional Muon Datasets

The Muon detectors could observe several different variables as opposed to HARPI's single variable. This provided many more plots that could be useful or are of specific interest. The scope of this project was limited to only counts, energies and altitude, so these plots were not included specifically in the body of the paper. Pictured in a few of these plots are a green and red bar, showing the time of the balloon's launch and burst. Data collection begins before launch to compare the resting background counts to the physical flight data. Data collection has been cropped to end at landing.

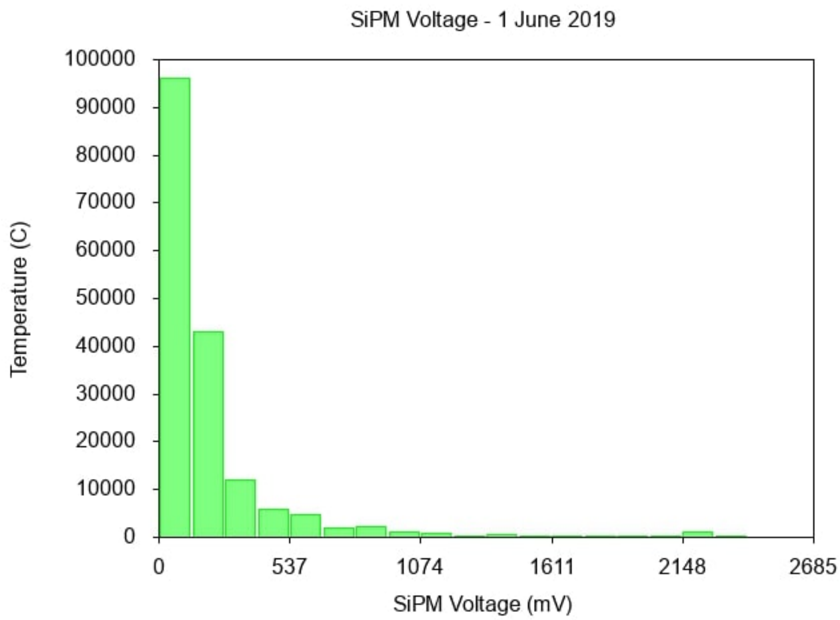

Figure D.1 A histogram of the SiPM voltage flown aboard a balloon on June 1, 2019. The SiPM voltage peaks very much in the lower mV spectrum, with specks in the higher mV range.

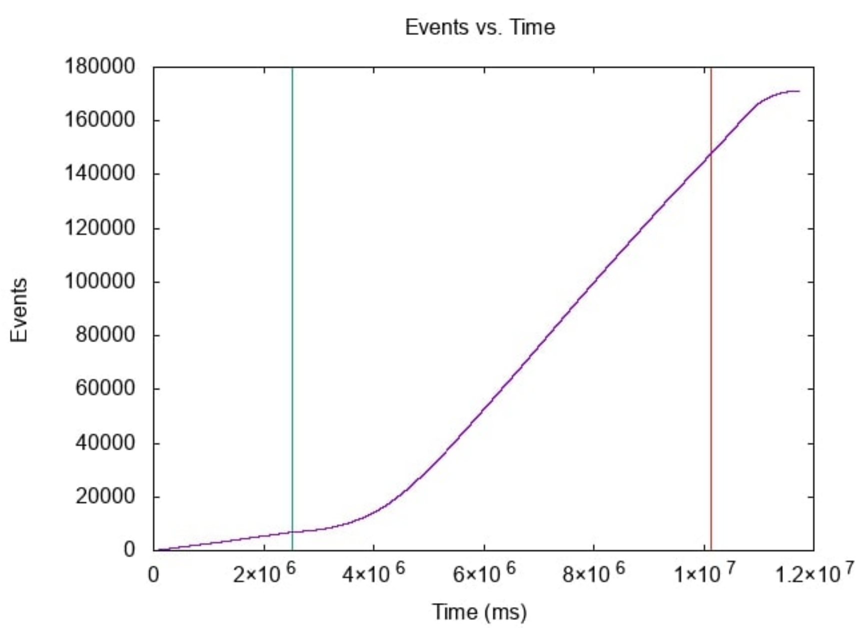

Figure D.2 A standard plot of the total number of events by each time in milliseconds.

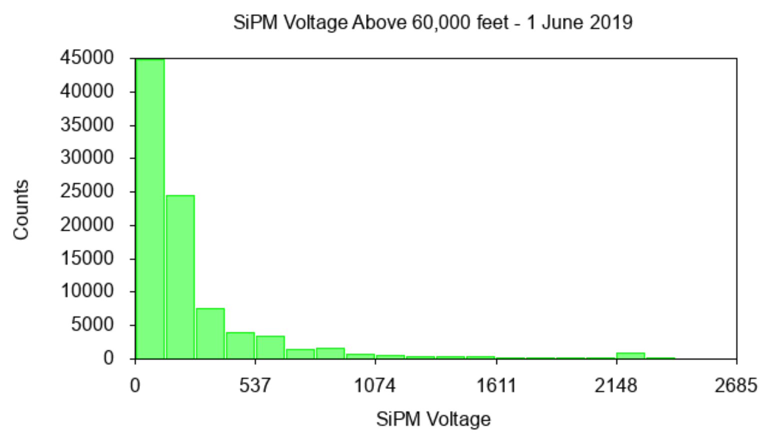

Figure D.3 SiPM voltage histogram against time is imaged here. Filtered here are the data points above 60,000 feet. This was used in contrast with the next visual.

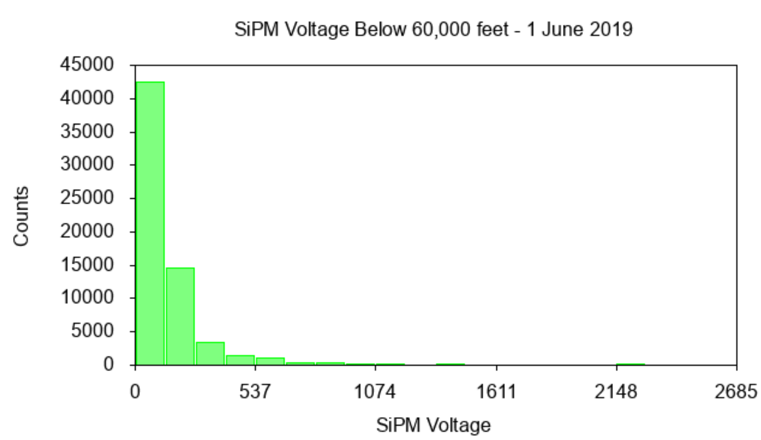

Figure D.4 SiPM voltage histogram for points filtered below 60,000 feet.

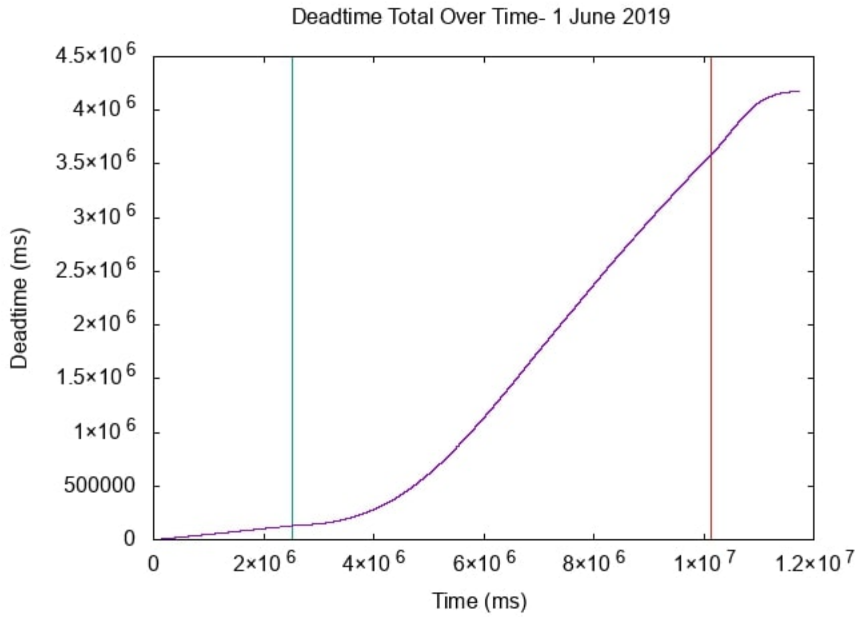

Figure D.5 Total deadtime of the detector as a function of the total time of detector operation.

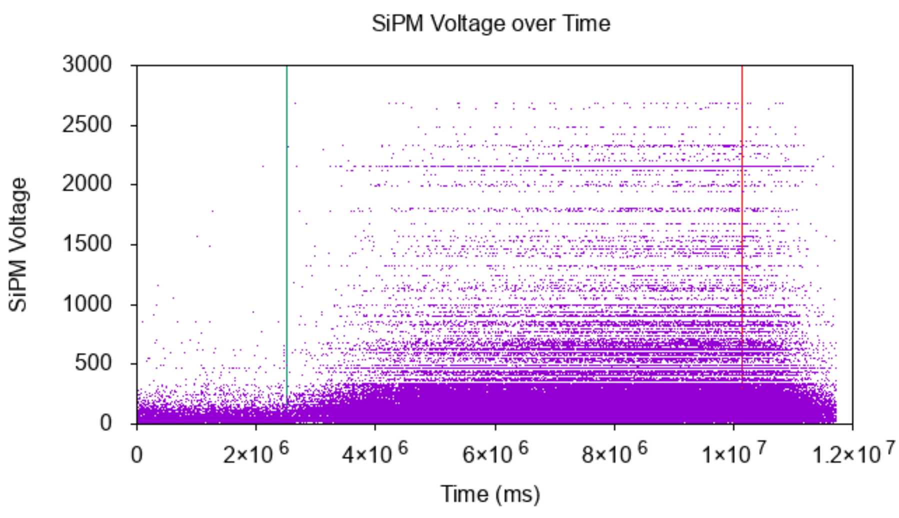

Figure D.6 Distribution of SiPM voltages by time. It's an extremely cluttered plot but there are noticeably higher voltages as time, and thus altitude, increases.

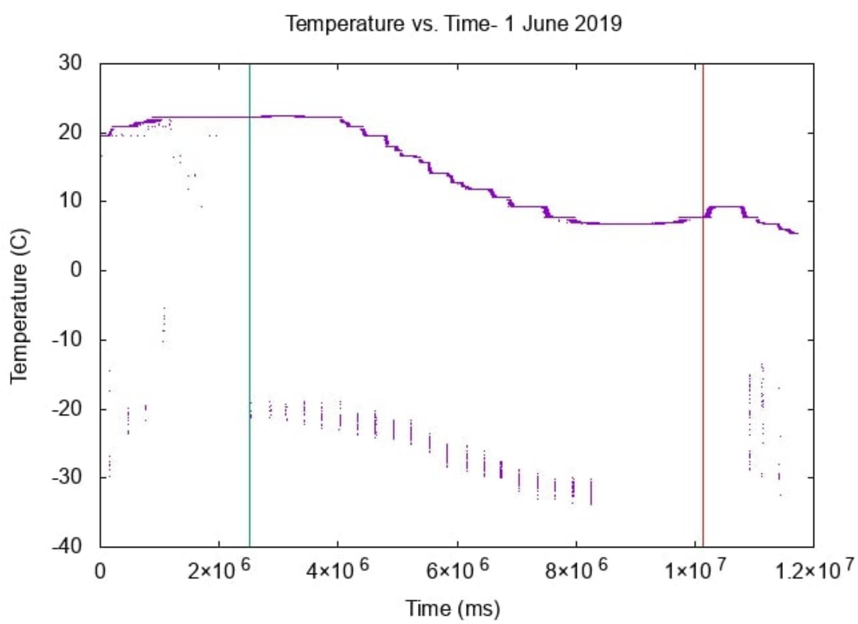

Figure D.7 Temperature of the Muon detector's internal temperature sensor.