# DEVELOPMENT OF RADSIGPRO - AN OPEN SOURCE CODE FOR FAST AND REAL TIME RADIATION DETECTION

A Thesis

by

BENJAMIN S. WELLONS

Submitted to the Graduate and Professional School of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

| | |
|---|---|
| Chair of Committee, | Shikha Prasad |
| Committee Members, | Sunil S. Chirayath |
| | Grigory V. Rogachev |
| Head of Department, | Michael Nastasi |

August 2022

Major Subject: Nuclear Engineering

Copyright 2022 Benjamin S. Wellons

ABSTRACT

In this work, an open source code RadSigPro 1.0 is developed and used for fast processing of nanosecond long pulses from scintillation detectors. The pulse processing and identification involves pulse height distribution (PHD), pulse shape discrimination (PSD), and time-of-flight (TOF). For the goal of better particle segregation, processed particle waveforms are supplied to test machine learning techniques along with TOF labeled neutrons and gamma-rays to train the data. The code is used to model the programmable logic design of an field programmable gate array (FPGA) design for on-the-fly processing of neutron and gamma-ray pulses, along with testing the results. Finally, a comparison between CAEN's CoMPASS Data Acquisition (DAQ) Software and RadSigPro's resulting tallies is attempted.

When trained on pulse waveform data, classification accuracy of 96% could be achieved with less than 100 ns of data, but 400 ns were required to get the accuracy to 97%. This indicates the information relevant to labeling a pulse as a neutron or gamma-ray is mostly found at the pulse's start. Principle component analysis (PCA) extracts information from the entire pulse, so relevant information is not lost when the number of components is trimmed. As a result, support vector machine (SVM) models trained on two principal components could accurately classify pulses over 94% of the time. To achieve 97% accuracy, models with nonlinear kernels required fewer than 50 principal components for training. Misclassification results displayed a 1.97% false gamma-ray rate and a 2.27% false neutron rate.

A weighted average of the percent difference of the results for RadSigPro 1.0 implemented on a central processing unit (CPU) and an FPGA logic design is calculated. This shows a 0% difference for the PHD data sets, a 0.458% and 0.344% difference for the designated gamma-detector and neutron-detector's PSD data sets respectively, and a 0% difference for the TOF data set. When the FPGA logic design is applied and simulated, it computes the total and tail pulse areas within 5 nanoseconds of the arrival of the final data point used for accumulation and also captures the pulse

height value within 2 nanoseconds of the arrival of the pulse maximum's data point.

# DEDICATION

To all the people who continuously chose to wish the best for me.

ACKNOWLEDGMENTS

I would like to thank my committee chair, Dr. Shikha Prasad, and my committee members, Dr. Sunil Chirayath, Dr. Grigory V. Rogachev, for their support and expertise throughout the course of this research.

Thanks also go to my friend Harrison Hall and colleague Xiaodong Tang.

## CONTRIBUTORS AND FUNDING SOURCES

### Contributors

This work was supported by a thesis dissertation committee consisting of Professors Shikha Prasad and Sunil S. Chirayath of the Department of Nuclear Engineering and Professor Grigory V. Rogachev of the Department of Department of Physics and Astronomy.

All other work conducted for the thesis was completed by the student independently.

### Funding Sources

Graduate study was supported by 2021 T3 Round 4 and by Texas AM University.

NOMENCLATURE

| | |
|---|---|
| FPGA | Field Programmable Gate Array |
| DAQ | Data Acquisition |
| CPU | Central Processing Unit |
| MC&A | Materials Control and Accounting |
| ps | Picoseconds |
| PSD | Pulse Shape Discrimination |
| PMT | Photomultiplier Tube |
| ns | Nanoseconds |
| TTT | Tail-to-Total |
| TOF | Time-of-Flight |
| PHD | Pulse Height Distribution |
| GEM | Gas Electron Multiplier |
| CSV | Comma Seperated Value |
| LSB | Least Significant Bit |
| mV | Millivolts |
| CFD | Constant Fraction Discrimination |
| TCC | Time Cross-Correlation |
| RPC | Raw_Pulse_Correction |
| SBZC | Sample Before Zero Crossing |
| PHDP | PHD_Plot |
| PSDA | PSD_Analysis |
| TOFA | TOF_Analysis |

| | |
|---|---|
| TOFC | TOF_Comparison |
| MAPE | Mean Absolute Percentage Error |
| SVM | Support Vector Machine |
| RBF | Radial Basis Function |
| PCA | Principal Component Analysis |
| fC | femtoCoulombs |
| LWR | Light Water Reactor |
| NRC | U.S. Nuclear Regulatory Authority |
| KP-FHR | Kairos Power's Fluoride salt-cooled High-temperature Reactor |
| MCRE | Molten Chloride Reactor Experiment |
| ARDP | Advanced Reactor Demonstration Program |
| DOE | Department of Energy |
| PBR | Pebble Bed Reactor |
| IAEA | International Atomic Energy Agency |

TABLE OF CONTENTS

Page

ABSTRACT .......... ii

DEDICATION .......... iv

ACKNOWLEDGMENTS .......... v

CONTRIBUTORS AND FUNDING SOURCES .......... vi

NOMENCLATURE .......... vii

TABLE OF CONTENTS .......... ix

LIST OF FIGURES .......... xi

LIST OF TABLES .......... xiv

I. INTRODUCTION .......... 1

I.1 Objective .......... 1
I.2 Motivation .......... 1
I.3 Chapter Descriptions .......... 4

II. LITERATURE REVIEW .......... 5

II.1 Background .......... 5
II.1.1 Particle Interaction Mechanisms .......... 5
II.1.2 Organic Scintillation Detectors .......... 7
II.1.3 Waveform Processing Methods .......... 11
II.2 Previous Works .......... 14

III. RADSIGPRO 1.0 .......... 16

III.1 RadSigPro Development .......... 16
III.1.1 Raw Pulse Correction Function .......... 16
III.1.2 Measurement Tallies .......... 19
III.2 Setup and Data Acquisition .......... 20
III.2.1 Experiment Setup .......... 20
III.2.2 Data Acquisition .......... 23
III.3 Results from RadSigPro 1.0 Implemented on CPU .......... 24
III.3.1 PHD .......... 24

III.3.2 PSD ........ 25
III.3.3 TOF ........ 26

IV. NEUTRON-GAMMA LABELS FOR SUPERVISED MACHINE LEARNING ........ 29

IV.1 Data Labeling ........ 29
IV.2 Supervised Machine Learning Results ........ 31

V. CPU VS FPGA IMPLEMENTATIONS ........ 34

V.1 RadSigPro Edits ........ 34
V.2 FPGA vs CPU ........ 34
V.2.1 PHD ........ 35
V.2.2 PSD ........ 36
V.2.3 TOF ........ 37

VI. RADSIGPRO VS COMPASS COMPARISON ........ 39

VI.1 CoMPASS Plot Data ........ 39
VI.2 Current Discrepancies ........ 40

VII. SUMMARY AND CONCLUSIONS ........ 45

VII.1 Conclusion ........ 45
VII.2 Future Steps ........ 46
VII.2.1 RadSigPro Usage ........ 47
VII.2.2 CoMPASS Comparison ........ 47
VII.2.3 Online Data Processing and FPGA ........ 47
VII.2.4 Machine Learning ........ 47
VII.2.5 Applications ........ 47

REFERENCES ........ 48

APPENDIX A. RADSIGPRO PYTHON CODE ........ 52

APPENDIX B. RADSIGPRO FPGA IMPLEMENTATION ........ 72

LIST OF FIGURES

FIGURE Page


I.1 Reactor design of X-Energy's Xe-100 Reactor Model, with online refueling using pebble-bed fuel, creating a high signal environment. Reprinted from [1]. .......... 2

II.1 Gamma-ray interaction mechanism regions of dominance shown by atomic number and incident energy (hv). Reprinted from [2]. .......... 5

II.2 A schematic of Compton scattering including relevant energy formula for each particle involved; where $E$ is energy, $p$ is momentum, $h$ is Planck's constant, $m_0$ is initial mass, $c$ is the speed of light, and $v$ is frequency. Reprinted from [3]. .......... 6

II.3 A labeled diagram of an EJ-309 organic scintillation detector, displaying the process by which it receives incident radiation and converts it into an electronic pulse. .......... 9

II.4 The energy level of an organic molecule with $\pi$-electron structure. Reprinted from [4]. 9

II.5 A gamma-ray pulse and neutron pulse detected from the same fission event of $^{252}$Cf, collected using two EJ-309 organic scintillation detectors. .......... 11

II.6 Classical implementation of the CFD showing each step of the process. .......... 12

II.7 Example CFD shaped signal for timing showing the zero crossing and SBZC. .......... 13

II.8 Acquisition window of particle waveform, displaying the moment of triggering, along with long gate, short gate, gate offset, pre-trigger, trigger hold-off, record length, and all other relevant values. Reprinted from [5]. .......... 14

III.1 Example raw pulse output by CoMPASS, detected using $^{252}$Cf source. .......... 18

III.2 Detection setup technology connections configuration, displaying the HV, both EJ-309 detectors, the digitizer, and the computer with CoMPASS software. .......... 21

III.3 Image of detection setup including source, equipment, and scintillator specifications. 22

III.4 PHD histograms of the specified gamma-detector (left) and neutron-detector (right), made from collected $^{252}$Cf pulses. The error bars in gray represent the variance of each bin. .......... 24

III.5 PSD histogram from collected $^{252}$Cf pulse pairs, processed with RadSigPro, and presented for two detectors set to record either gamma-ray or neutron particles in a TOF measurement based on the incident particles PSD value. The error bars shown in gray represent the variance of each bin. . . . 26

III.6 TOF histogram from collected $^{252}$Cf pulse pairs, or TCC events, including additional indicators which highlight the pulse pair identities which make up the histograms features. The error bars in gray represent the variance of each bin. . . . 27

IV.1 PSD histogram of time-cross-correlated pulse pairs from $^{252}$Cf, whose TCC fell in the range of 10 - 40 ns. The error bars in gray represent the variance of each bin. . . . 30

IV.2 Mean classification accuracy of the SVM using an RBF kernel: a as a function of the length of the waveform used for training and b the number of principal components that were extracted and used for training. The red band surrounding the mean test accuracy represents 2 standard deviations of the test accuracy for all folds. Reprinted from [6]. . . . 32

V.1 Example gamma-ray PHD histogram of collected $^{252}$Cf pulses, using the FPGA implementation. The error bars in gray represent the variance of each bin. . . . 35

V.2 Example neutron PHD histogram of collected $^{252}$Cf pulses, using the FPGA implementation. The error bars in gray represent the variance of each bin. . . . 36

V.3 Example PSD histogram of collected $^{252}$Cf pulse pairs, using the FPGA implementation. The error bars in gray represent the variance of each bin. . . . 37

V.4 Example TOF histogram of collected $^{252}$Cf pulse pairs - RadSigPro CPU implementation (left) and FPGA implementation (right). . . . 38

VI.1 TOF histograms of collected $^{252}$Cf pulses, displaying the result of CoMPASS and RadSigPro's TOF tallies. . . . 40

VI.2 PHD histograms of collected $^{137}$Cs pulses, displaying the result of CoMPASS and RadSigPro's pulse height tallies using a baseline held at 8141 ADC units, after a calibration was applied to the data sets based off the Compton edge locations. . . . 41

VI.3 PSD histograms of collected $^{252}$Cf pulse pairs displaying the result of CoMPASS and RadSigPro's TTT tallies using a baseline held at 8142 ADC units for the 'gamma-detector' (left) and a baseline held at 8127 ADC units for the 'neutron-detector' (right). . . . 43

VI.4 PSD histograms of collected $^{252}$Cf pulse pairs displaying the result of CoMPASS and RadSigPro's TTT tallies, where CoMPASS is using a baseline held at 8142 ADC units for the 'gamma-detector' (left) and a baseline held at 8142 ADC units for the 'neutron-detector' (right) while RadSigPro uses an averaged baseline value unique to each pulse for both the "gamma-detector" (left) and "neutron-detector" (right)........ 44

LIST OF TABLES

TABLE Page

III.1 Example format of CoMPASS's output CSV files, displaying each pulse's: instance of detection (timetag), calibration energy (calib_energy), channel selected for energy calibration (energyshort), a flag displaying a code which indicated any errors in the pulse data (flags), and the sample data which make up each pulse (samples). ......... 17

# I. INTRODUCTION

## I.1 Objective

This project includes detection of radioactive particles emitted by spontaneous fission and identification analysis using developed code. For the goal of better particle segregation, processed particle waveforms are supplied to test machine learning techniques along with TOF labeled neutrons and gamma-rays to train the data. The code is used to model the programmable logic design of an field programmable gate array (FPGA), and to test the results. Finally, the developed code's tallies will be compared with CAEN's CoMPASS Data Acquisition (DAQ) Software.

The goal of my research is the creation of a tool for fast signal processing of particles detected with organic/inorganic scintillation detectors. This work is towards the development of an autonomous detection method for advanced and harsh reactor environments. The specific objectives towards achieving this goal are:

i The creation of a pulse processing code/method (RadSigPro) for calculating tallies of and applying separations to particle waveforms collected using organic and inorganic scintillation detectors.

ii Generation of filtered data using RadSigPro to create neutron-gamma labels for supervised machine learning to allow better particle determination, further discussed in *Patrick et al* [6].

iii Comparison between the central processing unit (CPU) and FPGA implementations of RadSigPro. The FPGA version was implemented by another researcher in the research group, based on the RadSigPro method, further discussed in detail in *Kumaran et al* [7].

iv Comparison of RadSigPro tallies and separation with CAEN's CoMPASS DAQ Software.

## I.2 Motivation

High flux radiation measurements such as in-core measurements or next-to-core measurements are challenging endeavors due to extremely high signal, or detection-rates. However, such capabili-

ties are becoming especially important as many Generation IV advanced reactor designs such as X-energy, Kairos Power and others are proposing on-line refueling [8][9]. One such design can be seen in Fig. I.1 below, using pebble-bed fuel with on-line refueling, is the Xe-100. This reactor model was awarded 80 million dollars in government funding to conduct a reactor demonstration within 5 years. The Xe-100 along with Kairos Power's fluoride salt-cooled high-temperature reactor (KP-FHR) and Southern Company and Terra Power's molten chloride reactor experiment (MCRE) won the Advanced Reactor Demonstration Program (ARDP) by the Department of Energy (DOE); where the KP-FHR and MCRE are planned for demonstration within 10-14 years [10, 11].

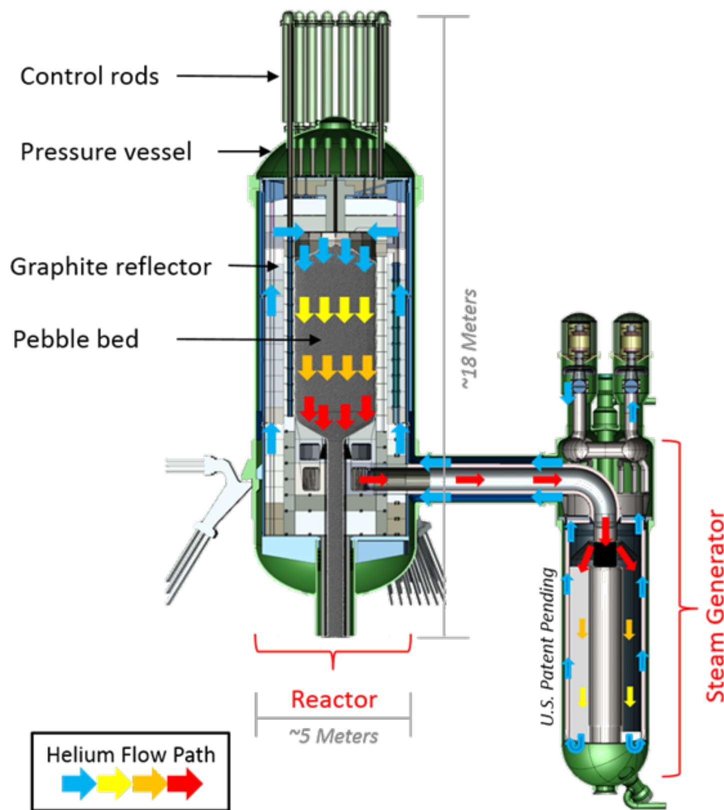



Figure I.1: Reactor design of X-Energy's Xe-100 Reactor Model, with online refueling using pebble-bed fuel, creating a high signal environment. Reprinted from [1].

With online refueling, fuel elements with very high radioactivity levels will need to be measured for various operations, safety, and security purposes. Measurements in highly radioactive environ-

ments cause very high fluxes incident on detectors. For instance, an irradiated pebble circulating out of a pebble-bed-core may have activities on the order of kilo-Curies [8]. These in-core and next-to-core measurements are for burnup and materials control and accounting (MC&A).

The current fleet of light water reactors (LWR) fall into Category III for MC&A requirements as set by the U.S. Nuclear Regulatory Authority (NRC). Many of the advanced reactor designs on the other hand will likely fall under Category I or II MC&A requirements having enrichment levels which exceed 10%. The pebble bed reactor (PBR) design's online refueling makes the reactor more susceptible to diversion at the loading and unloading process to and from the reactor. An integration of safety comes as the reactor core fissile material inventory is directly impacted by the reactivity, so monitoring that reactivity is very important. Understanding and monitoring for the existing range of isotropic compositions of the pebble fuel as they pass through the PBR core is important especially for tracking the Pu quantities in the pebbles [12]. Some molten salt reactor (MSR) fuel salts contain a high concentration of nuclear material meaning less volume will be required to be diverted to get the same amount of material as compared to PBR and LWR designs. These MSR designs will need to have onsite analytical capabilities for the International Atomic Energy Agency (IAEA) accountancy. Important roles for process monitoring and measurement for MSRs includes instruments for measuring salt composition, quantities, flows, etc [13]. MSRs can have a conservative estimate of about 1000 rem/h from just 17 kg of fuel salt. During draining and refueling measurements need to be bade to identify the isotopic makeup and amount of fuel salt added or removed. Dynamic inventory calculations for available designs can help inform the MC&A [14].

Current state-of-the-art semiconductor detectors such as HPGe detectors result in tens of microsecond-long pulses; however, certain scintillation detectors such as barium-fluoride can produce scintillation light with a mean decay time of 630 picoseconds (ps) for its fast component [15]. Another example is organic scintillation detectors such as EJ-309 which have a mean decay time of its short component $\sim$3.5 ns [16]. Thus, this and similar detectors can quickly detect in high radiation fields without being crippled by deadtime or pulse pileup issues. Theoretically, we

can expect to see 1,000 to 10,000 times faster detection rates, provided the accompanying signal processing and readout systems are also developed with comparable speeds. Fortunately, readout systems are being made available with sampling rates of a point every 100-200 ps (giga samples per second, 5-10 GHz sampling rate) to enable fast data acquisition [17]. Further, the benefits of such fast detection application systems will be realized with fast processing of detector signals. An important processing application includes discriminating between neutrons and gamma-rays using pulse shape discrimination (PSD) methods, a process which becomes especially difficult with low energy depositions. Insuring particle labeling can remain fast, accurate, and reasonably autonomous is necessary for operations, safety, and security considerations of a nuclear reactor.

## I.3 Chapter Descriptions

Chapter II lists background information key to understanding the work of this thesis in section II.1 while section II.2 highlights publications relevant to this work.

Chapter III details the construction of the RadSigPro code in section III.1, the experimentation process by which events were collected in section III.2, and the resulting histograms from RadSigPro processing collected data on the CPU in section III.3.

Chapter IV explains the process used to provide labels to a $^{252}$Cf spontaneous fission data set of neutrons and gamma-rays in section IV.1 and then notable results from the data sets use in supervised machine learning for allowing better particle determination in section IV.2.

Chapter V discusses edits made to the RadSigPro to allow comparison with the FPGA implementation in section V.1 and then compares the results of RadSigPro CPU vs FPGA implementations in section V.2.

Chapter VI details the ongoing process of directly comparing RadSigPro's results and CoMPASS's. Section VI.1 discusses the retrieval of CoMPASS's plot data while section VI.2 highlights the issues which currently limit the completion of comparison along with displaying initial results.

Chapter VII discusses the conclusions gained from the accumulation of work in section VII.1 along with potential future work in section VII.2.

# II. LITERATURE REVIEW

This chapter lists background information key to understanding the work of this thesis in section II.1 while section II.2 highlights publications relevant to this work.

## II.1 Background

In understanding background information, the relevant types of particle interaction mechanisms of both neutrons and gamma-rays must be explained. How organic scintillators record particle waveforms is also important, along with the methods which are used to process those waveforms.

### II.1.1 Particle Interaction Mechanisms

Interactions between incident radiation and the scintillation material will be dictated by the known interaction mechanisms of the incident radiation. As shown in Fig. II.1, gamma-rays can interact via photoelectric effect, Compton scattering, and pair production.

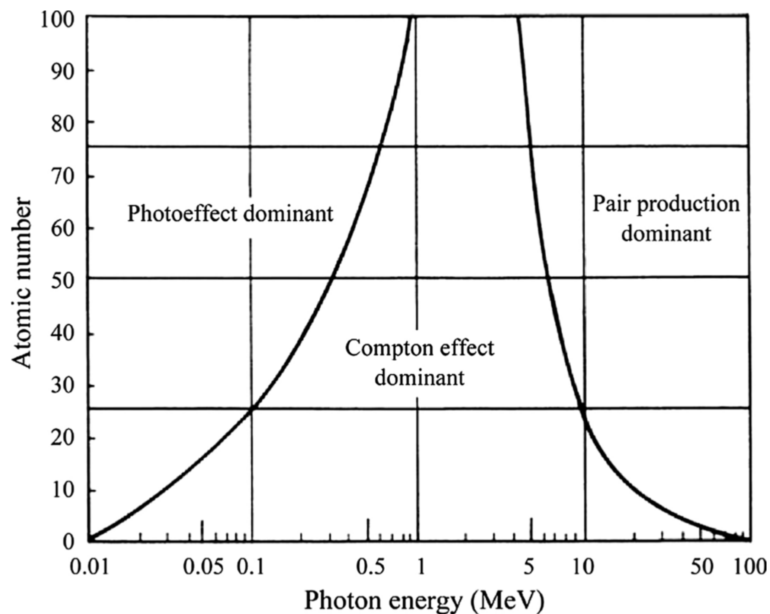



Figure II.1: Gamma-ray interaction mechanism regions of dominance shown by atomic number and incident energy (hv). Reprinted from [2].

The likelihood of each interaction mechanism occurring can be seen to depend on the atomic number of the absorbing material, in this case the scintillation material, and the energy of the incident gamma-ray. The organic scintillation material in our EJ-309 detectors is based on the solvent xylene, which has a chemical formula of $C_8H_{10}$. Both carbon and hydrogen have low atomic numbers, 6 and 1 respectively, so it is clear to see that Compton scattering will overwhelmingly dominate the gamma-ray interactions which occur.

Compton scattering, as shown in Fig. II.2, involves an incident gamma-ray (photon) with energy $hv$ interacting with an outer shell electron of a molecule. The gamma-ray transfers a portion of its energy to the electron, which is then scattered. This process also releases lower energy photons. These photons are the light which enters the photocathode and are turned into electrons for detection. Compton scattering was discovered in 1922 by Arthur H. Compton, for which he received the Nobel Prize in 1927. Because Compton scattering involves interactions with the outer shell of electrons, the binding energy of the target electrons is very small and can be ignored.

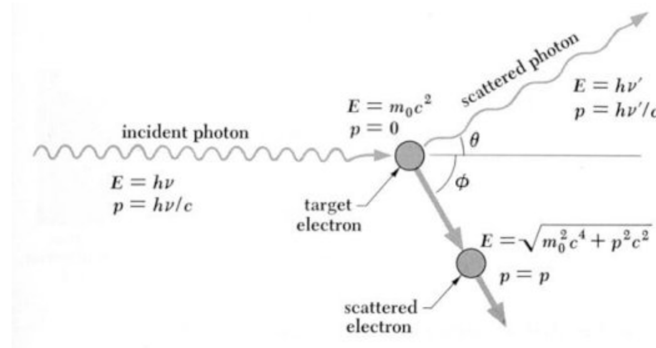



Figure II.2: A schematic of Compton scattering including relevant energy formula for each particle involved; where $E$ is energy, $p$ is momentum, $h$ is Planck's constant, $m_0$ is initial mass, $c$ is the speed of light, and $v$ is frequency. Reprinted from [3].

Neutron interactions with the scintillation material will mainly consist on the neutrons scattering off the nuclei of the hydrogen atoms, resulting in them emitting photons. Elastic scattering occurs with neutrons scattering off nuclei through billiard ball-type collisions. This leads to highly energetic

recoil nuclei, which then must lose energy by excitation and ionization of the surrounding material. The neutron shares its initial kinetic energy with the nucleus, which suffers recoil only and is not left in an excited state [18]. Equation II.2 displays the scattering distribution probability of a neutron striking a stationary nuclei. This probability is not based on the final energy ($E_f$) but rather the initial energy ($E_i$) of the neutron and $\alpha$, which is displayed in Equation II.1. This shows that the scattering probability of a neutron will be based on its $E_i$ and the atomic mass $A$ of the nuclei which it is striking. In the case of a neutron striking a hydrogen atom (atomic mass of 1), $\alpha$ becomes 0 meaning the probability of scatter is equal to (1/$E_i$) [19].

$$\alpha = \left(\frac{A-1}{A+1}\right)^2 \tag{II.1}$$

$$P(E_i \to E_f) = \begin{cases} \frac{1}{(1-\alpha)E_i}, & \alpha E_i \leq E_f \leq E_i \\ 0, & \text{otherwise} \end{cases} \tag{II.2}$$

Now Equation II.3 displays the average final energy ($\overline{E_f}$) as a function of $\alpha$ and $E_i$. Again when considering a hydrogen atom, it becomes clear that the average final energy of a neutron scattering with a hydrogen atom will be half of the intial energy of said neutron [19].

$$\overline{E_f} = \int_{\alpha E_i}^{E_i} dE_f E_f P(E_i \to E_f) = \left(\frac{1+\alpha}{2}\right) E_i \tag{II.3}$$

While elastic scattering dominates lower energy interactions, fast neutrons can scatter inelastically with a particle nuclei. The scattered neutron will carry less energy than the incident neutron and the nuclei goes into an excited state. This process will result in the nucleus emitting a gamma-ray or remaining metastable [18].

### II.1.2 Organic Scintillation Detectors

Neutral particles cannot interact via electromagnetic forces. This prohibits neutral particles from being able to be detected by a photomultiplier tube (PMT). Scintillation material produces

luminescence when excited by ionizing radiation, and that allows the PMT to record the light and produce an electronic signal. Because of this scintillation detectors are used to detect neutral particles such as neutrons and gamma-rays, which can still interact with the scintillation material.

This whole process works by having the incident neutron or gamma-ray enter the scintillation material, usually kept in some housing chamber. In there they interact with the material exciting it and causing scintillation, the re-emitting of absorbed energy in the form of light. It is important to note that this excited state can sometimes be metastable, causing the drop back down to a lower stable state to be delayed some time (nanoseconds to hours based on the material used). Now the low energy photons emitted in the scintillator pass through the photocathode where gamma-ray (photon) interaction mechanisms produce electrons. These photoelectrons or primary photoelectrons are passed through the focusing electrode directing them into the PMT's collection geometry to be multiplied by a factor of $10^6$. The primary electrons are electrostatically accelerated by an electric potential so that they strike the first dynode with a high kinetic energy, releasing a number of secondary electrons with each impact. The subsequent dynode interactions continue to produce more and more secondary electrons, increasing potential at each dynode and providing an acceleration field. The output signal produced by the anode is an electronic signal pulse carrying information about the energy of the incident radiation on the scintillator. This process is diagramed in Fig. II.3, which displays an EJ-309 organic scintillation detector as was used in this research.

Organic scintillators produce light by both prompt and delayed fluorescence. The prompt decay time is typically a few nanoseconds (ns), while the delayed decay time is normally on the order of hundreds of ns. The majority of the light is produced by the prompt decay; however, the amount of light in the delayed component often varies as a function of the type of particle causing the excitation [15]. The physics process which allows organic scintillators to produce fluorescence comes from molecular structures transitioning energy levels. This process is shown in Fig. II.4, where the scintillation material molecules can be seen absorbing energy from incident radiation before then emitting light.

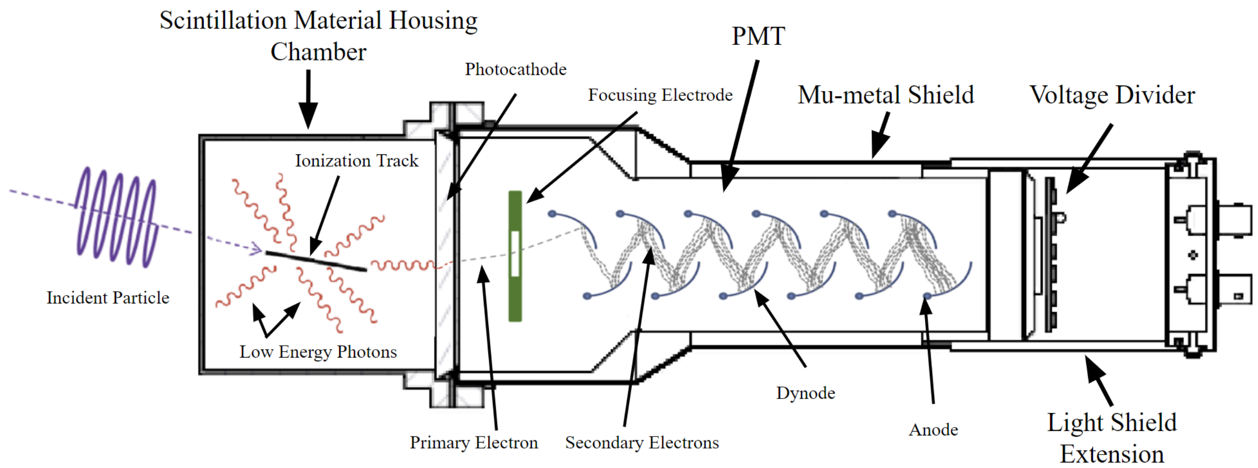



Figure II.3: A labeled diagram of an EJ-309 organic scintillation detector, displaying the process by which it receives incident radiation and converts it into an electronic pulse.

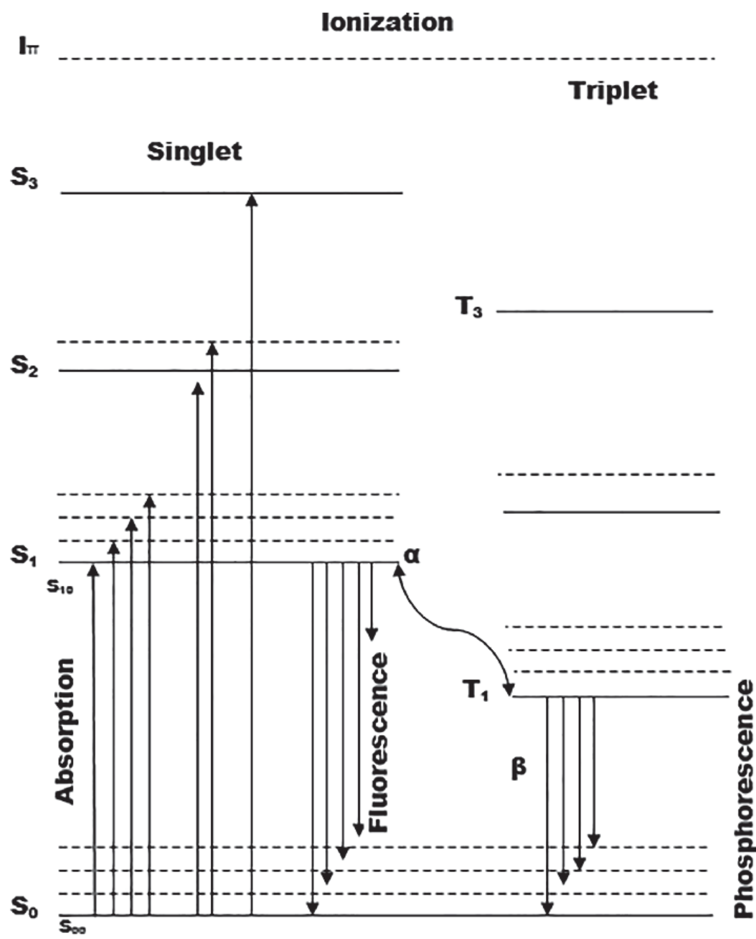



Figure II.4: The energy level of an organic molecule with $\pi$-electron structure. Reprinted from [4].

The organic scintillation material is synthesized such that its structure allows $\pi$-elections to be excited by incident radiation. A $\pi$-electron resides in $\pi$-bonds of a double or triple bond, or in a conjugated $\pi$ orbital. The $\pi$ molecular orbital component is known as an anti-bonding molecular orbital, which means it consists of electrons which spend their time outside the nuclei of two connected atoms. These electrons are what we refer to as $\pi$-electrons, and they are characterized by their relative mobility. The excitation of $\pi$-electrons can elevate them from the ground state $S_0$, to a number of singlet states ($S_1$, $S_2$, $S_3$, etc.). Direct decay from one of these singlet states is what causes prompt fluorescence, as the decay emits a photon. This fluorescence decays exponentially, and is therefore gone within a couple nanoseconds. The delayed fluorescence occurs from an alternative decay path where the excited $\pi$-electron's spin is reversed from the $S_0$ singlet state to the $T_1$ triplet state. Because the triplet state $T_1$ is below $S_0$ singlet state, this process involves a radiationless decay [15]. Once the $\pi$-electron is in the triplet state, there are interactions which can occur. Firstly, the $\pi$-electron can decay directly from the triplet state back to the $S_0$ ground singlet state emitting light. This emission is known as phosphorescence, having a longer wavelength and decay time than fluorescence, however it is strongly forbidden due to the multiplicity selection rule and is thus weak compared to fluorescence [20]. The second potential decay has the $\pi$-electron gain enough energy to jump to the $S_1$ singlet state. This energy gain can involve thermal energy or two $\pi$-electrons in the $T_1$ state interacting such that one ends up in the $S_0$ state and the other in $S_1$. During the interaction between two $\pi$-electrons phonons are emitted. The subsequent decay of the $\pi$-electron from the $S_1$ state to ground $S_0$ emits light which is the delayed fluorescence [20]. This fluorescence has unique traits which very importantly allows for particle differentiation.

Unlike prompt fluorescence, delayed fluorescence does not follow an exponential decay structure. The amount of delayed fluorescence in a pulse is related to the triplet density in the wake of the incident particle. This relationship is based on evidence that that the bimolecular reaction rate is related to the square of the density of triplet states [15]. The triplet state density is determined by the rate of energy loss, d$E$/dx, of an exciting particle. Particles with large d$E$/dx will result in greater densities of triplet states, and therefore greater amounts of delayed fluorescence in a pulse. The

d$E$/dx of a particle refers to the amount of energy $E$ that particle deposits over the distance it travels. Neurons are much heavier particles carrying a lot of energy and interact with matter often during their relatively shorter range. Gamma-rays travel very far before depositing their energy. The drastic difference in d$E$/dx between neutrons and gammas is then what allows for their differentiation as incident neutron particles result in high density triplet states while incident gammas result in lower triplet state densities.

### II.1.3 Waveform Processing Methods

The amount of delayed light produced can then be used to differentiate neutrons and gamma-rays, known as pulse shape discrimination (PSD). By taking the integral of the tail (retrieving its area) and comparing that as a ratio over the total integral of the pulse (the total pulse area) one obtains a value which is distinctly different for most neutrons and gamma-rays. Because the light from gamma-rays decays more quickly, the gamma-ray PSD values, also known as the tail-to-total (TTT) ratios, will typically be lower than that of the neutron PSD values. This comparison of light decay is seen in Fig. II.5, as the gamma-ray pulse decays much more quickly than the neutron pulse. These pulses do not show drastic differences, as the prompt components decay around the same rate, but the delayed component of the neutron pulse trails off more slowly and differently.

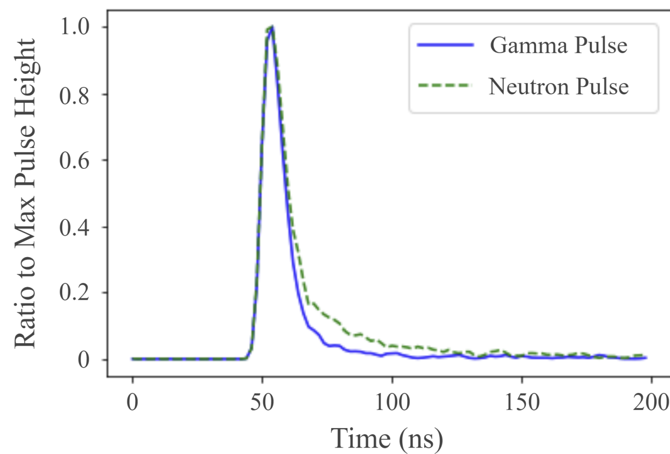



Figure II.5: A gamma-ray pulse and neutron pulse detected from the same fission event of $^{252}$Cf, collected using two EJ-309 organic scintillation detectors.

The travel time of a particle, called the time-of-flight (TOF), is another relevant value to understanding a particle's characteristics. TOF measurements can be used to determine important characteristics of detected neutron particles, such as its incident energy. They can also be used to differentiate gamma-rays from neutrons. This is understood as gamma-rays travel at the speed of light while neutrons are relatively heavy particles that travel more slowly.

Pulse height distribution (PHD) histograms are important to nuclear engineering and nuclear physics. These measurements can be most effectively used to differentiate and identify sources present [21].

Constant fraction discrimination (CFD) is a method of electronic signal processing that can be used for detection of particle waveforms received by scintillation detectors. This allows the triggering event for particle detection to not simply exist as a threshold on pulse height. The CFD method, as detailed in Fig. II.6, takes each input signal and attenuates them by a factor $f$. The signals are also simultaneously inverted and delayed by a time $d$. The two signals are then added together to generate a bipolar pulse, from which the zero crossing can be extracted. The zero crossing is located at the instance where the shaped signal goes from below the baseline to above it.

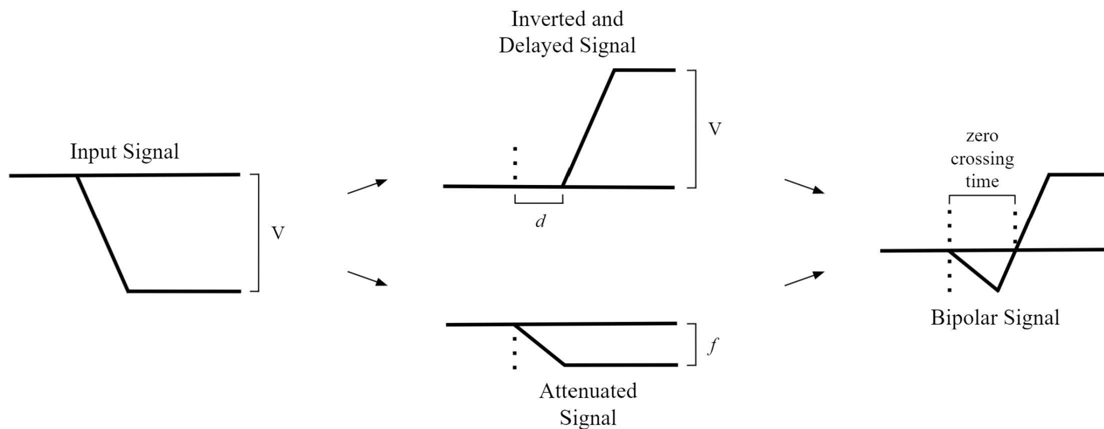



Figure II.6: Classical implementation of the CFD showing each step of the process.

A reference of the shaped signal is clearly shown in Fig. II.7. The time value at the instance

of zero crossing is known to be the timetag, which in our case is the instance in time at which a particle was detected.

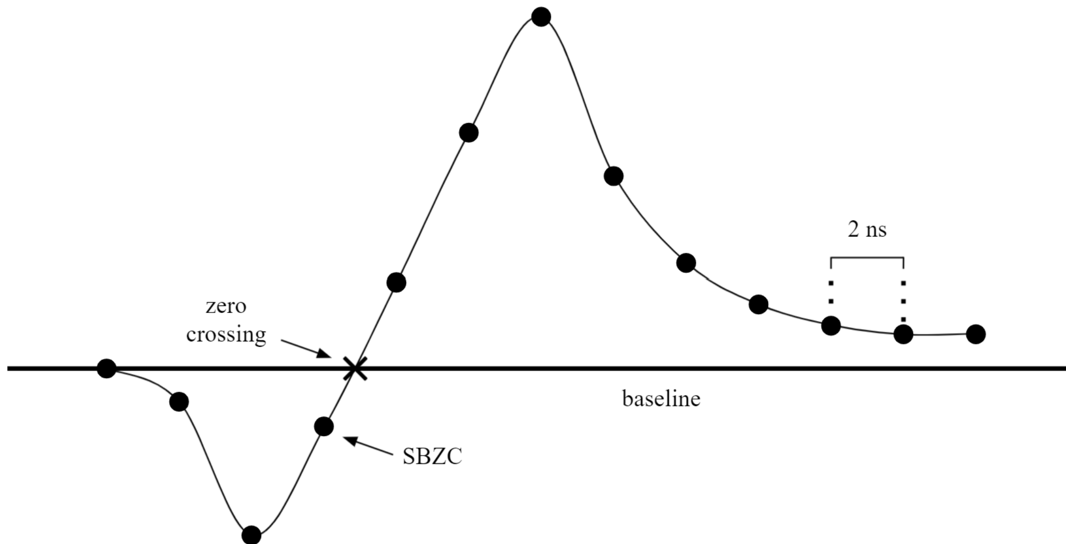



Figure II.7: Example CFD shaped signal for timing showing the zero crossing and SBZC.

To calculate the TTT ratios, or PSD values, windows of integration must be specified. These windows are dependent on 'gates' which the digitizer uses to segregate each particle waveform. These gates are shown in Fig. II.8. Both the short and long gates begin at a moment in time equal to the timetag minus the gate offset. This must be done because each electronic pulse is not recorded as a particle waveform until the moment of the timetag. The gate offset basically just corrects for this fact, moving the start of integration to be before the pulse has begun to rise. The start of integration for the total area then begins at the start of the short and long gate and stretches until the end of the long gate. The start of tail integration begins at the end of the short gate and continues until the end of the long gate. The area values are also dependent on the baseline which is set for each pulse, so an accurate baseline value must be determined.

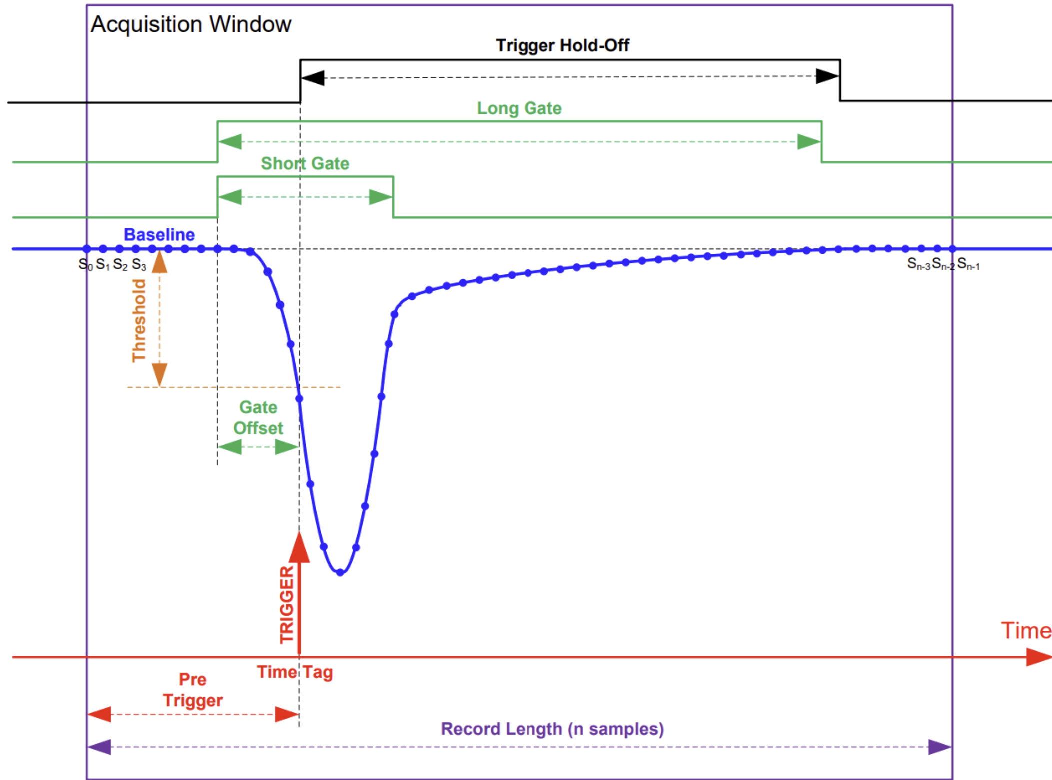



Figure II.8: Acquisition window of particle waveform, displaying the moment of triggering, along with long gate, short gate, gate offset, pre-trigger, trigger hold-off, record length, and all other relevant values. Reprinted from [5].

## II.2 Previous Works

The differentiation of gamma-ray and neutron pulses using PSD analysis and TOF labeling is discussed in a 2018 publication R. Wurtz, B. Blair, et al. [22]. These methods are important as much of my research and the RadSigPro code focuses on them. The publication used test data collected from a $^{252}$Cf source via a TOF setup resulting in a low contamination rate. A similar setup and even the same source is used in my data collection, and the differentiation of gammas and neutrons based on their travel times is a point which is greatly discussed and used in my research. The publication further notes that one of the most promising applications for digital pulse processing algorithms is FPGA implementation for real time event characterization [22].

FPGAs have become increasingly useful in the field of radiation detection. Especially when it

comes to on-the-fly processing tools, FPGAs are comparatively flexible, cost effective, easier to design, and have fast processing times [23]. Many publications have discussed the implementation of FPGAs. In one such publication by Martin Klein and Christian Schmidt et al., [24], a similar scope to our paper was provided. It details the CASCADE detector, a solid converter gas detector using several gas electron multiplier (GEM) foils as charge transparent substrates to carry solid $^{10}$B layers, which is designed for high-flux neutron applications ($10^7$ n/cm$^2$s) with high demands on the dynamic range, contrast, as well as, background. The CASCADE detector uses an ASIC electronic front-end paired with an adaptable integrated FPGA data processing unit to provide high rate capacity and real time event reconstruction [24]. In their set configuration, each module was able to detect $10^6$ neutrons per second with 10% dead time. The FPGA recorded data at 10 MHz (each data counter had a depth of 32 bits) extracting pulse height values.

# III. RADSIGPRO 1.0

This chapter details the construction of the RadSigPro code in section III.1, the experimentation process by which events were collected in section III.2, and the resulting histograms from RadSigPro processing collected data on the CPU in section III.3. The CPU version of the RadSigPro code is displayed in Appendix A, and the python file itself is included as a supplementary material along with this thesis.

## III.1 RadSigPro Development

This section discusses the developed method used to process raw pulse data and present tallied measurements. These tallies, including PHD, PSD, and TOF, are all relevant to nuclear measurements data analysis. The initial processing from raw data format is conducted with the Raw_Pulse_Correction (RPC) function. To understand the development of the pulse processing functions the files output by CoMPASS must first be characterized.

### III.1.1 Raw Pulse Correction Function

I will first explain the resulting data provided by CoMPASS from the experiment. The output files are given in comma separated value (CSV) format, and structured as shown in Table III.1. The delimeter used is a semicolon. The TOF measurement produces two files each pertaining to a scintillator.

Table III.1: Example format of CoMPASS's output CSV files, displaying each pulse's: instance of detection (timetag), calibration energy (calib_energy), channel selected for energy calibration (energyshort), a flag displaying a code which indicated any errors in the pulse data (flags), and the sample data which make up each pulse (samples).

| TIMETAG [ps] | CALIB_ENERGY [keV] | ENERGYSHORT [channel] | FLAGS | SAMPLES [ADC units] | | |
|---|---|---|---|---|---|---|
| 407 | — | — | — | 8142 | ... | 8130 |
| 5606 | — | — | — | 8138 | ... | 8134 |
| 12374 | — | — | — | 8141 | ... | 8152 |
| ... | — | — | — | ... | ... | ... |
| 980074 | — | — | — | 8140 | ... | 8142 |

Each pulse is given a timetag that signifies the point at which the pulse was officially tallied, the zero crossing. The timetags are given in ps and are accurate by $\pm 2$ ps due to the digitizer used. The calib_energy, energyshort, and flags data are not relevant to the analysis conducted and are therefore not used. The calib_energy shows the calibration energy value in keV corresponding to the channel number given by the energyshort column. The flags column gives a number code references potential flaws that CoMPASS found with the given pulse. All pulses with properly recorded timestamps have the flag 0x4000. The data labeled samples consists of the values which make up the pulse. A plot of these raw values can be seen in Fig. III.1. These values are given in ADC units, known in CoMPASS as the least significant bits (LSB). Each pulse consisted of 496 sample values recorded every 2 ns, with the sampling rate of 2 ns coming from the digitizer us.

The RPC function is used to process the raw data output. Each row of the raw data includes all relevant pulse data. As such the code manipulates one row after the other. First the raw sample data (as displayed in Fig. III.1) is isolated, inverted to a positive pulse, normalized with the baseline value to 0 on the x-axis, and converted from ADC units to millivolts (mV). The resulting appearance of the pulses can be observed in Fig. II.5. Time data is created, corresponding to each sample point,

from the given timetag. The time data displayed is in ns and the pulse data in mV.

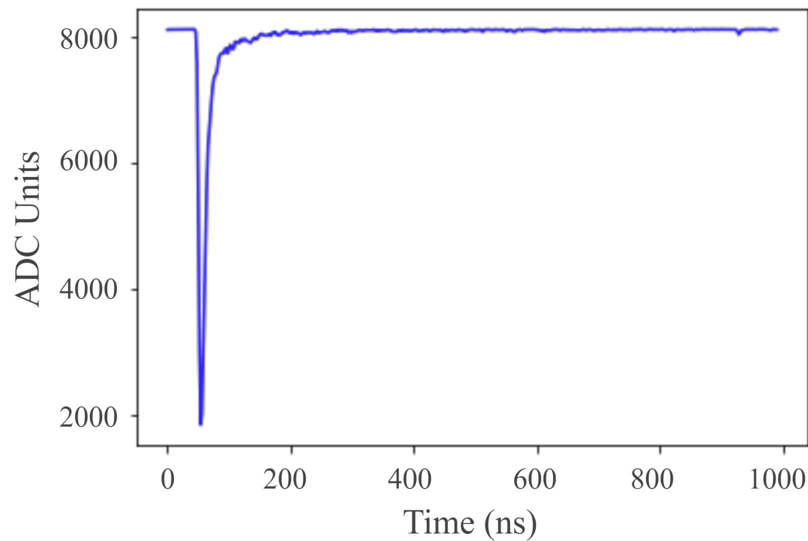



Figure III.1: Example raw pulse output by CoMPASS, detected using $^{252}$Cf source.

The following explains the structure of the python function itself. The inputs also include: the baseline used in CoMPASS for data collection, the number of pulses to be put in each CSV output file, and the parameters of CFD delay value and attenuation factor used for particle detection. The baseline and number of pulses have already been explained, while the CFD delay and attenuation factor both pertain to the method from which CoMPASS determines what voltage spikes are actually pulses and not noise. This method will be explained further on in this section. The baseline value must be specified, however, the other values have preset inputs: pulses_per_csv = 100000, cfd_delay = 6, and attenuation_fraction = 0.25.

The exact processing involves each sample value being made negative and then added to the baseline. This inverts the pulses and normalizes them to the x-axis. The units are also converted in this step from ADC to mV units. Now that the pulses have been preliminarily processed, just as would be done in CoMPASS, a new baseline is subtracted from each pulse. This second baseline is subtracted which allows the processed data to more accurately construct a PSD plot. An average of 5 data points (10 ns) located before the pulse begins to rise are used as this baseline. The waveforms

recorded using the technologies listed have a rising edge of prompt light which takes between 4-6 ns to reach its peak. With this information in mind, the 5 data points are selected from at least 6 ns before the pulse's maximum, assuming no other highly varying values populate that data range. This process is referred to as the baseline freeze method, which sets a unique baseline value for each pulse.

The next part of the code directly replicates the CFD method, displayed in Fig. II.6, used by CoMPASS to filter pulses. Its purpose is to allow the addition of accurate time values for each recorded pulse's samples. This exact instance is recorded as the timetag in ps. For the digitizer in use, DT5730, the precision of the timetag is 2 ps. Interpolation can be used to determine the time value for the zero crossing, as seen in Fig. II.7. Then a time value can be extracted corresponding to the closest sample before the SBZC. In the python code, the pulse data is manipulated just as in Fig. II.6, with the fraction $f$ being equal to 0.25 and the delay as set earlier.

### III.1.2 Measurement Tallies

The three main measurement tallies of PHD, PSD, and TCC histograms result from four functions, not including the RPC function used to process the raw data.

The PHD_Plot (PHDP) function extracts the maximum values of the processed data and creates histograms with those values in ADC units, mV, and keVee if there has been an energy calibration. The PHDP function requires a raw output data file, along with a set x-axis maximum, and a baseline. All of these inputs must be specified directly as they impact the resulting plot. Because extraction of pulse heights not need much data manipulation, the PHDP function uses unprocessed raw data files.

The PSD_Analysis (PSDA) function calculates the ratio of each pulse's tail area over the pulse's total area as found through integration, and then plots them as a histogram. The PSDA function's output data file is formatted same as the RPC function's output except the time and pulse columns are accompanied by a third column which repeats that pulse's PSD value. The inputs for the function include: a Corrected Pulses gamma file, a Corrected Pulses neutron file, a value for minimum pulse height allowance, a value for maximum pulse height allowed, the current detection setup's values for the long gate, the pregate, the short gate, the CFD delay, and the attenuation factor. The gamma file

and neutron file have to be Corrected Pulses files, which were processed with the RPC function, and therefore have the naming convention of its outputs. For an actual PSD comparison to be plotted, both the inputs "gamma_file" and "neutron_file" must be filled. If only one file is input, a histogram will be created with just its PSD data included. All inputs other than the gamma file have predefined values, the neutron file's value being a list which reads 'Not a File'. The rest of the preset values and their python name are: min_height_allowed = 0, max_height_allowed = 1000, long_gate = 360, pregate = 50, short_gate = 70, cfd_delay = 6, attenuation_fraction = 0.25. The minimum and maximum preset height ranges are set such that all pulses will naturally be allowed, the maximum possible recorded height being that of 1000 mV. The rest of the preset parameters were all chosen as the values used for our detection system.

The TOF histograms are created with one function, however the complete TOF analysis extends to two separate functions. These functions are the TOF_Analysis (TOFA) and TOF_Comparison (TOFC). The TOFA function subtracts the timetag value of one particle from the timetag value of its pair, and then plots those resulting values on a histogram. It also outputs a TOF PSD CSV data file showing pulse pair PSD values and TOF values. The TOFC function uses input parameters to selectively limit pulse pairs and display the ones that pass as plotted waveforms. It also creates two Discriminated Pulses data files which consist of the pulse pairs that satisfied the input parameters.

## III.2 Setup and Data Acquisition

The technology and environment of my experiment, the calculation of TOF and time cross-correlation (TCC) values, and the methodology of particle segregation using TOF and PSD values is detailed. Then the process of data acquisition is listed explicitly.

### III.2.1 Experiment Setup

The experiment uses a 35-kBq $^{252}$Cf, two ELJEN-309 scintillators, a CAEN DT5730 digitizer, a CAEN DT1471ET 4 Channel HV supply, and CAEN's CoMPASS Software. The EJ-309 organic scintillation detectors include a cylindrical scintillation liquid housing of 7.62 cm diameter by 7.62 cm height, a 17 cm long Hamamatsu R11833-100 photomultiplier tube, a negative HV voltage

divider, light-tight aluminum housing, mu-metal shielding, and back-cap with three connectors (SHV-neg.HV, BNC-signal, BNC-dynode). The two ELJEN-309 scintillators are placed 110 cm apart from each other on stands. The cable connections for the setup are shown in Fig. III.2.

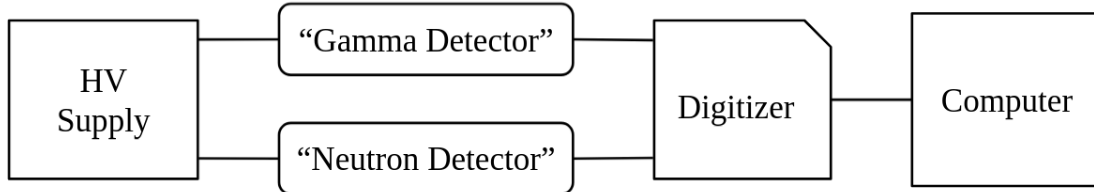



Figure III.2: Detection setup technology connections configuration, displaying the HV, both EJ-309 detectors, the digitizer, and the computer with CoMPASS software.

CAEN’s Multi-Parameter Spectroscopy Software (CoMPASS) was used for data acquisition (DAQ) of the pulses. The voltage range was set at 2 V ($\sim$16,000 ADC units), however, only half of that is used. This is because the digitizer is set to record both positive and negative pulses even though I only receive negative pulses for the data collection, meaning the baseline level for recorded pulses actually exists around 8,000 ADC units. This can be seen in Fig. III.1, as the negative pulse extends down from around 8,000 ADC units. CoMPASS allowed for the manipulation of parameters in particle detection for the purpose of particle differentiation, namely applied PSD value cuts limiting the types of particles recorded. These cuts are implemented such that one detector records almost entirely gamma-rays while the other records almost entirely neutrons. The image in Fig. III.3 shows each scintillator’s label signifying which particle it is set to mostly record.

A distance of 55 cm was chosen as seen in Fig. III.3 largely due to restraints of the detection environment. The distance between detector and source is important as it influences particle TOF. To understand this connection, lets take the classical physics kinetic energy formula shown in Equation III.1. This formula can be transformed to solve for velocity, which in turn equals distance over time, as shown in Equation III.2.

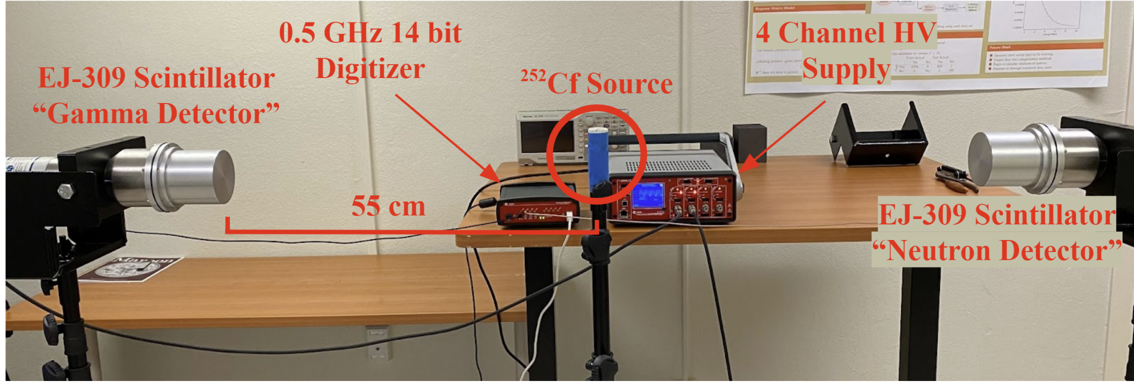



Figure III.3: Image of detection setup including source, equipment, and scintillator specifications.

$$E = \frac{1}{2}mv^2 \tag{III.1}$$

$$v = \frac{d}{t} = \sqrt{E}\sqrt{\frac{2}{m}} \tag{III.2}$$

The TOF of a particle will be impacted by both the distance *d* it travels, the energy *E* it begins with, and finally its mass *m*. When spontaneous fission occurs, particles are emitted in various directions from the source. $^{252}$Cf will emit both gamma-rays and neutrons from the same spontaneous fission event, thus with detectors placed on either side of the source and appropriate PSD cuts, a gamma-ray can be recorded in one and a neutron in the other. TCC values can be calculated by subtracting the times at which particles were recorded by one detector from the times at which particles were recorded by the other detector. When the two recorded particles consist of a gamma-ray and a neutron, the TCC value will then correspond to the TOF value of the neutron minus that of a gamma-ray. The TCC values recorded and referenced in this paper from here on out are directly equal to the timetag of the particle detected by the “gamma-detector” minus the timetag of the particle detected by the “neutron-detector”. The timetag is the instance at which the DAQ software determined a particle was detected. TOF values of neutrons are then TCC values from true neutron-gamma pairs with the time value corrected by adding back the TOF of a gamma-ray.

The PSD cut applied during measurement does not entirely filter between gammas and neutrons

so neutron-neutron and gamma-gamma pairs are also recorded. These neutron-neutron and gamma-gamma pairs result in a peak in the TCC value at approximately 0 ns, because both detectors are equidistant from the source. The unwanted pairs contribute to a small but uniform background throughout the TOF distribution. With this in mind, proper neutron-gamma pairs' TOF values should be able to be distinguished from the distribution of bad pairs which exist around the 0 ns mark. This process of separation is the reason why TCC values are important. To find the TCC window which should be expected from neutron-gamma pairs, the energy range of the neutron energies detected must be calculated. Neutrons emitted from the spontaneous fission of $^{252}$Cf fall mostly in the energy range of 0 to 10 MeV; however, from earlier experiments I had calculated that the setup records neutrons only as low as around 871 $\pm$ 25 keV. Using this energy value and assuming a 10 MeV high energy limit, the expected neutron TOF range as well as gamma-ray TOF can be determined. With the mass of a neutron and the distance 0.55 m, the neutron TOF range is calculated. These calculations result in the expected maximum recorded neutron TOF of 43 $\pm$ 1.2 ns, minimum recorded neutron TOF of 12.57 $\pm$ 0.03 ns, and gamma-ray TOF of 1.83 ns. From here I can estimate that the TCC histogram resulting from the measurement can include both neutron-gamma and neutron-neutron pairs from the TCC values of 10.7 $\pm$ 1.2 ns to 30 $\pm$ 1.2 ns.

### III.2.2 Data Acquisition

An energy calibration is carried out using a $^{137}$Cs check source. The detection setup is held constant with power being supplied to all detectors. Two 15-minute runs with $^{137}$Cs placed 5 cm from each scintillator are collected, and PHD plots of the data created. With the PHD plots and the known energy spectrum of $^{137}$Cs, the Compton edge is used to gain an energy conversion value from the digitizer's 'ADC units' to 'keVee' for both scintillators. After both scintillators have their energy calibration value, a $^{252}$Cf source is placed in the middle of each scintillator, as seen in Fig. III.3. A 10-minute test run is conducted so that preliminary PSD plots can be observed and each scintillator's waveform baseline can be determined. The PSD plots are needed for selecting the values of the PSD cuts and the baseline values are needed to set a fixed baseline throughout the run. The resulting PSD plots are visually analyzed such that the "gamma-detector" is limited to

only recording its gamma-ray distribution and the “neutron-detector” is limited to only its neutron distribution. With each new experiment recurring PSD value limits were found to be from 0.02 to 0.12 for the “gamma-detector” and from 0.1 to 0.2 for the “neutron-detector.” For the purpose of allowing offline TCC cuts, the PSD limits were set such that CoMPASS allowed each detector to record particles which had PSD values within the valley of the PSD plots two distributions. This valley can then be later split based on the TCC values of each recorded pulse pair. To determine the average baseline values of each scintillator, the output data is processed with an edited version of the Raw_Pulse_Correction (RPC) python function I developed.

### III.3 Results from RadSigPro 1.0 Implemented on CPU

This section documents the results of each function detailed in section III.1, explaining the histograms and analyzing the pulse pairs with PSD and TOF measurement cuts. These cuts indicate the usefulness of applying multiple filters to data in order to differentiate gamma-ray and neutron particles detected.

#### III.3.1 PHD

PHD histograms output by the PHDP function can be seen in Fig. III.4. The maximum heights of each pulse in keVee is displayed for both the neutron and gamma-detector.

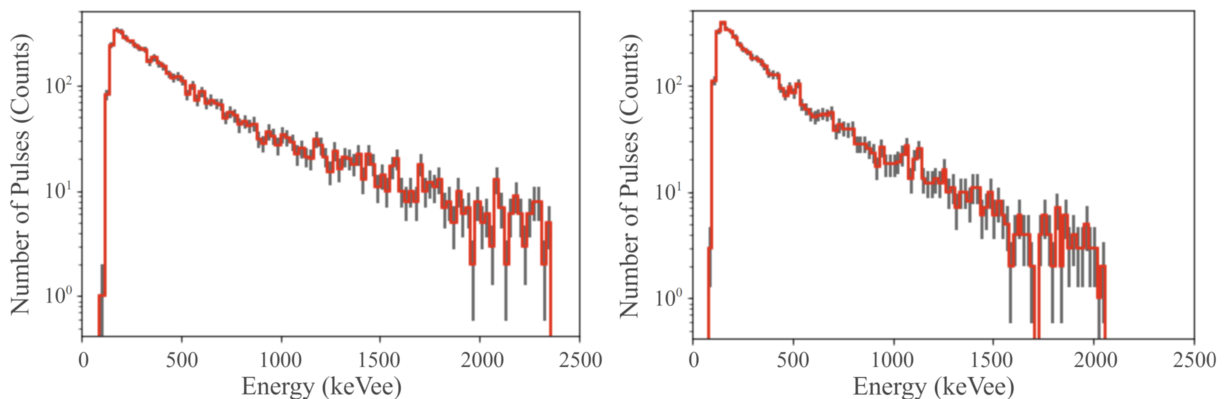



Figure III.4: PHD histograms of the specified gamma-detector (left) and neutron-detector (right), made from collected $^{252}$Cf pulses. The error bars in gray represent the variance of each bin.

Each histogram contains the same number of total pulses, it is clear that on average the gamma-ray particles had higher pulse heights. The simple average pulse height from the gamma-detector data set was 762 keVee while the neutron-detector data set was 685 keVee.

The gray error bars seen in Fig. III.4 represent the propagation of the stochastic error which occurs from the random nature of counting statistics, equal to the standard deviation. The most probable pulse height for the specified gamma-detector was 167 keVee, consisting of 330 pulses of the total 5201 and having a stochastic percentage uncertainty of 6%. The most probable neutron-detector pulse height was 145 keVee with 381 pulses and a stochastic percentage uncertainty of 5%.

### III.3.2 PSD

The PSD histogram output by the PSDA function can be seen in Fig. III.5. There is a clearly visible overlap where some neutrons were detected by the gamma-detector and some gamma-rays detected by the neutron-detector. Because detectors record particles as pairs, some of the pairs will consist of two neutrons or two gamma-rays. In order to further segregate types of particle pairs, more processing must be conducted using PSD cuts.

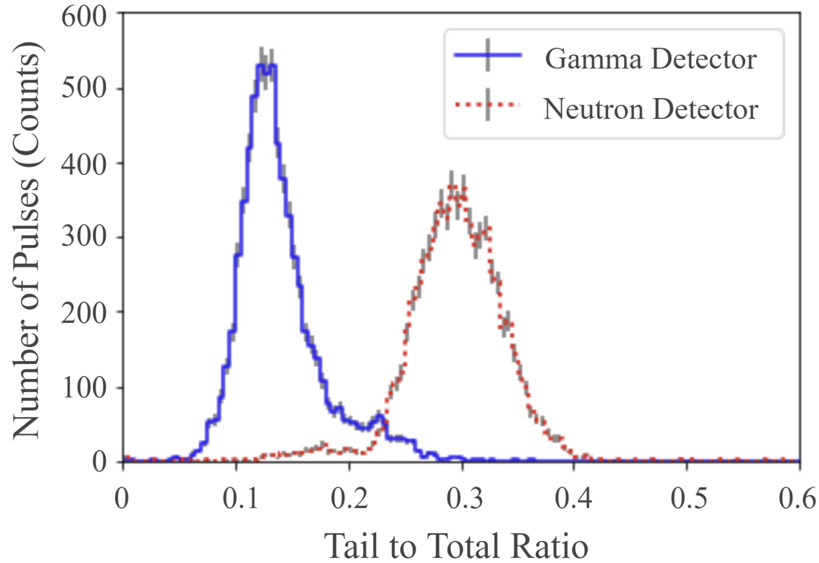



Figure III.5: PSD histogram from collected $^{252}$Cf pulse pairs, processed with RadSigPro, and presented for two detectors set to record either gamma-ray or neutron particles in a TOF measurement based on the incident particles PSD value. The error bars shown in gray represent the variance of each bin.

The average PSD values are skewed by the existence of overlap as seen in Fig. III.5. The peak of the gamma-ray distribution, seen on the left side of Fig. III.5, occurs at 0.125. The peak of the neutron distribution, seen on the right side of Fig. III.5, is located at the tail-to-total ratio of 0.285.

The stochastic error in this data, represented by gray error bars (one standard deviation) seen in Fig. III.5, occurs from the random nature of counting statistics. The gamma-ray distribution peak of 0.125 consisted of 421 pulses, and had a stochastic percentage uncertainty of 4.9%. The neutron distribution peak of 0.285 consisted of 295 pulses, and had 5.8% stochastic percentage uncertainty.

### III.3.3 TOF

The TOF histogram output by the TOFA function can be seen in Fig. III.6.

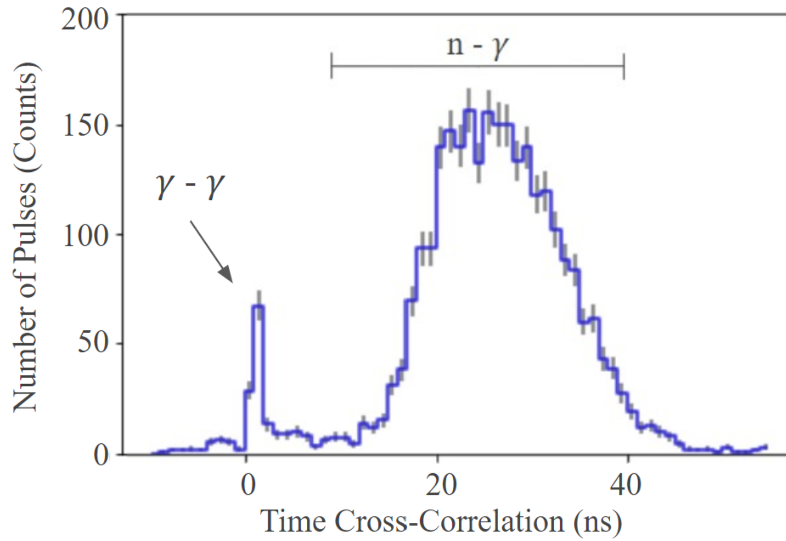



Figure III.6: TOF histogram from collected $^{252}$Cf pulse pairs, or TCC events, including additional indicators which highlight the pulse pair identities which make up the histograms features. The error bars in gray represent the variance of each bin.

The gamma-gamma doubles peak occurs at 0 ns. This is because gamma-rays travel at the speed of light, so their TOF will not vary greatly over the distance of 55 cm. These gamma-gamma doubles are visible in Fig. III.6. The neutron-gamma pairs were explained previously. The range of neutron TOFs was calculated previously to be around 13 - 43 ns, so with the gamma-ray TOFs calculated the TCC range of neutron-gamma pairs would be about 10 - 40 ns. Using this estimated neutron-gamma range, a TOF cut can be applied to the data sets only keeping pulse pairs with a TCC between 10 - 40 ns. The resulting data sets can then be made into PSD histograms again and displayed as in Fig. IV.1. The differences in this Fig. as compared with the previously displayed PSD histogram, Fig. III.5, is that it does not include much overlap between neutron and gamma distributions. Since the TOF cut involved a minimum TCC of 10 ns, the gamma-gamma doubles which were all distributed around a TCC of 0 ns were eliminated resulting in very little gamma-ray overlap. Neutron-neutron overlap is still not completely eliminated, as the TCC of neutron-neutron doubles can still fall in the 10 - 40 ns range. While these doubles do exist in the data set, it can be visibly seen in Fig. IV.1 that they only account for a small fraction.

The stochastic error in the TOF histogram occurs from the nature of counting statistics. The error bars seen in Fig. III.6 represent the stochastic uncertainty. The error bar values equal the standard deviation, being calculated by taking the square root of each bin value. The TCC values of 0 ns, presumably consisting mostly of gamma-gamma doubles, is made up of 164 of the 5201 total pulses and has a stochastic percent uncertainty of 7.8%.

# IV. NEUTRON-GAMMA LABELS FOR SUPERVISED MACHINE LEARNING

This chapter explains the process used to provide labels to a $^{252}$Cf spontaneous fission data set of neutrons and gamma-rays in section IV.1 and then notable results from the data sets use in supervised machine learning for allowing better particle determination in section IV.2.

## IV.1 Data Labeling

In order to label the detected events as either neutrons or gamma-rays, PSD and TOF cuts are used in conjunction. As previously discussed, the TOFs of neutrons and gamma-rays will differ noticeably given the detection environment present for data collection.

The data set collected as discussed in section III.2 was used, being processed by RadSigPro. First the raw data was processed by the RPC function, and then the PSDA function to generate PSD values for each pulse. Taking the neutron-gamma TCC range, a TOF cut was applied to the data set using the TOFA function only keeping pulse pairs with a TCC value between 10 - 40 ns. This eliminated all gamma-gamma doubles present along with some of the neutron-neutron doubles. The resulting data set's PSD histograms is displayed in Fig. IV.1.

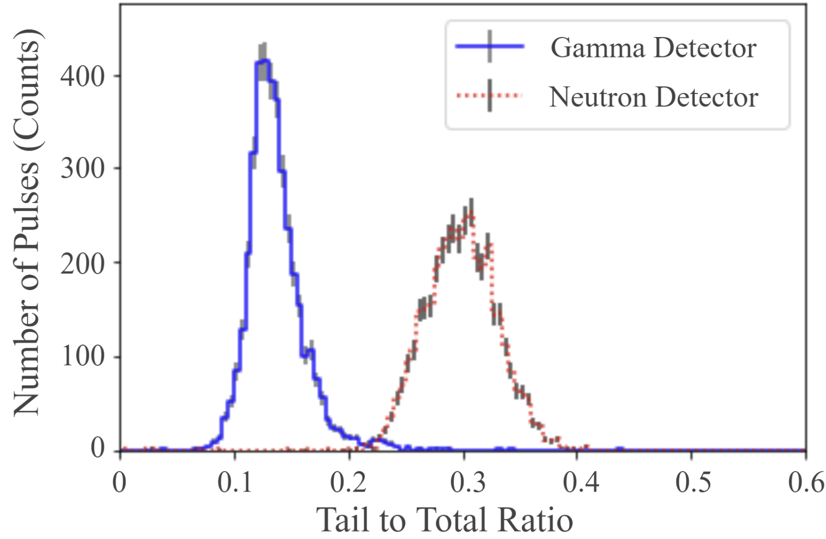



Figure IV.1: PSD histogram of time-cross-correlated pulse pairs from $^{252}$Cf, whose TCC fell in the range of 10 - 40 ns. The error bars in gray represent the variance of each bin.

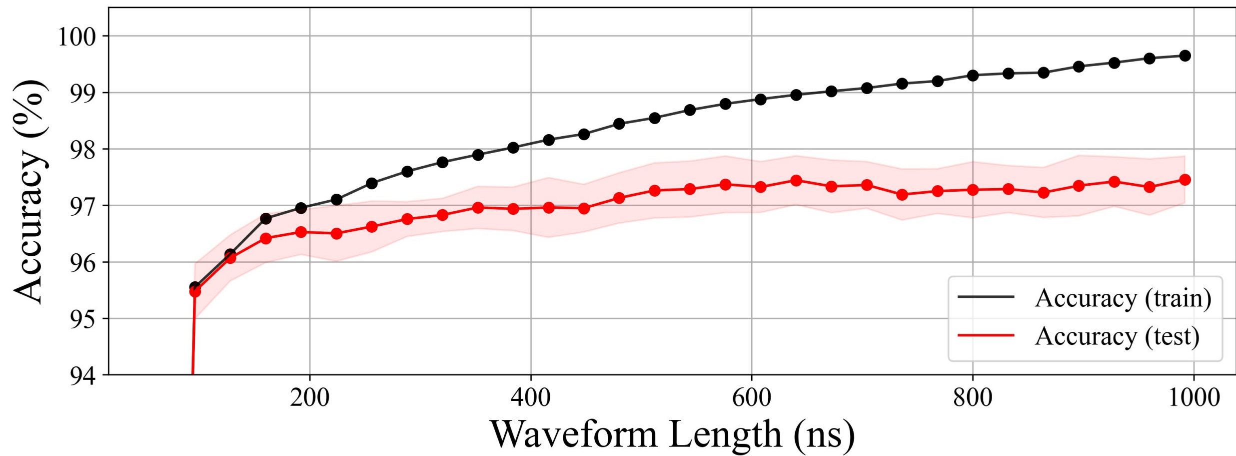
The differences between Fig. IV.1 and the previously displayed PSD histogram in Fig. III.5 (note that Fig. IV.1 and III.5 were made from the same initial data set, with Fig. IV.1 including the mentioned TOF cut) is that Fig. IV.1 includes little overlap in the valley between the known particle distributions. Specifically the overlap of gamma-rays, displayed as the parts of the "neutron detector" data which exist clearly under the "gamma detector" distribution, is reduced. Neutron-neutron overlaps will still be partly present, as the TCC of neutron-neutron doubles can still fall in the 10 - 40 ns range. When it comes to properly identifying detected neutral particles, the use of both PSD and TOF cuts is often applied.

In order to train and test the machine learning techniques, both unlabeled and labeled pulses needed to be provided. The unlabeled data provided consisted of the original $^{252}$Cf data set after having been processed with RadSigPro's RPC function. The labeled data provided consisted of all the pulse pairs from the initial set who passed the additional TOF cut.

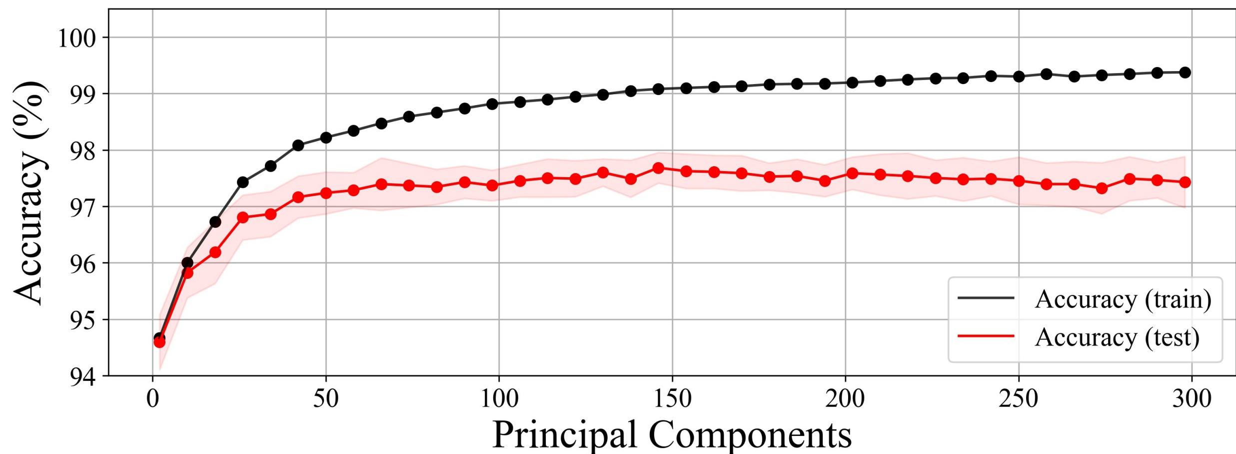
## IV.2 Supervised Machine Learning Results

The results shown display the accumulation of work done by myself and another researcher. While I provided the labeled data discussed in section IV.1, the other researcher carried out all further tasks including training and testing the machine learning techniques along with creating the figures which display the results of said training and testing. Support vector machines (SVM) using linear, radial basis function (RBF), and exponential kernel functions were trained and tested on data provided in two different forms. The first form being the labeled and unlabeled $^{252}$Cf data which I had processed and the second being the principle components extracted from the data provided. The results I will highlight come from the use of the RBF kernel.

Principle component analysis (PCA) transforms a high-dimensional data set into a set of principle components that exist in a lower-dimensional subspace [6]. The original data features are linearly combined to form the principle components, where each successive principal component has a lower variance and is uncorrelated to the principal components that came before. A reduction was made to the number of principle components through dimensionality reduction. The amount of information lost is minimized during this process because the components discarded contained less information than the components kept [25] [26]. The resulting accuracy of particle differentiation during the training and then testing is shown in Fig. IV.2.




(a) SVM using an RBF kernel on provided processed and labeled data set.




(b) SVM using an RBF kernel on principle components extracted from provided data set.

Figure IV.2: Mean classification accuracy of the SVM using an RBF kernel: a as a function of the length of the waveform used for training and b the number of principal components that were extracted and used for training. The red band surrounding the mean test accuracy represents 2 standard deviations of the test accuracy for all folds. Reprinted from [6].

For the waveform pulse training, a test classification accuracy over 95% is reached when training on pulses that cover only the first 92 ns of an event. As a classifier trains on pulse waveforms of increasing length, its accuracy when evaluated on a test set steadily grows. Shown in Fig. IV.2a, the training accuracy for the RBF kernel increases gradually and is nearly 100% when training on the full waveform. The misclassification results for the RBF when trained on waveforms was 2.35% for gamma-rays and 2.31% for neutrons [6].

Only 2 principal components are needed to accurately label an event over 94% of the time. The accuracy of each classifier evaluated on a test set continues to quickly rise as the number of principal components used for training increases until around 50 principal components. After this point, increasing the number of principal components that a classifier is fitted on has a very slight positive effect on the test accuracy. However, that positive trend is so small that it is almost non-existent. This is demonstrated in Fig. IV.2b as it was able to achieve a classification accuracy of 97.68% on the test set while only training on 146 principal components. The misclassification results when trained on principle components was 1.98% for gamma-rays and 2.27% for neutrons [6].

# V. CPU VS FPGA IMPLEMENTATIONS

This chapter discusses edits made to the RadSigPro to allow comparison with the FPGA implementation in section V.1 and then compares the results of RadSigPro CPU vs FPGA implementations in section V.2. The RadSigPro code was edited to allow better comparison with the FPGA version, this altered code is displayed in Appendix B, and the corresponding python file is included as a supplementary material along with this thesis.

## V.1 RadSigPro Edits

In order to implement the RadSigPro design onto an FPGA, a few changes were required. The RadSigPro CPU version replicates the CFD method, as was discussed in section III.1.1, by using the known location of the timetag in offline processing. This process cannot be implemented on the FPGA's programmable logic because calculating the location of the timetag during online processing was deemed to take too much time. To work around this, the RadSigPro code was edited to decide the windows of integration based on the location of the pulse's maximum height value, something which the FPGA can complete much more quickly. The integration windows remained constant, though the gate offset value was changed to account for the nearly 6 ns average time from the start of each pulse's rise to its maximum height. The integration method was changed from trapezoidal integration to simply summing the values in the range for each pulse. The FPGA was also limited by the amount of data it could store for each pulse data point, so the RadSigPro was edited to make the number of significant figures stored equal. Lastly, the waveforms were edited such that any data points which fell below the averaged baseline were set to 0. These changes were observed to not affect the overall plot distributions.

## V.2 FPGA vs CPU

The same input data set which was given to RadSigPro on CPU was also given to the RadSigPro FPGA design. In simulations, the FPGA computed the total and tail pulse areas within 5 ns of the arrival of the final data point used for accumulation and also captured the maximum height value of

the pulse within 2 ns of the arrival of pulse maximum's data point. The TCC took 1 clock cycle to calculate the difference. Hence, the theoretical processing time of the FPGA design to give out all the results is within 5 ns after the arrival of the last data point in an ideal scenario. However, this timing can be realized only if the FPGA is running on a clock period of 1 ns. The IP block will also have registers (memory) allocated to obtain the input data points from the digitizer. This memory access time will also contribute to the overall processing time of the FPGA system. The results from FPGA and CPU on the same set of data are presented and compared in this section. The results comparison for PHD, PSD, and TOF are presented in the following section.

### V.2.1 PHD

For the PHD histograms, the results from RadSigPro CPU and FPGA implementation are identical, as Figures V.1 and V.2 show when compared to Fig. III.4. The pulse heights were reported in terms of the ADC scale units which received whole numbers, so there was no difference between the results. Because of this the weighted average of the percent difference between the results from the FPGA and CPU is 0%.

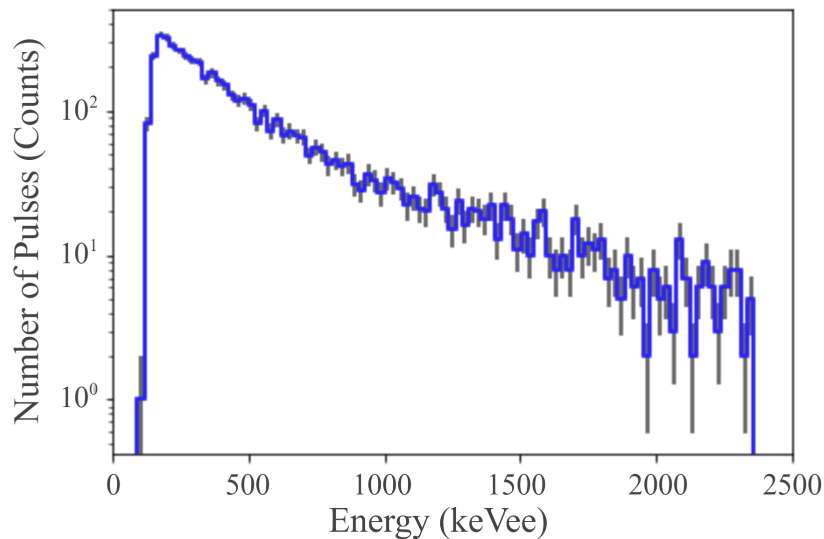



Figure V.1: Example gamma-ray PHD histogram of collected $^{252}$Cf pulses, using the FPGA implementation. The error bars in gray represent the variance of each bin.

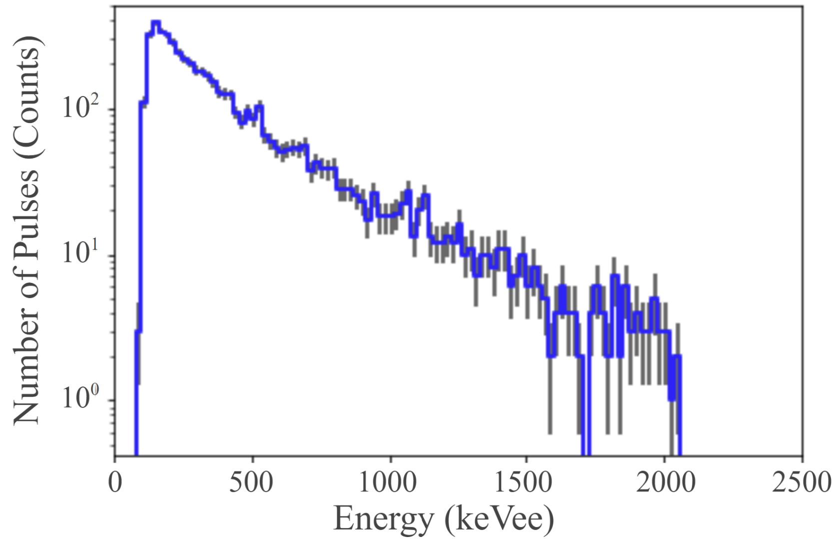



Figure V.2: Example neutron PHD histogram of collected $^{252}$Cf pulses, using the FPGA implementation. The error bars in gray represent the variance of each bin.

### V.2.2 PSD

As seen in Fig. V.3 below and Fig. III.5 previously, the PSD histograms from RadSigPro CPU and FPGA design are similar. The RadSigPro CPU code was altered so that it selects an integration window with reference to the pulse height index and only integrates using pulses in ADC units. The remaining difference comes from the ability of the software, the computer running RadSigPro, to store a longer decimal length of data as compared to the hardware, and the FPGA. This results in minute differences in the tail-to-total ratios. The mean absolute percentage error (MAPE) was calculated for the tail-to-total ratio results between the FPGA and CPU implementations for 200 bins. For the pulse data received from the designated gamma-detector there was a weighted percent difference of 0.458% and for the designated neutron-detector there was a weighted percent difference of 0.344%. Despite these existing differences, it can be seen that the FPGA is still capable of producing results very similar to what the software can produce.

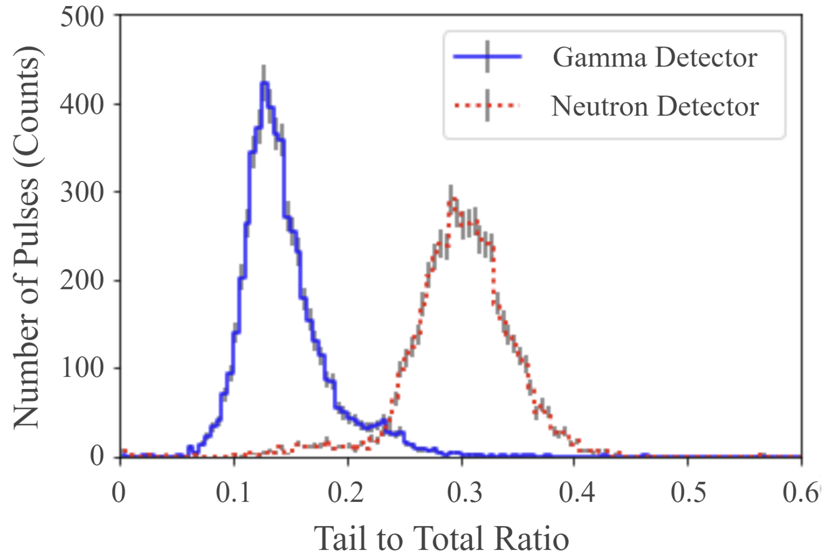



Figure V.3: Example PSD histogram of collected $^{252}$Cf pulse pairs, using the FPGA implementation. The error bars in gray represent the variance of each bin.

### V.2.3 TOF

From the histograms shown in Fig. V.4, it can be seen that the results of RadSigPro CPU and FPGA are the same. A one-to-one comparison of the difference in timetags from the FPGA and CPU was conducted and the results were identical. This is because the timetags used to calculate the TOFs are received from the digitizer as whole numbers. Therefore, the weighted average of the percent difference between the two methods of implementation is 0%.

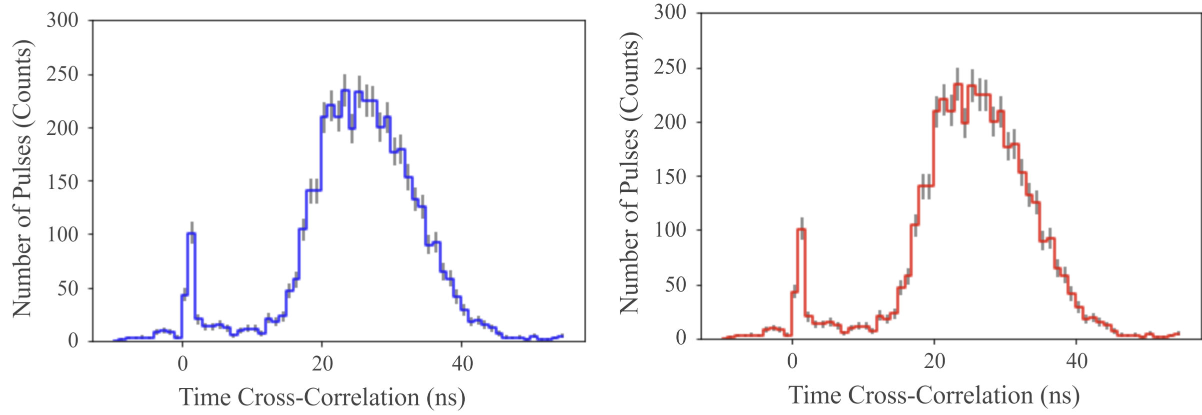



Figure V.4: Example TOF histogram of collected $^{252}$Cf pulse pairs - RadSigPro CPU implementation (left) and FPGA implementation (right).

# VI. RADSIGPRO VS COMPASS COMPARISON

This chapter details the ongoing process of directly comparing RadSigPro's results and CoMPASS's. Section VI.1 discusses the retrieval of CoMPASS's plot data while section VI.2 highlights the issues which currently limit the completion of comparison along with displaying initial results.

## VI.1 CoMPASS Plot Data

CAEN's CoMPASS DAQ software displays live PHD, PSD, and TOF histograms during data collection. Replicating these histograms while keeping knowledge of the specific particles which filled each bin was an important goal of the RadSigPro code. CoMPASS includes text files containing data from their PHD, PSD, and TOF plots, however, this data is in the form of the number of events per bin, with the total number of data points provided being the total number of bins. For the PHD plot data 16383 bins were used while the PSD and TOF plot data included 16384 bins. To properly achieve the comparison, my resulting tallies and CoMPASS's plot data would need to be put into histograms of equal binning, in this case consolidating the number of bins from the CoMPASS plot data. For the PHD plots the histograms chosen ranges need to be equal, being based on the ADC channels which CoMPASS's plot used. The PSD plots would have a range from 0 to 1. The TOF ranges would be again based on CoMPASS's plotting ranges.

An important factor is that CoMPASS plots are affected by the energy course gain value as set in CoMPASS's input parameters. The energy course gain works as a scaling factor, corresponding to the weight of a single bin in femtoCoulombs (fC). The higher the value of fC per bin, the more the spectrum is squeezed into smaller values. The course gain value depends on the input dynamics, having units of fC/(LSB $\times$ Vpp), where Vpp denotes the voltage range. This means that the real units are fC/channel, where the actual channel value changes as the voltage range changes. As explained by CAEN support, a single peak which occurs at channel 200 when the energy course gain is 40 fC/channel and dynamic is 2 Vpp will go to channel 50 when the energy course gain is set 160 fC/channel for the same input range of 2 Vpp.

For the data collection in question, the course gain was set to 10 fC/(LSB $\times$ Vpp) and the input dynamic set to 2 Vpp.

## VI.2 Current Discrepancies

Despite many attempts to perfectly replicate and compare PHD, PSD, and TOF tally results; noticeable differences in the PHD and PSD tallies persist. A comparison of the TOF histograms can be seen in Fig. VI.1. Because the timetags which CoMPASS provides in its output data file are being used to calculate the TOF values, the resulting values are the exact same.

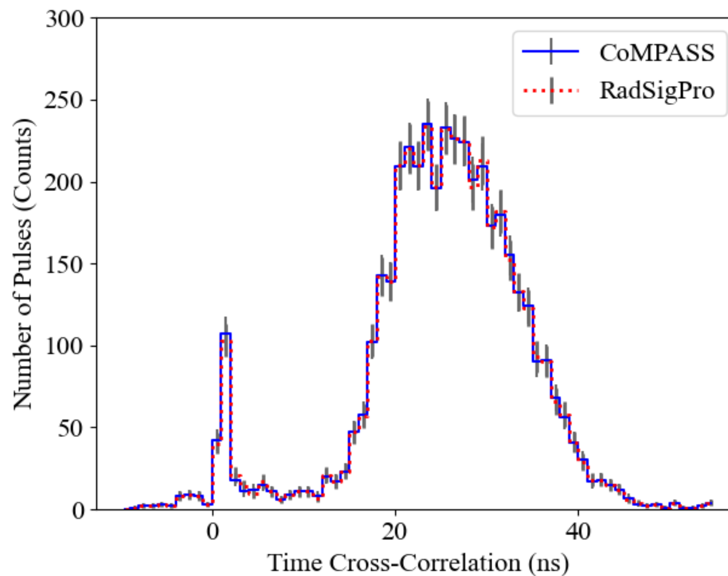



Figure VI.1: TOF histograms of collected $^{252}$Cf pulses, displaying the result of CoMPASS and RadSigPro's TOF tallies.

For the PHD histograms, this was especially concerning after a comparison was made between CoMPASS and an edited version of RadSigPro which insured that the baseline used to calculate pulse height (pulse height = baseline - pulse maximum height value) was the exact same for both CoMPASS and RadSigPro. Initial comparisons showed that while the plotted pulse heights shared similar distribution shapes, the actual data sets were misaligned from each other by some

conversion factor. This conversion factor was assumed to be coming from the energy course gain value mentioned earlier, so an energy calibration between both data sets was needed. A run was collected using a $^{137}$Cs source just as the energy calibration run was collected in section III.2.2. The location of the Compton edge in ADC units was found for both RadSigPro and CoMPASS's histograms being 1664 and 1408 respectively, and the ratio between those values was used to align both data sets around that known energy value. The resulting plot can be seen in Fig. VI.2.

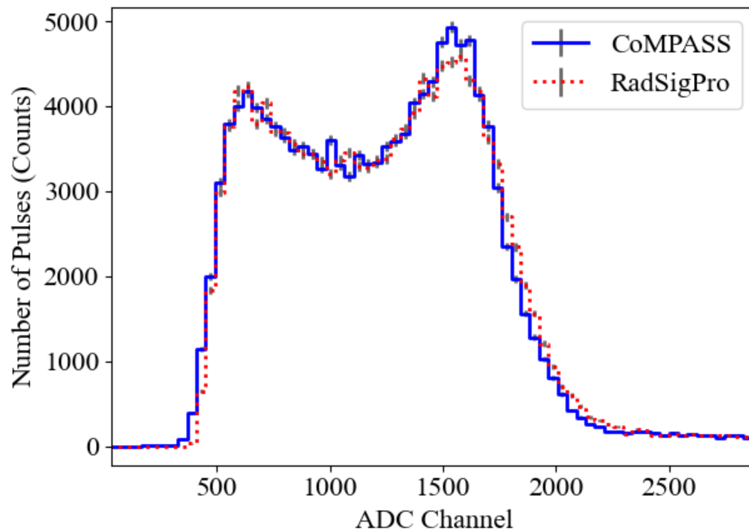



Figure VI.2: PHD histograms of collected $^{137}$Cs pulses, displaying the result of CoMPASS and RadSigPro's pulse height tallies using a baseline held at 8141 ADC units, after a calibration was applied to the data sets based off the Compton edge locations.

Unfortunately this did not solve the issue, as it can be seen that the two data sets show variance especially with the number of events located around the Compton edge. This difference persists even though as mentioned previously, the method of pulse height calculation via RadSigPro was created to very simply mirror CoMPASS's. This has lead me to believe that there is some additional processing which CoMPASS is carrying out on the pulses before tallying pulse height, which I am not yet aware of.

When attempting the comparison for PSD histograms, the baseline was held at a constant value for all events, the intervals of integration were made to be the same as CoMPASS's, and the CFD process of detecting pulses was recreated to insure the location of integration start would be the same. The resulting plots, as seen in Fig. VI.3, displayed extreme differences. Recalling from section III.2.2 that the data collected was by the first detector ("gamma-detector") was limited to only the pulses whose PSD value fell between 0.02 and 0.12 while the second detector ("neutron-detector") was held from 0.1 to 0.2, the CoMPASS data is shown to mirror the initial cut. The detected events which CoMPASS displays and which I received and ran through RadSigPro were all pulses which even with a constant baseline being held, had their resulting TTT ratios fall within the preset PSD cuts. Enforcing the same baseline through RadSigPro, for the "gamma-detector" 65.2% of the PSD values were 0 or negative and for the "neutron-detector" that value was 60.9%. That is why in Fig. VI.3 the RadSigPro data sets appear to have less events. When a number of pulses with 0 or negative PSD values were plotted, it became apparent that the result came from the constant baseline being too low for those specific pulses causing the tail areas to be calculated as an accumulation of negative values. It is unknown how CoMPASS, using the same preset baseline and integration windows on the same data set, was able to calculate proper PSD values for these pulses. Considering the pulses which did have a positive PSD value calculated, it can be further seen in Fig. VI.3 that RadSigPro's PSD values are higher than CoMPASS's for both neutrons and gamma-rays.

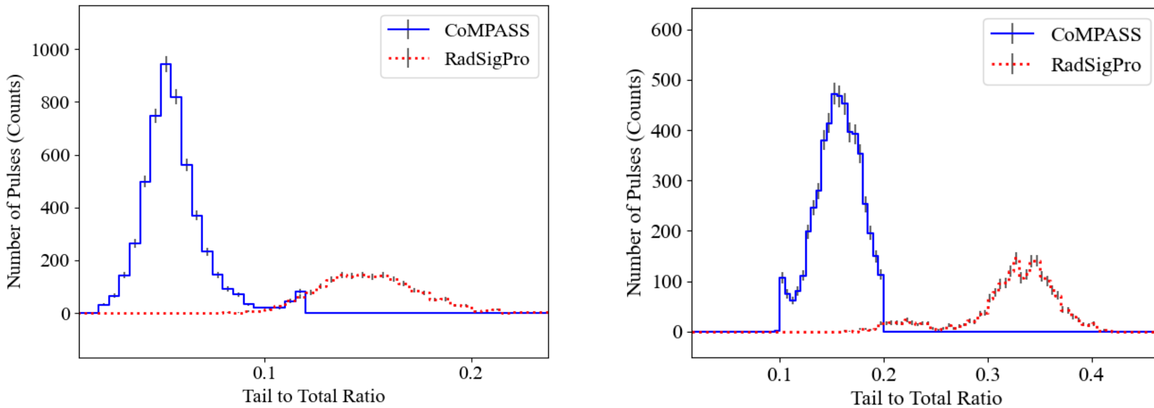



Figure VI.3: PSD histograms of collected $^{252}Cf$ pulse pairs displaying the result of CoMPASS and RadSigPro's TTT tallies using a baseline held at 8142 ADC units for the 'gamma-detector' (left) and a baseline held at 8127 ADC units for the 'neutron-detector' (right).

Because using the constant baselines resulted in 0 or negative PSD values, the same comparison was made however this time RadSigPro was allowed to use its method of creating an averaged baseline value unique to each pulse. Fig. VI.4 shows the results of this comparison, where the RadSigPro pulses with 0 or negative PSD values dropped to 0% for the "gamma-detector" data set and to 0.096% for the "neutron-detector" data set. While this change solved the issue of improper PSD values, the locations of the neutron and gamma-ray distributions remained consistently different between RadSigPro and CoMPASS. RadSigPro shows the distribution of neutron events from a little after 0.2 to around 0.4 while the distribution of gamma-ray events is from around 0.1 to a little before 0.2. An interesting point to note is that RadSigPro shows an increased separation of the overlap between neutron and gamma-ray distributions.

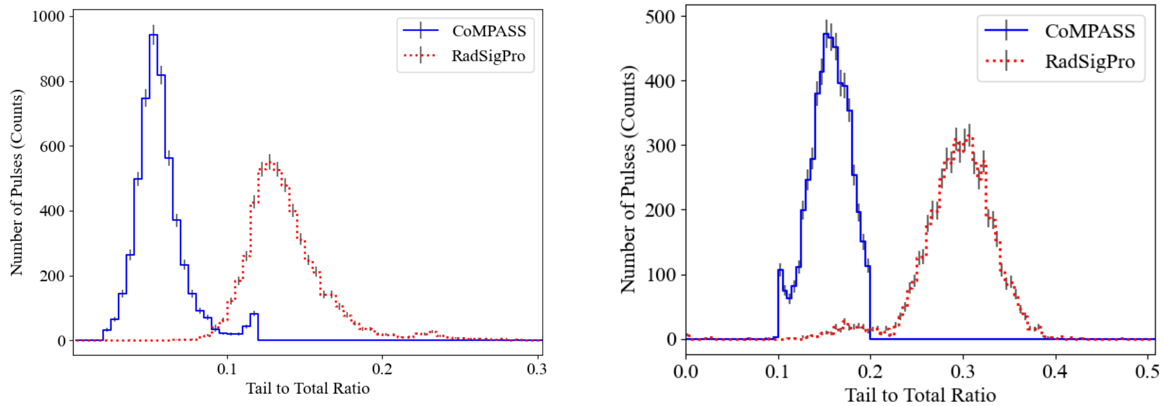



Figure VI.4: PSD histograms of collected $^{252}$Cf pulse pairs displaying the result of CoMPASS and RadSigPro's TTT tallies, where CoMPASS is using a baseline held at 8142 ADC units for the 'gamma-detector' (left) and a baseline held at 8142 ADC units for the 'neutron-detector' (right) while RadSigPro uses an averaged baseline value unique to each pulse for both the "gamma-detector" (left) and "neutron-detector" (right).

The consistent difference between data sets could occur again from the energy course gain which has been observed in PHD plots to reduce the ADC unit values for CoMPASS's data. However, it would be generally assumed that any reduction to the pulses height would affect all pulse values and therefore the TTT ratio would still remain constant. Further work must be done to identify the source of these discrepancies.

# VII. SUMMARY AND CONCLUSIONS

This chapter discusses the conclusions gained from the accumulation of work in section VII.1 along with potential future work in section VII.2.

## VII.1 Conclusion

The RadSigPro code processes pulse data sets allowing for their insightful analysis. The PHD histograms allow for analysis of incident particle energies, the PSD histograms differentiate neutron and gamma-ray pulses efficiently, and the TOF histograms indicate both key energy ranges of particles detected and shed light on the identities of the pulse pairs. This ability to conduct such analysis is important in the nuclear engineering and physics fields. A TOF experiment was conducted using a $^{252}$Cf and two scintillation detectors to collect particles, before the data was processed with the developed RadSigPro code implemented on a CPU. For the detector limited to record particles in the PSD range associated with gamma-rays, called the “gamma-detector", the average processed pulse height of the particles recorded was 762 keVee. The other detector limited to the PSD range associated with neutrons, the “neutron-detector", had an average processed pulse height of 685 keVee. The PSD histograms created from the processed data did not show much overlap between PSD values of neutron and gamma-ray pulses. The peak of the gamma-ray distribution occurred at 0.125 while the peak of the neutron distribution was at 0.285. These two distributions are clearly identifiable from each other, a necessity for PSD analysis, and the valley between them was sparsely populated. The TOF histogram displayed distinct distributions of both pulse doubles and neutron-gamma pairs. The visible TCC data allowed for calculation of particle energies. PSD cuts were applied to the TCC data, keeping only the pulse doubles whose TCC value existed in the possible neutron-gamma pair’s range. A PSD histogram created from this further filtered pulse data showed that most neutron and gamma-ray doubles had been eliminated. This illustrates the use of applying both a PSD and TOF cut to pulse data when attempting to properly determine pulse particle identities. The stochastic uncertainty in each of the histogram data sets was

represented as the standard deviation. The stochastic uncertainty of each histogram's maximum bin value ranged from 2.9-5.0%.

When training a model on pulse waveforms, it was found that a classification accuracy over 96% could be achieved with vectors that covered less than 100 ns, or around one tenth of the original pulse. To reach 97% however, the classifiers needed to be trained on much longer pulse vectors; around 400 ns for the RBF kernel. This indicates that the information relevant to determining if a pulse is either a neutron or gamma-ray is mostly found at the beginning of said pulse. As the length of the waveforms used during training increases, the accuracy on a test set also gradually increases. PCA extracts information about entire pulse. Each principal component is extracted from the the pulse in its entirety, so relevant information is not discarded when less-useful principal components are discarded. The first few principal components will always be the most useful because of the ordering inherent to PCA. As a result, SVM models trained on just two principal components could accurately classify pulses over 94% of the time. To achieve 97% accuracy, models with nonlinear kernels required fewer than 50 principal components for training. It was determined that extracting principal components from the waveforms increased the efficiency of the classifiers without impacting their effectiveness [6].

The results obtained from the FPGA were very similar when compared to those obtained with the RadSigPro code implemented on a CPU. The MAPE between the data sets was 0% for both PHD and TOF; the MAPE was within 0.344-0.458% for PSD. In the FPGA implementation, the theoretical processing time matched the expected 5 ns time. In simulations, the FPGA was found to tally pulse height within 2 ns of the latest pulse data point's arrival, while the area integrations were completed 5 ns after the 365 ns waveform datum has been received by the FPGA [7].

### VII.2 Future Steps

This section highlights directions of furthering the work discussed in this Thesis.

### VII.2.1 RadSigPro Usage

The RadSigPro pulse processing method was used in this work to process waveforms created by EJ-309 organic scintillation detectors. The processing method however can be applied more generally to any detected waveform, and is not limited simply to organic scintillators. The only requirement for such applications is that the parameters of RadSigPro be tailored to the specific shapes of the particle waveforms generated by a given sensor.

### VII.2.2 CoMPASS Comparison

To resolve the discrepancies hindering a direct RadSigPro vs CoMPASS comparison, a better understanding of how the energy course gain affects the raw pulse waveform data must be attained.

### VII.2.3 Online Data Processing and FPGA

To obtain more accurate timing results for the FPGA implementation, we need to validate the design using live data from a digitizer. Once that data movement between digitizer and FPGA board is implemented, this custom IP can be directly used to process the incoming data. On the other hand, the CAEN DT5730 digitizer is also an FPGA based digitizer and this IP can also be added as an additional functionality in the FPGA inside the digitizer.

### VII.2.4 Machine Learning

In the future we would like to improve our method by training and testing on data with smaller pulse heights corresponding to lower neutron and gamma-ray energies. We would also like to train and test this method on other independent data sets collected by the community.

### VII.2.5 Applications

The RadSigPro pulse processing method was developed and verified using EJ-309 organic scintillation detectors and a spontaneous fission source ($^{252}$Cf), which emits predominantly high-energy neutrons and gamma rays. Hence, for MC&A applications in advanced nuclear reactors, appropriate scintillation detectors suitable for the radiation field (fast or thermal neutrons, gamma rays, etc.) shall be selected. RadSigPro can be used wit any choice of radiation detector.

## REFERENCES


[1] R. Adams, Atomic show 248 - dr. pete pappano, vp fuel production x-energy, https://atomicinsights.com/atomic-show-24pment-x-energy/, accessed: 2022-05-04 (2020).

[2] H. Marshall, R. Supervisor, Stodilka, Mri-based attenuation correction in emission computed tomography, Ph.D. thesis (06 2012). doi:10.13140/RG.2.2.11594.80323.

[3] P. Ganesan, P. Joshi, R. Palit, Measurement of electron mass using compton scattering (12 2015). doi:10.13140/RG.2.1.2781.9280.

[4] S. Gupta, Y. Mao, Nano Scintillator-Book, 2020.

[5] CAEN, User Manual UM5960, COMPASS: Multiparametric DAQ Software for Physics Applications, accessed: 2022-05-04 (2021).

[6] P. Maedgen, B. Wellons, S. Prasad, J. Tao, Improving pulse shape discrimination in organic scintillation detectors by understanding underlying data structure, Nuclear Technology (2022). doi:10.1080/00295450.2022.2045533.

[7] R. S. Kumaran, B. S. Wellons, S. Prasad, High speed computation using an fpga with neutron-gamma scintillation detectors, Transactions of the American Nuclear Society 125 (2021) 292–295.

[8] E. Mulder, W. Boyes, Neutronics characteristics of a 165MWth xe-100 reactor, Nuclear Engineering and Design 357 (2020) 110415. doi:https://doi.org/10.1016/j.nucengdes.2019.110415.
URL https://www.sciencedirect.com/science/article/pii/S0029549319304467

[9] E. Blandford, K. Brumback, L. Fick, C. Gerardi, B. Haugh, E. Hillstrom, K. Johnson, P. F. Peterson, F. Rubio, F. S. Sarikurt, S. Sen, H. Zhao, N. Zweibaum, Kairos power thermal

hydraulics research and development, Nuclear Engineering and Design 364 (2020) 110636. doi:https://doi.org/10.1016/j.nucengdes.2020.110636.
URL https://www.sciencedirect.com/science/article/pii/S0029549320301308

[10] X. E. LLC, Advanced reactor demonstration program, https://x-energy.com/ardp, accessed: 2022-20-04 (2022).

[11] O. of Nuclear Energy, Energy department's advanced reactor demonstration program awards $30 million in initial funding for risk reduction projects, https://www.energy.gov/ne/articles/energy-departments-advanced-reactor-demonstration-program-awards-30-mil accessed: 2022-20-04 (2020).

[12] P. Gibbs, J. Hu, D. Kovacic, L. Scott, Pebble bed reactor domestic safeguards fy21 summary report, Tech. rep., Oak Ridge National Laboratory (2021).

[13] A. Garrett, S. Garrett, R. Marek, M. Mitchell, C. Orton, R. Otto, T. Sobolev, D. Springfels, Advanced reactor safeguards: Lessons from the iaea safeguards domain, Tech. rep., Pacific Northwest National Laboratory (2021).

[14] M. P. Dion, M. S. Greenwood, K. K. Hogue, S. E. O'Brien, L. M. Scott, G. T. Westphal, Pebble bed reactor domestic safeguards fy21 summary report, Tech. rep., Oak Ridge National Laboratory (2021).

[15] G. F. Knoll, Radiation Detection and Measurement, 4th Edition, Wiley, 2010.

[16] Eljen Technology, NEUTRON/GAMMA PSD EJ-301, EJ-309.
URL https://eljentechnology.com/products/liquid-scintillators/ej-301-ej-309

[17] Teledyne Technologies, ADQ7DC DATASHEET - 17-2017-D 2021-11-23 (2021) 1–31.
URL https://www.spdevices.com/documents/datasheets/26-adq7dc-datasheet/file

[18] V. Valković, Chapter 5 - measurements of radioactivity, in: V. Valković (Ed.), Radioactivity in the Environment, Elsevier Science, Amsterdam, 2000, pp. 117–258. doi:https://doi.org/10.1016/B978-044482954-2.50005-8. URL https://www.sciencedirect.com/science/article/pii/B9780444829542500058

[19] L. Hamilton, J. Duderstadt, Nuclear Reactor Analysis, Wiley, New York, 1976.

[20] F. Brooks, Development of organic scintillators, Nuclear Instruments and Methods 162 (1) (1979) 477–505. doi:https://doi.org/10.1016/0029-554X(79)90729-8. URL https://www.sciencedirect.com/science/article/pii/0029554X79907298

[21] S. Pozzi, S. Clarke, M. Flaska, P. Peerani, Pulse-height distributions of neutron and gamma rays from plutonium-oxide samples, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 608 (2) (2009) 310–315. doi:https://doi.org/10.1016/j.nima.2009.07.007. URL https://www.sciencedirect.com/science/article/pii/S0168900209013874

[22] R. Wurtz, B. Blair, C. Chen, A. Glenn, A. D. Kaplan, P. Rosenfield, J. Ruz, L. M. Simms, Methodology and performance comparison of statistical learning pulse shape classifiers as demonstrated with organic liquid scintillator, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 901 (2018) 46–55. doi:https://doi.org/10.1016/j.nima.2018.06.001. URL https://www.sciencedirect.com/science/article/pii/S0168900218307058

[23] L. C. Cinzia Bernardeschi, A. Domenici, Sram-based fpga systems for safety-critical applications: A survey on design standards and proposed methodologies, Journal of Computer

Science and Technology 30 (1) (2015) 373–390. doi:https://doi.org/10.1007/s11390-015-1530-5.

[24] M. Klein, C. J. Schmidt, Cascade, neutron detectors for highest count rates in combination with asic/fpga based readout electronics, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 628 (1) (2011) 9–18, vCI 2010. doi:https://doi.org/10.1016/j.nima.2010.06.278.
URL https://www.sciencedirect.com/science/article/pii/S0168900210014683

[25] I. T. Jolliffe, J. Cadima, Principal component analysis: a review and recent developments 374 (2065) 20150202. doi:10.1098/rsta.2015.0202.
URL https://royalsocietypublishing.org/doi/10.1098/rsta.2015.0202

[26] T. Alharbi, Principal component analysis for pulse-shape discrimination of scintillation radiation detectors, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 806 (2016) 240–243. doi:https://doi.org/10.1016/j.nima.2015.10.030.
URL https://www.sciencedirect.com/science/article/pii/S0168900215012371

# APPENDIX A

# RADSIGPRO PYTHON CODE

```
import csv
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
import numpy as np
import pandas as pd
import math
from os import path

def Raw\_Pulse\_Correction(file, baseline, pulses\_per\_csv = 100000, cfd\_delay = 6, attenuation\_fraction = 0.25):

## This code corrects the raw compass data files into more readable files, better units (ADC to mV), and creates
## a list of time data for each pulse.

    mpl.rc('font',family='Times␣New␣Roman')
    mpl.rc('font', size = 16)

    num\_rows = sum(1 for line in open(file)) - 1      #This part finds out the number of rows of data in the file
                                                      # so that arrays can be pre-allocated of that number later.
    #num\_rows = 30                    #This part allows you to manually limit the number of rows that will be pre-allocated,
                                      #use this when only looking at a set number of rows.

    sample\_start\_index = 0
    timetag\_index = 0
    sample\_length = 0
    with open(file, newline='') as f:                 #This part sets up the length of the samples, it should be 496
                                                      #but this allows for if it isn't.
        csv\_reader = csv.reader(f, delimiter=';')
        for counter,line in enumerate(csv\_reader):
            if counter == 0:
                for counter2,i in enumerate(line):
                    if i == 'TIMETAG':
                        timetag\_index = counter2
                    if i == 'SAMPLES':
                        sample\_start\_index = counter2
            if counter > 0:
                sample\_length = len(line[sample\_start\_index:])
            if counter > 0:                #Don't change the counter limiter in this if statement, its form only checks the length
                                           #of the first row of samples to save time.
                break

    ##These sections pre-allocate arrays to then later fill with data edited from the csv file, this is done to save computing
    #time.
    if num\_rows >= pulses\_per\_csv:
        pulses\_corrected\_first = np.zeros(sample\_length)
        pulses\_corrected = np.zeros(shape=(pulses\_per\_csv, sample\_length))
        inverted\_delayed\_signal = np.zeros(sample\_length)
```

```
    attenuated\_signal = np.zeros(sample\_length)
    times\_of\_pulses = np.zeros(shape=(pulses\_per\_csv, sample\_length))
else:
    pulses\_corrected = np.zeros(shape=(num\_rows, sample\_length))
    times\_of\_pulses = np.zeros(shape=(num\_rows, sample\_length))

#max\_pulse\_heights = np.zeros(num\_rows)        #This would allow us to store the height of each pulse, its not necessary
                                                  #for tail to total.
#max\_height\_times = np.zeros(num\_rows)         #This would allow us to store the times of the height of each pulse, its not
                                                  #necessary for tail to total.

raw\_pulse = np.zeros(sample\_length)

##This section takes the sample data and time tags in the csv file, manipulates them, and then puts them into arrays of each
#pulse and their corresponding times.
for y in range(math.ceil(num\_rows/pulses\_per\_csv)):

    start\_pulse = y*pulses\_per\_csv
    final\_pulse = start\_pulse + pulses\_per\_csv
    if num\_rows < final\_pulse:
        final\_pulse = num\_rows

    with open(file, newline='') as f:
        csv\_reader = csv.reader(f, delimiter=';')       #Opens the csv file and reads it (not the most effecient way but
                                                          #changing code would take too much time)
        for counter,line in enumerate(csv\_reader):
            if counter > start\_pulse and counter <= final\_pulse:

                '''#Prints out a raw pulse which CoMPASS outputs.
                if int(min(line[sample\_start\_index:])) <= 2000:

                    print(counter)
                    time = np.zeros(496)
                    for counter,i in enumerate(time):
                        time[counter] = counter*2

                    raw\_pulse[:] = [float(p) for p in line[sample\_start\_index:]]

                    plt.plot(time,raw\_pulse, color='b')
                    plt.xlabel('Time (ns)')
                    plt.ylabel('ADC Units')
                    plt.show()'''

                #print(line[sample\_start\_index], line[sample\_start\_index+1], line[sample\_start\_index+2],
                #line[sample\_start\_index+3], line[sample\_start\_index+4])
                #Constant Baseline Method
                #baseline = float(line[sample\_start\_index])
                #pulses\_corrected[counter-1] = [(-float(p)+baseline)*0.1220703125 for p in line[sample\_start\_index:]]

                #Baseline Freeze at Pulse Start Method
                pulses\_corrected\_first[:] = [(-float(p)+baseline)*0.1220703125 for p in line[sample\_start\_index:]]
                for i,height in enumerate(pulses\_corrected\_first):
                    if height >= max(pulses\_corrected\_first):
                        pulses\_corrected[counter-start\_pulse-1] =
                        [float(p-float(np.average(pulses\_corrected\_first[i-10:i-5]))) for p in pulses\_corrected\_first]
                        '''if pulses\_corrected\_first[i-5] >= 0.15*max(pulses\_corrected\_first):
```

```
                plt.scatter(np.array(range(len(pulses\_corrected\_first))), pulses\_corrected\_first, s=3,
                color='r')
                plt.scatter(i, pulses\_corrected\_first[i])
                plt.scatter(i-5, pulses\_corrected\_first[i-5], s=3)
                plt.show()
                plt.plot(np.array(range(len(pulses\_corrected\_first))), pulses\_corrected\_first, color='r')
                plt.show()'''
            break

        #print(f'Pulse {counter-1} Length: {len(line[sample\_start\_index:-cfd\_delay])}')
        inverted\_delayed\_signal[(cfd\_delay//2):] = [(-float(p)+baseline) for p in
        line[sample\_start\_index:-cfd\_delay//2]]
        #inverted\_delayed\_signal[(cfd\_delay//2):] = inverted\_delayed\_signal\_first[cfd\_delay::2]

        attenuated\_signal[:] = [attenuation\_fraction*(float(p)-baseline) for p in line[sample\_start\_index:]]
        #attenuated\_signal[:] = attenuated\_signal\_first[::2]

        shaped\_signal = inverted\_delayed\_signal + attenuated\_signal
        shaped\_signal\_flipped = np.flip(shaped\_signal)

        index\_max\_height = 0
        index\_sbzc = 0
        index\_sazc = 0
        for counter1,i in enumerate(shaped\_signal\_flipped):
            if i >= max(shaped\_signal\_flipped):
                index\_max\_height = counter1
        for counter2,i in enumerate(shaped\_signal\_flipped[index\_max\_height:]):
            print(i)
            if i <= 0:

                index\_sbzc = (sample\_length - 1) - (counter2 + index\_max\_height)
                index\_sazc = index\_sbzc + 1
                #print(f'sample length = {sample\_length}, Counter = {counter2}, max height index = {index\_max\_height}')
            break
        if shaped\_signal[index\_sbzc] == 0:
            t\_sbzc = float(line[timetag\_index])/(10**3)
        else:
            #print(f'SAZC = {shaped\_signal[index\_sazc]}, index = {index\_sazc}')
            #print(f'SBZC = {shaped\_signal[index\_sbzc]}, index = {index\_sbzc}')
            t\_fine = (-float(shaped\_signal[index\_sbzc])/(float(shaped\_signal[index\_sazc]) -
            float(shaped\_signal[index\_sbzc]))) * 2.0  #interpolation to find time from SBZC to ZC
            t\_sbzc = float(line[timetag\_index])/(10**3) - t\_fine

        time\_list = (np.arange(-index\_sbzc+1, sample\_length+1 - index\_sbzc, dtype=np.float64)*2) + t\_sbzc
        times\_of\_pulses[counter-start\_pulse-1] = time\_list      #Creates array of time correspondiong to each pulse
                                                                 #data point, starting from timetag bin location,
                                                                 #and increasing/decreasing by 2ns around it.

    #Makes all negative values 0
    pulses\_corrected = np.where(pulses\_corrected < 0, 0, pulses\_corrected)


#Writes files with the processed data in it.
'''f = open(f'{file[:-4]}\_CorrectedPulses{y+1}.csv', 'w')
f.write(",".join([f"time {k+1},pulse {k+1}" for k in range(start\_pulse, final\_pulse)])+ "\n")
for j in range(sample\_length):
```

```
            f.write(",".join([f"{times\_of\_pulses[k][j]},{pulses\_corrected[k][j]}" for k in range(0, pulses\_per\_csv)]) + "\n")
        f.close()'''


    for i in range(math.ceil(num\_rows/pulses\_per\_csv)):
        start\_pulse = i*pulses\_per\_csv
        final\_pulse = start\_pulse + pulses\_per\_csv
        if num\_rows < final\_pulse:
            final\_pulse = num\_rows
        f = open(f'{file[:-4]}\_CorrectedPulses{i+1}.csv', 'w')
        f.write(",".join([f"time␣{k+1},pulse␣{k+1}" for k in range(start\_pulse, final\_pulse)])+ "\n")
        for j in range(sample\_length):
            f.write(",".join([f"{times\_of\_pulses[k][j]},{pulses\_corrected[k][j]}" for k in range(start\_pulse, final\_pulse)])
            + "\n")
        f.close()



def PSD\_Analysis(gamma\_file, neutron\_file = 'Not␣a␣File', min\_height\_allowed = 0, max\_height\_allowed = 1000, long\_gate =
360, pregate = 50, short\_gate = 70, cfd\_delay = 6, attenuation\_fraction = 0.25):
## This function makes a pulse shape discrimination plot of both the gamma and neutron corrected data on the same graph, with
#selected max and min pulse heights (pulse start and end shouldn't usually be changed).

    mpl.rc('font',family='Times␣New␣Roman')
    mpl.rc('font', size = 16)

    num\_total\_pulses = sum(1 for line in open(f'{gamma\_file[:-21]}.csv')) - 1
    sample\_length = (sum(1 for line in open(f'{gamma\_file[:-5]}1.csv')) - 1)
    first\_file = pd.read\_csv(f'{gamma\_file[:-5]}1.csv')
    pulses\_per\_csv = len(first\_file.columns)//2

    time\_data1 = np.zeros(shape=(num\_total\_pulses, sample\_length))
    pulse\_data1 = np.zeros(shape=(num\_total\_pulses, sample\_length))
    time\_data2 = np.zeros(shape=(num\_total\_pulses, sample\_length))
    pulse\_data2 = np.zeros(shape=(num\_total\_pulses, sample\_length))

    num\_pulses\_inserted1 = 0
    for i in range(math.ceil(num\_total\_pulses/pulses\_per\_csv)):
        if path.exists(f'{gamma\_file[:-5]}{i+1}.csv') == True:
            data = pd.read\_csv(f'{gamma\_file[:-5]}{i+1}.csv')
            num\_pulses = len(data.columns)//2
            time\_data1[num\_pulses\_inserted1:(num\_pulses + num\_pulses\_inserted1)] =
            data[[f'time␣{j+1}' for j in range(num\_pulses\_inserted1, num\_pulses + num\_pulses\_inserted1)]].to\_numpy().T
            pulse\_data1[num\_pulses\_inserted1:(num\_pulses + num\_pulses\_inserted1)] =
            data[[f'pulse␣{j+1}' for j in range(num\_pulses\_inserted1, num\_pulses + num\_pulses\_inserted1)]].to\_numpy().T
            num\_pulses\_inserted1 += num\_pulses
        else:
            break

    num\_pulses\_inserted2 = 0
    for i in range(math.ceil(num\_total\_pulses/pulses\_per\_csv)):
        if path.exists(f'{neutron\_file[:-5]}{i+1}.csv') == True:
            data = pd.read\_csv(f'{neutron\_file[:-5]}{i+1}.csv')
            num\_pulses = len(data.columns)//2
            time\_data2[num\_pulses\_inserted2:(num\_pulses + num\_pulses\_inserted2)] =
            data[[f'time␣{j+1}' for j in range(num\_pulses\_inserted2, num\_pulses + num\_pulses\_inserted2)]].to\_numpy().T
```

```
            pulse\_data2[num\_pulses\_inserted2:(num\_pulses + num\_pulses\_inserted2)] =
            data[[f'pulse_{j+1}' for j in range(num\_pulses\_inserted2, num\_pulses + num\_pulses\_inserted2)]].to\_numpy().T
            num\_pulses\_inserted2 += num\_pulses
        else:
            break


total\_integrals1 = np.zeros(num\_total\_pulses)
tail\_integrals1 = np.zeros(num\_total\_pulses)
gamma\_tail\_to\_total = np.zeros(num\_total\_pulses)
total\_integrals2 = np.zeros(num\_total\_pulses)
tail\_integrals2 = np.zeros(num\_total\_pulses)
neutron\_tail\_to\_total = np.zeros(num\_total\_pulses)

inverted\_delayed\_signal1 = np.zeros(sample\_length)
attenuated\_signal1 = np.zeros(sample\_length)
shaped\_signal1 = np.zeros(sample\_length)
inverted\_delayed\_signal2 = np.zeros(sample\_length)
attenuated\_signal2 = np.zeros(sample\_length)
shaped\_signal2 = np.zeros(sample\_length)

index\_of\_discrimination1 = []
index\_of\_discrimination2 = []

gamma\_negative = 0
neutron\_negative = 0
total\_number = 0
##This section does the intergration for tail and total and then calculates the ratio of the two.
for counter, (pulse1,times1,pulse2,times2) in enumerate(zip(pulse\_data1,time\_data1,pulse\_data2,time\_data2)):

    max\_height1 = max(pulse1)
    max\_height2 = max(pulse2)

    inverted\_delayed\_signal1[(cfd\_delay//2):] = pulse1[:-(cfd\_delay//2)]
    attenuated\_signal1 = -attenuation\_fraction*pulse1
    shaped\_signal1 = inverted\_delayed\_signal1 + attenuated\_signal1
    shaped\_signal\_flipped1 = np.flip(shaped\_signal1)
    inverted\_delayed\_signal2[(cfd\_delay//2):] = pulse2[:-(cfd\_delay//2)]
    attenuated\_signal2 = -attenuation\_fraction*pulse2
    shaped\_signal2 = inverted\_delayed\_signal2 + attenuated\_signal2
    shaped\_signal\_flipped2 = np.flip(shaped\_signal2)

    index\_max\_height1 = 0
    index\_sbzc1 = 0
    index\_of\_tail\_start1 = 0
    index\_max\_height2 = 0
    index\_sbzc2 = 0
    index\_of\_tail\_start2 = 0
    for counter1,i in enumerate(shaped\_signal\_flipped1):
        if i >= max(shaped\_signal\_flipped1):
            index\_max\_height1 = counter1
    for counter2,i in enumerate(shaped\_signal\_flipped1[index\_max\_height1:]):
        if i <= 0:
            index\_sbzc1 = (sample\_length - 1) - (counter2 + index\_max\_height1)
            index\_of\_tail\_start1 = index\_sbzc1 - pregate//2 + short\_gate//2
            break
```

```
for counter3 ,i in enumerate(shaped\_signal\_flipped2 ):
    if i >= max(shaped\_signal\_flipped2 ):
        index\_max\_height2 = counter3
for counter4 ,i in enumerate(shaped\_signal\_flipped2[index\_max\_height2 :]):
    if i <= 0:
        index\_sbzc2 = (sample\_length - 1) - (counter4 + index\_max\_height2)
        index\_of\_tail\_start2 = index\_sbzc2 - pregate//2 + short\_gate//2
        break



#Method where integration starts at pregate.
if max\_height1 >= min\_height\_allowed and max\_height1 <= max\_height\_allowed and (index\_sbzc1 - pregate//2) > 0:
                                                            #Sets #a max mV limit on the pulses counted towards the tail to total.
    total\_number += 1
    tail\_integrals1[counter] = np.trapz(pulse1[index\_of\_tail\_start1 : index\_sbzc1 - pregate//2 +
    (long\_gate//2)], times1[index\_of\_tail\_start1 : index\_sbzc1 - pregate//2 + (long\_gate//2)]) #Takes tail integral
                                                            #from tail start to after pulse body.
    total\_integrals1[counter] = np.trapz(pulse1[index\_sbzc1 - pregate//2 : index\_sbzc1 - pregate//2 + (long\_gate//2)],
    times1[index\_sbzc1 - pregate//2 : index\_sbzc1 - pregate//2 +
    (long\_gate//2)])                #Takes the total intergral from pulse start to after pulse body.
    index\_of\_discrimination1.append(counter)
    if  tail\_integrals1[counter]/total\_integrals1[counter] < 0:
        gamma\_negative += 1
        plt.plot(times1, pulse1)
        plt.axvline(x=times1[index\_sbzc1 - pregate//2], label = 'Pulse␣Start', color='g')
        plt.axvline(x=times1[index\_of\_tail\_start1], label = 'Tail␣Start', color='r')
        plt.axvline(x=times1[index\_sbzc1 - pregate//2], label = 'Pulse␣End', color='y')
        plt.plot(times1, np.zeros(len(times1)), alpha=0.7, label = '0␣mV␣Mark', color='cyan')
        plt.legend()
        plt.xlabel('Time␣(ns)')
        plt.ylabel('Voltage␣(mV)')
        plt.title(f'Gamma␣Pulse␣{counter}')
        plt.show()
if max\_height1 >= min\_height\_allowed and max\_height1 <= max\_height\_allowed and (index\_sbzc1 - pregate//2) <= 0:
#Sets a max mV limit on the pulses counted towards the tail to total.
    total\_number += 1
    tail\_integrals1[counter] = np.trapz(pulse1[index\_of\_tail\_start1 : index\_sbzc1 - pregate//2 + (long\_gate//2)],
    times1[index\_of\_tail\_start1 : index\_sbzc1 - pregate//2 + (long\_gate//2)]) #Takes tail integral from tail start
                                                                                        #to after pulse body.
    total\_integrals1[counter] = np.trapz(pulse1[: index\_sbzc1 -
    pregate//2 + (long\_gate//2)], times1[: index\_sbzc1 - pregate//2 + (long\_gate//2)])   #Takes the total integral
                                                                                        #from pulse start to after pulse body.
    index\_of\_discrimination1.append(counter)
    if  tail\_integrals1[counter]/total\_integrals1[counter] < 0:
        gamma\_negative += 1
        plt.plot(times1, pulse1)
        plt.axvline(x=times1[0], label = 'Pulse␣Start', color='g')
        plt.axvline(x=times1[index\_of\_tail\_start1], label = 'Tail␣Start', color='r')
        plt.axvline(x=times1[index\_sbzc1 - pregate//2], label = 'Pulse␣End', color='y')
        plt.plot(times1, np.zeros(len(times1)), alpha=0.7, label = '0␣mV␣Mark', color='cyan')
        plt.legend()
        plt.xlabel('Time␣(ns)')
        plt.ylabel('Voltage␣(mV)')
        plt.title(f'Gamma␣Pulse␣{counter}')
        plt.show()
```

```
if max\_height2 >= min\_height\_allowed and max\_height2 <= max\_height\_allowed and (index\_sbzc2 - pregate//2) > 0:
    tail\_integrals2[counter] = np.trapz(pulse2[index\_of\_tail\_start2 : index\_sbzc2 - pregate//2 + (long\_gate//2)],
    times2[index\_of\_tail\_start2 : index\_sbzc2 - pregate//2 + (long\_gate//2)]) #Takes tail integral from tail start
                                                                                    #to after pulse body.
    total\_integrals2[counter] = np.trapz(pulse2[index\_sbzc2 - pregate//2 : index\_sbzc2 - pregate//2 + (long\_gate//2)],
    times2[index\_sbzc2 - pregate//2 : index\_sbzc2 - pregate//2 + (long\_gate//2)])          #Takes the total
                                                                          #integral from pulse start to after pulse body.
    index\_of\_discrimination2.append(counter)
    if  tail\_integrals2[counter]/total\_integrals2[counter] < 0:
        neutron\_negative += 1
        plt.plot(times2, pulse2)
        plt.axvline(x=times2[index\_sbzc2 - pregate//2], label = 'Pulse␣Start', color='g')
        plt.axvline(x=times2[index\_of\_tail\_start2], label = 'Tail␣Start', color='r')
        plt.axvline(x=times2[index\_sbzc2 - pregate//2], label = 'Pulse␣End', color='y')
        plt.plot(times2, np.zeros(len(times2)), alpha=0.7, label = '0␣mV␣Mark', color='cyan')
        plt.legend()
        plt.xlabel('Time␣(ns)')
        plt.ylabel('Voltage␣(mV)')
        plt.title(f'Neutron␣Pulse␣{counter}')
        plt.show()
if max\_height2 >= min\_height\_allowed and max\_height2 <= max\_height\_allowed and (index\_sbzc2 - pregate//2) <= 0:
    tail\_integrals2[counter] = np.trapz(pulse2[index\_of\_tail\_start2 : index\_sbzc2 - pregate//2 + (long\_gate//2)],
    times2[index\_of\_tail\_start2 : index\_sbzc2 - pregate//2 + (long\_gate//2)]) #Takes tail integral from tail start
                                                                                  #to after pulse body.
    total\_integrals2[counter] = np.trapz(pulse2[: index\_sbzc2 - pregate//2 + (long\_gate//2)],
    times2[: index\_sbzc2 - pregate//2 + (long\_gate//2)])          #Takes the total
                                                  #integral from pulse start to after pulse body.
    index\_of\_discrimination2.append(counter)
    if  tail\_integrals2[counter]/total\_integrals2[counter] < 0:
        neutron\_negative += 1
        plt.plot(times2, pulse2)
        plt.axvline(x=times2[0], label = 'Pulse␣Start', color='g')
        plt.axvline(x=times2[index\_of\_tail\_start2], label = 'Tail␣Start', color='r')
        plt.axvline(x=times2[index\_sbzc2 - pregate//2], label = 'Pulse␣End', color='y')
        plt.plot(times2, np.zeros(len(times2)), alpha=0.7, label = '0␣mV␣Mark', color='cyan')
        plt.legend()
        plt.xlabel('Time␣(ns)')
        plt.ylabel('Voltage␣(mV)')
        plt.title(f'Neutron␣Pulse␣{counter}')
        plt.show()


#Method where integration starts at 3 indexes before pulse height.
'''pulse1\_max\_index = 0
pulse2\_max\_index = 0
for counter5,i in enumerate(pulse1):
    if i >= max(pulse1):
        pulse1\_max\_index = counter5
for counter6,i in enumerate(pulse2):
    if i >= max(pulse2):
        pulse2\_max\_index = counter6

if max\_height1 >= min\_height\_allowed and max\_height1 <= max\_height\_allowed:   #Sets a max mV limit on the pulses
                                                                                 #counted towards the tail to total.
    total\_number += 1
```

```
        tail\_integrals1[counter] = np.trapz(pulse1[index\_of\_tail\_start1 : index\_sbzc1 - pregate//2 + (long\_gate//2)],
        times1[index\_of\_tail\_start1 : index\_sbzc1 - pregate//2 + (long\_gate//2)]) #Takes tail integral from tail
                                                                                      #start to after pulse body.
        total\_integrals1[counter] = np.trapz(pulse1[pulse1\_max\_index - 3 : index\_sbzc1 - pregate//2 + (long\_gate//2)],
        times1[pulse1\_max\_index - 3 : index\_sbzc1 - pregate//2 + (long\_gate//2)])            #Takes the total
                                                            #integral from pulse start to after pulse body.
        index\_of\_discrimination1.append(counter)
        if  tail\_integrals1[counter]/total\_integrals1[counter] < 0:
            gamma\_negative += 1
            plt.plot(times1, pulse1)
            plt.axvline(x=times1[index\_sbzc1 - pregate//2], label = 'Pulse Start', color='g')
            plt.axvline(x=times1[index\_of\_tail\_start1], label = 'Tail Start', color='r')
            plt.axvline(x=times1[index\_sbzc1 - pregate//2], label = 'Pulse End', color='y')
            plt.plot(times1, np.zeros(len(times1)), alpha=0.7, label = '0 mV Mark', color='cyan')
            plt.legend()
            plt.xlabel('Time (ns)')
            plt.ylabel('Voltage (mV)')
            plt.title(f'Gamma Pulse {counter}')
            plt.show()

    if max\_height2 >= min\_height\_allowed and max\_height2 <= max\_height\_allowed:
        tail\_integrals2[counter] = np.trapz(pulse2[index\_of\_tail\_start2 : index\_sbzc2 - pregate//2 + (long\_gate//2)],
        times2[index\_of\_tail\_start2 : index\_sbzc2 - pregate//2 + (long\_gate//2)]) #Takes tail integral from tail
                                                                                  #start to after pulse body.
        total\_integrals2[counter] = np.trapz(pulse2[pulse2\_max\_index - 3 : index\_sbzc2 - pregate//2 + (long\_gate//2)],
        times2[pulse2\_max\_index - 3 : index\_sbzc2 - pregate//2 + (long\_gate//2)])            #Takes the total
                                                                     #integral from pulse start to after pulse body.
        index\_of\_discrimination2.append(counter)
        if  tail\_integrals2[counter]/total\_integrals2[counter] < 0:
            neutron\_negative += 1
            plt.plot(times2, pulse2)
            plt.axvline(x=times2[pulse2\_max\_index - 3], label = 'Pulse Start', color='g')
            plt.axvline(x=times2[index\_of\_tail\_start2], label = 'Tail Start', color='r')
            plt.axvline(x=times2[index\_sbzc2 - pregate//2], label = 'Pulse End', color='y')
            plt.plot(times2, np.zeros(len(times2)), alpha=0.7, label = '0 mV Mark', color='cyan')
            plt.legend()
            plt.xlabel('Time (ns)')
            plt.ylabel('Voltage (mV)')
            plt.title(f'Neutron Pulse {counter}')
            plt.show()
print(total\_integrals1)
print(total\_integrals2)'''


for counter7, (tail1,total1,tail2,total2) in enumerate(zip(tail\_integrals1,total\_integrals1,tail\_integrals2,
total\_integrals2)):    #Calculates the tail to total ratio
    if total1 == 0:
        gamma\_tail\_to\_total[counter7] = 0
    else:
        gamma\_tail\_to\_total[counter7] = tail1/total1
    if total2 == 0:
        neutron\_tail\_to\_total[counter7] = 0
    else:
        neutron\_tail\_to\_total[counter7] = tail2/total2
    #print(gamma\_tail\_to\_total[counter])
    #print(neutron\_tail\_to\_total[counter])
```

```
    #print(gamma\_tail\_to\_total)

#Creates a plot of gamma and neutron pulses on one plot.
'''legend\_counter = 0
for counter, (pulse1,times1,pulse2,times2,g\_psd,n\_psd) in enumerate(zip(pulse\_data1,time\_data1,pulse\_data2,time\_data2,
gamma\_tail\_to\_total,neutron\_tail\_to\_total)):
    legend\_counter += 1
    height1 = max(pulse1)
    height2 = max(pulse2)
    if g\_psd <= 0.17 and n\_psd >= 0.25 and height1 >= 400 and height2 >= 400:
        time = np.zeros(100)
        for counter,i in enumerate(time):
            time[counter] = counter*2
        plt.plot(time, pulse1[:100]/height1, label = 'Gamma Pulse', color='b')
        plt.plot(time, pulse2[:100]/height2, label = 'Neutron Pulse', color='g',linestyle='--')
        plt.xlabel('Time (ns)')
        plt.ylabel('Ratio to Max Pulse Height')
        plt.legend()
        plt.show()'''

#Used to compare with CoMPASS plots.
'''spectrum\_data = pd.read\_csv('BaselineHeld\_G8138\_and\_N8125\_PSDCut\_CompassSpectrums.csv')
#total\_pulses\_psd = []
total\_pulses\_psd1 = []
total\_pulses\_psd2 = []
data\_interval = 1/16383
psd\_values = np.arange(0, 1 + data\_interval, data\_interval)
psd\_data1 = spectrum\_data['Gamma PSD'].to\_numpy()
psd\_data2 = spectrum\_data['Neutron PSD'].to\_numpy()
#psd\_data1 = spectrum\_data['LE - PSD'].to\_numpy()
#psd\_data2 = spectrum\_data['CFD - PSD'].to\_numpy()
for i in range(len(psd\_data1)):
    how\_many = psd\_data1[i]
    if how\_many > 0:
        total\_pulses\_psd1 += [psd\_values[i] for j in range(how\_many)]
for i in range(len(psd\_data2)):
    how\_many = psd\_data2[i]
    if how\_many > 0:
        total\_pulses\_psd2 += [psd\_values[i] for j in range(how\_many)]'''

pd.DataFrame(gamma\_tail\_to\_total).to\_csv('C:/Users/bswellons.AUTH/Documents/ben\_gamma\_PSD.csv')
pd.DataFrame(neutron\_tail\_to\_total).to\_csv('C:/Users/bswellons.AUTH/Documents/ben\_gamma\_PSD.csv')
if path.exists(f'{neutron\_file[:-5]}{1}.csv'):

    #Plot without error bars
    plt.hist(gamma\_tail\_to\_total, bins = 500, range = (0,1), label = 'Gamma␣Detector', color='blue', histtype='step')
                                                        #Plots histogram of tail to total ratio for each pulse.
    plt.hist(neutron\_tail\_to\_total, bins = 500, range = (0,1), alpha=0.6, label = 'Neutron␣Detector', color='orange',
    histtype='step')
    plt.legend(loc='upper␣right',)
    plt.xlabel('Tail␣to␣Total␣Ratio')
    plt.ylabel('Number␣of␣Pulses␣(Counts)')
    plt.xticks(np.arange(0, 1.1, 0.1))
    plt.show()
    #plt.hist(total\_pulses\_psd1, bins = 2000, range = (0,1), alpha=0.7, label = 'Compass PSD Plot', color='cyan')
```

```
    #Plot with error bars
    y, bin\_edges = np.histogram(gamma\_tail\_to\_total, bins= 200, range = (0, 1))
    bin\_centers = 0.5*(bin\_edges[1:] + bin\_edges[:-1])
    plt.errorbar(bin\_centers, y, yerr = y**0.5, capsize = 2, marker = '.', drawstyle = 'steps-mid', ecolor = 'dimgrey',
    elinewidth = 1.5, linewidth = 1.5, markeredgewidth = 0, markersize = 0.1, label = 'Gamma␣Detector', color='blue')
    '''interval = 0.005
    highest = 0
    highest\_index = 0
    average\_index = 0
    for counter,i in enumerate(y):
        if i >= highest:
            highest = i
            highest\_index = counter
        if i >= float(np.mean(y)):
            average\_index = counter
    print(f'Number of Pulses at Average Height: {np.mean(y)}, PSD of Average: {average\_index*interval}')
    print(f'Number of Pulses at Highest: {highest}, PSD of Highest: {highest\_index*interval}')'''
    y, bin\_edges = np.histogram(neutron\_tail\_to\_total, bins= 200, range = (0, 1))
    bin\_centers = 0.5*(bin\_edges[1:] + bin\_edges[:-1])
    plt.errorbar(bin\_centers, y, yerr = y**0.5, capsize = 2, marker = '.', drawstyle = 'steps-mid', ecolor = 'dimgrey',
    elinewidth = 1.5, linewidth = 2, markeredgewidth = 0, markersize = 0.1, label = 'Neutron␣Detector', color='red', ls = ':')
    '''interval = 0.005
    highest = 0
    highest\_index = 0
    average\_index = 0
    for counter,i in enumerate(y):
        if i >= highest:
            highest = i
            highest\_index = counter
        if i >= float(np.mean(y)):
            average\_index = counter
    print(f'Number of Pulses at Average Height: {np.mean(y)}, PSD of Average: {average\_index*interval}')
    print(f'Number of Pulses at Highest: {highest}, PSD of Highest: {highest\_index*interval}')'''
    plt.ylim(0,500)
    plt.xlim(0, 0.6)
    plt.legend(loc='upper␣right')
    plt.xlabel('Tail␣to␣Total␣Ratio')
    plt.ylabel('Number␣of␣Pulses␣(Counts)')
    plt.xticks(np.arange(0, 0.7, 0.1))
    plt.show()

else:
    if f'{gamma\_file[0]}' == 'G':
        plt.hist(gamma\_tail\_to\_total, bins = 500, range = (0,1), label = 'My␣PSD␣Plot', color='red')
                                                    #Plots histogram of tail to total ratio for each pulse.
        #plt.hist(total\_pulses\_psd1, bins = 500, range = (0,1), alpha=0.7, label = 'Compass PSD Plot', color='blue')
        #plt.hist(total\_pulses\_psd1, bins = 500, range = (0,1),  label = 'Leading Edge PSD')
        #plt.hist(total\_pulses\_psd2, bins = 500, range = (0,1), alpha=0.6, label = 'CFD PSD')
        plt.legend(loc='upper␣right')
        plt.xlabel('Tail␣to␣Total␣Ratio')
        plt.ylabel('Number␣of␣Pulses␣(Counts)')
        plt.xticks(np.arange(0, 1.1, 0.1))
        #plt.yscale('log')
        plt.show()
    if f'{gamma\_file[0]}' == 'N':
        plt.hist(gamma\_tail\_to\_total, bins = 500, range = (0,1), label = 'My␣PSD␣Plot', color='red')
```

```
                                                                                #Plots histogram of tail to total ratio for each pulse.
            #plt.hist(total\_pulses\_psd2, bins = 500, range = (0,1), alpha=0.7, label = 'Compass PSD Plot', color='blue')
            #plt.hist(total\_pulses\_psd1, bins = 500, range = (0,1),  label = 'Leading Edge PSD')
            #plt.hist(total\_pulses\_psd2, bins = 500, range = (0,1), alpha=0.6, label = 'CFD PSD')
            plt.legend(loc='upper␣right')
            plt.xlabel('Tail␣to␣Total␣Ratio')
            plt.ylabel('Number␣of␣Pulses␣(Counts)')
            plt.xticks(np.arange(0, 1.1, 0.1))
            #plt.yscale('log')
            plt.show()

    if path.exists(f'{gamma\_file[:-21]}\_PSD\_Data{1}.csv'):
        x = 1
    else:
        for i in range(math.ceil(len(index\_of\_discrimination1)/50000)):
            start\_pulse = i*50000
            final\_pulse = start\_pulse + 50000
            if len(index\_of\_discrimination1) < final\_pulse:
                final\_pulse = len(index\_of\_discrimination1)
            f = open(f'{gamma\_file[:-21]}\_PSD\_Data{i+1}.csv', 'w')
            f.write(",".join([f"time␣{index\_of\_discrimination1[k]+1},pulse␣{index\_of\_discrimination1[k]+1},
␣␣␣␣␣␣␣␣␣␣␣␣tail/total␣{index\_of\_discrimination1[k]+1}" for k in range(start\_pulse, final\_pulse)])+ "\n")
            for j in range(sample\_length):
                f.write(",".join([f"{time\_data1[index\_of\_discrimination2[k]][j]},{pulse\_data1[index\_of\_discrimination1[k]][j]},
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣{gamma\_tail\_to\_total[index\_of\_discrimination1[k]]}" for k in range(start\_pulse, final\_pulse)]) + "\n")
            f.close()

    if path.exists(f'{neutron\_file[:-21]}\_PSD\_Data{1}.csv'):
        x = 1
    else:
        if path.exists(f'{neutron\_file[:-5]}{1}.csv'):
            for i in range(math.ceil(len(index\_of\_discrimination2)/50000)):
                start\_pulse = i*50000
                final\_pulse = start\_pulse + 50000
                if len(index\_of\_discrimination2) < final\_pulse:
                    final\_pulse = len(index\_of\_discrimination2)
                f = open(f'{neutron\_file[:-21]}\_PSD\_Data{i+1}.csv', 'w')
                f.write(",".join([f"time␣{index\_of\_discrimination2[k]+1},pulse␣{index\_of\_discrimination2[k]+1},
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣tail/total␣{index\_of\_discrimination2[k]+1}" for k in range(start\_pulse, final\_pulse)])+ "\n")
                for j in range(sample\_length):
                    f.write(",".join([f"{time\_data2[index\_of\_discrimination2[k]][j]},{pulse\_data2[index\_of\_discrimination2[k]][j]},
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣{neutron\_tail\_to\_total[index\_of\_discrimination2[k]]}" for k in range(start\_pulse, final\_pulse)]) + "\n")
                f.close()

    print(f'Number␣of␣Negative␣Gamma␣PSD␣Pulses␣=␣{gamma\_negative}')
    print(f'Number␣of␣Negative␣Neutron␣PSD␣Pulses␣=␣{neutron\_negative}')
    print(f'Total␣Number␣of␣Pulses␣=␣{total\_number}')


def PHD\_Plot(file, adc\_axis\_max, baseline):
## This function makes a pulse heigh distribution plot of a raw data file (not a file already run through the pulse correction function)

    mpl.rc('font',family='Times␣New␣Roman')
    mpl.rc('font', size = 16)

    num\_rows = sum(1 for line in open(file)) - 1
```

```
pulse\_heights\_ADC = np.zeros(num\_rows)
pulse\_heights\_mV = np.zeros(num\_rows)

sample\_start\_index = 0
with open(file, newline='') as f:
    csv\_reader = csv.reader(f, delimiter=';')
    for counter,line in enumerate(csv\_reader):
        if counter == 0:
            for counter2,i in enumerate(line):
                if i == 'SAMPLES':
                    sample\_start\_index = counter2
        if counter > 0:
            #baseline = max([float(p) for p in line[sample\_start\_index:len(line)]])
            pulse\_heights\_ADC[counter-1] = (-float(min(line[sample\_start\_index:len(line)]))+baseline)
            pulse\_heights\_mV[counter-1] = (-float(min(line[sample\_start\_index:len(line)]))+baseline)*0.1220703125

 #Compass comparrison
'''spectrum\_data = pd.read\_csv('BaselineHeld\_G8139\_and\_N8127\_PSDCut\_CompassSpectrums.csv')
total\_pulses\_energy = []
#total\_pulses\_energy1 = []
#total\_pulses\_energy2 = []

energy\_channels = np.arange(0, 16384, 1, dtype=np.float64)
energy\_data = spectrum\_data['Neutron Energy'].to\_numpy()
#energy\_data1 = spectrum\_data['LE - PHD'].to\_numpy()
#energy\_data2 = spectrum\_data['CFD - PHD'].to\_numpy()'''

'''for i in range(len(energy\_data)-1):
    how\_many = int(energy\_data[i])
    if how\_many > 0:
        total\_pulses\_energy += [int(energy\_channels[i]) for j in range(how\_many)]'''
"""for i in range(len(energy\_data1)-1):
    how\_many = float(energy\_data1[i])
    if how\_many > 0:
        total\_pulses\_energy1 += [float(energy\_channels[i]) for j in range(how\_many)]
for i in range(len(energy\_data2)-1):
    how\_many = float(energy\_data2[i])
    if how\_many > 0:
        total\_pulses\_energy2 += [float(energy\_channels[i]) for j in range(how\_many)]"""

#Plot without error bars
plt.hist(pulse\_heights\_ADC, bins = int(adc\_axis\_max//11), range = (0, adc\_axis\_max), label = 'My␣PHD␣Plot', color='red')
#plt.hist(pulse\_heights\_ADC, bins = int(adc\_axis\_max//11), range = (0, adc\_axis\_max), color='red', histtype='step')
#plt.hist(total\_pulses\_energy, bins = int(adc\_axis\_max//11), range = (0, adc\_axis\_max), alpha=0.6, label = 'Compass PHD Plot',
color='y')
#plt.hist(total\_pulses\_energy1, bins = float(adc\_axis\_max//11), range = (0, adc\_axis\_max), label = 'Leading Edge PHD')
#plt.hist(total\_pulses\_energy2, bins = float(adc\_axis\_max//11), range = (0, adc\_axis\_max), alpha=0.6, label = 'CFD PHD')
plt.legend(loc='upper␣right')
plt.xlabel('ADC␣Channel')
plt.ylabel('Number␣of␣Pulses␣(Counts)')
#plt.ylim((0, 20000))
#plt.xticks(np.arange(0, adc\_axis\_max+1, 500))
plt.show()
plt.hist(pulse\_heights\_mV, bins = int(adc\_axis\_max*0.1220703125//1.15), range = (0, int(adc\_axis\_max*0.1220703125)))
plt.xlabel('Voltage␣(mV)')
```

```
    plt.ylabel('Number␣of␣Pulses␣(Counts)')
    #plt.ylim((0, 20000))
    plt.show()

    #Plot with error bars
    y, bin\_edges = np.histogram(pulse\_heights\_mV, bins =
    int(adc\_axis\_max*0.1220703125//11), range = (0, int(adc\_axis\_max*0.1220703125)))
    bin\_centers = 0.5*(bin\_edges[1:] + bin\_edges[:-1])
    plt.errorbar(bin\_centers, y, yerr = y**0.5, capsize = 2, marker = '.', drawstyle = 'steps-mid', ecolor = 'dimgrey',
    elinewidth = 2, linewidth = 2, markeredgewidth = 0, markersize = 0.1, color='red')
    plt.yscale('log')
    plt.xlim((0,900))
    plt.xlabel('Voltage␣(mV)')
    plt.ylabel('Number␣of␣Pulses␣(Counts)')
    plt.show()



def ToF\_Analysis(gamma\_file, neutron\_file, time\_axis\_min, time\_axis\_max, long\_gate = 360, pregate = 50, short\_gate = 70,
cfd\_delay = 6, attenuation\_fraction = 0.25):
## This function makes a time of flight plot of the neutron times - gamma times, where you select the minimum and maximum
#points on the x-axis.

    mpl.rc('font',family='Times␣New␣Roman')
    mpl.rc('font', size = 16)

    num\_total\_pulses = sum(1 for line in open(f'{gamma\_file[:-21]}.csv')) - 1
    sample\_length = (sum(1 for line in open(f'{gamma\_file[:-5]}1.csv')) - 1)
    first\_file = pd.read\_csv(f'{gamma\_file[:-5]}1.csv')
    pulses\_per\_csv = len(first\_file.columns)//2

    time\_data1 = np.zeros(shape=(num\_total\_pulses, sample\_length))
    pulse\_data1 = np.zeros(shape=(num\_total\_pulses, sample\_length))
    time\_data2 = np.zeros(shape=(num\_total\_pulses, sample\_length))
    pulse\_data2 = np.zeros(shape=(num\_total\_pulses, sample\_length))
    gamma\_psd = 0
    neutron\_psd = 0

    num\_pulses\_inserted1 = 0
    for i in range(math.ceil(num\_total\_pulses/pulses\_per\_csv)):
        if path.exists(f'{gamma\_file[:-5]}{i+1}.csv'):
            data = pd.read\_csv(f'{gamma\_file[:-5]}{i+1}.csv')
            num\_pulses = len(data.columns)//2
            time\_data1[num\_pulses\_inserted1:(num\_pulses + num\_pulses\_inserted1)] = data[[f'time␣{j+1}' for j in
            range(num\_pulses\_inserted1, num\_pulses + num\_pulses\_inserted1)]].to\_numpy().T
            pulse\_data1[num\_pulses\_inserted1:(num\_pulses + num\_pulses\_inserted1)] = data[[f'pulse␣{j+1}' for j in
            range(num\_pulses\_inserted1, num\_pulses + num\_pulses\_inserted1)]].to\_numpy().T
            num\_pulses\_inserted1 += num\_pulses
        else:
            break
    num\_pulses\_inserted2 = 0
    for i in range(math.ceil(num\_total\_pulses/pulses\_per\_csv)):
        if path.exists(f'{neutron\_file[:-5]}{i+1}.csv'):
            data = pd.read\_csv(f'{neutron\_file[:-5]}{i+1}.csv')
            num\_pulses = len(data.columns)//2
            time\_data2[num\_pulses\_inserted2:(num\_pulses + num\_pulses\_inserted2)] = data[[f'time␣{j+1}' for j in
```

```
        range(num\_pulses\_inserted2 , num\_pulses + num\_pulses\_inserted2 )]].to\_numpy().T
        pulse\_data2[num\_pulses\_inserted2:(num\_pulses + num\_pulses\_inserted2)] = data[[f'pulse␣{j+1}' for j in
        range(num\_pulses\_inserted2 , num\_pulses + num\_pulses\_inserted2 )]].to\_numpy().T
        num\_pulses\_inserted2 += num\_pulses
    else:
        break


num\_discriminated\_gammas = 0
num\_discriminated\_neutrons = 0
if path.exists(f'{gamma\_file[:-21]}\_PSD\_Data{1}.csv') and path.exists(f'{neutron\_file[:-21]}\_PSD\_Data{1}.csv'):
    for i in range(50):
        if path.exists(f'{gamma\_file[:-21]}\_PSD\_Data{i+1}.csv'):
            gamma\_data = pd.read\_csv(f'{gamma\_file[:-21]}\_PSD\_Data{i+1}.csv')
            num\_discriminated\_gammas += len(gamma\_data.columns)//3
        if path.exists(f'{neutron\_file[:-21]}\_PSD\_Data{i+1}.csv'):
            neutron\_data = pd.read\_csv(f'{neutron\_file[:-21]}\_PSD\_Data{i+1}.csv')
            num\_discriminated\_neutrons += len(neutron\_data.columns)//3

gamma\_time\_data = np.zeros(shape=(num\_discriminated\_gammas, sample\_length))
gamma\_pulse\_data = np.zeros(shape=(num\_discriminated\_gammas, sample\_length))
gamma\_psd = np.zeros(num\_discriminated\_gammas)
neutron\_time\_data = np.zeros(shape=(num\_discriminated\_neutrons, sample\_length))
neutron\_pulse\_data = np.zeros(shape=(num\_discriminated\_neutrons, sample\_length))
neutron\_psd = np.zeros(num\_discriminated\_neutrons)

num\_pulses\_inserted3 = 0
if path.exists(f'{gamma\_file[:-21]}\_PSD\_Data{1}.csv') and path.exists(f'{neutron\_file[:-21]}\_PSD\_Data{1}.csv'):
    for i in range(math.ceil(num\_discriminated\_gammas/50000)):
        if path.exists(f'{gamma\_file[:-21]}\_PSD\_Data{i+1}.csv'):
            gamma\_data = pd.read\_csv(f'{gamma\_file[:-21]}\_PSD\_Data{i+1}.csv')
            num\_pulses = len(gamma\_data.columns)//3
            gamma\_time\_data[num\_pulses\_inserted3:(num\_pulses + num\_pulses\_inserted3)] = gamma\_data[[f'time␣{j+1}' for
            j in range(num\_pulses\_inserted3 , num\_pulses + num\_pulses\_inserted3 )]].to\_numpy().T
            gamma\_pulse\_data[num\_pulses\_inserted3:(num\_pulses + num\_pulses\_inserted3)] = gamma\_data[[f'pulse␣{j+1}' for
            j in range(num\_pulses\_inserted3 , num\_pulses + num\_pulses\_inserted3 )]].to\_numpy().T
            gamma\_psd[num\_pulses\_inserted3:(num\_pulses + num\_pulses\_inserted3)] = gamma\_data[[f'tail/total␣{j+1}' for
            j in range(num\_pulses\_inserted3 , num\_pulses + num\_pulses\_inserted3 )]].loc[0].to\_numpy().T
            num\_pulses\_inserted3 += num\_pulses

num\_pulses\_inserted4 = 0
if path.exists(f'{gamma\_file[:-21]}\_PSD\_Data{1}.csv') and path.exists(f'{neutron\_file[:-21]}\_PSD\_Data{1}.csv'):
    for i in range(math.ceil(num\_discriminated\_neutrons/50000)):
        if path.exists(f'{neutron\_file[:-21]}\_PSD\_Data{i+1}.csv'):
            neutron\_data = pd.read\_csv(f'{neutron\_file[:-21]}\_PSD\_Data{i+1}.csv')
            num\_pulses = len(neutron\_data.columns)//3
            neutron\_time\_data[num\_pulses\_inserted4:(num\_pulses + num\_pulses\_inserted4)] = neutron\_data[[f'time␣{j+1}' for
            j in range(num\_pulses\_inserted4 , num\_pulses + num\_pulses\_inserted4 )]].to\_numpy().T
            neutron\_pulse\_data[num\_pulses\_inserted4:(num\_pulses + num\_pulses\_inserted4)] = neutron\_data[[f'pulse␣{j+1}' for
            j in range(num\_pulses\_inserted4 , num\_pulses + num\_pulses\_inserted4 )]].to\_numpy().T
            neutron\_psd[num\_pulses\_inserted4:(num\_pulses + num\_pulses\_inserted4)] = neutron\_data[[f'tail/total␣{j+1}' for
            j in range(num\_pulses\_inserted4 , num\_pulses + num\_pulses\_inserted4 )]].loc[0].to\_numpy().T
            num\_pulses\_inserted4 += num\_pulses


gamma\_timetags = np.zeros(num\_total\_pulses)
```

```
neutron\_timetags = np.zeros(num\_total\_pulses)
times\_of\_flight = np.zeros(num\_total\_pulses)
gamma\_timetags\_psd = np.zeros(num\_discriminated\_gammas)
neutron\_timetags\_psd = np.zeros(num\_total\_pulses)
if num\_discriminated\_gammas >= num\_discriminated\_neutrons:
    times\_of\_flight\_psd = np.zeros(num\_discriminated\_neutrons)
else:
    times\_of\_flight\_psd = np.zeros(num\_discriminated\_gammas)

inverted\_delayed\_signal1 = np.zeros(sample\_length)
attenuated\_signal1 = np.zeros(sample\_length)
shaped\_signal1 = np.zeros(sample\_length)
inverted\_delayed\_signal2 = np.zeros(sample\_length)
attenuated\_signal2 = np.zeros(sample\_length)
shaped\_signal2 = np.zeros(sample\_length)


for counter,(pulse1,time1,pulse2,time2) in enumerate(zip(pulse\_data1, time\_data1, pulse\_data2, time\_data2)):

    inverted\_delayed\_signal1[(cfd\_delay//2):] = pulse1[:-(cfd\_delay//2)]
    attenuated\_signal1 = -attenuation\_fraction*pulse1
    shaped\_signal1 = inverted\_delayed\_signal1 + attenuated\_signal1
    shaped\_signal\_flipped1 = np.flip(shaped\_signal1)
    inverted\_delayed\_signal2[(cfd\_delay//2):] = pulse2[:-(cfd\_delay//2)]
    attenuated\_signal2 = -attenuation\_fraction*pulse2
    shaped\_signal2 = inverted\_delayed\_signal2 + attenuated\_signal2
    shaped\_signal\_flipped2 = np.flip(shaped\_signal2)

    for counter1,i in enumerate(shaped\_signal\_flipped1):
        if i >= max(shaped\_signal\_flipped1):
            index\_max\_height1 = counter1
    for counter2,i in enumerate(shaped\_signal\_flipped1[index\_max\_height1:]):
        if i <= 0:
            index\_sbzc1 = (sample\_length - 1) - (counter2 + index\_max\_height1)
            index\_sazc1 = index\_sbzc1 + 1
            t\_fine1 = (-float(shaped\_signal1[index\_sbzc1])/(float(shaped\_signal1[index\_sazc1]) -
            float(shaped\_signal1[index\_sbzc1]))) * 2.0
            timetag1 = float(time1[index\_sbzc1]) + float(t\_fine1)
            break
    for counter3,i in enumerate(shaped\_signal\_flipped2):
        if i >= max(shaped\_signal\_flipped2):
            index\_max\_height2 = counter3
    for counter4,i in enumerate(shaped\_signal\_flipped2[index\_max\_height2:]):
        if i <= 0:
            index\_sbzc2 = (sample\_length - 1) - (counter4 + index\_max\_height2)
            index\_sazc2 = index\_sbzc2 + 1
            t\_fine2 = (-float(shaped\_signal2[index\_sbzc2])/(float(shaped\_signal2[index\_sazc2]) -
            float(shaped\_signal2[index\_sbzc2]))) * 2.0
            timetag2 = float(time2[index\_sbzc2]) + float(t\_fine2)
            break

    gamma\_timetags[counter] = timetag1
    neutron\_timetags[counter] = timetag2

for counter, (gamma\_timetag,neutron\_timetag) in enumerate(zip(gamma\_timetags,neutron\_timetags)):
    times\_of\_flight[counter] = neutron\_timetag - gamma\_timetag
```

```
if path.exists(f'{gamma\_file[:-21]}\_PSD\_Data{1}.csv') and path.exists(f'{neutron\_file[:-21]}\_PSD\_Data{1}.csv'):
    for counter,(pulse1,time1,pulse2,time2) in enumerate(zip(gamma\_pulse\_data, gamma\_time\_data, neutron\_pulse\_data,
    neutron\_time\_data)):

        inverted\_delayed\_signal1[(cfd\_delay//2):] = pulse1[:-(cfd\_delay//2)]
        attenuated\_signal1 = -attenuation\_fraction*pulse1
        shaped\_signal1 = inverted\_delayed\_signal1 + attenuated\_signal1
        shaped\_signal\_flipped1 = np.flip(shaped\_signal1)
        inverted\_delayed\_signal2[(cfd\_delay//2):] = pulse2[:-(cfd\_delay//2)]
        attenuated\_signal2 = -attenuation\_fraction*pulse2
        shaped\_signal2 = inverted\_delayed\_signal2 + attenuated\_signal2
        shaped\_signal\_flipped2 = np.flip(shaped\_signal2)

        for counter1,i in enumerate(shaped\_signal\_flipped1):
            if i >= max(shaped\_signal\_flipped1):
                index\_max\_height1 = counter1
        for counter2,i in enumerate(shaped\_signal\_flipped1[index\_max\_height1:]):
            if i <= 0:
                index\_sbzc1 = (sample\_length - 1) - (counter2 + index\_max\_height1)
                index\_sazc1 = index\_sbzc1 + 1
                t\_fine1 = (-float(shaped\_signal1[index\_sbzc1])/(float(shaped\_signal1[index\_sazc1]) -
                float(shaped\_signal1[index\_sbzc1]))) * 2.0
                timetag1 = float(time1[index\_sbzc1]) + float(t\_fine1)
                break
        for counter3,i in enumerate(shaped\_signal\_flipped2):
            if i >= max(shaped\_signal\_flipped2):
                index\_max\_height2 = counter3
        for counter4,i in enumerate(shaped\_signal\_flipped2[index\_max\_height2:]):
            if i <= 0:
                index\_sbzc2 = (sample\_length - 1) - (counter4 + index\_max\_height2)
                index\_sazc2 = index\_sbzc2 + 1
                t\_fine2 = (-float(shaped\_signal2[index\_sbzc2])/(float(shaped\_signal2[index\_sazc2]) -
                float(shaped\_signal2[index\_sbzc2]))) * 2.0
                timetag2 = float(time2[index\_sbzc2]) + float(t\_fine2)
                break

        gamma\_timetags\_psd[counter] = timetag1
        neutron\_timetags\_psd[counter] = timetag2

for counter, (gamma\_timetag,neutron\_timetag) in enumerate(zip(gamma\_timetags\_psd,neutron\_timetags\_psd)):
    times\_of\_flight\_psd[counter] = neutron\_timetag - gamma\_timetag


#plt.hist(times\_of\_flight, bins = int((abs(time\_axis\_min)+abs(time\_axis\_max))*2), range = (int(time\_axis\_min),
int(time\_axis\_max)))
plt.hist(times\_of\_flight, bins = int((abs(time\_axis\_min)+abs(time\_axis\_max))*2), range = (int(time\_axis\_min),
int(time\_axis\_max)), histtype='step')
#plt.yscale('log')
plt.xlabel('Time␣of␣Flight␣(ns)')
plt.title('Total␣Time␣of␣Flight')
plt.ylabel('Number␣of␣Pulses␣(Counts)')
plt.show()

#Plotting PSD histograms after a TOF cut, includes error bars
```

```python
    tof\_cut\_gamma\_psd = []
    tof\_cut\_neutron\_psd = []
    for counter,(g\_psd, n\_psd, tof) in enumerate(zip(gamma\_psd, neutron\_psd, times\_of\_flight\_psd)):
        if tof >= 10 and tof <= 40:
            tof\_cut\_gamma\_psd.append(g\_psd)
            tof\_cut\_neutron\_psd.append(n\_psd)
    y, bin\_edges = np.histogram(tof\_cut\_gamma\_psd, bins= 200, range = (0, 1))
    bin\_centers = 0.5*(bin\_edges[1:] + bin\_edges[:-1])
    plt.errorbar(bin\_centers, y, yerr = y**0.5, capsize = 2, marker = '.', drawstyle = 'steps-mid', ecolor = 'dimgrey',
    elinewidth = 1.5, linewidth = 1.5, markeredgewidth = 0, markersize = 0.1, label = 'Gamma␣Detector', color='blue')
    y, bin\_edges = np.histogram(tof\_cut\_neutron\_psd, bins= 200, range = (0, 1))
    bin\_centers = 0.5*(bin\_edges[1:] + bin\_edges[:-1])
    plt.errorbar(bin\_centers, y, yerr = y**0.5, capsize = 2, marker = '.', drawstyle = 'steps-mid', ecolor = 'dimgrey',
    elinewidth = 2, linewidth = 1.5, markeredgewidth = 0, markersize = 0.1, label = 'Neutron␣Detector', color='red', ls = ':')
    plt.legend(loc='upper␣right')
    plt.xlabel('Tail␣to␣Total␣Ratio')
    plt.ylabel('Number␣of␣Pulses␣(Counts)')
    plt.ylim(0,475)
    plt.xlim(0, 0.6)
    plt.xticks(np.arange(0, 0.7, 0.1))
    plt.show()

    #Plot with error bars
    y, bin\_edges = np.histogram(times\_of\_flight, bins= int((abs(time\_axis\_min)+abs(time\_axis\_max))), range =
    (int(time\_axis\_min), int(time\_axis\_max)))
    bin\_centers = 0.5*(bin\_edges[1:] + bin\_edges[:-1])
    plt.errorbar(bin\_centers, y, yerr = y**0.5, capsize = 2, marker = '.', drawstyle = 'steps-mid', ecolor = 'dimgrey',
    elinewidth = 1.5, linewidth = 1.5, markeredgewidth = 0, markersize = 0.1, color = 'red')
    #print(y)
    plt.xlabel('Time␣Cross-Correlation␣(ns)')
    #plt.title('Total Time of Flight')
    plt.ylabel('Number␣of␣Pulses␣(Counts)')
    plt.ylim(0,300)
    plt.show()

    if path.exists(f'{gamma\_file[:-21]}\_PSD\_Data{1}.csv') and path.exists(f'{neutron\_file[:-21]}\_PSD\_Data{1}.csv'):
        plt.hist(times\_of\_flight\_psd, bins = int((abs(time\_axis\_min)+abs(time\_axis\_max))*2), range =
        (int(time\_axis\_min), int(time\_axis\_max)))
        plt.title('Time␣of␣Flight␣from␣PSD␣Restriction')
        #plt.yscale('log')
        plt.xlabel('Time␣of␣Flight␣(ns)')
        plt.ylabel('Number␣of␣Pulses␣(Counts)')
        plt.show()

    if path.exists(f'{gamma\_file[:-21]}\_PSD\_Data{1}.csv') or path.exists(f'{neutron\_file[:-21]}\_PSD\_Data{1}.csv'):
        f = open(f'{gamma\_file[6:-21]}\_ToF\_&\_PSD.csv', 'w')
        f.write(",".join(['pulse,time␣of␣flight,gamma␣psd,neutron␣psd'])+ "\n")
        for i in range(len(times\_of\_flight)):
            if times\_of\_flight[i] <= 40 and times\_of\_flight[i] >= 10:
                f.write(",".join([f"{i},{times\_of\_flight\_psd[i]},{gamma\_psd[i]},{neutron\_psd[i]}"]) + "\n")
        f.close()



def ToF\_Comparison(gamma\_file, neutron\_file, max\_time\_difference, min\_time\_difference = 2, show\_times = 'No',
max\_height\_cutoff = 10000, min\_height\_cutoff = 0, long\_gate = 360, pregate = 50, short\_gate = 70, cfd\_delay = 6,
```

```
attenuation\_fraction = 0.25):
## This function compares gamma and neutron pulse pairs, then plots and records the pairs that fit the specified constraints.

    mpl.rc('font',family='Times New Roman')
    mpl.rc('font', size = 16)

    num\_total\_pulses = sum(1 for line in open(f'{gamma\_file[:-21]}.csv')) - 1
    sample\_length = (sum(1 for line in open(f'{gamma\_file[:-5]}1.csv')) - 1)
    first\_file = pd.read\_csv(f'{gamma\_file[:-5]}1.csv')
    pulses\_per\_csv = len(first\_file.columns)//2

    time\_data1 = np.zeros(shape=(num\_total\_pulses, sample\_length))
    pulse\_data1 = np.zeros(shape=(num\_total\_pulses, sample\_length))
    time\_data2 = np.zeros(shape=(num\_total\_pulses, sample\_length))
    pulse\_data2 = np.zeros(shape=(num\_total\_pulses, sample\_length))

    num\_pulses\_inserted1 = 0
    for i in range(math.ceil(num\_total\_pulses/pulses\_per\_csv)):
        if path.exists(f'{gamma\_file[:-5]}{i+1}.csv') == True:
            data = pd.read\_csv(f'{gamma\_file[:-5]}{i+1}.csv')
            num\_pulses = len(data.columns)//2
            time\_data1[num\_pulses\_inserted1:(num\_pulses + num\_pulses\_inserted1)] = data[[f'time {j+1}' for j in
            range(num\_pulses\_inserted1, num\_pulses + num\_pulses\_inserted1)]].to\_numpy().T
            pulse\_data1[num\_pulses\_inserted1:(num\_pulses + num\_pulses\_inserted1)] = data[[f'pulse {j+1}' for j in
            range(num\_pulses\_inserted1, num\_pulses + num\_pulses\_inserted1)]].to\_numpy().T
            num\_pulses\_inserted1 += num\_pulses
        else:
            break
    num\_pulses\_inserted2 = 0
    for i in range(math.ceil(num\_total\_pulses/pulses\_per\_csv)):
        if path.exists(f'{neutron\_file[:-5]}{i+1}.csv') == True:
            data = pd.read\_csv(f'{neutron\_file[:-5]}{i+1}.csv')
            num\_pulses = len(data.columns)//2
            time\_data2[num\_pulses\_inserted2:(num\_pulses + num\_pulses\_inserted2)] = data[[f'time {j+1}' for j in
            range(num\_pulses\_inserted2, num\_pulses + num\_pulses\_inserted2)]].to\_numpy().T
            pulse\_data2[num\_pulses\_inserted2:(num\_pulses + num\_pulses\_inserted2)] = data[[f'pulse {j+1}' for j in
            range(num\_pulses\_inserted2, num\_pulses + num\_pulses\_inserted2)]].to\_numpy().T
            num\_pulses\_inserted2 += num\_pulses
        else:
            break

    inverted\_delayed\_signal1 = np.zeros(sample\_length)
    attenuated\_signal1 = np.zeros(sample\_length)
    shaped\_signal1 = np.zeros(sample\_length)
    inverted\_delayed\_signal2 = np.zeros(sample\_length)
    attenuated\_signal2 = np.zeros(sample\_length)
    shaped\_signal2 = np.zeros(sample\_length)

    index\_of\_discrimination = []
    for i,(pulse1,time1,pulse2,time2) in enumerate(zip(pulse\_data1, time\_data1, pulse\_data2, time\_data2)):

        max\_height1 = max(pulse1)
        max\_height2 = max(pulse2)

        inverted\_delayed\_signal1[(cfd\_delay//2):] = pulse1[:-(cfd\_delay//2)]
        attenuated\_signal1 = -attenuation\_fraction*pulse1
```

```
        shaped\_signal1 = inverted\_delayed\_signal1 + attenuated\_signal1
        shaped\_signal\_flipped1 = np.flip(shaped\_signal1)
        inverted\_delayed\_signal2[(cfd\_delay//2):] = pulse2[:-(cfd\_delay//2)]
        attenuated\_signal2 = -attenuation\_fraction*pulse2
        shaped\_signal2 = inverted\_delayed\_signal2 + attenuated\_signal2
        shaped\_signal\_flipped2 = np.flip(shaped\_signal2)

        for counter1,i in enumerate(shaped\_signal\_flipped1):
            if i >= max(shaped\_signal\_flipped1):
                index\_max\_height1 = counter1
        for counter2,i in enumerate(shaped\_signal\_flipped1[index\_max\_height1:]):
            if i <= 0:
                index\_sbzc1 = (sample\_length - 1) - (counter2 + index\_max\_height1)
                index\_sazc1 = index\_sbzc1 + 1
                t\_fine1 = (-float(shaped\_signal1[index\_sbzc1])/(float(shaped\_signal1[index\_sazc1]) -
                float(shaped\_signal1[index\_sbzc1]))) * 2.0
                timetag1 = index\_sbzc1 + t\_fine1
                break
        for counter3,i in enumerate(shaped\_signal\_flipped2):
            if i >= max(shaped\_signal\_flipped2):
                index\_max\_height2 = counter3
        for counter4,i in enumerate(shaped\_signal\_flipped2[index\_max\_height2:]):
            if i <= 0:
                index\_sbzc2 = (sample\_length - 1) - (counter4 + index\_max\_height2)
                index\_sazc2 = index\_sbzc2 + 1
                t\_fine2 = (-float(shaped\_signal2[index\_sbzc2])/(float(shaped\_signal2[index\_sazc2]) -
                float(shaped\_signal2[index\_sbzc2]))) * 2.0
                timetag2 = index\_sbzc2 + t\_fine2
                break

        if ((timetag1 + max\_time\_difference) >= timetag2) and ((timetag1 + min\_time\_difference) <= timetag2) and
        (max\_height1 <= max\_height\_cutoff and max\_height2 <= max\_height\_cutoff) and
        (max\_height1 >= min\_height\_cutoff and max\_height2 >= min\_height\_cutoff):
            plt.plot(time1, pulse1, label = 'Gamma␣Pulse', color='b')
            plt.plot(time2, pulse2, label = 'Neutron␣Pulse', color='g',linestyle='--')
            plt.xlabel('Time␣(ns)')
            plt.ylabel('Voltage␣(mV)')
            plt.legend()
            plt.show()

            if (show\_times == 'Show␣Times') or (show\_times == 'show␣times') or (show\_times == 'Show␣times') or (show\_times
            == 'Show') or (show\_times ==
            'show') or (show\_times == 'Yes') or (show\_times == 'yes'):
                print(f'Gamma␣{i}␣Arrival␣Time:␣{timetag1}␣ns,␣Neutron␣{i}␣Arrival␣Time:␣{timetag2}␣ns')

            index\_of\_discrimination.append(i)
        else:
            continue


    f = open(f'{gamma\_file[:-21]}\_DiscriminatedPulses\_MaxTime{max\_time\_difference}\_MinTime{min\_time\_difference}
␣␣␣␣\_MaxHeight{max\_height\_cutoff}\_MinHeight{min\_height\_cutoff}.csv', 'w')
    f.write(",".join([f"time␣{k},pulse␣{k}" for k in index\_of\_discrimination])+ "\n")
    for j in range(sample\_length):
        f.write(",".join([f"{time\_data1[k][j]},{pulse\_data1[k][j]}" for k in index\_of\_discrimination]) + "\n")
    f.close()
```

```
    f = open(f'{neutron\_file[:-21]}\_DiscriminatedPulses\_MaxTime{max\_time\_difference}\_MinTime{min\_time\_difference}
␣␣␣␣\_MaxHeight{max\_height\_cutoff}\_MinHeight{min\_height\_cutoff}.csv', 'w')
    f.write(",".join([f"time␣{k},pulse␣{k}" for k in index\_of\_discrimination])+ "\n")
    for j in range(sample\_length):
        f.write(",".join([f"{time\_data2[k][j]},{pulse\_data2[k][j]}" for k in index\_of\_discrimination]) + "\n")
    f.close()
```

# APPENDIX B

# RADSIGPRO FPGA IMPLEMENTATION

```
import csv
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
import numpy as np
import pandas as pd
import math
from os import path

def Raw\_Pulse\_Correction(file, baseline, pulses\_per\_csv = 100000, cfd\_delay = 6, attenuation\_fraction = 0.25):
## This code corrects the raw compass data files into more readable files, better units (ADC to mV), and creates a list of
#time data for each pulse.

    num\_rows = sum(1 for line in open(file)) - 1      #This part finds out the number of rows of data in the file so that
                                                       #arrays can be pre-allocated of that number later.
    #num\_rows = 30                    #This part allows you to manually limit the number of rows that will be pre-allocated,
                                                    #use this when only looking at a set number of rows.

    sample\_length = 0
    with open(file, newline='') as f:                 #This part sets up the length of the samples, it should be 496 but this
                                                                                   #allows for if it isnt.
        csv\_reader = csv.reader(f, delimiter=';')
        for counter,line in enumerate(csv\_reader):
            if counter > 0:
                sample\_length = len(line[4:])
            if counter > 0:                #Don't change the counter limiter in this if statement, its form only
                                              #checks the length of the first row of samples to save time.
                break

    ##These sections pre-allocate arrays to then later fill with
    #data edited from the csv file, this is done to save computing time.
    pulses\_corrected\_first = np.zeros(sample\_length)
    pulses\_corrected = np.zeros(shape=(num\_rows, sample\_length))
    inverted\_delayed\_signal = np.zeros(sample\_length)
    attenuated\_signal = np.zeros(sample\_length)
    times\_of\_pulses = np.zeros(shape=(num\_rows, sample\_length))

    #max\_pulse\_heights = np.zeros(num\_rows)         #This would allow us to store the height of each pulse,
                                                       #its not necessary for tail to total.
    #max\_height\_times = np.zeros(num\_rows)          #This would allow us to store the times of the height of each pulse,
                                                      #its not necessary for tail to total.

    #raw\_pulse = np.zeros(sample\_length)

    ##This section takes the sample data and time tags in the csv file, manipulates them, and then puts them into arrays of
    #each pulse and their corresponding times.
    with open(file, newline='') as f:
```

```python
csv\_reader = csv.reader(f, delimiter=';')        #Opens the csv file and reads it (not the most effecient way but
                                                  #changing code would take too much time)
for counter,line in enumerate(csv\_reader):
    if counter > 0:

        #Prints out a raw pulse which CoMPASS outputs.
        '''if int(min(line[4:])) <= 2000:

            time = np.zeros(496)
            for counter,i in enumerate(time):
                time[counter] = counter*2

            raw\_pulse[:] = [float(p) for p in line[4:]]
            mpl.rc('font',family='Times New Roman')
            mpl.rc('font', size = 16)
            plt.plot(time,raw\_pulse, color='b')
            plt.xlabel('Time (ns)')
            plt.ylabel('ADC Units')
            plt.show()'''

        #print(line[4], line[5], line[6], line[7], line[8])
        #Constant Baseline Method
        #baseline = float(max(line[4:len(line)]))
        #pulses\_corrected[counter-1] = [(-float(p)+baseline)*0.1220703125 for p in line[4:]]

        #Baseline Freeze at Pulse Start Method
        pulses\_corrected\_first[:] = [(-float(p)) for p in line[4:len(line)]]
        for i,height in enumerate(pulses\_corrected\_first):
            if height >= max(pulses\_corrected\_first):
                pulses\_corrected[counter-1] = [float(p-float(np.average(pulses\_corrected\_first[i-9:i-4])))*
                0.1220703125 for p in pulses\_corrected\_first]

                '''if counter == 3 or counter == 5:
                    print(f'Baseline Average: {int(np.average(pulses\_corrected\_first[i-9:i-4]))}, Values Averaged:
                    {pulses\_corrected\_first[i-9:i-4]}, Maximum Index: {i}')'''

                '''if pulses\_corrected\_first[i-5] >= 0.15*max(pulses\_corrected\_first):
                    plt.scatter(np.array(range(len(pulses\_corrected\_first))), pulses\_corrected\_first, s=3, color='r')
                    plt.scatter(i, pulses\_corrected\_first[i])
                    plt.scatter(i-5, pulses\_corrected\_first[i-5], s=3)
                    plt.show()
                    plt.plot(np.array(range(len(pulses\_corrected\_first))), pulses\_corrected\_first, color='r')
                    plt.show()'''
                break


        #print(f'Pulse {counter-1} Length: {len(line[4:-cfd\_delay])}')
        inverted\_delayed\_signal[(cfd\_delay//2):] = [(-float(p)+baseline) for p in line[4:-cfd\_delay//2]]
        #inverted\_delayed\_signal[(cfd\_delay//2):] = inverted\_delayed\_signal\_first[cfd\_delay::2]

        attenuated\_signal[:] = [attenuation\_fraction*(float(p)-baseline) for p in line[4:len(line)]]
        #attenuated\_signal[:] = attenuated\_signal\_first[::2]

        shaped\_signal = inverted\_delayed\_signal + attenuated\_signal
        shaped\_signal\_flipped = np.flip(shaped\_signal)
```

```
                index\_max\_height = 0
                index\_sbzc = 0
                index\_sazc = 0
                for counter1,i in enumerate(shaped\_signal\_flipped):
                    if i >= max(shaped\_signal\_flipped):
                        index\_max\_height = counter1
                for counter2,i in enumerate(shaped\_signal\_flipped[index\_max\_height:]):
                    if i <= 0:
                        index\_sbzc = (sample\_length - 1) - (counter2 + index\_max\_height)
                        index\_sazc = index\_sbzc + 1
                        #print(f'sample length = {sample\_length}, Counter = {counter2}, max height index = {index\_max\_height}')
                        break
                if shaped\_signal[index\_sbzc] == 0:
                    t\_sbzc = float(line[0])/(10**3)
                else:
                    #print(f'SAZC = {shaped\_signal[index\_sazc]}, index = {index\_sazc}')
                    #print(f'SBZC = {shaped\_signal[index\_sbzc]}, index = {index\_sbzc}')
                    t\_fine = (-float(shaped\_signal[index\_sbzc])/(float(shaped\_signal[index\_sazc]) -
                    float(shaped\_signal[index\_sbzc]))) * 2.0  #interpolation to find time from SBZC to ZC
                    t\_sbzc = float(line[0])/(10**3) - t\_fine

                time\_list = (np.arange(-index\_sbzc+1, sample\_length+1 - index\_sbzc, dtype=np.float64)*2) + t\_sbzc
                times\_of\_pulses[counter-1] = time\_list      #Creates array of time corresponding to each pulse data point,
                                     #starting from timetag bin location, and increasing/decreasing by 2ns around it.

        pulses\_corrected = np.where(pulses\_corrected < 0, 0, pulses\_corrected)


    for i in range(math.ceil(num\_rows/pulses\_per\_csv)):
        start\_pulse = i*pulses\_per\_csv
        final\_pulse = start\_pulse + pulses\_per\_csv
        if num\_rows < final\_pulse:
            final\_pulse = num\_rows
        f = open(f'{file[:-4]}\_CorrectedPulses{i+1}.csv', 'w')
        f.write(",".join([f"time␣{k+1}␣[ns],pulse␣{k+1}␣[mV]" for k in range(start\_pulse, final\_pulse)])+ "\n")
        for j in range(sample\_length):
            f.write(",".join([f"{times\_of\_pulses[k][j]},{pulses\_corrected[k][j]}" for k in range(start\_pulse, final\_pulse)])
            + "\n")
        f.close()



def PSD\_Analysis(gamma\_file, neutron\_file = 'Not␣a␣File', min\_height\_allowed = 0, max\_height\_allowed = 1000, long\_gate = 360,
pregate = 50, short\_gate = 70, cfd\_delay = 6, attenuation\_fraction = 0.25):
## This function makes a pulse shape discrimination plot of both the gamma and neutron corrected data on the same graph, with
#selected max and min pulse heights (pulse start and end shouldn't usually be changed).

    num\_total\_pulses = sum(1 for line in open(f'{gamma\_file[:-21]}.csv')) - 1
    sample\_length = (sum(1 for line in open(f'{gamma\_file[:-5]}1.csv')) - 1)
    first\_file = pd.read\_csv(f'{gamma\_file[:-5]}1.csv')
    pulses\_per\_csv = len(first\_file.columns)//2

    time\_data1 = np.zeros(shape=(num\_total\_pulses, sample\_length))
    pulse\_data1 = np.zeros(shape=(num\_total\_pulses, sample\_length))
    time\_data2 = np.zeros(shape=(num\_total\_pulses, sample\_length))
    pulse\_data2 = np.zeros(shape=(num\_total\_pulses, sample\_length))
```

```
num\_pulses\_inserted1 = 0
for i in range(math.ceil(num\_total\_pulses/pulses\_per\_csv)):
    if path.exists(f'{gamma\_file[:-5]}{i+1}.csv') == True:
        data = pd.read\_csv(f'{gamma\_file[:-5]}{i+1}.csv')
        num\_pulses = len(data.columns)//2
        time\_data1[num\_pulses\_inserted1:(num\_pulses + num\_pulses\_inserted1)] = data[[f'time␣{j+1}␣[ns]' for j in
        range(num\_pulses\_inserted1, num\_pulses + num\_pulses\_inserted1)]].to\_numpy().T
        pulse\_data1[num\_pulses\_inserted1:(num\_pulses + num\_pulses\_inserted1)] = data[[f'pulse␣{j+1}␣[mV]' for j in
        range(num\_pulses\_inserted1, num\_pulses + num\_pulses\_inserted1)]].to\_numpy().T
        num\_pulses\_inserted1 += num\_pulses
    else:
        break

num\_pulses\_inserted2 = 0
for i in range(math.ceil(num\_total\_pulses/pulses\_per\_csv)):
    if path.exists(f'{neutron\_file[:-5]}{i+1}.csv') == True:
        data = pd.read\_csv(f'{neutron\_file[:-5]}{i+1}.csv')
        num\_pulses = len(data.columns)//2
        time\_data2[num\_pulses\_inserted2:(num\_pulses + num\_pulses\_inserted2)] = data[[f'time␣{j+1}␣[ns]' for j in
        range(num\_pulses\_inserted2, num\_pulses + num\_pulses\_inserted2)]].to\_numpy().T
        pulse\_data2[num\_pulses\_inserted2:(num\_pulses + num\_pulses\_inserted2)] = data[[f'pulse␣{j+1}␣[mV]' for j in
        range(num\_pulses\_inserted2, num\_pulses + num\_pulses\_inserted2)]].to\_numpy().T
        num\_pulses\_inserted2 += num\_pulses
    else:
        break


total\_integrals1 = np.zeros(num\_total\_pulses)
tail\_integrals1 = np.zeros(num\_total\_pulses)
gamma\_tail\_to\_total = np.zeros(num\_total\_pulses)
total\_integrals2 = np.zeros(num\_total\_pulses)
tail\_integrals2 = np.zeros(num\_total\_pulses)
neutron\_tail\_to\_total = np.zeros(num\_total\_pulses)

inverted\_delayed\_signal1 = np.zeros(sample\_length)
attenuated\_signal1 = np.zeros(sample\_length)
shaped\_signal1 = np.zeros(sample\_length)
inverted\_delayed\_signal2 = np.zeros(sample\_length)
attenuated\_signal2 = np.zeros(sample\_length)
shaped\_signal2 = np.zeros(sample\_length)

index\_of\_discrimination1 = []
index\_of\_discrimination2 = []

gamma\_negative = 0
neutron\_negative = 0
total\_number = 0
##This section does the intergration for tail and total and then calculates the ratio of the two.
for counter, (pulse1,times1,pulse2,times2) in enumerate(zip(pulse\_data1,time\_data1,pulse\_data2,time\_data2)):

    max\_height1 = max(pulse1)
    max\_height2 = max(pulse2)

    inverted\_delayed\_signal1[(cfd\_delay//2):] = pulse1[:-(cfd\_delay//2)]
    attenuated\_signal1 = -attenuation\_fraction*pulse1
```

```
shaped\_signal1 = inverted\_delayed\_signal1 + attenuated\_signal1
shaped\_signal\_flipped1 = np.flip(shaped\_signal1)
inverted\_delayed\_signal2[(cfd\_delay//2):] = pulse2[:-(cfd\_delay//2)]
attenuated\_signal2 = -attenuation\_fraction*pulse2
shaped\_signal2 = inverted\_delayed\_signal2 + attenuated\_signal2
shaped\_signal\_flipped2 = np.flip(shaped\_signal2)

index\_max\_height1 = 0
index\_sbzc1 = 0
index\_of\_tail\_start1 = 0
index\_max\_height2 = 0
index\_sbzc2 = 0
index\_of\_tail\_start2 = 0
for counter1,i in enumerate(shaped\_signal\_flipped1):
    if i >= max(shaped\_signal\_flipped1):
        index\_max\_height1 = counter1
for counter2,i in enumerate(shaped\_signal\_flipped1[index\_max\_height1:]):
    if i <= 0:
        index\_sbzc1 = (sample\_length - 1) - (counter2 + index\_max\_height1)
        index\_of\_tail\_start1 = index\_sbzc1 - pregate//2 + short\_gate//2
        break
for counter3,i in enumerate(shaped\_signal\_flipped2):
    if i >= max(shaped\_signal\_flipped2):
        index\_max\_height2 = counter3
for counter4,i in enumerate(shaped\_signal\_flipped2[index\_max\_height2:]):
    if i <= 0:
        index\_sbzc2 = (sample\_length - 1) - (counter4 + index\_max\_height2)
        index\_of\_tail\_start2 = index\_sbzc2 - pregate//2 + short\_gate//2
        break


#Method to replicate Rishya's style of integration
pulse1\_max\_index = 0
pulse2\_max\_index = 0
for counter5,i in enumerate(pulse1):
    if i >= max(pulse1):
        pulse1\_max\_index = counter5
for counter6,i in enumerate(pulse2):
    if i >= max(pulse2):
        pulse2\_max\_index = counter6

for i in pulse1[pulse1\_max\_index - 3 : pulse1\_max\_index + 155]:
    if i >= 0:
        total\_integrals1[counter] += i
for i in pulse1[pulse1\_max\_index + 10 : pulse1\_max\_index + 155]:
    if i >= 0:
        tail\_integrals1[counter] += i
index\_of\_discrimination1.append(counter)
for i in pulse2[pulse2\_max\_index - 3 : pulse2\_max\_index + 155]:
    if i >= 0:
        total\_integrals2[counter] += i
for i in pulse2[pulse2\_max\_index + 10 : pulse2\_max\_index + 155]:
    if i >= 0:
        tail\_integrals2[counter] += i
index\_of\_discrimination2.append(counter)
```

```
mpl.rc('font',family='Times New Roman')
mpl.rc('font', size = 16)
#Method where integration starts at 3 indexes before pulse height.
'''pulse1\_max\_index = 0
pulse2\_max\_index = 0
for counter5,i in enumerate(pulse1):
    if i >= max(pulse1):
        pulse1\_max\_index = counter5
for counter6,i in enumerate(pulse2):
    if i >= max(pulse2):
        pulse2\_max\_index = counter6

if max\_height1 >= min\_height\_allowed and max\_height1 <= max\_height\_allowed:   #Sets a max mV limit on the pulses
                                                                                  #counted towards the tail to total.
    total\_number += 1
    tail\_integrals1[counter] = np.trapz(pulse1[index\_of\_tail\_start1 : index\_sbzc1 - pregate//2 + (long\_gate//2)],
    times1[index\_of\_tail\_start1 : index\_sbzc1 - pregate//2 + (long\_gate//2)]) #Takes tail integral from tail
                                                                                  #start to after pulse body.
    total\_integrals1[counter] = np.trapz(pulse1[pulse1\_max\_index - 3 : index\_sbzc1 - pregate//2 + (long\_gate//2)],
    times1[pulse1\_max\_index - 3 : index\_sbzc1 - pregate//2 + (long\_gate//2)])        #Takes the total integral
                                                                                  #from pulse start to after pulse body.
    index\_of\_discrimination1.append(counter)
    if  tail\_integrals1[counter]/total\_integrals1[counter] < 0:
        gamma\_negative += 1
        plt.plot(times1, pulse1)
        plt.axvline(x=times1[index\_sbzc1 - pregate//2], label = 'Pulse Start', color='g')
        plt.axvline(x=times1[index\_of\_tail\_start1], label = 'Tail Start', color='r')
        plt.axvline(x=times1[index\_sbzc1 - pregate//2], label = 'Pulse End', color='y')
        plt.plot(times1, np.zeros(len(times1)), alpha=0.7, label = '0 mV Mark', color='cyan')
        plt.legend()
        plt.xlabel('Time (ns)')
        plt.ylabel('Voltage (mV)')
        plt.title(f'Gamma Pulse {counter}')
        plt.show()

if max\_height2 >= min\_height\_allowed and max\_height2 <= max\_height\_allowed:
    tail\_integrals2[counter] = np.trapz(pulse2[index\_of\_tail\_start2 : index\_sbzc2 - pregate//2 + (long\_gate//2)],
    times2[index\_of\_tail\_start2 : index\_sbzc2 - pregate//2 + (long\_gate//2)]) #Takes tail integral from tail
                                                  #start to after pulse body.
    total\_integrals2[counter] = np.trapz(pulse2[pulse2\_max\_index - 3 : index\_sbzc2 - pregate//2 + (long\_gate//2)],
    times2[pulse2\_max\_index - 3 : index\_sbzc2 - pregate//2 + (long\_gate//2)])        #Takes the total integral
                                                            #from pulse start to after pulse body.
    index\_of\_discrimination2.append(counter)
    if  tail\_integrals2[counter]/total\_integrals2[counter] < 0:
        neutron\_negative += 1
        plt.plot(times2, pulse2)
        plt.axvline(x=times2[pulse2\_max\_index - 3], label = 'Pulse Start', color='g')
        plt.axvline(x=times2[index\_of\_tail\_start2], label = 'Tail Start', color='r')
        plt.axvline(x=times2[index\_sbzc2 - pregate//2], label = 'Pulse End', color='y')
        plt.plot(times2, np.zeros(len(times2)), alpha=0.7, label = '0 mV Mark', color='cyan')
        plt.legend()
        plt.xlabel('Time (ns)')
        plt.ylabel('Voltage (mV)')
        plt.title(f'Neutron Pulse {counter}')
```

```
            plt.show()  '''

    for counter7, (tail1,total1,tail2,total2) in enumerate(zip(tail\_integrals1,total\_integrals1,tail\_integrals2,total\_integrals2)):
#Calculates the tail to total ratio
        gamma\_tail\_to\_total[counter7] = tail1/total1
        neutron\_tail\_to\_total[counter7] = tail2/total2
        #print(gamma\_tail\_to\_total[counter])
        #print(neutron\_tail\_to\_total[counter])
        #print(gamma\_tail\_to\_total)

    #Find center values of gamma and neutron distributions along with average PSD value of each data set.
    '''y1, x1, \_ = plt.hist(gamma\_tail\_to\_total, bins = 200, range = (0,0.6))
    print(f'Most Common Gamma Ratio: {x1[np.where(y1 == y1.max())][0]}, Number of Pulses: {y1.max()}')
    y2, x2, \_ = plt.hist(neutron\_tail\_to\_total, bins = 200, range = (0,0.6))
    print(f'Most Common Neutron Ratio: {x2[np.where(y2 == y2.max())][0]}, Number of Pulses: {y2.max()}')
    print(f'Average Gamma PSD: {np.mean(gamma\_tail\_to\_total)}')
    print(f'Average Neutron PSD: {np.mean(neutron\_tail\_to\_total)}')'''

    #Creates a plot of gamma and neutron pulses on one plot.
    '''legend\_counter = 0
    for counter, (pulse1,times1,pulse2,times2,g\_psd,n\_psd) in enumerate(zip(pulse\_data1,time\_data1,pulse\_data2,time\_data2,
    gamma\_tail\_to\_total,neutron\_tail\_to\_total)):
        legend\_counter += 1
        height1 = max(pulse1)
        height2 = max(pulse2)
        if g\_psd <= 0.17 and n\_psd >= 0.25 and height1 >= 400 and height2 >= 400:
            time = np.zeros(100)
            for counter,i in enumerate(time):
                time[counter] = counter*2
            plt.plot(time, pulse1[:100]/height1, label = 'Gamma Pulse', color='b',)
            plt.plot(time, pulse2[:100]/height2, label = 'Neutron Pulse', color='g',linestyle='--')
            plt.xlabel('Time (ns)')
            plt.ylabel('Ratio to Max Pulse Height')
            plt.legend()
            plt.show()'''

    #Used to compare with CoMPASS plots.
    '''spectrum\_data = pd.read\_csv('BaselineHeld\_G8138\_and\_N8125\_PSDCut\_CompassSpectrums.csv')
    #total\_pulses\_psd = []
    total\_pulses\_psd1 = []
    total\_pulses\_psd2 = []
    data\_interval = 1/16383
    psd\_values = np.arange(0, 1 + data\_interval, data\_interval)
    psd\_data1 = spectrum\_data['Gamma PSD'].to\_numpy()
    psd\_data2 = spectrum\_data['Neutron PSD'].to\_numpy()
    #psd\_data1 = spectrum\_data['LE - PSD'].to\_numpy()
    #psd\_data2 = spectrum\_data['CFD - PSD'].to\_numpy()
    for i in range(len(psd\_data1)):
        how\_many = psd\_data1[i]
        if how\_many > 0:
            total\_pulses\_psd1 += [psd\_values[i] for j in range(how\_many)]
    for i in range(len(psd\_data2)):
        how\_many = psd\_data2[i]
        if how\_many > 0:
            total\_pulses\_psd2 += [psd\_values[i] for j in range(how\_many)]'''
```

```
if path.exists(f'{neutron\_file[:-5]}{1}.csv'):
    plt.hist(gamma\_tail\_to\_total, bins = 200, range = (0,0.6), label = 'Gamma Detector', color='blue',
    histtype='step')        #Plots histogram of tail to total ratio for each pulse.
    plt.hist(neutron\_tail\_to\_total, bins = 200, range = (0,0.6), alpha=0.6, label = 'Neutron Detector', color='orange', histtype='step')
    plt.legend(loc='upper right')
    plt.xlabel('Tail to Total Ratio')
    plt.ylabel('Number of Pulses (Counts)')
    plt.xticks(np.arange(0, 0.7, 0.1))
    plt.ylim(0,800)
    plt.show()
    #plt.hist(total\_pulses\_psd1, bins = 2000, range = (0,1), alpha=0.7, label = 'Compass PSD Plot', color='cyan')

    #Plot with error bars
    y, bin\_edges = np.histogram(gamma\_tail\_to\_total, bins= 200, range = (0, 0.6))
    bin\_centers = 0.5*(bin\_edges[1:] + bin\_edges[:-1])
    plt.errorbar(bin\_centers, y, yerr = y**0.5, capsize = 2, marker = '.', drawstyle = 'steps-mid', ecolor = 'dimgrey',
    elinewidth = 1.5, linewidth = 1.5, markeredgewidth = 0, markersize = 0.1, label = 'Gamma Detector', color='blue')

    y, bin\_edges = np.histogram(neutron\_tail\_to\_total, bins= 200, range = (0, 0.6))
    bin\_centers = 0.5*(bin\_edges[1:] + bin\_edges[:-1])
    plt.errorbar(bin\_centers, y, yerr = y**0.5, capsize = 2, marker = '.', drawstyle = 'steps-mid', ecolor = 'dimgrey',
    elinewidth = 1.5, linewidth = 2, markeredgewidth = 0, markersize = 0.1, label = 'Neutron Detector', color='red', ls = ':')

    plt.ylim(0,800)
    plt.legend(loc='upper right')
    plt.xlabel('Tail to Total Ratio')
    plt.ylabel('Number of Pulses (Counts)')
    plt.xticks(np.arange(0, 0.7, 0.1))
    plt.show()

else:
    if f'{gamma\_file[0]}' == 'G':
        plt.hist(gamma\_tail\_to\_total, bins = 500, range = (0,1), label = 'My PSD Plot', color='red')
                        #Plots histogram of tail to total ratio for each pulse.
        #plt.hist(total\_pulses\_psd1, bins = 500, range = (0,1), alpha=0.7, label = 'Compass PSD Plot', color='blue')
        #plt.hist(total\_pulses\_psd1, bins = 500, range = (0,1),  label = 'Leading Edge PSD')
        #plt.hist(total\_pulses\_psd2, bins = 500, range = (0,1), alpha=0.6, label = 'CFD PSD')
        plt.legend(loc='upper right')
        plt.xlabel('Tail to Total Ratio')
        plt.ylabel('Number of Pulses (Counts)')
        plt.xticks(np.arange(0, 1.1, 0.1))
        #plt.yscale('log')
        plt.show()
    if f'{gamma\_file[0]}' == 'N':
        plt.hist(gamma\_tail\_to\_total, bins = 500, range = (0,1), label = 'My PSD Plot', color='red')
                            #Plots histogram of tail to total ratio for each pulse.
        #plt.hist(total\_pulses\_psd2, bins = 500, range = (0,1), alpha=0.7, label = 'Compass PSD Plot', color='blue')
        #plt.hist(total\_pulses\_psd1, bins = 500, range = (0,1),  label = 'Leading Edge PSD')
        #plt.hist(total\_pulses\_psd2, bins = 500, range = (0,1), alpha=0.6, label = 'CFD PSD')
        plt.legend(loc='upper right')
        plt.xlabel('Tail to Total Ratio')
        plt.ylabel('Number of Pulses (Counts)')
        plt.xticks(np.arange(0, 1.1, 0.1))
        #plt.yscale('log')
```

```
            plt.show()

    if path.exists(f'{gamma\_file[:-21]}\_PSD\_Data{1}.csv'):
        x = 1
    else:
        for i in range(math.ceil(len(index\_of\_discrimination1)/50000)):
            start\_pulse = i*50000
            final\_pulse = start\_pulse + 50000
            if len(index\_of\_discrimination1) < final\_pulse:
                final\_pulse = len(index\_of\_discrimination1)
            f = open(f'{gamma\_file[:-21]}\_PSD\_Data{i+1}.csv', 'w')
            f.write(",".join([f"time␣{index\_of\_discrimination1[k]+1}␣[ns],pulse␣{index\_of\_discrimination1[k]+1}␣[mV],
␣␣␣␣␣␣␣␣␣␣␣␣tail/total␣{index\_of\_discrimination1[k]+1}" for k in range(start\_pulse, final\_pulse)])+ "\n")
            for j in range(sample\_length):
                f.write(",".join([f"{time\_data1[index\_of\_discrimination2[k]][j]},{pulse\_data1[index\_of\_discrimination1[k]][j]},
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣{gamma\_tail\_to\_total[index\_of\_discrimination1[k]]}" for k in range(start\_pulse, final\_pulse)]) + "\n")
            f.close()

    if path.exists(f'{neutron\_file[:-21]}\_PSD\_Data{1}.csv'):
        x = 1
    else:
        if path.exists(f'{neutron\_file[:-5]}{1}.csv'):
            for i in range(math.ceil(len(index\_of\_discrimination2)/50000)):
                start\_pulse = i*50000
                final\_pulse = start\_pulse + 50000
                if len(index\_of\_discrimination2) < final\_pulse:
                    final\_pulse = len(index\_of\_discrimination2)
                f = open(f'{neutron\_file[:-21]}\_PSD\_Data{i+1}.csv', 'w')
                f.write(",".join([f"time␣{index\_of\_discrimination2[k]+1}␣[ns],pulse␣{index\_of\_discrimination2[k]+1}␣[mV],
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣tail/total␣{index\_of\_discrimination2[k]+1}" for k in range(start\_pulse, final\_pulse)])+ "\n")
                for j in range(sample\_length):
                    f.write(",".join([f"{time\_data2[index\_of\_discrimination2[k]][j]},{pulse\_data2[index\_of\_discrimination2[k]][j]},
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣{neutron\_tail\_to\_total[index\_of\_discrimination2[k]]}" for k in range(start\_pulse, final\_pulse)]) + "\n")
                f.close()

    print(f'Number␣of␣Negative␣Gamma␣PSD␣Pulses␣=␣{gamma\_negative}')
    print(f'Number␣of␣Negative␣Neutron␣PSD␣Pulses␣=␣{neutron\_negative}')
    print(f'Total␣Number␣of␣Pulses␣=␣{total\_number}')


def PHD\_Plot(file, adc\_axis\_max, baseline):
## This function makes a pulse heigh distribution plot of a raw data file (not a file already run through the pulse correction function)

    num\_rows = sum(1 for line in open(file)) - 1

    pulse\_heights\_ADC = np.zeros(num\_rows)
    pulse\_heights\_mV = np.zeros(num\_rows)


    with open(file, newline='') as f:
        csv\_reader = csv.reader(f, delimiter=';')
        for counter,line in enumerate(csv\_reader):
            if counter > 0:
                #baseline = max([float(p) for p in line[4:len(line)]])
                pulse\_heights\_ADC[counter-1] = (-float(min(line[4:len(line)]))+baseline)
                pulse\_heights\_mV[counter-1] = (-float(min(line[4:len(line)]))+baseline)*0.1220703125
```

```
#print(f'Average Pulse Height: {np.mean(pulse\_heights\_mV)}')
'''y1, x1, \_ = plt.hist(pulse\_heights\_mV, bins = int(adc\_axis\_max*0.1220703125//11), range = (0,
int(adc\_axis\_max*0.1220703125)))
print(f'Most Common PHD: {x1[np.where(y1 == y1.max())][0]}, Number of Pulses: {y1.max()}')'''

'''spectrum\_data = pd.read\_csv('BaselineHeld\_G8139\_and\_N8127\_PSDCut\_CompassSpectrums.csv')
total\_pulses\_energy = []
#total\_pulses\_energy1 = []
#total\_pulses\_energy2 = []

energy\_channels = np.arange(0, 16384, 1, dtype=np.float64)
energy\_data = spectrum\_data['Neutron Energy'].to\_numpy()
#energy\_data1 = spectrum\_data['LE - PHD'].to\_numpy()
#energy\_data2 = spectrum\_data['CFD - PHD'].to\_numpy()'''

'''for i in range(len(energy\_data)-1):
    how\_many = int(energy\_data[i])
    if how\_many > 0:
        total\_pulses\_energy += [int(energy\_channels[i]) for j in range(how\_many)]'''
"""for i in range(len(energy\_data1)-1):
    how\_many = float(energy\_data1[i])
    if how\_many > 0:
        total\_pulses\_energy1 += [float(energy\_channels[i]) for j in range(how\_many)]
for i in range(len(energy\_data2)-1):
    how\_many = float(energy\_data2[i])
    if how\_many > 0:
        total\_pulses\_energy2 += [float(energy\_channels[i]) for j in range(how\_many)]"""

mpl.rc('font',family='Times␣New␣Roman')
mpl.rc('font', size = 16)
'''plt.hist(pulse\_heights\_ADC, bins = int(adc\_axis\_max//11), range = (0, adc\_axis\_max), label = 'My PHD Plot', color='red')
#plt.hist(pulse\_heights\_ADC, bins = int(adc\_axis\_max//11), range = (0, adc\_axis\_max), color='red', histtype='step')
#plt.hist(total\_pulses\_energy, bins = int(adc\_axis\_max//11), range = (0, adc\_axis\_max), alpha=0.6, label = 'Compass PHD Plot',
color='y')
#plt.hist(total\_pulses\_energy1, bins = float(adc\_axis\_max//11), range = (0, adc\_axis\_max), label = 'Leading Edge PHD')
#plt.hist(total\_pulses\_energy2, bins = float(adc\_axis\_max//11), range = (0, adc\_axis\_max), alpha=0.6, label = 'CFD PHD')
plt.legend(loc='upper right')
plt.xlabel('ADC Channel')
plt.ylabel('Number of Pulses (Counts)')
#plt.ylim((0, 20000))
#plt.xticks(np.arange(0, adc\_axis\_max+1, 500))
plt.show()
plt.hist(pulse\_heights\_mV, bins = int(adc\_axis\_max*0.1220703125//11), range = (0, int(adc\_axis\_max*0.1220703125)))
plt.xlabel('Voltage (mV)')
plt.ylabel('Number of Pulses (Counts)')
#plt.ylim((0, 20000))
plt.show()'''

#Plot with error bars
y, bin\_edges = np.histogram(pulse\_heights\_mV, bins = int(adc\_axis\_max*0.1220703125//11), range = (0,
int(adc\_axis\_max*0.1220703125)))
bin\_centers = 0.5*(bin\_edges[1:] + bin\_edges[:-1])
plt.errorbar(bin\_centers, y, yerr = y**0.5, capsize = 2, marker = '.', drawstyle = 'steps-mid', ecolor = 'dimgrey',
elinewidth = 2, linewidth = 2, markeredgewidth = 0, markersize = 0.1, color='red')
plt.yscale('log')
```

```
    plt.xlim((0,900))
    plt.xlabel('Voltage (mV)')
    plt.ylabel('Number of Pulses (Counts)')
    plt.show()


def ToF\_Analysis(gamma\_file, neutron\_file, time\_axis\_min, time\_axis\_max, long\_gate = 360, pregate = 50,
short\_gate = 70, cfd\_delay = 6, attenuation\_fraction = 0.25):
## This function makes a time of flight plot of the neutron times - gamma times, where you select the minimum and maximum
#points on the x-axis.

    num\_total\_pulses = sum(1 for line in open(f'{gamma\_file[:-21]}.csv')) - 1
    sample\_length = (sum(1 for line in open(f'{gamma\_file[:-5]}1.csv')) - 1)
    first\_file = pd.read\_csv(f'{gamma\_file[:-5]}1.csv')
    pulses\_per\_csv = len(first\_file.columns)//2

    time\_data1 = np.zeros(shape=(num\_total\_pulses, sample\_length))
    pulse\_data1 = np.zeros(shape=(num\_total\_pulses, sample\_length))
    time\_data2 = np.zeros(shape=(num\_total\_pulses, sample\_length))
    pulse\_data2 = np.zeros(shape=(num\_total\_pulses, sample\_length))
    gamma\_psd = 0
    neutron\_psd = 0

    num\_pulses\_inserted1 = 0
    for i in range(math.ceil(num\_total\_pulses/pulses\_per\_csv)):
        if path.exists(f'{gamma\_file[:-5]}{i+1}.csv'):
            data = pd.read\_csv(f'{gamma\_file[:-5]}{i+1}.csv')
            num\_pulses = len(data.columns)//2
            time\_data1[num\_pulses\_inserted1:(num\_pulses + num\_pulses\_inserted1)] = data[[f'time {j+1} [ns]' for j in
            range(num\_pulses\_inserted1, num\_pulses + num\_pulses\_inserted1)]].to\_numpy().T
            pulse\_data1[num\_pulses\_inserted1:(num\_pulses + num\_pulses\_inserted1)] = data[[f'pulse {j+1} [mV]' for j in
            range(num\_pulses\_inserted1, num\_pulses + num\_pulses\_inserted1)]].to\_numpy().T
            num\_pulses\_inserted1 += num\_pulses
        else:
            break
    num\_pulses\_inserted2 = 0
    for i in range(math.ceil(num\_total\_pulses/pulses\_per\_csv)):
        if path.exists(f'{neutron\_file[:-5]}{i+1}.csv'):
            data = pd.read\_csv(f'{neutron\_file[:-5]}{i+1}.csv')
            num\_pulses = len(data.columns)//2
            time\_data2[num\_pulses\_inserted2:(num\_pulses + num\_pulses\_inserted2)] = data[[f'time {j+1} [ns]' for j in
            range(num\_pulses\_inserted2, num\_pulses + num\_pulses\_inserted2)]].to\_numpy().T
            pulse\_data2[num\_pulses\_inserted2:(num\_pulses + num\_pulses\_inserted2)] = data[[f'pulse {j+1} [mV]' for j in
            range(num\_pulses\_inserted2, num\_pulses + num\_pulses\_inserted2)]].to\_numpy().T
            num\_pulses\_inserted2 += num\_pulses
        else:
            break



    num\_discriminated\_gammas = 0
    num\_discriminated\_neutrons = 0
    if path.exists(f'{gamma\_file[:-21]}\_PSD\_Data{1}.csv') and path.exists(f'{neutron\_file[:-21]}\_PSD\_Data{1}.csv'):
        for i in range(50):
            if path.exists(f'{gamma\_file[:-21]}\_PSD\_Data{i+1}.csv'):
                gamma\_data = pd.read\_csv(f'{gamma\_file[:-21]}\_PSD\_Data{i+1}.csv')
                num\_discriminated\_gammas += len(gamma\_data.columns)//3
```

```
        if path.exists(f'{neutron\_file[:-21]}\_PSD\_Data{i+1}.csv'):
            neutron\_data = pd.read\_csv(f'{neutron\_file[:-21]}\_PSD\_Data{i+1}.csv')
            num\_discriminated\_neutrons += len(neutron\_data.columns)//3

gamma\_time\_data = np.zeros(shape=(num\_discriminated\_gammas, sample\_length))
gamma\_pulse\_data = np.zeros(shape=(num\_discriminated\_gammas, sample\_length))
gamma\_psd = np.zeros(num\_discriminated\_gammas)
neutron\_time\_data = np.zeros(shape=(num\_discriminated\_neutrons, sample\_length))
neutron\_pulse\_data = np.zeros(shape=(num\_discriminated\_neutrons, sample\_length))
neutron\_psd = np.zeros(num\_discriminated\_neutrons)

num\_pulses\_inserted3 = 0
if path.exists(f'{gamma\_file[:-21]}\_PSD\_Data{1}.csv') and path.exists(f'{neutron\_file[:-21]}\_PSD\_Data{1}.csv'):
    for i in range(math.ceil(num\_discriminated\_gammas/50000)):
        if path.exists(f'{gamma\_file[:-21]}\_PSD\_Data{i+1}.csv'):
            gamma\_data = pd.read\_csv(f'{gamma\_file[:-21]}\_PSD\_Data{i+1}.csv')
            num\_pulses = len(gamma\_data.columns)//3
            gamma\_time\_data[num\_pulses\_inserted3:(num\_pulses + num\_pulses\_inserted3)] = gamma\_data[[f'time␣{j+1}␣[ns]' for
            j in range(num\_pulses\_inserted3, num\_pulses + num\_pulses\_inserted3)]].to\_numpy().T
            gamma\_pulse\_data[num\_pulses\_inserted3:(num\_pulses + num\_pulses\_inserted3)] = gamma\_data[[f'pulse␣{j+1}␣[mV]' for
            j in range(num\_pulses\_inserted3, num\_pulses + num\_pulses\_inserted3)]].to\_numpy().T
            gamma\_psd[num\_pulses\_inserted3:(num\_pulses + num\_pulses\_inserted3)] = gamma\_data[[f'tail/total␣{j+1}' for
            j in range(num\_pulses\_inserted3, num\_pulses + num\_pulses\_inserted3)]].loc[0].to\_numpy().T
            num\_pulses\_inserted3 += num\_pulses

num\_pulses\_inserted4 = 0
if path.exists(f'{gamma\_file[:-21]}\_PSD\_Data{1}.csv') and path.exists(f'{neutron\_file[:-21]}\_PSD\_Data{1}.csv'):
    for i in range(math.ceil(num\_discriminated\_neutrons/50000)):
        if path.exists(f'{neutron\_file[:-21]}\_PSD\_Data{i+1}.csv'):
            neutron\_data = pd.read\_csv(f'{neutron\_file[:-21]}\_PSD\_Data{i+1}.csv')
            num\_pulses = len(neutron\_data.columns)//3
            neutron\_time\_data[num\_pulses\_inserted4:(num\_pulses + num\_pulses\_inserted4)] = neutron\_data[[f'time␣{j+1}␣[ns]' for
            j in range(num\_pulses\_inserted4, num\_pulses + num\_pulses\_inserted4)]].to\_numpy().T
            neutron\_pulse\_data[num\_pulses\_inserted4:(num\_pulses + num\_pulses\_inserted4)] = neutron\_data[[f'pulse␣{j+1}␣[mV]' for
            j in range(num\_pulses\_inserted4, num\_pulses + num\_pulses\_inserted4)]].to\_numpy().T
            neutron\_psd[num\_pulses\_inserted4:(num\_pulses + num\_pulses\_inserted4)] = neutron\_data[[f'tail/total␣{j+1}' for
            j in range(num\_pulses\_inserted4, num\_pulses + num\_pulses\_inserted4)]].loc[0].to\_numpy().T
            num\_pulses\_inserted4 += num\_pulses


gamma\_timetags = np.zeros(num\_total\_pulses)
neutron\_timetags = np.zeros(num\_total\_pulses)
times\_of\_flight = np.zeros(num\_total\_pulses)
gamma\_timetags\_psd = np.zeros(num\_discriminated\_gammas)
neutron\_timetags\_psd = np.zeros(num\_total\_pulses)
if num\_discriminated\_gammas >= num\_discriminated\_neutrons:
    times\_of\_flight\_psd = np.zeros(num\_discriminated\_neutrons)
else:
    times\_of\_flight\_psd = np.zeros(num\_discriminated\_gammas)

inverted\_delayed\_signal1 = np.zeros(sample\_length)
attenuated\_signal1 = np.zeros(sample\_length)
shaped\_signal1 = np.zeros(sample\_length)
inverted\_delayed\_signal2 = np.zeros(sample\_length)
attenuated\_signal2 = np.zeros(sample\_length)
shaped\_signal2 = np.zeros(sample\_length)
```

```
for counter,(pulse1,time1,pulse2,time2) in enumerate(zip(pulse\_data1, time\_data1, pulse\_data2, time\_data2)):

    inverted\_delayed\_signal1[(cfd\_delay//2):] = pulse1[:-(cfd\_delay//2)]
    attenuated\_signal1 = -attenuation\_fraction*pulse1
    shaped\_signal1 = inverted\_delayed\_signal1 + attenuated\_signal1
    shaped\_signal\_flipped1 = np.flip(shaped\_signal1)
    inverted\_delayed\_signal2[(cfd\_delay//2):] = pulse2[:-(cfd\_delay//2)]
    attenuated\_signal2 = -attenuation\_fraction*pulse2
    shaped\_signal2 = inverted\_delayed\_signal2 + attenuated\_signal2
    shaped\_signal\_flipped2 = np.flip(shaped\_signal2)

    for counter1,i in enumerate(shaped\_signal\_flipped1):
        if i >= max(shaped\_signal\_flipped1):
            index\_max\_height1 = counter1
    for counter2,i in enumerate(shaped\_signal\_flipped1[index\_max\_height1:]):
        if i <= 0:
            index\_sbzc1 = (sample\_length - 1) - (counter2 + index\_max\_height1)
            index\_sazc1 = index\_sbzc1 + 1
            t\_fine1 = (-float(shaped\_signal1[index\_sbzc1])/(float(shaped\_signal1[index\_sazc1]) -
            float(shaped\_signal1[index\_sbzc1]))) * 2.0
            timetag1 = float(time1[index\_sbzc1]) + float(t\_fine1)
            break
    for counter3,i in enumerate(shaped\_signal\_flipped2):
        if i >= max(shaped\_signal\_flipped2):
            index\_max\_height2 = counter3
    for counter4,i in enumerate(shaped\_signal\_flipped2[index\_max\_height2:]):
        if i <= 0:
            index\_sbzc2 = (sample\_length - 1) - (counter4 + index\_max\_height2)
            index\_sazc2 = index\_sbzc2 + 1
            t\_fine2 = (-float(shaped\_signal2[index\_sbzc2])/(float(shaped\_signal2[index\_sazc2]) -
            float(shaped\_signal2[index\_sbzc2]))) * 2.0
            timetag2 = float(time2[index\_sbzc2]) + float(t\_fine2)
            break

    gamma\_timetags[counter] = timetag1
    neutron\_timetags[counter] = timetag2

for counter, (gamma\_timetag,neutron\_timetag) in enumerate(zip(gamma\_timetags,neutron\_timetags)):
    times\_of\_flight[counter] = neutron\_timetag - gamma\_timetag


if path.exists(f'{gamma\_file[:-21]}\_PSD\_Data{1}.csv') and path.exists(f'{neutron\_file[:-21]}\_PSD\_Data{1}.csv'):
    for counter,(pulse1,time1,pulse2,time2) in enumerate(zip(gamma\_pulse\_data, gamma\_time\_data, neutron\_pulse\_data,
    neutron\_time\_data)):

        inverted\_delayed\_signal1[(cfd\_delay//2):] = pulse1[:-(cfd\_delay//2)]
        attenuated\_signal1 = -attenuation\_fraction*pulse1
        shaped\_signal1 = inverted\_delayed\_signal1 + attenuated\_signal1
        shaped\_signal\_flipped1 = np.flip(shaped\_signal1)
        inverted\_delayed\_signal2[(cfd\_delay//2):] = pulse2[:-(cfd\_delay//2)]
        attenuated\_signal2 = -attenuation\_fraction*pulse2
        shaped\_signal2 = inverted\_delayed\_signal2 + attenuated\_signal2
        shaped\_signal\_flipped2 = np.flip(shaped\_signal2)
```

```
        for counter1,i in enumerate(shaped\_signal\_flipped1):
            if i >= max(shaped\_signal\_flipped1):
                index\_max\_height1 = counter1
        for counter2,i in enumerate(shaped\_signal\_flipped1[index\_max\_height1:]):
            if i <= 0:
                index\_sbzc1 = (sample\_length - 1) - (counter2 + index\_max\_height1)
                index\_sazc1 = index\_sbzc1 + 1
                t\_fine1 = (-float(shaped\_signal1[index\_sbzc1])/(float(shaped\_signal1[index\_sazc1]) -
                float(shaped\_signal1[index\_sbzc1]))) * 2.0
                timetag1 = float(time1[index\_sbzc1]) + float(t\_fine1)
                break
        for counter3,i in enumerate(shaped\_signal\_flipped2):
            if i >= max(shaped\_signal\_flipped2):
                index\_max\_height2 = counter3
        for counter4,i in enumerate(shaped\_signal\_flipped2[index\_max\_height2:]):
            if i <= 0:
                index\_sbzc2 = (sample\_length - 1) - (counter4 + index\_max\_height2)
                index\_sazc2 = index\_sbzc2 + 1
                t\_fine2 = (-float(shaped\_signal2[index\_sbzc2])/(float(shaped\_signal2[index\_sazc2]) -
                float(shaped\_signal2[index\_sbzc2]))) * 2.0
                timetag2 = float(time2[index\_sbzc2]) + float(t\_fine2)
                break

        gamma\_timetags\_psd[counter] = timetag1
        neutron\_timetags\_psd[counter] = timetag2

for counter, (gamma\_timetag,neutron\_timetag) in enumerate(zip(gamma\_timetags\_psd,neutron\_timetags\_psd)):
    times\_of\_flight\_psd[counter] = neutron\_timetag - gamma\_timetag

'''y1, x1, \_ = plt.hist(times\_of\_flight, bins= int((abs(time\_axis\_min)+abs(time\_axis\_max))*2), range =
(int(time\_axis\_min), int(time\_axis\_max)))
print(f'Most Common TOF: {x1[np.where(y1 == y1.max())][0]}, Number of Pulses: {y1.max()}')'''

mpl.rc('font',family='Times␣New␣Roman')
mpl.rc('font', size = 16)
#plt.hist(times\_of\_flight, bins = int((abs(time\_axis\_min)+abs(time\_axis\_max))*2), range = (int(time\_axis\_min),
int(time\_axis\_max)))
plt.hist(times\_of\_flight, bins = int((abs(time\_axis\_min)+abs(time\_axis\_max))*2), range = (int(time\_axis\_min),
int(time\_axis\_max)), histtype='step')
#plt.yscale('log')
plt.xlabel('Time␣of␣Flight␣(ns)')
plt.title('Total␣Time␣of␣Flight')
plt.ylabel('Number␣of␣Pulses␣(Counts)')
plt.show()

#Plotting PSD histograms after a TOF cut, includes error bars
tof\_cut\_gamma\_psd = []
tof\_cut\_neutron\_psd = []
for counter,(g\_psd, n\_psd, tof) in enumerate(zip(gamma\_psd, neutron\_psd, times\_of\_flight\_psd)):
    if tof >= 20:
        tof\_cut\_gamma\_psd.append(g\_psd)
        tof\_cut\_neutron\_psd.append(n\_psd)
y, bin\_edges = np.histogram(tof\_cut\_gamma\_psd, bins= 200, range = (0, 0.6))
bin\_centers = 0.5*(bin\_edges[1:] + bin\_edges[:-1])
plt.errorbar(bin\_centers, y, yerr = y**0.5, capsize = 2, marker = '.', drawstyle = 'steps-mid', ecolor = 'dimgrey',
elinewidth = 1.5, linewidth = 1.5, markeredgewidth = 0, markersize = 0.1, label = 'Gamma␣Detector', color='blue')
```

```
    y, bin\_edges = np.histogram(tof\_cut\_neutron\_psd, bins= 200, range = (0, 0.6))
    bin\_centers = 0.5*(bin\_edges[1:] + bin\_edges[:-1])
    plt.errorbar(bin\_centers, y, yerr = y**0.5, capsize = 2, marker = '.', drawstyle = 'steps-mid', ecolor = 'dimgrey',
    elinewidth = 2, linewidth = 1.5, markeredgewidth = 0, markersize = 0.1, label = 'Neutron␣Detector', color='red', ls = ':')
    plt.legend(loc='upper␣right')
    plt.xlabel('Tail␣to␣Total␣Ratio')
    plt.ylabel('Number␣of␣Pulses␣(Counts)')
    plt.ylim(0,300)
    plt.xticks(np.arange(0, 0.7, 0.1))
    plt.show()

    #Plot with error bars
    y, bin\_edges = np.histogram(times\_of\_flight, bins= int((abs(time\_axis\_min)+abs(time\_axis\_max))*2), range =
    (int(time\_axis\_min), int(time\_axis\_max)))
    bin\_centers = 0.5*(bin\_edges[1:] + bin\_edges[:-1])
    plt.errorbar(bin\_centers, y, yerr = y**0.5, capsize = 2, marker = '.', drawstyle = 'steps-mid', ecolor = 'dimgrey',
    elinewidth = 1.5, linewidth = 1.5, markeredgewidth = 0, markersize = 0.1, color = 'blue')
    plt.xlabel('Time␣Cross-Correlation␣(ns)')
    #plt.title('Total Time of Flight')
    plt.ylabel('Number␣of␣Pulses␣(Counts)')
    plt.ylim(0,400)
    plt.show()

    if path.exists(f'{gamma\_file[:-21]}\_PSD\_Data{1}.csv') and path.exists(f'{neutron\_file[:-21]}\_PSD\_Data{1}.csv'):
        plt.hist(times\_of\_flight\_psd, bins = int((abs(time\_axis\_min)+abs(time\_axis\_max))*2), range = (int(time\_axis\_min),
        int(time\_axis\_max)))
        plt.title('Time␣of␣Flight␣from␣PSD␣Restriction')
        #plt.yscale('log')
        plt.xlabel('Time␣of␣Flight␣(ns)')
        plt.ylabel('Number␣of␣Pulses␣(Counts)')
        plt.show()

    if path.exists(f'{gamma\_file[:-21]}\_PSD\_Data{1}.csv') and path.exists(f'{neutron\_file[:-21]}\_PSD\_Data{1}.csv'):
        f = open(f'{gamma\_file[6:-21]}\_ToF\_and\_PSD.csv', 'w')
        f.write(",".join(['pulse,time␣of␣flight,gamma␣psd,neutron␣psd'])+ "\n")
        for i in range(len(times\_of\_flight)):
            if times\_of\_flight[i] <= 30 and times\_of\_flight[i] >= 21:
                f.write(",".join([f"{i},{times\_of\_flight\_psd[i]},{gamma\_psd[i]},{neutron\_psd[i]}"]) + "\n")
        f.close()



def ToF\_Comparison(gamma\_file, neutron\_file, show\_times, max\_time\_difference, min\_time\_difference = 2, max\_height\_cutoff =
10000, min\_height\_cutoff = 200, long\_gate = 360, pregate = 50, short\_gate = 70, cfd\_delay = 6, attenuation\_fraction = 0.25):
## This function compares gamma and neutron pulse pairs, then plots and records the pairs that fit the specified constraints.

    num\_total\_pulses = sum(1 for line in open(f'{gamma\_file[:-21]}.csv')) - 1
    sample\_length = (sum(1 for line in open(f'{gamma\_file[:-5]}1.csv')) - 1)
    first\_file = pd.read\_csv(f'{gamma\_file[:-5]}1.csv')
    pulses\_per\_csv = len(first\_file.columns)//2

    time\_data1 = np.zeros(shape=(num\_total\_pulses, sample\_length))
    pulse\_data1 = np.zeros(shape=(num\_total\_pulses, sample\_length))
    time\_data2 = np.zeros(shape=(num\_total\_pulses, sample\_length))
    pulse\_data2 = np.zeros(shape=(num\_total\_pulses, sample\_length))
```

```
num\_pulses\_inserted1 = 0
for i in range(math.ceil(num\_total\_pulses/pulses\_per\_csv)):
    if path.exists(f'{gamma\_file[:-5]}{i+1}.csv') == True:
        data = pd.read\_csv(f'{gamma\_file[:-5]}{i+1}.csv')
        num\_pulses = len(data.columns)//2
        time\_data1[num\_pulses\_inserted1:(num\_pulses + num\_pulses\_inserted1)] = data[[f'time␣{j+1}␣[ns]' for j in
        range(num\_pulses\_inserted1, num\_pulses + num\_pulses\_inserted1)]].to\_numpy().T
        pulse\_data1[num\_pulses\_inserted1:(num\_pulses + num\_pulses\_inserted1)] = data[[f'pulse␣{j+1}␣[mV]' for j in
        range(num\_pulses\_inserted1, num\_pulses + num\_pulses\_inserted1)]].to\_numpy().T
        num\_pulses\_inserted1 += num\_pulses
    else:
        break
num\_pulses\_inserted2 = 0
for i in range(math.ceil(num\_total\_pulses/pulses\_per\_csv)):
    if path.exists(f'{neutron\_file[:-5]}{i+1}.csv') == True:
        data = pd.read\_csv(f'{neutron\_file[:-5]}{i+1}.csv')
        num\_pulses = len(data.columns)//2
        time\_data2[num\_pulses\_inserted2:(num\_pulses + num\_pulses\_inserted2)] = data[[f'time␣{j+1}␣[ns]' for j in
        range(num\_pulses\_inserted2, num\_pulses + num\_pulses\_inserted2)]].to\_numpy().T
        pulse\_data2[num\_pulses\_inserted2:(num\_pulses + num\_pulses\_inserted2)] = data[[f'pulse␣{j+1}␣[mV]' for j in
        range(num\_pulses\_inserted2, num\_pulses + num\_pulses\_inserted2)]].to\_numpy().T
        num\_pulses\_inserted2 += num\_pulses
    else:
        break

inverted\_delayed\_signal1 = np.zeros(sample\_length)
attenuated\_signal1 = np.zeros(sample\_length)
shaped\_signal1 = np.zeros(sample\_length)
inverted\_delayed\_signal2 = np.zeros(sample\_length)
attenuated\_signal2 = np.zeros(sample\_length)
shaped\_signal2 = np.zeros(sample\_length)

index\_of\_discrimination = []
for i,(pulse1,time1,pulse2,time2) in enumerate(zip(pulse\_data1, time\_data1, pulse\_data2, time\_data2)):

    max\_height1 = max(pulse1)
    max\_height2 = max(pulse2)

    inverted\_delayed\_signal1[(cfd\_delay//2):] = pulse1[:-(cfd\_delay//2)]
    attenuated\_signal1 = -attenuation\_fraction*pulse1
    shaped\_signal1 = inverted\_delayed\_signal1 + attenuated\_signal1
    shaped\_signal\_flipped1 = np.flip(shaped\_signal1)
    inverted\_delayed\_signal2[(cfd\_delay//2):] = pulse2[:-(cfd\_delay//2)]
    attenuated\_signal2 = -attenuation\_fraction*pulse2
    shaped\_signal2 = inverted\_delayed\_signal2 + attenuated\_signal2
    shaped\_signal\_flipped2 = np.flip(shaped\_signal2)

    for counter1,i in enumerate(shaped\_signal\_flipped1):
        if i >= max(shaped\_signal\_flipped1):
            index\_max\_height1 = counter1
    for counter2,i in enumerate(shaped\_signal\_flipped1[index\_max\_height1:]):
        if i <= 0:
            index\_sbzc1 = (sample\_length - 1) - (counter2 + index\_max\_height1)
            index\_sazc1 = index\_sbzc1 + 1
            t\_fine1 = (-float(shaped\_signal1[index\_sbzc1])/(float(shaped\_signal1[index\_sazc1]) -
            float(shaped\_signal1[index\_sbzc1]))) * 2.0
```

```
                timetag1 = index\_sbzc1 + t\_fine1
                break
        for counter3 ,i in enumerate(shaped\_signal\_flipped2):
            if i >= max(shaped\_signal\_flipped2):
                index\_max\_height2 = counter3
        for counter4 ,i in enumerate(shaped\_signal\_flipped2[index\_max\_height2:]):
            if i <= 0:
                index\_sbzc2 = (sample\_length - 1) - (counter4 + index\_max\_height2)
                index\_sazc2 = index\_sbzc2 + 1
                t\_fine2 = (-float(shaped\_signal2[index\_sbzc2])/(float(shaped\_signal2[index\_sazc2]) -
                float(shaped\_signal2[index\_sbzc2]))) * 2.0
                timetag2 = index\_sbzc2 + t\_fine2
                break
        mpl.rc('font',family='Times␣New␣Roman')
        mpl.rc('font', size = 16)
        if ((timetag1 + max\_time\_difference) >= timetag2) and ((timetag1 + min\_time\_difference) <= timetag2) and
        (max\_height1 <= max\_height\_cutoff and max\_height2 <= max\_height\_cutoff) and (max\_height1 >= min\_height\_cutoff and
        max\_height2 >= min\_height\_cutoff):
            plt.plot(time1, pulse1)
            plt.xlabel('Time␣(ns)')
            plt.ylabel('Voltage␣(mV)')
            plt.title('Gamma')
            plt.show()
            plt.plot(time2, pulse2)
            plt.xlabel('Time␣(ns)')
            plt.ylabel('Voltage␣(mV)')
            plt.title('Neutron')
            plt.show()

            if (show\_times == 'Show␣Times') or (show\_times == 'show␣times') or (show\_times == 'Show␣times') or (show\_times == 'Show')
            or (show\_times == 'show') or (show\_times == 'Yes') or (show\_times == 'yes'):
                print(f'Gamma␣{i}␣Arrival␣Time:␣{timetag1}␣ns,␣Neutron␣{i}␣Arrival␣Time:␣{timetag2}␣ns')

            index\_of\_discrimination.append(i)
        else:
            continue


    f = open(f'{gamma\_file[:-21]}\_DiscriminatedPulses\_MaxTime{max\_time\_difference}\_MinTime{min\_time\_difference}
␣␣␣␣\_MaxHeight{max\_height\_cutoff}\_MinHeight{min\_height\_cutoff}.csv', 'w')
    f.write(",".join([f"time␣{k}␣[ns],pulse␣{k}␣[mV]" for k in index\_of\_discrimination])+ "\n")
    for j in range(sample\_length):
        f.write(",".join([f"{time\_data1[k][j]},{pulse\_data1[k][j]}" for k in index\_of\_discrimination]) + "\n")
    f.close()

    f = open(f'{neutron\_file[:-21]}\_DiscriminatedPulses\_MaxTime{max\_time\_difference}\_MinTime{min\_time\_difference}
␣␣␣␣\_MaxHeight{max\_height\_cutoff}\_MinHeight{min\_height\_cutoff}.csv', 'w')
    f.write(",".join([f"time␣{k}␣[ns],pulse␣{k}␣[mV]" for k in index\_of\_discrimination])+ "\n")
    for j in range(sample\_length):
        f.write(",".join([f"{time\_data2[k][j]},{pulse\_data2[k][j]}" for k in index\_of\_discrimination]) + "\n")
    f.close()
```