Implementing a New Data Acquisition System for the ALICE TRD Module at UCT

V. Bantchovski, T. Bjerring, S. Cohen, C. Faraday, A. Grindrod, N. Hlengane, K. Khareba, M. Kidson, D. Lewis, M. Matheussen, M. Mobida, T. Schlesinger, F. Smuts, D. T. Spies, S. Thornhill, D. Torbet, N. Yerolemou

November 3, 2021

Abstract

The principal objective of this project was to make use of a new data acquisition system in order to consolidate all operations involved in running a fully integrated setup of a Transition Radiation Detector (TRD) module, from the TRD at ALICE, in conjunction with two scintillation detectors that are used as a trigger for the TRD module, and analyse the resulting data in order to identify the origins of cosmic rays. There were many technical issues which prevented the final data acquisitions from being usable, however an analysis was performed on data from previous projects where possible and an outline of the methods that were to be implemented has been given. Many additions have been made to the overall scope of the project, such as the GUI for the TRDBOX, which simplifies some of the detector controls, and the CORSIKA simulation which gives an indication of expected results. The report includes a full Technical Guide which gives an outline of all of the basic technical skills required to use the tools available in this project. There is also a CORSIKA User Guide which gives direction on the relevant CORSIKA knowledge needed.

Contents

1 Introduction

This report is written in the context of a third-year Physics Major (PHY3004W) group project, under the supervision of Dr. Thomas Dietel.

Located at CERN in Switzerland, the Large Hadron Collider (LHC) is the largest and most energetic particle accelerator in the world. Two particle beams, travelling within the detector in opposite directions, are made to collide at ultra-relativistic speeds. These collisions result in huge amounts of data which is then analysed by experimental particle physicists. The LHC has allowed physicists to explore the theoretical predictions of particle physics, including the Higgs Boson, Supersymmetry and the nature of dark matter. [1]

A Large Ion Collider Experiment (ALICE) at CERN is detector designed for the study of heavy-ion physics, resulting from the ultra-relativistic nuclei collisions provided by the LHC. ALICE specialises in the study of high-density strongly interacting matter, the conditions under which quark-gluon plasma is formed.[2] Just moments after the Big Bang, the universe is thought to have consisted of this primeval state of matter. Understanding the quark-gluon plasma and how it evolves can answer many questions about the early universe as well as quantum chromodynamics — how quarks and gluons combine based on their different 'colors'. [1]

The Transition Radiation Detector (TRD) is situated in the central barrel of ALICE. The main objective of the TRD is to provide electron identification and tracking at momenta that exceed 1 GeV/c. [3]

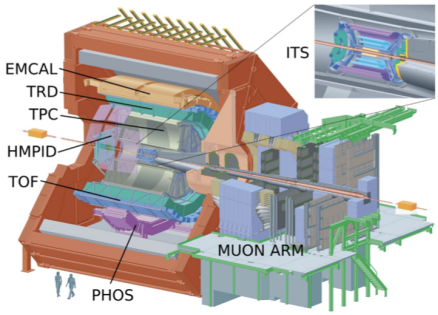


Figure 1.1: The ALICE detector consisting of: the ElectroMagnetic CALorimeter (EMCAL), the Transition Radiation Detector (TRD), the Time Projection Chamber (TPC), the High Momentum Particle Identification detector(HMPID), the Time of Flight detector (TOF), the Photon Spectrometer (PHOS), the Inner Tracking System (ITS) and the forward muon spectrometer arm. [3]

The University of Cape Town is fortunate enough to have a spare module from the TRD which is used, in conjunction with two scintillation detectors, to detect cosmic rays in this project.

The foremost objective of this project is to consolidate all operations necessary to run a fully integrated setup of the TRD module with the scintillation detectors, making use of a new data acquisition system (see section 5). We aim to analyse the captured data in such a way as to identify the origins of the detected cosmic rays. The group was split into different subgroups in charge of the various operations and tasks, each with their own goals that feed into the aim of the project as a whole. The roles of this project are outlined in section 9.

2 Theory

2.1 Cosmic Rays

Cosmic rays are atomic fragments created through various stellar processes which travel through space with energies ranging from the order of 100 MeV up to 1×10^{20} eV. The most common of these is the hydrogen nucleus (a single proton) which is the most abundant element in the universe. This accounts for 89% of the cosmic rays incident on the earth's atmosphere. A further 9% of cosmic rays consist of helium nuclei. The remaining fraction is composed of heavier element nuclei and electrons. For our purposes we are only considering protons in the energy interval between 1 TeV and 1 PeV. Below this interval, muon production becomes exceedingly rare, and cosmic rays above this energy interval are few and far in between.

The atmosphere of Earth is extremely dense in comparison to outer space. Hence, when a cosmic ray enters the atmosphere, it can interact with the particles in the atmosphere to produce electromagnetic radiation and various new

baryons, mesons, and leptons. These can, in turn, react with the atmosphere producing their own set of particles, albeit with reduced energy. The resulting phenomenon is known as a cosmic air shower.

Our interest in the process of cosmic ray showers lies in the production and subsequent detection of muons. The muons are generated by the primary cosmic ray at a distance much greater than the size of the detectors at the observation level, hence they can be assumed to approach the detectors parallel to one another. The surface area covered by these muon showers can range from several square meters to several square kilometers, depending on the energy of the primary cosmic ray.

2.2 The TRD

The TRD is made up of 540 ReadOut drift Chambers (ROC's) that make up 18 supermodules. These are arranged in 5 stacks and 6 layers [4]. Cylindrical coordinates are used to describe the TRD overall, with the origin at the beam intersection, the positive z-axis pointing towards the muon arm and the angle ϕ is the deflection angle in the magnetic field. An exception to this is when discussing the ROC's, where z remains the same, y is in the direction of the wires and x in the direction of electron drift [3]. The module used in this project is the same as that used in Layer 4 of the TRD with a chamber type 0.

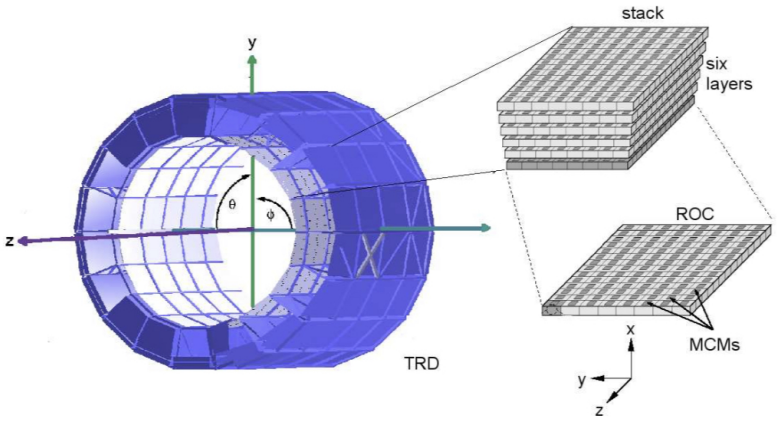


Figure 2.1: TRD barrel with 18 supermodules, showing a zoomed in view of one stack with 6 layers as well as one ROC

One module consists of a multi-wire proportional readout chamber filled with gas, with front end electronics installed on top. The readout chamber is subdivided into three regions: a radiator, a drift region and an amplification region. The radiator sits at the bottom of the chamber and is the region in which transition radiation would occur. The multi-wire proportional chamber, which contains both the drift and amplification regions, is filled with an ionising gas (CERN uses a Xe/CO_2 mixture, while the University of Cape Town module uses an Ar/CO_2 mixture) and is surrounded by two parallel conducting plates called electrodes or pads. In between these plates are two planes of wires, the anode and cathode. These wires serve as the separation points between the drift region and amplification region within the multi-wire proportional chamber. Specifically the cathode wire plane decouples the drift electric field from the amplification region which allows independent adjustment of the drift velocity and gas gain for more precise control. This is discussed more in section 4.4.

Inside the drift region, free electrons in the gas are ionised by high energy particles and then drift towards the cathode wires with a constant drift velocity. As electrons enter the amplification region, they are accelerated towards the anode wires and create an electron avalanche, ionising the gas surrounding the anode wires. Positive ions from this avalanche induce a positive signal on the cathode pad plane. The signals are then digitised using analogue to digital converters (ADC) [5].

Transition radiation occurs when an ultra-relativistic particle crosses the interface between materials with differing dielectric constants [6]. This effect is however not measurable using the module at the University of Cape Town as the momenta of the cosmic rays are typically not sufficient for transition radiation to occur. Additionally, the argon does not have a high enough X-ray photoabsorption probability to see transition radiation in most cases.

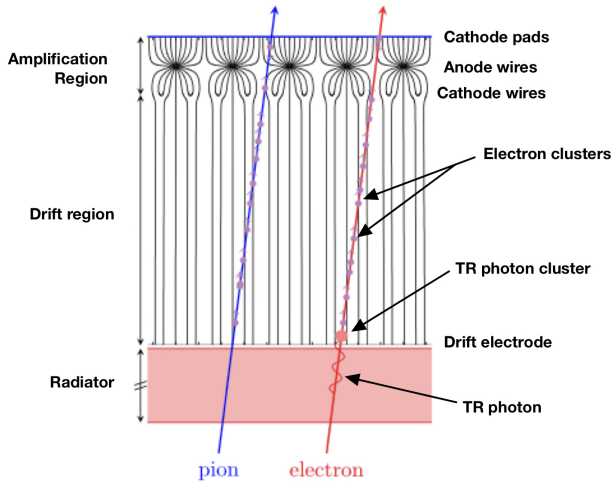


Figure 2.2: Cross section through one ROC with tracks of a pion and electron in order to illustrate the typical ionisation when using the Xe/CO_2 gas mixture as used at CERN. For the ROC used in this project, transition radiation is not measured. The ionisation is represented by so called 'clusters'.

2.3 Scintillation Detectors

The two scintillation detectors used in this experiment were constructed in 2018 and 2019 respectively [7][8]. Their design is based on the detectors used in the HiSPARC project [9], which were designed to detect cosmic rays. The detectors built at UCT were made from an EJ200 scintillation plate, a photomultiplier tube (PMT), and a light guide. They each have a cross-sectional area of $0.5\,\mathrm{m}^2$.

Figure 2.3: One of the scintillation detectors used in this experiment. The large rectangular section houses the scintillation material. The triangular section houses the light guide. The small rectangle on the end houses the photomultiplier tube. The detectors require power from a small 12 V adapter and output their signal via a coaxial cable.

As charged particles enter the scintillation material, they excite electrons in the material which, as they de-excite, emit photons. These photons are always the same energy, so they hold no information about the energy of the charged particle, however the number of excited electrons, and thus the number of photons emitted, is proportional to the energy of the charged particle. The photons are emitted isotropically, so they need to be guided towards the PMT in order to be detected. To do this, the scintillation material was covered in a reflective material and a light guide was fitted to

one side of the scintillation material. This feeds the photons onto the cathode plate of the PMT. Once a photon hits the cathode plate, it produces an electron by the photoelectric effect. This electron, along with others produced by photons from the same event, gets multiplied through an avalanche effect in the PMT before being output as a voltage pulse from the anode. More details on how scintillation detectors work and how they interact with PMTs can be found in [10] and details on the construction of the detectors can be found in [7] and [8].

The primary use of the scintillation detectors in this experiment was to trigger the TRD to begin recording data. This is so that we can be sure the TRD actually has something to detect. However, the scintillation detectors only tell us when a charged particle passes through the scintillation material of that specific detector. This tells us nothing about anything passing through the TRD, so we must turn to the theory of what we are detecting in order to proceed.

As described in section 2.1, when muon showers occur, they usually result in muons that travel parallel to each other, at similar speeds. Using this knowledge, we can set up the scintillation detectors and the TRD such that any coincident pulses from the two detectors likely come from the same muon shower event, and thus we have a better chance of a muon passing through the TRD at the same time. The details of the set-up used are in section 3.

2.4 CORSIKA

CORSIKA (COsmic Ray SImulation KAscade) is a comprehensive cosmic ray shower simulation program used for high-end applications. It is a powerful tool for investigating all manners of cosmic air shower particle interaction scenarios. Detailed parameter and module selection make it capable of simulating very specific conditions. We will be using this software to investigate the behaviour of muons produced when cosmic rays interact with Earth's atmosphere. We can determine their energies and momenta, and their tracks through the sky. The cosmic rays we will be focusing on are protons in the energy range 1 TeV to 1 PeV. Further details on how to obtain and use the software can be found in section 11.

3 Experimental Set-up

Figure 3.1 shows the set-up chosen for operation of the TRD module. The scintillation detectors were used as triggers for the TRD module. One lay underneath the module in order to guarantee that a muon passed through the it, and the other lay a few metres away to confirm that the events which trigger the module are muon showers, as we expect there to be multiple muons travelling together as a result of a shower. The separation distance was chosen as a result of the calculations made in section 6.4.

The two scintillation detectors sent their output signal to both the oscilloscope and the TRDBOX. Inside the TRDBOX, as described in section 4.1, is a system which sends a trigger pulse when two signals from the scintillation detectors arrive within the specified time window. This trigger pulse goes to the oscilloscope as well as the TRD module. The oscilloscope was set to hold the data when it received a trigger, so that the data could be collected if needed. The details on the program used to take data are discussed in section 4.2 and section 5.3.

In order to connect the scintillation detectors to the oscilloscope and TRDBOX, coaxial cables were used. These ran from the detector to a splitter which took the signal to the oscilloscope as well as the TRDBOX. The large diameter cables were used up to the splitter, and then the smaller, brown cables were used from the splitter to the TRDBOX. The splitter plugged directly into the oscilloscope. `PRE-OUT` was connected by another small diameter brown cable to a splitter at the oscilloscope, and then with another brown cable to the TRD module.

An important note about these connections: The coaxial cables need to be terminated properly, or else they produce signals with large oscillations. We had some trouble with terminating the trigger output as the adapters from small diameter to large diameter were lacking. The adapters needed can usually be found in the orange box in the TRD room.

3.1 Set-up of Scintillation Detectors for Determination of Optimal Separation Distance

The set-up used to take data for the event rate at certain separation distances found in section 6.4 was the same as in Figure 3.1, but the TRD module was not included. The code used to take this data was `optimalDistance.py` in [11]. The code waits for the TRDBOX to send a trigger, by monitoring the `0x102` register and seeing if it increases, and then collects the data from the oscilloscope. It ran until 100 events were recorded, for three separation distances, while keeping track of how long the program was running for. The three separation distances were 2.082 m, 2.865 m,

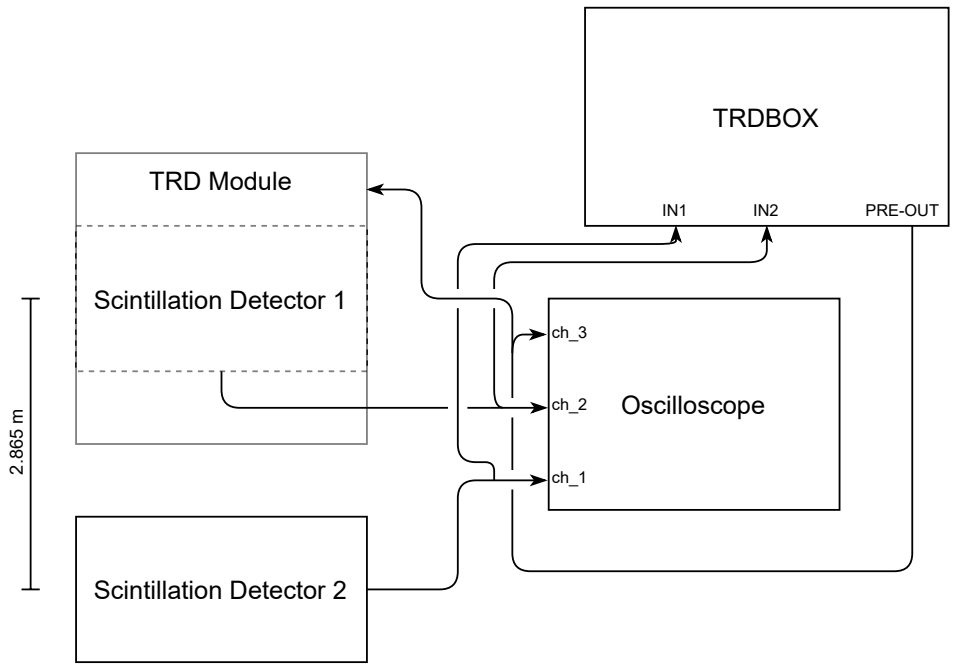


Figure 3.1: Simplified diagram of the set-up of the scintillation detectors with respect to the TRD module for the operation of the scintillation detectors as the triggers for the TRD module. Scintillation detector 1 was underneath the TRD module. Both Scintillation detectors were on the floor and the TRD module was 0.65 m above them. Note that the TRDBOX output labelled "PRE-OUT" is the output that sends the trigger signal. Lengths are not to scale.

and 3.625 m. The distances were chosen as a result of the geometry of the location of the experiment. A fourth distance, 7.5 m, was chosen much further away in order to get an idea of the event rate at far distances and to confirm that closer gave better event rates. Only 50 events were recorded to save time.

3.2 Set-up of Scintillation Detectors for Time Resolution

A specific configuration of the scintillation detectors was used for the measurement of the time resolution. The set-up simply had one of the scintillation detectors on top of the other. Data was taken for the 2018 detector on top and then the detectors were swapped and readings were taken again.

This simple configuration was chosen so that many particles would travel through both detectors therefore leading to many events that could be measured. In addition, the particles would only have to travel a negligible distance between getting detected by the two detectors. Therefore the only contribution to the difference in time between the two outputted signals would be the time resolution of the detectors.

The code used to take this data was `timeResolution.py` in [11]. It works in much the same way as the code used in section 3.1, but 1000 events were recorded per orientation of the detectors as this configuration allowed for a much higher rate of coincident events.

3.3 Set-up of Scintillation Detectors for Angular Resolution

The angular resolution of the TRD was intended to be investigated, but we ran into issues recording the data. The physical set-up was the same as in section 3 but the data from the oscilloscope was needed in order to determine the angular resolution. The details of this calculation are in section 6.5, but we were unable to get the necessary data, so only the process is outlined. The problem we ran into was one of timing. The TRD module requires a trigger to be told when to collect data. As far as we could tell, this had to be manually sent to the TRD with the command mentioned in section 5.1. Unfortunately, this also sent a physical trigger signal to the oscilloscope, effectively wiping the data of the triggering event from the memory of the oscilloscope. We tried collecting the oscilloscope data before sending the trigger, but it took too long to take the data and by that time, the event was no longer “visible” in the TRD module.

This is an area that can be improved upon. We believe that if either the data collection from the oscilloscope can be sped up, as mentioned in section 5.3, or the TRD module can be told to take data without needing to trigger the oscilloscope, this problem would be fixed.

3.4 TRDBOX Settings

The initial TRDBOX settings used for the vast majority of this experiment are given below in Table 3.1. The delay gate generators (DGGs) were set such that the scintillation detectors could read events within 20 cycles of 8.33 ns and still be considered coincident. This was a fairly wide range and, combined with the relatively low thresholds for the discriminators, this meant that the number of coincident counts was likely biased towards the higher side. These settings were intentionally chosen to ensure data could be taken from the TRD chamber at a reasonable rate given the stringent time constraints during this practical. Further details about these settings, along with the commands to change them, can be found below in section 4.1.3.

Setting description	Setting value
Discriminator thresholds	2020 for both
DGG widths	10 cycles at 8.33 ns for both
DGG delays	12 cycles at 8.33 ns for both
LUT value	`08` for automatic triggering
Last value of `pre-conf`	`5` for enabling autoblock and the pretrigger

Table 3.1: Table of specific settings used during this practical for the TRDBOX.

4 TRD Operation

4.1 TRDBOX

The TRDBOX is the link between the TRD module, the scintillation detectors, and the control computer. It is comprised of logic gates and other circuitry whose purpose is to take in a signal from the scintillation detectors and determine if the signal should be sent further downstream to trigger the TRD module to record data. The TRDBOX has a set number of registers that are accessible from the control computer, each of which contains information about different aspects of the TRDBOX. Below is an overview of the different aspects to the TRDBOX. Section 10.3.4 contains more information about the various commands and registers that one can alter.

4.1.1 TRDBOX Operation

In the current setup, the TRDBOX takes in signals from both scintillation detectors at all times. These signals are passed through two discriminators within the TRDBOX. The discriminators act as a gateway, preventing signals which are too weak from being passed on to the TRD module. If the signal is strong enough, i.e. above the user-set threshold, then the signal is passed on to the two delay gate generators (DGGs). These allow for a delay between one scintillation detector's signal and the other, and they consider two particles to be coincident if they fall within a user-set time frame of one another.

The settings of the DGGs depends on the geometry of the scintillation detector setup, along with the features one is wanting to measure with the TRD. Applied to cosmic muon showers, the DGG settings should be set such that, for a specific angle, parallel muons are recorded as coincident regardless of the angle. For an angle which is normal to the scintillation detectors, the DGG settings should be such that both have the same delay. For angles which are not normal, the incident speed of the particles needs to be considered when calculating the delay of the DGGs.

4.1.2 The Physical TRDBOX

The TRDBOX contains nine different physical input pins, of which only three were connected for this setup. Each pin has an LED next to it that flashes if a signal is sent to it. The LEDs are extremely useful when it comes to troubleshooting or seeing how the TRDBOX responds to certain setting changes, as they provide instant feedback on certain settings. Figure 4.1 to the right shows the face of the TRDBOX, with all nine input pins shown.

4.1.3 Interacting with the TRDBOX

The most important TRDBOX settings have dedicated commands that one can use to change these settings. For example, the discriminator threshold voltages are paramount to deciding what is passed through the TRDBOX to the TRD, and there exists a dedicated command for changing these settings. Table 4.1 below gives a list of these commands, along with a brief description of their functionality.

In terms of actually changing the settings, a few preliminary actions need to be taken. The first is to log into the control computer remotely (using `ssh` - see section 10.1.3), and the second is to ensure that Setuptools is properly configured. This requires creating a `venv` directory within the main directory that contains `trdbox.py`. Generically, `venv` can be installed under `alicetrd-python` [12]. Once the `venv` is activated, then any TRDBOX command can be executed from anywhere else on the control computer. See section 10.1.4 for more information on Setuptools and Python virtual environments.

All settings are changed by doing `trdbox [COMMAND] [ARGUMENT(S)]`. Table 4.1 lists possible commands and possible arguments as of the writing of this report. For commands which accept arguments in the format `[ADDRESS]`, the values that it accepts are three digit hexadecimal numbers of the form `0xFFF`, where `FFF` identifies the specific address of the register-of-interest. Commands which accept arguments in the format `DATA` require eight digit hexadecimal numbers of the form `0xFFFFFFFF`. For instance, the command `trdbox set-pre-conf` takes an eight digit hexadecimal number where each set of two digits means something different. This command is extremely important as it controls what the TRDBOX needs to read before sending a trigger to the TRD module, as well as controlling the pretrigger (which is the signal that is actually sent to the TRD module). It also controls whether or not autoblock is on, which is where the TRDBOX will not send further trigger signals to the TRD module until is is unblocked via manually inputting `trdbox unblock` into the command line, or by another program. Using autoblock is very useful for taking data with automatic triggers so that the TRDBOX does not continue to send triggers to the TRD module while the TRD module is actively taking data from an event.

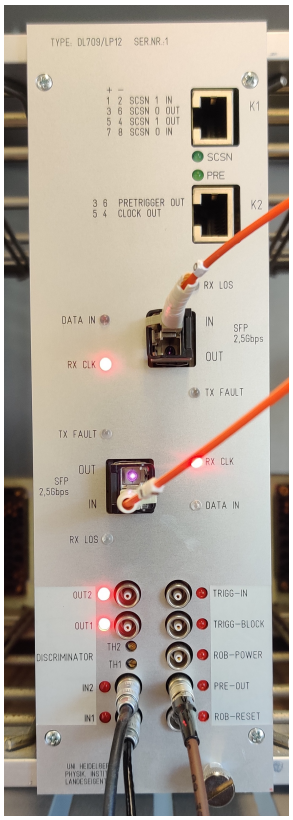


Figure 4.1: Image of the TRDBOX face, where the nine input pins are shown in the bottom of the image. OUT1 and OUT2 flash each time a signal that is above the set threshold passes into IN1 and IN2, respectively. The scintillation detectors are connected to IN1 and IN2. PRE-OUT is the pretrigger output, and it is connected to the TRD module as well as the oscilloscope. The orange cables are optical fibre cables which transmit data to and from the TRD module.

For the settings used here, the `pre-conf` register was set to `0x00000805`, while the `pre-dgg` register was set to be `0x10121012` for normal angles of incidence. The `dis-thr` register was initially set such both discriminators had the threshold of `2020`.

Command	Description of command
`trdbox`	Prints out a list of all commands, and should be used as the prefix for every following command.
`status`	Prints out the value in all registers, along with their addresses.
`unblock`	Unblocks the TRDBOX, allowing triggers to be sent through to the TRD module.
`pretrigger`	Used to manually send a software trigger to the TRD.
`dis-thr [NUM] [VALUE] [NUM] [VALUE]`	Sets the value of both discriminator thresholds. `NUM` corresponds to discriminator number, and is either 0 or 1. `VALUE` corresponds to the threshold and is on a scale from 0 to 4095, with 2048 corresponding to 0 V.
`set-pre-conf [DATA]`	Changes the external triggering, look-up-table (LUT) value, and other discriminator settings.
`set-dgg [WIDTH0] [TIME0] [WIDTH1] [TIME1]`	Changes the DGG width and delay settings. Both widths and times are in LHC clock cycles, i.e. each unit is equal to 8.33 ns. The maximum value is 99 cycles.
`reg-read [ADDRESS]`	Reads the value from a specified address where the `ADDRESS` can be found by running `trdbox status`. See section 10.3.4 for more information.
`reg-write [ADDRESS] [DATA]`	Writes `DATA` to the specified address where `DATA` is an eight digit hexadecimal number. See section 10.3.4 for more information.

Table 4.1: Table of all current commands that one can use to alter TRDBOX settings, along with brief descriptions of what each command does.

Register	Description	Input value
`0xFF000000`	External trigger width	Not relevant (set to 00)
`0x00FF0000`	External trigger delay	Not relevant (set to 00)
`0x0000FF00`	LUT value	See Table 4.3
`0x000000F0`	Monitoring output	Not relevant (set to 0)
`0x0000000F`	Enable autoblock	1
	Invert pretrigger	2
	Enable pretrigger	4
	Enable ROB power	8

Table 4.2: Table giving an overview of the various options for the `trdbox set-pre-conf` command. For the last digit, it can be set to any sum of the possible values. For example, setting it to 5 means autoblock will be enabled and the pretrigger will be enabled, since 4 and 1 sum to 5.

LUT value	128	64	32	16	8	4	2	1
HEX value	`80`	`40`	`20`	`10`	`08`	`04`	`02`	`01`
External signal	1	1	1	1	0	0	0	0
Disc. 0	1	1	0	0	1	1	0	0
Disc. 1	1	0	1	0	1	1	0	0

Table 4.3: All the possible LUT values for the `0x0000FF00` digits in the `pre-conf` register. For taking data with automatic triggering, the LUT value of 8 was used, with HEX value `08`.

4.2 Scintillation Detector Operation

In order to collect the data displayed on the oscilloscope, a modified version of the OpenWave-1KB [13] program was used. The version used can be found at [11], with a README outlining operation, as well as the quirks, but we will repeat the important parts here.

The oscilloscope can either be connected to the TRD computer directly using a USB cable, or over the network by ethernet. We strongly recommend using the USB connection as the interface written for ethernet had some very strange bugs that we weren't able to fix. The USB connection is not without its bugs but it works much better than ethernet. From this point on, instructions will be given assuming the USB connection is being used. This section will also reference the code in [11]. The most important part of that repository is the `oscilloscopeRead` package. It has been written to be installed as a package on the TRD computer, just like `numpy` or `matplotlib`, and contains all the vital programs. The rest of the programs in the repository were written in order to actually use the `oscilloscopeRead` package to take data from the scintillation detectors. They can be used as is, with perhaps some modification of the file path to save the data, or they could simply be used as an example for how the `oscilloscopeRead` package is best used.

There are two methods of interfacing with the oscilloscope remotely. The first uses the `dso1kb` module. We found this method to be the most useful when we wanted to change specific settings on the oscilloscope, or just get to grips with the commands that the oscilloscope takes. This was best done within a python interpreter session (preferably within a venv that has `oscilloscopeRead` installed, see section 10.1.4), and running the command `from oscilloscopeRead import dso1kb`. The name of the device in the system is then needed. This can be found by running `dmesg | grep -A 1 631D108G1 | tail -1 | grep -E -o "tty.{0,4}"` in the linux command line after plugging the oscilloscope into the TRD computer via USB. It should return something like `'ttyACM1\n'`. That `ttyACM1` is the device name. Note here that the `631D108G1` in the command above is the serial number for the oscilloscope. This shouldn't change, but if a new oscilloscope is used this will create problems.

Now a `Dso` object can be created by running `dso = dso1kb.Dso('/dev/{devicename}')` in the interpreter session. Note that the path to the device is needed, which should always be `/dev/`. After creating the object, commands can be sent to the oscilloscope using `dso.write(command)`. The commands that can be sent to the oscilloscope are all contained within the Programming Manual found in the Download section of `https://www.gwinstek.com/en-global/products/detail/GDS-1000B`. Some of the commands are used to change settings on the oscilloscope and as a result, do not return a value, however some of the commands, usually those that end in a "?", do return values. In order to access the returned value, `dso.read()` can be run in the python interpreter. This should return a byte string containing the returned value. Note that if the value is not read and another command is sent that also returns a value, it will be added to the buffer of returned values, separating each value with a newline character, and running `dso.read()` will return the value from the first command. Reading again will then return the value from the second command. The command `dso.clearBuf()` can be used to clear the buffer entirely.

A very important note about the commands being sent to the oscilloscope: When sending a command such as `:CHAN1:DISP ON`, which sets channel 1 to display, it is ***ESSENTIAL*** that a newline character is added to the end of the command. In the python interpreter session, that would look like `dso.write(':CHAN1:DISP ON\n')`. Without this, the command will not be executed.

The second method of interfacing with the oscilloscope, and the method we recommend when it comes to collecting data from it in conjunction with the data taken from the TRD module, is using the `scopeRead` module. This again requires the installation of the `oscilloscopeRead` package, but is best used in a python program. The `scopeRead` module contains a class called `Reader`. Creating a `Reader` object opens the connection to the oscilloscope, sets the oscilloscope to some settings that allow for good vertical resolution as well as setting some things back to defaults, and then allows a few methods to be called. To create a `Reader` object, simply include the line `scope = scopeRead.Reader('{devicename}')` after importing the module. Note that here only the name of the device is needed, not the file path. The device name can also be omitted entirely and the program will find the device name itself. We recommend using it this way, but the ability to input the device name is retained in case something breaks.

The methods that can be called are `scope.getData([1,2,3])` and `scope.takeScreenshot()`. The second of these simply takes a screenshot of the oscilloscope display and then saves it to the screenshots folder. The first method is the most important. It takes two arguments: A list of channel numbers to take data from, and an optional

boolean value that tells the program whether to create a plot of the output and save it to the screenshots folder. The first argument needs to be a list of integers and can only contain the numbers 1 to 4. Numbers can be repeated, but this will simply return the same data. The default is the first three channels. The second argument is designed to be used as a debugging tool, not for data collection, and is by default set to False in order to not plot the data.

One of the shortcomings of the current program should be highlighted. When the `scopeRead.Reader` object gets data from the oscilloscope, part of what it does is call the `dso.read()` method. This will read the buffer in the oscilloscope and return it as a byte string. The problem arises from the fact that the `dso.read()` method reads up to the next newline character in the buffer. Under usual operation, each newline character separates the returned value from a separate command (aside from the command `:ACQ{ch}:MEM?`, which returns a header and then the data on the specified channel, separated by a newline character), and everything inside the returned value for a given command should contain no newline characters. However, sometimes `:ACQ{ch}:MEM?` randomly includes newline characters in the data section of the output. We are not sure why this happens and thought the fix would simply be to continue reading the buffer until the end is reached, then appending all the outputs, but this didn't seem to work and we didn't have the time to get it working. The current fix just sets all the data in the affected channel to zero, prints a '−' character, and then continues, but this is an imperfect fix and can be improved. The problem and its current fix lie in the `dso1kb.py` file, lines 294-303.

One more note on the code written to interact with the oscilloscope: This code was used in two separate ways. First it was used to record data for both time resolution and optimal distance measurements, the details of which are in section 3.2 and section 3.1. This was done using the programs in [11]. The second implementation is discussed in section 5.3, as part of the `minidaq.py` program. The code used in the second implementation was modified slightly in order to suit the needs of that program, but only when it came to the imports of `oscilloscopeRead`. The function of the programs were still identical, but it must be noted of course that any modifications made to fix the problems outlined in this section must be applied to both instances, or simply the two implementations can be integrated into the same program.

4.3 TRD Control GUI

A Graphical User Interface for controlling the TRDBOX was designed. This was done to ensure more intuitive control over the TRDBOX, as well as to allow the user to set up and review a sequence of commands before executing them, which will reduce the chance of user error negatively affecting the TRD's operations. This ensures ease of operations, and allows new users of the TRD computer to have an easier time setting up everything.

The GUI was designed to run either locally on the user's machine or remotely on the TRD computer. These are two separate files, as the version that runs locally needs to connect via SSH to the machine, so it requires the user to input their SSH passcode. This version can't run on the TRD computer, as this computer has no knowledge of the SSH keys the user can use, and therefore needs the TRD computer's password. Therefore the version that runs on the TRD computer does not request to create a connection, which necessitates the creation of separate files but allows it to run seamlessly on the TRD computer.

The GUI consists of the following sets of components: a variable amount of pairs of action buttons, consisting of Function buttons to add commands and Configure buttons to configure the commands executed by the Function buttons; a pair of buttons to add or remove pairs of action buttons; an entry field and button to add custom functions to the command sequence; a text box displaying the current command sequence; an entry field and button to remove a command from the command sequence; and a button to execute the command sequence.

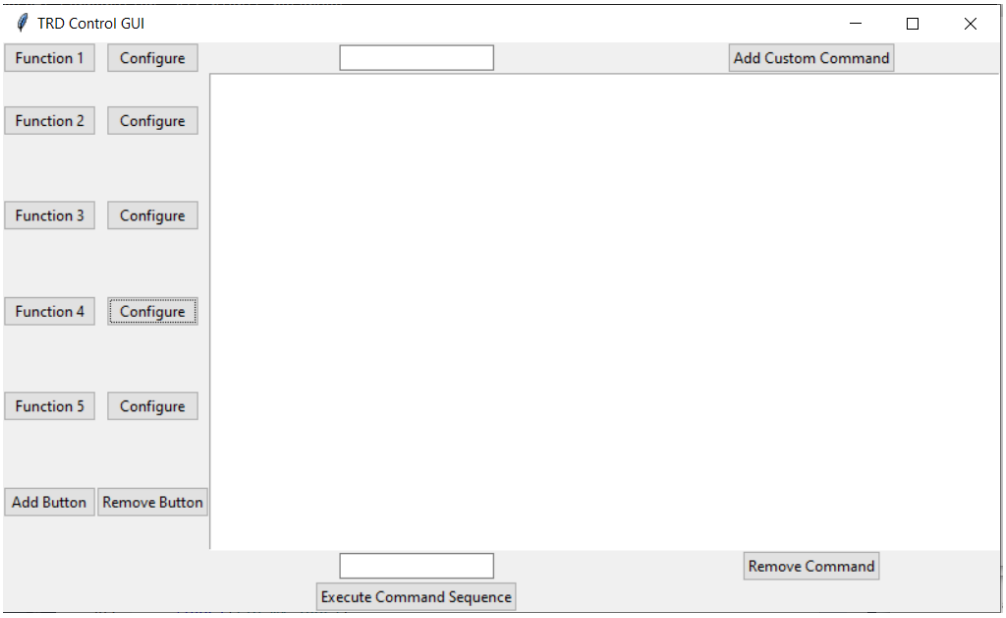


Figure 4.2: The GUI as it looks upon startup

Each pair of action buttons consists of one function button and one configure button. Each function button has a command associated with it. When a function button is pressed, it adds an instance of its associated command to the command sequence. Each associated command is initially set to be empty, as this allows the user to play around with the features without accidentally executing unwanted commands. This feature can be improved by allowing the binding of multiple commands to a single button. Each configure button opens another GUI, which allows the user to set the caption and command of its associated function button.

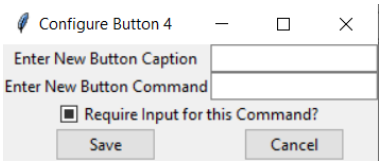


Figure 4.3: The screen for configuring a button

The configuration GUI also allows the user to set the action button to request input from the user every time it is clicked, which allows multiple commands with the same command but different arguments to be executed without having to configure the button multiple times. This is ideal for actions such as writing to registers, as it is unlikely that the user would write the same value to the same register more than once.

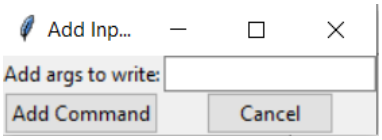


Figure 4.4: GUI Input for a Write Command

The pair of buttons that add or remove action buttons are positioned directly below the action buttons. The adding button adds a pair of action buttons to the bottom of the action button list. Like all other action buttons, the new action button starts out with a generic caption and no command set. This can again be configured using the newly created configure button, as described above.

The custom command section consists of one entry field and one button. The user can enter a custom command into the entry field, and press the button to append it to the command sequence. This allows the user to easily add once-off commands to the command sequence without having to go through the trouble of having to configure a button.

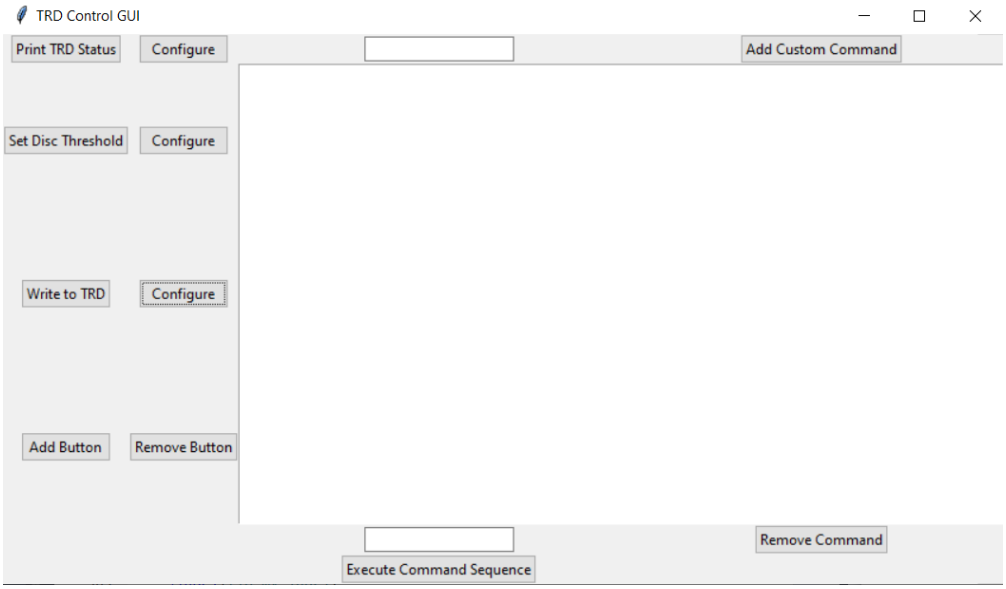


Figure 4.5: A GUI with a set of configured buttons

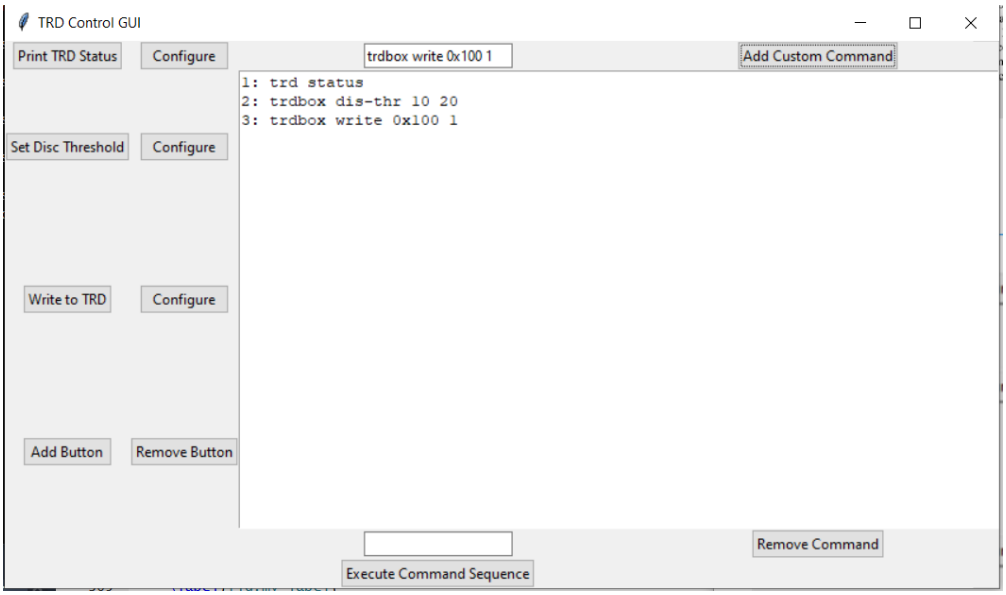


Figure 4.6: A GUI with Commands Added to it

4.4 Services: High and Low Voltage and Gas Supply

The anode and cathode wires within the readout chamber are connected to a high voltage power supply, NHQ 222M, which provides the necessary current and voltages to the wires to create the amplification and drift regions within the multi-wire proportional chamber. The cathode wires are kept at the same potential as the pad plane, that being ground. Meanwhile the anode voltage is set to a positive voltage reading by the high voltage power supply.

The gas used within the multi-wire proportional chamber is a combination of argon and carbon-dioxide at a ratio of 85:15. The carbon-dioxide was introduced as a quencher to ensure that the complete breakdown of the argon gas did not occur. Other organic gases could have been used but carbon-dioxide was chosen as it is nonflammable and did not contain hydrogen, which would create additional background readings from energetic knock-on protons.

When it comes to detection it is important that gas gain is set appropriately for legible results. Gas gain is the ratio of the total number of electrons detected on a wire or cathode pad to the number of primary ionisations produced by

some incident particle. To achieve the required gas gain of around 3500 the anode wires require a voltage of around 1.2 kV as gas gain increases with voltage. In this experiment the high voltage power supplies were set to provide 1350V and were set up with a significantly large voltage ramp down time in the case of overcurrent, overvoltage or sudden shutdown of electronics.

On the readout chamber, there is a plane of circuitry and electronics, made up of multichip modules, wires and the cathode pads. On top of this the lines for the digital readout, power and ground are laid out. These electronic components of the readout chamber are powered by three low voltage power supplies, EA-PS 3016-20 B Laboratory Power Supplies, which supply between 3.5 V and 7 V.

5 Data Acquisition

A new data acquisition system was written to record all relevant data resulting from events identified by the TRD module. These were used to study the typical response of the module, and identify cosmic rays. As of the time of writing, this implementation is available on Github [14].

Previous DAQ systems did not fully integrate waveform data from the scintillation detectors in conjunction with events from the TRD module. The inclusion of this data allows for measurements of the angular resolution of the detector, when tracing the paths of cosmic ray muons through the detector. This would be a powerful extension to the capability of the detector unit, and was made a primary goal for our project. See section 6.5 for a detailed discussion on the analysis of angular resolution on the TRD module.

In order to implement this, a new DAQ system was built with the ZeroMQ messaging library. The previous DAQ system was based on a full C++/C implementation, in the hopes of producing a system with optimal performance - however, this attempt was not successful. ZeroMQ allows for highly concurrent interaction with the TRD module through a Python wrapper, which should enable rapid prototyping of programs for data acquisition. At the start of the project, only a minimal component of the new DAQ had been implemented. We greatly expanded this implementation to meet our requirements for data collection.

For each event, ADC counts were recorded, as well as corresponding oscilloscope waveform data from the scintillation detectors. ADC counts are related to voltages induced by particles passing through the drift region of the TRD module. These are indexed by row and column (12×144), as well as 30 evenly spaced time bins. It follows that the ADC counts are useful for measuring the paths and energies of cosmic rays passing through the detector. This is discussed in much more depth in the introduction and analysis sections (sections 2.3 and 6.6). We are also interested in the waveforms recorded from the scintillation detectors when ADC counts are recorded from the TRDBOX. Events may take place in all the detectors simultaneously as a result of cosmic ray showers, which allows for the scintillators to function as a trigger for the TRD module.

5.1 Interception and Storage of TRD Events

At this point, it is necessary to precisely define an event in the TRD module. A subevent is composed of a fixed number of ADC count sets read out by the TRDBOX in a short time interval, as well as a number of encoded instructions that are parsed in order to organise the data. Subevents are output frequently, and need to be intercepted by the data acquisition system. The TRD module is composed of two sub-detectors, each of which produce subevents. A subevent recorded from both detectors simultaneously is combined to form an event.

The structure of the data acquisition system is laid out in Figure 5.1. Subevent readouts are facilitated by the TRDBOX daemon (trdboxd), drawn in red, and two subevent builders instantiated by the subevent builder daemon (subevd), drawn in orange. The TRDBOX daemon runs a `logicbox_proxy` method, that makes the TRDBOX accessible to ZeroMQ at a low level. The subevent builder threads read data blocks from the TRDBOX daemon for both sub-detectors, until they read an end-of-data marker. These data blocks correspond to subevents from the TRD.

The miniDAQ (`minidaq.py`) initiates readouts and collection of ADC counts and oscilloscope waveform data through the `readevent()` method. Before attempting data collection, a request is sent to the TRDBOX to dump the event buffers before taking a new reading. This is done to ensure there is a clean slate for new data to be stored in. A software trigger is sent to the TRDBOX to prompt a new readout, with the line `ctx.obj.trdbox.send_string(f"write 0x08 1")`. This is followed by requests to read data from each of the sub-detectors. At this point, both subevent builders attempt to read data blocks from the TRDBOX. If the subevent builders are successful, the raw data is stored

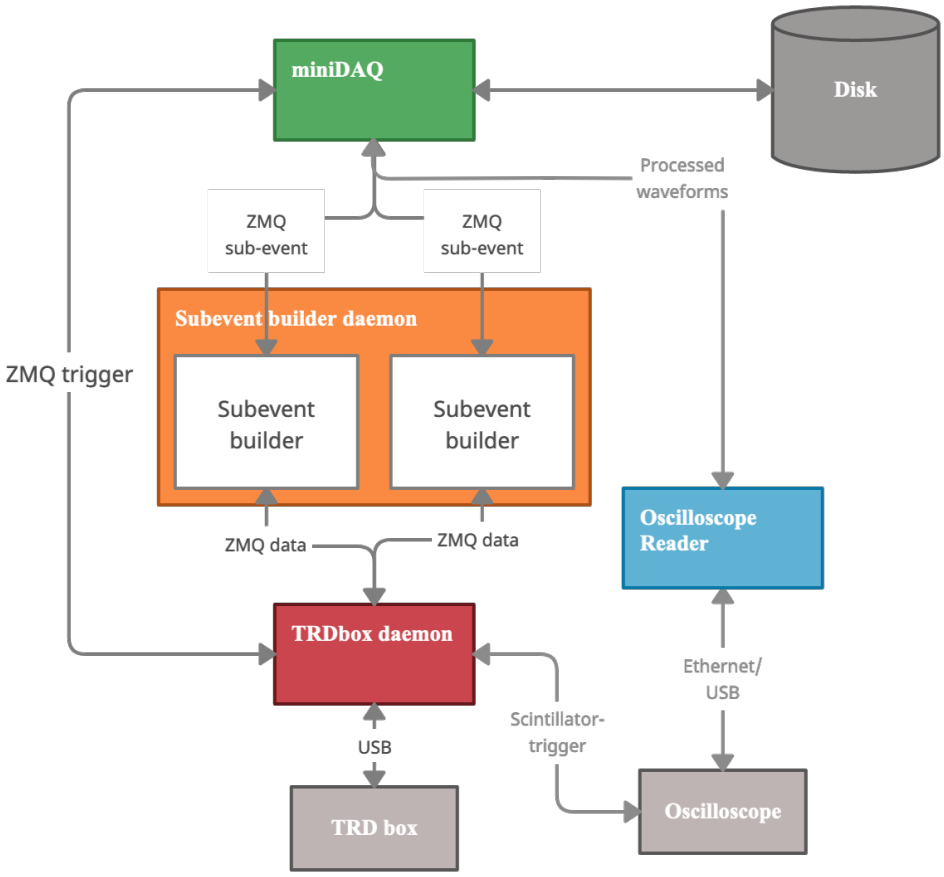


Figure 5.1: A diagram of the layout for the data acquisition system. The miniDAQ program initiates data collection runs, and stores the recorded ADC counts and oscilloscope waveforms to disk for each event. Data collection for an event may be initiated by a manual ZMQ trigger in the miniDAQ, or by a hardware trigger produced when the discriminator identifies a coincidence from the scintillation detectors.

in integer arrays in `subevent_t` objects, that also store information on the equipment type and ID used to take the measurements. The `event_t` objects are passed to an `eventToFile()` method to store on the disk.

The raw TRD data is stored in a `.o32` format, that can be parsed with `o32reader.py`. The file is given a main header with information on the format version of the file, the timestamp associated with the data collection for the whole event, and the number of data blocks stored in the file. In this case, one data block corresponds to one subevent. Each subevent's raw data (also called a payload) is converted from integer to hex and appended to the file, along with a subheader specifying the number of lines to be read. When parsing the data with `evdump.py`, a few key features are observed. The first couple of lines of data produce tracklets; these should not be found past the first 256 lines, and there are typically about a dozen of them. They are not useful in the analysis of the ADC counts. Tracklets should be followed by signpost lines in the `.o32` file (`0x10001000` in hex format). The rest of the output should consist of tables of ADC counts separated by headers. The ADC counts are indexed by TRD channel and time bin. TRD channels are used as a unique identifier for the row and column of the pad that recorded the ADC counts. The last two lines should have an end-of-data marker `0x0`. The headers make a reference to MCMs - these form a part of the front-end electronics for the TRD readout chamber, but are not necessary to consider for analysis.

5.1.1 Adapting Subevent Builders to Both Chambers

The original `minidaq.py` file could only read from one of the TRD chambers, cutting our available data by half. The DAQ system was extended to read data from both chambers, by modifying `minidaq.py` and the subevent

builders. The subevent builders work by utilising ZeroMQ's message parsing system. They set up a socket subscribing to either of the two TRD chambers and then infinitely wait for a request to read data. Once it receives this request it will read data from the TRDBOX daemon's buffer corresponding to each chamber. In order to read from both chambers simultaneously, we created a new subevent program called `subevd` which takes two arguments, `-sfp0-enable=<bool>` and `-sfp1-enable=<bool>`. When both arguments are set to true, `subevd` will create two instances of `trd_subevent_builder`, parsing the same ZeroMQ contexts but different sockets. The first chamber resides at the address `tcp://localhost:7750` and the other at `tcp://localhost:7751`. These instances are created in two separate threads, making sure that the subevent builders are run concurrently and are only stopped on a termination command (i.e. Ctrl+C on the command line.) The `trd_subevent_builder` objects have an infinite loop in their main method which waits for a user trigger to then read the buffer from the TRDBOX daemon at the ZeroMQ address of `tcp://localhost:7766`.

5.2 Background and Cosmic Ray Measurements

Two different types of data runs were needed for data analysis - background noise measurements, and cosmic ray detection measurements. Background noise measurements aim to study the typical response of the TRD to its environment. A method suggested in [7] is to record a large number of events with manual triggering, and aggregate the ADC count data to determine the mean response of the detector. These runs were implemented in the miniDAQ through the `background_read()` method. The final implementation looped for a fixed number of iterations - at each step, the method slept for two seconds before calling the `readevent()` method to send a manual trigger to the TRDBOX and collect data for an event. All events for a run were stored in a single folder, with name including the starting timestamp of the run. After calling readevent, the hardware trigger is unblocked (see section 4.1.3). Background measurements are recommended to be taken with non-zero-suppression, which can be set in the TRDBOX with the `nginject all 100` command. Unfortunately, this could not be carried out - see section 5.4.

Cosmic ray detection measurements utilise the scintillation detectors to detect events and send a trigger for recording data from the TRD. As discussed in section 2.3, this is achieved by feeding the waveforms recorded by the oscilloscope through a discriminator. If pulses recorded from both scintillation detectors are sufficiently high to trigger the discriminator, and the triggers overlap in time, a signal is sent to the TRDBOX to increment its `0x102` register. This register can be monitored to trigger data collection for an event when its value increments. The purpose of this hardware-based trigger is to use the scintillation detectors as a filter for detecting possible high-energy cosmic ray events through the TRD. This is discussed in much more depth in sections 4.1 and 4.2.

The data runs for cosmic ray measurements were carried out in the `trigger_event()` method of the miniDAQ. The `trigger_event()` method checked for a fixed number of changes to the `0x102` register, in a continuous while loop. Whenever a change was found between subsequent loops, a counter was incremented and the `readevent()` method was called. Upon completion of the `readevent()` method, the hardware trigger for the TRDBOX is unblocked. The module was set to take zero-suppressed readings with the `nginject all 101` command.

For more details on how to set up the TRDBOX and take runs, see section 10.3 in the technical guide.

5.3 Oscilloscope Waveforms

The `readevent` method was extended to also record and store oscilloscope waveforms from the scintillation detectors, over the same time frame that each event was recorded. The implementation used the `oscilloscopeRead` package (see section 4.2 for an in-depth discussion on its usage). For both of the data run types, an `oscilloscopeRead.scopeRead.Reader` object was instantiated outside the main loop, and passed to the `readevent()` method. After collecting ADC count data from the subevent builders, the `Reader.getData()` method was run to collect waveform data from the oscilloscope. The output was a two-dimensional numpy array containing three integer arrays, each corresponding to a different channel of the oscilloscope. Two of the channels are waveforms from the two scintillation detectors, and the third is the trigger signal sent from the discriminator to the TRD when a coincidence is detected with the scintillation detectors. The data was output to a standard `.csv` file in the same folder as the ADC count data. An attempt was made to extend the `.o32` file format, to include the oscilloscope waveform data in the same file as the ADC counts. The integer arrays were appended to the `.o32` files one after the other, with a short header containing information on the timestamp and array length. The `evdump.py` program was extended to handle this new data. Currently, this is just a csv and not an actual `.o32` file. A future goal would be to complete this implementation to

produce a compact `.o32` file format, that can be parsed to extract ADC counts and waveform data in one go, preferably by condensing the oscilloscope readings from the three channels into a single hexadecimal string.

5.4 Technical Issues

Data collection was a seemingly intractable problem for the full duration of the project. The TRD module was not connected to a generator, and loadshedding would frequently paralyse the equipment for days on end, unless there was experienced intervention from our laboratory supervisor. An attempt was made to circumvent loadshedding issues, by powering all components of the TRD with a backup UPS power supply for the last week of the project. Switching over to the backup power supply resulted in a two-day hiatus where the TRD module was inactive (as measured through the `smmon` command) - this was attributed to a Linux computer on the DCS board not successfully turning on after making the switch. Any attempts to change the configuration of the TRD module would result in a similar inability to take measurements.

We were unable to identify any cosmic ray events when taking data runs for cosmic ray measurements. Cosmic rays are expected to generate an extremely large number of ADC counts (on the order of 10^3) - this was never seen in the events read out from the TRD module. Excluding noise from identifiably faulty pads in the readout chamber, that produced approximately 400 to 500 counts, the ADC counts were almost universally on the order of 10^1. At the time of writing, this failure to capture cosmic rays has been attributed to timing issues with the miniDAQ, where readouts are taken after or before the cosmic ray passed through the TRD. This can happen, for instance, if too much time transpires between the interception of a coincidence signal from the discriminator and the execution of the `readevent` method. Fixing this could entail improving the speed of the miniDAQ, or adjusting gate delay timing settings in the electronics. Timing issues were also identified in the synchronisation of the oscilloscope data, also likely related to the execution of the program. A possible solution was proposed to run the oscilloscope and ADC count collection with different threads running in parallel - however, there was insufficient time to implement and test this.

The format of ADC count data recorded in the readout was frequently faulty. In some cases this was easily diagnosed. For instance, readings can be taken when the TRD box has not been configured - this may lead to the data consisting purely of tracklets or skipped parses when parsing the data with `evdump.py`. An issue that has yet to be explained at the time of writing, that arose late in the project, were readouts where a few sets of ADC counts would be followed by an endless string of data that was skipped by the parser. This was not an issue when we first started taking measurements.

Issues were also encountered when trying to read non-zero-suppressed data from the TRD, which is recommended when taking background noise measurements. Data read out with this setting (set with `nginject all 100`) could not be parsed by `evdump`, and a cause has yet to be determined.

As an overall consequence, we were only able to take background noise measurements for the TRD, which presented a major setback for analysis. Therefore, studies of cosmic rays were conducted with data produced from a previous laboratory.

It is worth noting some small things to keep in mind when dealing with the module. The high voltage power supply should be turned off before loadshedding, or any situation where power plugs are switched off or moved. One of the programs required to run nginjects on the TRD box, `dimcoco`, interacts with an external database that is not connected to a backup power supply, and needs to be restarted after loadshedding (see section 10.3.3 in the technical guide).

5.4.1 Stalling

A persistent stalling issue was encountered when implementing data runs in the miniDAQ, for both background and trigger measurements. At the time of writing, this was attributed to a race condition issue with the two subevent builder threads instantiated by `subevd`. When the two threads attempt to read data from the TRDBOX daemon simultaneously, they likely conflict with each other and leave the trdbox with half-written or unreceived data. This is a result of `ZeroMQ` using a sort of stop-and-wait protocol for its request/response framework.

Attempted solutions:

1. First, it was attempted to have a loop that would reattempt a request of data until it received. This didn't work because the trdbox was still in a stalled state.

2. A timeout was then implemented that killed the threads when they took too long (i.e. were most likely in a stalling state), and then reattempted. This did not work as the trdbox still maintained something that was blocking

it.

3. It was unclear how the stalling was actually unblocked, as it seemed to unblock between consecutive runs but no attempt to unblock it during a single run was successful.

4. It was also attempted to communicate with each subevent builder thread in a different process, but this gave no improvement.

5. A wait was implemented between the 'send' and 'receive' methods, and this had some small positive effect in reducing the race conditions, allowing a meaningful quantity of data to be taken.

5.4.2 Oscilloscope readings

The collection of oscilloscope waveform data was a noticeable bottleneck for the speed of the `readevent()` method, adding on the order of 100ms for the completion of one readout. A likely consequence of this was that there would be issues with the timing of data collection for an event in the TRD. The bottleneck was attributed to the fact that the `getData()` method post-processes the raw data from the oscilloscope. A possible solution to this would be to store the original raw data to file, and process it later for analysis. It seems to be a viable option to use the raw data from the oscilloscope, which is output in 2-bit hex format. As there are four channels, this could be easily converted to a single 8-bit hex number and then returned. The issue lies in the conversion from hex to voltage, as that requires information from the header of the data, output by the oscilloscope. This can be extracted, and already is extracted for use in the `convertWaveform()` method of the `dso1kb` program, and then stored in a header for later use.

5.5 Additional Functionality in the miniDAQ

Methods were written to instantiate the TRDBOX and subevent builder daemons through the miniDAQ using a `minidaq setup` command. These could be launched with console commands `/usr/local/sbin/trdboxd` and `/usr/local/sbin/subevd` at the time of writing. Both processes could be terminated with a `minidaq terminate` command.

6 Results and Analysis

6.1 Data Pre-processing

The data that comes directly from the detector is in a binary (`.o32`) format and there are regions of the detector which do not output correct data. In order to correctly format the data and exclude the broken regions of the detector pre-processing of the data is needed. Similarly to [15], the scripts `o32reader.py`, and `adcarray.py` were both used and were written by Jan-Albert Viljoen for his Honours project in 2016[16]. The details of how these scripts work are not important for this report, and their functionality is implemented by a script `raw2npy.py`[17], which takes in an `.o32` file as an input and returns a `numpy` array file (`.npy`) which can be used for later data analysis.

The `raw2npy.py` script was adapted from [15] and it returns a four dimensional `numpy` array which has dimensions of "event number", "pad row", "pad number", and "time bin". The value in each element of the array is the ADC count recorded by the TRD. The broken region of the detector is in a different place to where it was located in 2020, and is now located in the rectangular region from columns 72 to 144 and rows 4 to 8.

6.2 Background

As mentioned in section 5.2 the background readings were obtained by sending a manual trigger to the TRDbox for each event. Such events can be used to measure the baseline value and influence of the noise on the pads. These influences may then be used to determine whether the pads are working expected. 1000 runs for Background ADC values were obtained with the high voltage turned off. The raw data was stored in a hexadecimal format and was extracted using the python program `raw2npy.py`. This can be accessed at `https://bitbucket.org/uctalice/pytrd/`. Using the background readings and measuring the noise in units of counts per time bin, the mean can be determined and a heat map showing how the noise is distributed throughout the detector can thus be created. The figure below shows an output of the obtained background data, before zero suppression. From this plot it is clear that the region from

rows 4 to 8 and columns 72 to 144 is the broken region of the detector. This is a useful check to do on any data that comes in, to identify the region that needs to be zero suppressed.

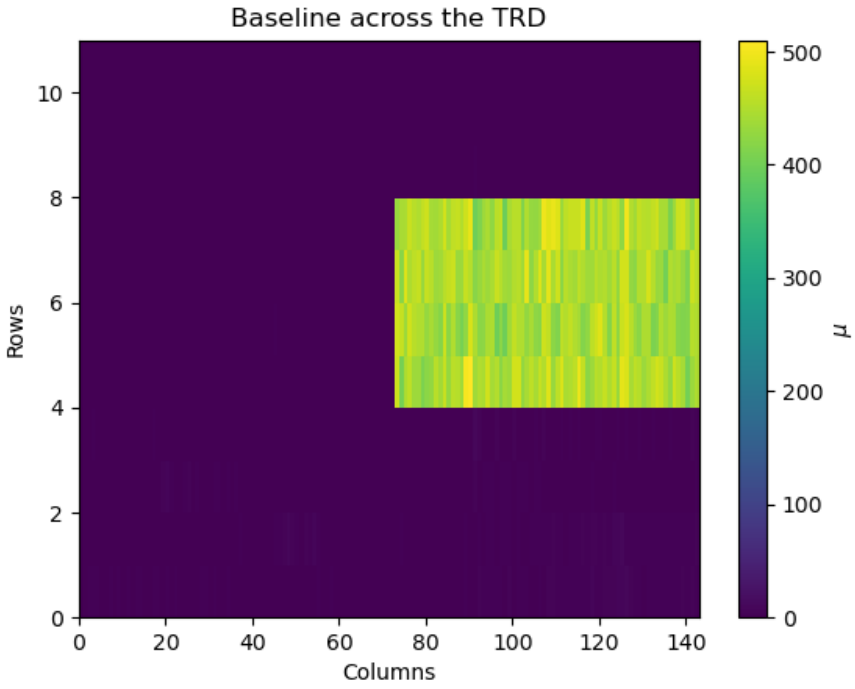


Figure 6.1: Heat map showing the broken region of the detector.

The variable `DATA_EXCLUDE_MASK` (see Table 10.1 for more details) can now be edited to contain this region. For the specific region shown in Figure 6.1, the variable would need to be changed to `DATA_EXCLUDE_MASK [:,4:8, 72:,:] = True`. Doing this allows us to generate a background noise plot (Figure 6.2), which we expect to be fairly evenly distributed across pad columns and rows. From this the background ADC count for our detector is found to be 9.3, and this can be subtracted off in later analysis.

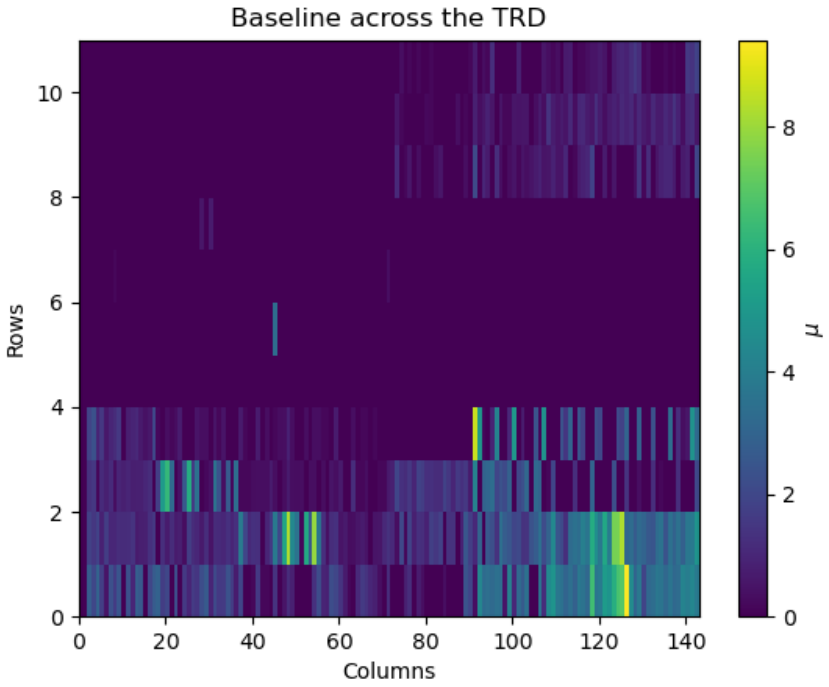


Figure 6.2: Heat map of the ADC value on each pad, averaged over time and events. This is used to ensure that there are no abnormalities in the data and to find the average background ADC count.

6.3 Time Resolution

Naturally, the scintillation detectors do not produce a pulse at the instant that a particle is detected. The avalanche and detection process described in section 2.3 takes time as it must travel through the length of the scintillation detector. The travel time is not the only thing that contributes to it however, the electronics and construction of the scintillation detector itself also contribute. The time resolution is an essential measurement due to its importance in determining times of muon arrivals at the detectors, which extends to being able to pinpoint the origin of the muon shower that the muon originated from. If we did not take the time resolution into account, any measurement pertaining to the timing of the muons would be inaccurate and hence not useful in determining a final result. The experimental set-up used for these measurements is described in section 3.2.

6.3.1 Time Difference Between Two Oscilloscope events

To analyse the time between the two oscilloscope events we used the analysis described in [15] as a framework. While we agreed that finding the time difference between the minimum of the two pulses lead to inaccurate values due to the peaks being skewed, the improvements that were proposed seemed arbitrary; particularly the value of the threshold that was chosen. Therefore the aim of this section was to find a method that potentially improved the previous analysis with an ultimate goal of obtaining a method that future years could use.

The method that was implemented involved creating a linear fit from the point just before the pulse started to the minimum point of that particular pulse. A visualisation of this can be seen in Figure 6.3. A fit is necessary because the data extracted from the oscilloscope came in intervals of 4 ns. While a very small time interval, for measuring quantities such as the time resolution, which has a very rough estimate of 5 ns, we require smaller time intervals to be accurate,

While the pulse should have a curved slope towards the minimum, this was not an option due to the fluctuations that were observed when no pulse was present. The fluctuations from the oscilloscope data were of the order 1×10^{-2} V at a maximum and 1×10^{-3} V at a minimum. In addition, both channels had different observed fluctuations. While this could have just been a result of the equipment used we decided to keep it in the data and not assume such. As mentioned before, these fluctuations made creating a curve very difficult as they would result in the fit being very inaccurate for

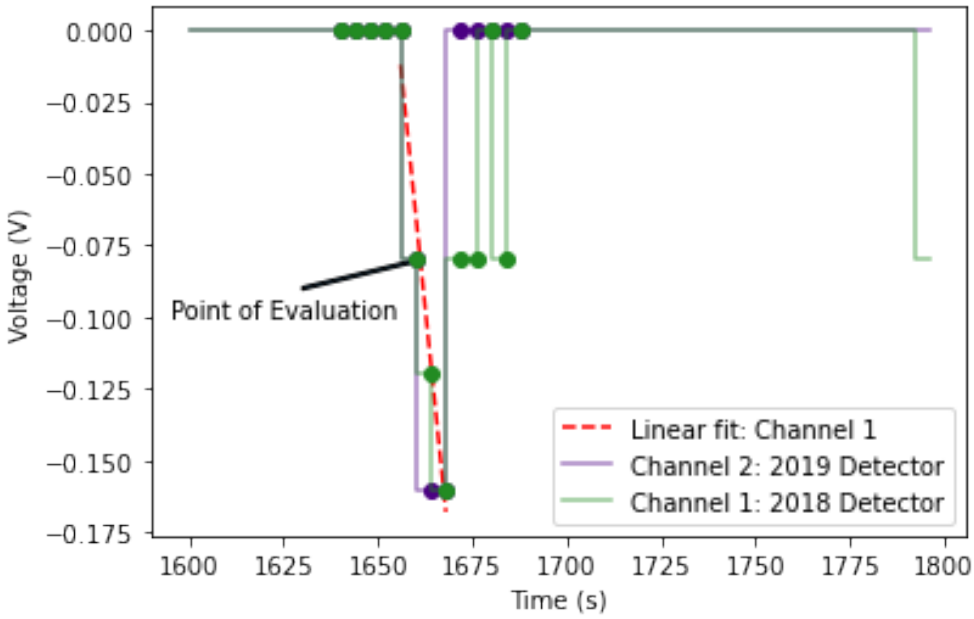


Figure 6.3: Pulse isolated from the 1000th data set obtained for the configuration with the 2018 scintillation detector on top. Both channels are plotted visually showcasing the small time difference between the two pulses. The linear fit of channel 1 is plotted alongside the peak.The point at which this line is evaluated is indicated. Only the linear fit of channel 1 is done to increase clarity.

certain data sets. The linear approximation is also not terrible as we are comparing time values. Therefore as long as the method remains constant for both channels, the inaccuracy due to the linear fit should be nullified.

The value at which we used the linear fit to extract the data was directly obtained from the fluctuations that were observed. At the point where the line exceeded twice the fluctuation is the point that we can confidently say the signal starts. However, as the fluctuations might be different for each channel we take the largest of the fluctuations that were observed. It should be noted that this was a rare occurrence. We evaluate the linear fit at the new "start of signal" and compare the time values.

The code use for this analysis is contained in `Time_Difference.py` and can be found in the git repository [17].

6.3.2 Comparison of Methods and Calculation of Time Resolution

We decided to perform a comparison between all the methods, used to calculate the time difference, to see whether the proposed improved method performed better than the two methods already in place. From the results of section 6.3.1 an appropriate distribution was chosen to model the data. We model the results as simply calculating the sample mean and standard deviation renders the values obtained very dependent on that particular run of data acquisition. If we model the results as a distribution the values obtained won't be as specific to the run providing a more accurate representation of what the actual values of the mean and standard deviation should be. It is a more rigorous and general way of obtaining values from the results. In addition, modelling the results will exclude any potential outliers in a non-arbitrary way.

We chose to model the results as a gamma distribution, whose probability density function is given by:

$$f(x; A, \beta, \alpha) = A\frac{\beta^{\alpha}}{\Gamma(\alpha)}x^{\alpha-1}e^{-\beta x} \tag{6.1}$$

A gamma distribution was chosen as it matched the shape of the results that we wanted to measure as depicted in Figure 6.4. It should be noted the method of using a threshold value to obtain time differences was not compared as it did not output any meaningful results (all time difference values were zero). From this point we shall refer to the method comparing the minimums of the pulses as 'Method 1' and the method comparing the start of the signals as 'Method 3'.

The two methods compared above were not analysed with the same number of bins. The reason for this is that method 1, when the number of bins increased past 110, did not create statistically analysable data (data where each

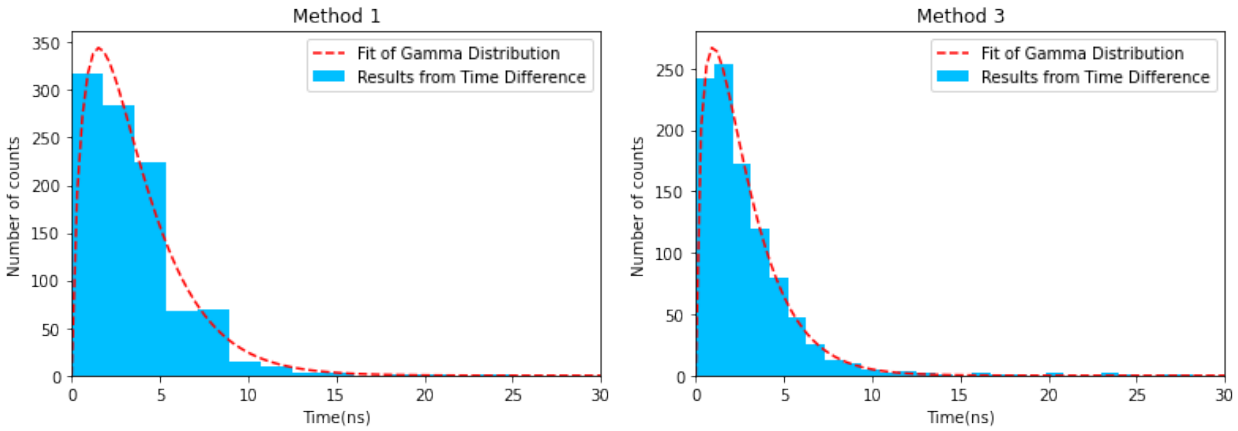


Figure 6.4: Histogram of results obtained from comparing the minimums of the pulses. One thousand readings were used with the 2018 scintillation detector on top of the 2019 detector as described in section 3.2. The results were split into 110 bins for method 1 (left) and 1000 bins for method 3 (right). A gamma distribution that has been fit to the data is also imposed on the plot in red

bin has a sufficient number of points). However when done using a different run of the results from section 6.3.1 one could increase the number of bins past 200 with the data still being analysable. Method 3 on the other hand was split into 1000 bins for every run of and produced analysable results every time. Therefore, we can make the statement that method 1 is dependent on the particular data set given and might not provide a great estimate for what the actual values of the time difference.

As the number of bins between the two method was not the same, we cannot compare the Chi-square per degrees of freedom, or any other equivalent value, or the standard uncertainty of the fitting parameters. We can however compare the standard deviation of the mean, as a result of the fitting parameters, and see which is smaller. The variance of a gamma distribution is given by:

$$\sigma^2 = \frac{\alpha}{\beta^2} \tag{6.2}$$

Which yields the following values for method 1 and 3 respectively.

$$\sigma_1^2 = 7.670 \pm 0.020, \quad \sigma_3^2 = 4.557\,526 \pm 0.000\,080 \tag{6.3}$$

We notice that the variance for method 1 is significantly larger than the variance for method 3. While this cannot be used in isolation to decide that method 3 is superior to method 1, coupled with the fact that method 1 is more reliant on the data set used we can conclude that method 3 is superior for trying to determine the time difference between two oscilloscope pulses.

The time resolution of the two scintillation detectors can now be found using method 3 and the model of the data. For a particular scintillation detector the time resolution will be the mean produced by the gamma distribution when that detector is underneath the other one. For a gamma distribution the mean is given by:

$$\mu = \frac{\alpha}{\beta} \tag{6.4}$$

We do not assume that the two detectors have the same time resolution as they are not constructed exactly the same however we do expect the values to be similar. The uncertainties for these values are the standard deviation obtained from the model added in quadrature to a term that takes the uncertainty of the fitting parameters into account. It should be noted that the uncertainty due to the fitting parameters can be ignored for simplicity as it is significantly smaller than the standard deviation. The time resolution for the scintillation detectors are:

2018 Detector: (2.8 ± 2.4) ns
2019 Detector: (2.7 ± 2.1) ns

Details of this analysis can be found in `Method_Comp_And_Time_Reso.py` in the git repository [17].

6.4 Optimising the Distance Between the Scintillation Detectors

In order to determine the optimal separation distance for the scintillation detectors, the angular resolution of the two-detector system, as well as the event rate, at a given separation, needed to be investigated. There would need to be a trade-off between the angular resolution and the event rate, when optimising the setup.

6.4.1 Angular Resolution of the Two-Scintillation Detector System

The angular resolution of the system of the two detectors in detecting muon showers was investigated without the use of data from the scintillation detectors, and instead from knowledge of the details of the setup and making predictions on angular resolution given these details.

Two muons arriving at each scintillation detector, in a single event, can be assumed to have parallel trajectories towards and through the detectors (since they come from very far distances away). They are therefore incident at the same angle with respect to the straight line through the two detectors (placed at the same height above the ground). This is illustrated in Figure 6.5.

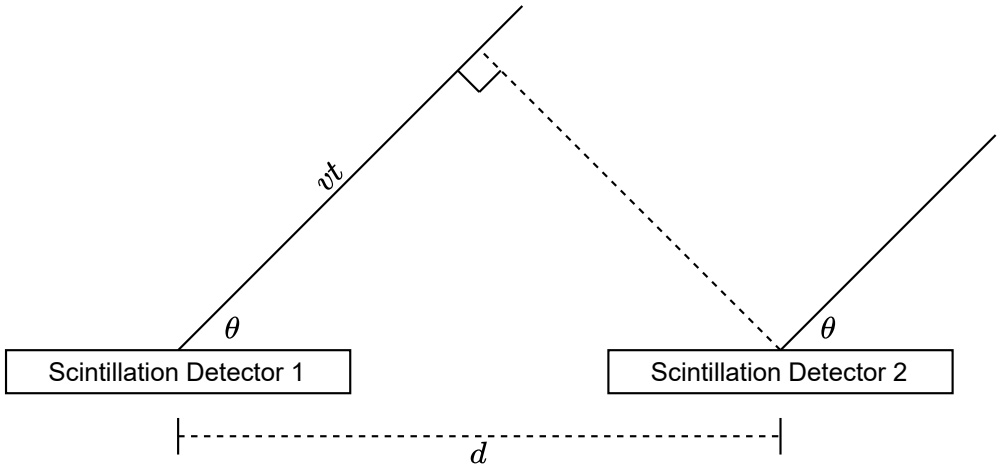


Figure 6.5: The geometry of the two-scintillation detector system (for a given separation distance, d), used to measure θ, the angle of incidence, in the plane depicted, of a muon shower. In the case where the angle was between 0 and 90 degrees, a muon was expected to arrive at the right-hand detector (2) at an earlier time than the left-hand detector (1), whereas when the angle was between 90 and 180 degrees, a muon was expected to arrive at detector 1 at an earlier time than detector 2.

It was further assumed that the two muons travel at exactly the same speed, taken to be $v = 0.994c$ [18], which depends on the angle of incidence. Hence, for a given angle θ, the time difference, t between the scintillation detectors for a single muon shower event can be predicted, assuming that two muons from a shower arriving at the detectors do not have an inherent time difference (i.e. a time difference arising from causes other than them being incident at different angles). This is achieved by simple trigonometry according to Figure 6.5. Since,

$$\cos\theta = \frac{vt}{d} \tag{6.5}$$

it follows that

$$t = \frac{d\cos\theta}{v} \tag{6.6}$$

The uncertainty on the value of $\cos(\theta)$ would then be determined via propagation according to:

$$u(\cos\theta) = \cos(\theta) \times \sqrt{\left[\frac{v \times (u(t))}{vt}\right]^2 + \left[\frac{u(d)}{d}\right]^2} \tag{6.7}$$

Equivalently, by equation (6.3)

$$u(\cos\theta) = \cos(\theta) \times \sqrt{\left[\frac{v \times (u(t))}{d\cos\theta}\right]^2 + \left[\frac{u(d)}{d}\right]^2} \tag{6.8}$$

The incidence angle, θ found from the cosine, would then have an associated uncertainty found by

$$u(\theta) = \frac{u(\cos\theta)}{\sin\theta} \tag{6.9}$$

Therefore, the angular uncertainty, which would indicate the angular resolution, can be found as a function of angle, from equations 6.4 and 6.5, and knowing the parameter values and the necessary uncertainties.

If coincident muon events from the same shower were to be detected by the scintillation detectors, the uncertainty on the value of the time difference between the muons (which could be found, for instance, in the manner similar to that in section 6.3.1) would have a value found via propagation, according to

$$u(t) = u(t_1 - t_2) = \sqrt{[u(t_1)]^2 + [u(t_2)]^2} \tag{6.10}$$

where the values of $u(t_1)$ and $u(t_2)$ are equal to the time resolution of the scintillation detector 1 and 2, respectively, as determined in section 6.3.

The determination of $u(d)$ is slightly more complicated. When a coincident event is detected by the two detectors, the position on the detector at which the muon is incident is not precisely known, and so, while one could say that the distance between the points at which two muons in a shower are incident on each detector is the center-to-center distance of the detectors (i.e. the separation distance, d as shown above), this would carry a large uncertainty, primarily owing to the fact that the position of an incident muon on each detector follows a normal distribution.

6.4.2 Monte Carlo Simulation to Determine Distance Uncertainty

To calculate the uncertainty of the distance between two muons incident on the scintillation detectors, a Monte Carlo simulation was made, `montecarlo_distance_uncertainty.py` which can be found in [14]. In the simulation it was assumed that the incident muons were uniformly distributed across the surface area of the detectors while they were separated by a known distance D. Then two sets of random 2D points (x and y) were generated, one set corresponding to the coordinate of incidence on each respective detector. This generation process was repeated 100 000 times. In a loop the distance between each of the two sets of points was then calculated and put in an array of data, giving us a large data set of distances between two incident muons on the detectors. Then, by taking the mean and variance of the data, we could get an average of the distance between some two muons as well as the overall uncertainty of that result. This simulation for acquiring the uncertainty and total average distance was then repeated four times, with each new run using a new distance, D, of separation between the detectors, thus giving us four sets of results.

D	d	σ	σ^2
2.082	2.121356023135194	0.20489820266625581	0.04198327345586204
2.865	2.8937191662266284	0.20420478171570725	0.04169959287555964
3.625	3.648298351387677	0.20463296712826512	0.04187465123571763
7.500	7.511187886758412	0.20353034133137357	0.04142459984246543

Table 6.1: Table giving the results of the described Monte Carlo simulation where D is the set separation distance between the two scintillation detectors, d is the average of the distance data between two incident muons and σ and σ^2 are the standard deviation and variance respectively

Figure 6.6: A histogram of 100 bins showing the separation distances sets of two muons, each being incident on one of the scintillating detectors, with the scintillators being separated by a distance of 2.865m

6.4.3 Scintillation Detector Angular Resolution for different separation distances

The uncertainty as a function of incident angle θ was plotted, for 4 different separation distances using `angular_res_plots.py`, found in [17]. The results are shown below.

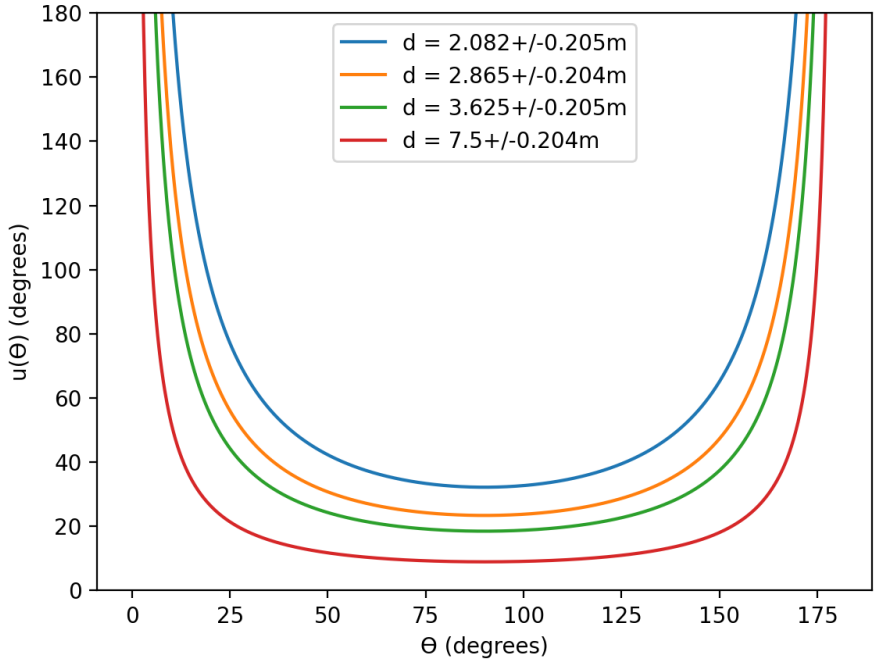


Figure 6.7: Graph of the uncertainty expected for different possible angle measurements by the scintillation detectors, when separated by certain separation distances. The uncertainty tends to infinity at angles of 0° and 180°

6.4.4 Determining Optimal Separation Distance

While it is clear that, as expected, for larger separation distances, globally the angular resolution is improved (the uncertainty values are lower for all possible θ values), it was not practical, due to the time constraints of this experiment, to use very large separation distances, since this would lead to a reduced event rate (at further separation distances fewer coincident events are expected), which was not ideal for data acquisition. A trade-off between the event rate and the angular resolution had to be made, which constituted an optimisation of the two-detector system.

In order to find the event rate at 4 different separation distances, the scintillation detectors were set up in the manner described in section 3.1. A series of 4 runs (1 for each distance) of coincidence-triggered events (events for which each detector detected what was assumed to be a muon from the same shower, within a certain time interval of each other) was taken, using the `optimalDistance.py` in [11], detailed in section 3.1. The event rates were found by dividing the number of events detected in each run, which could be given as an argument when running `optimalDistance.py`, by the total acquisition time, which was found simply by using the `time` python package. The results for different separation distances are shown in the table below.

Separation Distance (m)	Event Rate (s^{-1})	Number of Trials
2.082	0.140	100
2.865	0.089	100
3.625	0.067	100
7.500	0.031	50

Table 6.2: Event Rate and Angular Resolution for different Separation Distances

The largest separation distance with a high enough event rate, was 2.865 m. Since a reasonable angular resolution was expected at this distance (the uncertainty as a function of angle for this distance (the orange line in Figure 6.7 is near an intermediate value compared to all the other distances shown), this was determined to be the optimal separation distance between the two scintillation detectors.

6.5 Angular Resolution of the TRD

This section of the analysis could not be implemented as planned, since no usable scintillator-triggered TRD data was acquired: the data from the scintillator channels of the oscilloscope (channel 1 and 2) for the scintillator-triggered TRD setup detailed in section 3.3 contained no non-noise single (i.e. no pulse associated with an event was present in any of the data). That this data acquisition did not take place as planned was likely the result of timing issues as detailed in section 3.3. This data was necessary to conduct analysis to determine the angular resolution of the TRD. However, we outline here the analysis process that would be followed, in the case where the data had been obtained correctly, in order to determine the resolution of the TRD. The TRD and scintillation detectors would have been set up in the manner described in section 3.3 so that both would measure the same incidence angle of a muon shower event. A run of a number of events would have been taken using the process described in section 5.2. Only certain of these events would be muon shower events, with TRD data which could be analysed in the manner described in section 6.6.1 (i.e. using `path_reconstruct.py`) to find an incidence angle measurements, according to the TRD.

For these events, the same muon shower incidence angle would be found using the oscilloscope data (specifically the pulse data for channel 1 and 2, connected to scintillation detector 1 and 2, respectively) from the corresponding scintillation detectors events. (The oscilloscope data would have been read into a three-dimensional python array using `TRD_angular_res.py`, found in [17]). To find such events, out of all those from the single run, the index of the event could be used (since the events would follow the same order in the TRD and scintillation detector data). `TRD_angular_res.py` would then be used to find the angle for each event. To do so, it would first be necessary to obtain the difference between the times at which each scintillation detector had detected a muon from the shower event. This would be done using the method outlined in the section 6.3.1 (i.e. `Time_Difference.py` found in [17] was adapted and integrated into `TRD_angular_res.py`). `TRD_angular_res.py` would then find the incidence angle, according to the scintillation detectors, using Equation 6.5 (taking the speed of muons to be $v = 0.994c$ and the distance between the positions of incidence of the muons on each detector to be the center-to-center distance of the scintillation detectors, 2.865 m).

The difference in the angle, for each of the events, as measured by the TRD and by the scintillation detectors, could then be found. The difference in angles determined by the two independent methods would be assumed to follow a normal distribution, and so the standard deviation resulting from this distribution of angular differences could be taken as the uncertainty on the difference. Using this uncertainty and the uncertainty on the incidence angle (previously determined in as a function of the angle in section 6.4.3) for the scintillation detector system at the optimal separation distance a value for the uncertainty on angles measured by the TRD, i.e. its angular resolution, could be found via uncertainty propagation:

$$u(\theta_{\text{TRD}}) = \sqrt{[u(\theta_{\text{TRD}} - \theta_{\text{Scintillators}})]^2 - [u(\theta_{\text{Scintillators}})]^2} \tag{6.11}$$

6.6 Path reconstruction

The main goal of this section is to give the reader an understanding of how data outputted by the TRD can be used to reconstruct the trajectory that a particle took through the detector. This is basic functionality which will be useful for future reports and the code has been written in an extensible way with lots of work put into documentation. The hope here is that reports can extend the functionality to do more interesting analysis using the TRD in the future. For more detail on any of the scripts used and the exact steps needed refer to section 10.4, Table 10.1, and the analysis GitHub[17]. Unfortunately this year, the data that was taken from the TRD was not usable for path reconstruction as previously mentioned in section 5.4. Specifically for this section it meant that all of the analysis had to be performed on datasets produced in the 2020 report[15]. The dataset which was used was the one named `0793` and found in `/data/raw` on the TRD computer. This dataset was used as it had the most noticeable particle paths, while other datasets from 2020 had no noticeable particle paths presumably from their own issues in taking data.

The TRD can only precisely reconstruct the angle of the incoming particle along its column dimension, as the column width is much smaller than the row width (7.55 mm vs 90 mm respectively[4, tbl. 3.2]), meaning that particles usually stay within one row of the TRD and there is a large region wherein the trajectory could fall. The fact that particles almost always pass through multiple columns allows us to reconstruct the trajectory in the xz plane where x is the direction covered by the columns and z is the depth. The depth value, z, can be reconstructed from the time dimension since the electron drift velocity is constant (from section 2.2).

We can actually reconstruct the position of a hit to within a single pad using methods described in Wulff's thesis[4, sec. 6.1]. The effect that allows us to do this is called *charge sharing*[4]. In most cases a signal is distributed

among three adjacent pads and for our purposes we will assume that this is always the case. Under this assumption, Equation 6.12 can be used to describe the *position resolution function*, which is defined for every pad as $Q_{\text{pad}}/Q_{\text{tot}}$, and can be parameterised as a Gaussian distribution as shown[19].

$$\text{PRF}(y) = \frac{Q_{\text{pad}}}{Q_{\text{tot}}} = \frac{Q_i}{Q_{i-1} + Q_i + Q_{i+1}} = Ae^{-\frac{y^2}{2\sigma^2}} \tag{6.12}$$

Applying Equation 6.12 to a set of three adjacent pads leads us to Equation 6.13 which tells us the distance from the center of the pad y of a particle (sub-pad precision) as a function the ADC counts in the adjacent pads. Note that Q_i represents the ADC count in the pad i, and $W = 7.55\,\text{mm}$ is the width of the pad (in the column direction). Importantly, this equation for the position depends only on known quantities and the unknown standard deviation σ is no longer present in the equation.

$$y = \frac{W}{2} \frac{\ln(Q_{i+1}/Q_{i-1})}{\ln\left(Q_i^2/Q_{i-1}Q_{i+1}\right)} \tag{6.13}$$

6.6.1 Regions of interest

The zero-suppressed, pre-processed data outputted by `raw2npy.py` is used as an input to `roi.py` both of which were adapted from the analysis GitHub from last year[20] and current versions can be found in this years analysis GitHub[17]. The script `roi.py` outputs a Numpy array file of all of the continuous regions of interest in each event. A continuous region of interest (for a specific event) is defined as a set of pads with total ADC counts (summed over time) above a certain threshold value, which are all in the same row and are in consecutive columns. The regions of interest are then potential paths that the incident particle travelled through. The threshold value used for this report was 700 and this value was found mostly via trial and error. In reality we would prefer to have the threshold a lot higher, around 1000, however with the data we had this was too stringent of a requirement and returned no regions. In the future, with better data, this could be implemented.

6.6.2 Reconstructing paths and calculating angles

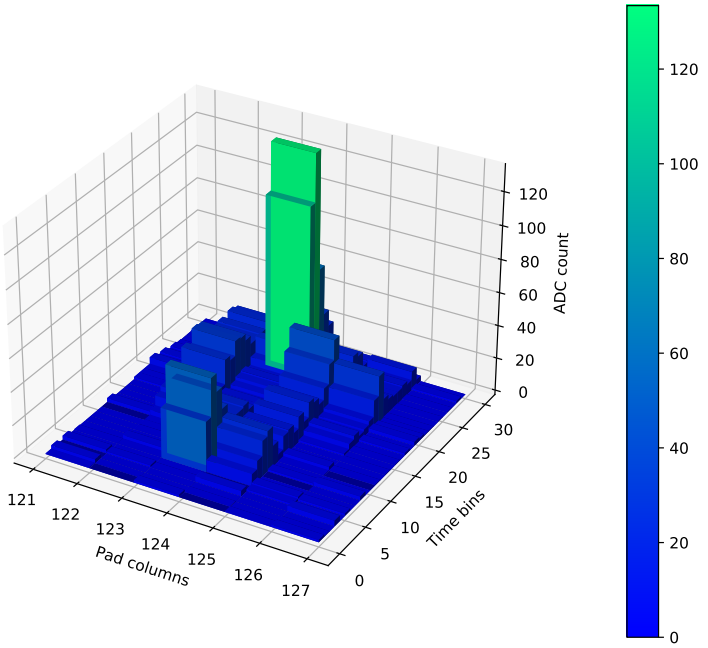
The script `path_reconstruct.py` takes in the regions of interest array from `roi.py` and uses Equation 6.13 to reconstruct the sub-pad position of the particle in each pad. This is used to reconstruct the path that a particle took through the detector in the time vs pad-column plane (which corresponds to the xz plane). This script works by looping through the regions and calculating the position of the particle within each column using Equation 6.13. For each time bin, the column which is the *center* of the hit is defined as the column with the highest ADC count. These columns are then the Q_i values in Equation 6.13 and the corresponding Q_{i-1} and Q_{i+1} values are found by offsetting from the center column. Now the output of Equation 6.13 can be added to the center column value to obtain a set of points which trace out the particle hits through the detector. Since muons move in straight lines, we can use a linear least squares fit on these data points and obtain a trajectory in the xz plane. This can simply be turned into an incidence angle (the zenith angle) using the relation $m = \tan\theta$ with θ the incidence angle and m the gradient of the best fit line. This process is repeated for each region and the angles are stored in an array for later use in working out the angular distribution. In figs. 6.8a and 6.9a, two potential particles paths through the detector are visualized. The highest ADC counts correspond to the center columns previously mentioned, and the exact position of the particle hit is still to be reconstructed using Equation 6.13. In figs. 6.8b and 6.9b, the lines of best fit corresponding respectively to the same previously referenced regions are drawn.

6.6.3 Uncertainties in position of the hit

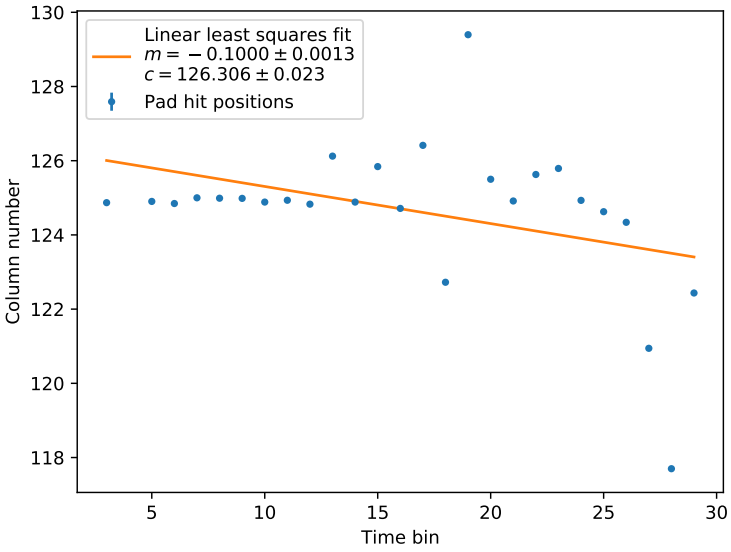
Uncertainties in this section haven't been explored as much as they could have been, given more time with the data and this is definitely an area that can be improved upon. This section will discuss what in principle could be contributing to the uncertainty of the trajectory that the particle takes through the detector. The *position residuals* are defined as

$$\Delta y = y_{\text{rec}} - y_{\text{fit}}, \tag{6.14}$$

where y_{rec} is the recreated position value found by adding the column offset from Equation 6.13 to the center column, and t_{fit} is the fitted y value from the best fit. Wulff shows numerically that these form a Gaussian distribution

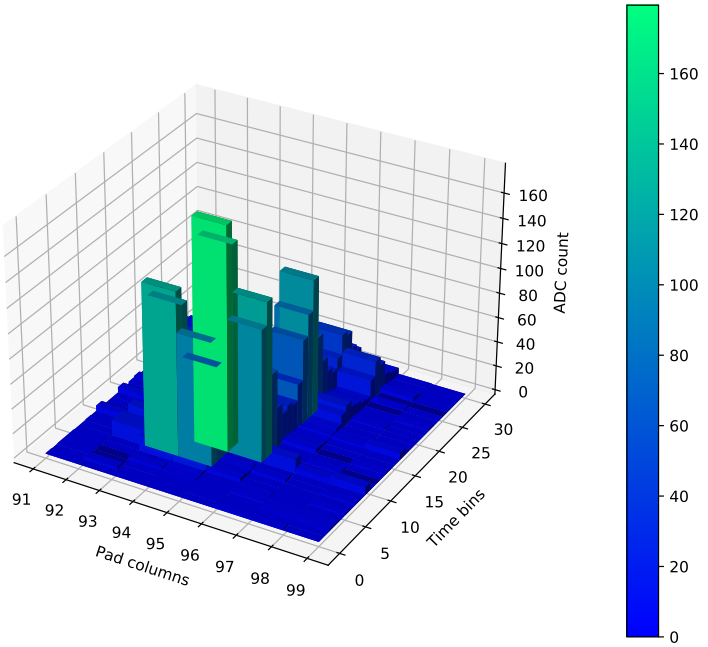


(a) A possible particle path through the TRD in the depth-column plane. Note the high ADC counts which correspond to the center columns talked about previously (approximate center of the pad hit).

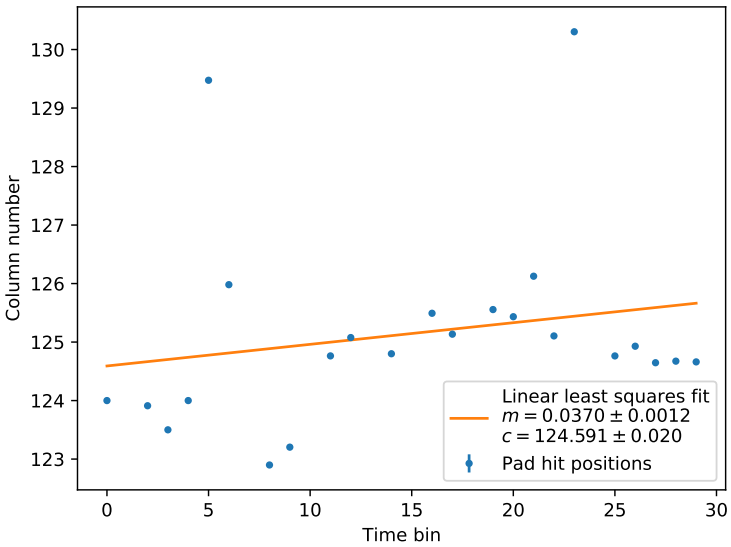


(b) The best fit line of the pad hits of the potential particle in the detector. Note time bins which resulted in angles with very high uncertainties were discarded as these usually were from noise.

Figure 6.8: Illustration of what a particle passing through the detector looks like in the data. This is a specific particle path trajectory of event 6 in the data file `0793` from 2020[15]



(a) A possible particle path through the TRD in the depth-column plane. Note the high ADC counts which correspond to the center columns talked about previously (approximate center of the pad hit).



(b) The best fit line of the pad hits of the potential particle in the detector. Note time bins which resulted in angles with very high uncertainties were discarded as these usually were from noise.

Figure 6.9: Illustration of what a particle passing through the detector looks like in the data. This is a specific possible particle path trajectory of event 11 in the data file `0793` from 2020[15]

and the variance of this distribution σ is defined to be the *position resolution* of the detector. The position resolution was calculated by Wulff[4, tbl. 6.1] for our detector layer (L4) to be $\sigma = 408.9 \pm 1.0$ µm, or approximately $1/20^{th}$ of the pad width. There are lots of other things which could contribute to the uncertainty which are not taken into account but are listed for future reports:

- The likelihood that the region we see is signal and not just noise. For our purposes we chose a threshold value of ADC counts that was needed in a column, however the likelihood that this was signal was not calculated.
- The uncertainty on more than one particle passing through a similar region of the TRD and being detected by our scripts as a single particle. This would result in the wrong angle being calculated. Maybe there is a technique to identify these occurrences?
- The uncertainties on the ADC converters in the detector.

Now that we at least have a lower bound of the uncertainty, given by the position resolution, we can include this in the best fit calculations for the trajectory. Since the uncertainties are constant this won't affect the actual trajectory but it will have an affect on the uncertainty in the measured angle. Finally we can calculate the distribution of zenith incident angles, and plot this in a histogram as shown in Figure 6.10.

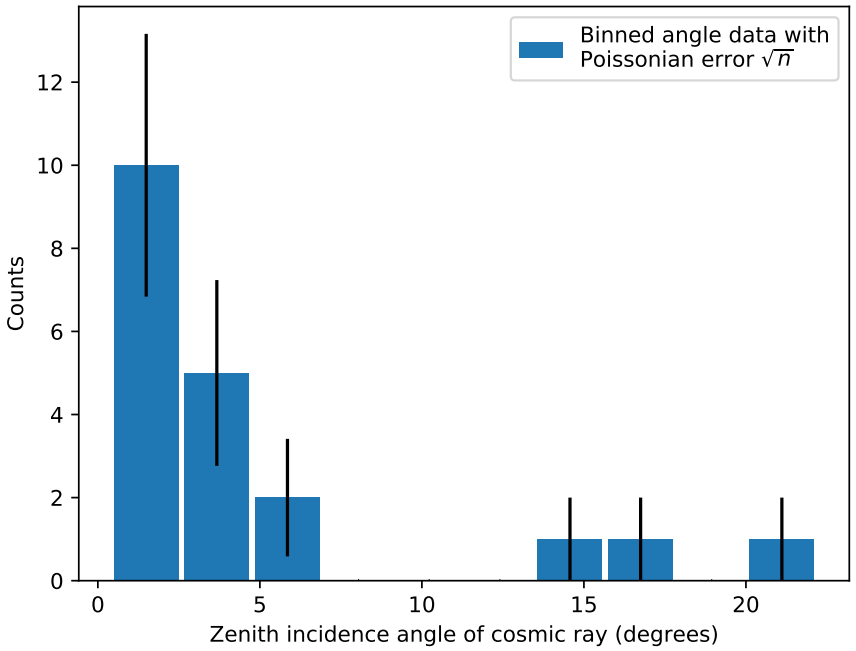


Figure 6.10: Histogram of incident zenith angles of possible cosmic rays passing through the detector along with associated Poisson uncertainties.

At the start of this project there were a lot of (possibly overambitious) goals for comparing the analysed data to the predictions made by simulations in section 2.4. Unfortunately due to the problems with taking data, a lot of these goals could not be realised, however this is one of the most exciting areas for further research. We can still however make comparisons between Figure 6.10 and Figure 6.14. Although there is not enough data to make a definitive comparison, we can qualitatively see that the intensity of muons is much higher closer to zero degrees incidence angle which agrees with the simulation. With more data, it would be possible to try and fit the binned data to the predicted curve if uncertainties were handled with care. This is a possibility for future research.

6.7 Angular Distribution

6.7.1 Necessary angular components

Due to the nature of the TRD only one angle could be accurately measured and as such the second angle necessary for computing the distribution of the muons could be determined using the scintillation detectors. Regrettably this was not

attainable for us and thus we resorted to using data from the previous year [15] in order to refine the method for plotting the angular distribution.

6.7.2 Angular distribution in polar coordinates

The process of plotting the distributions was computed using the *angulardistribution.py* script available at GitHub [17]. The location and time zone of the TRD had to first be defined in terms of the astropy module, where the TRD was located at Latitude: -33.955° and Longitude: 18.462° and had a time zone of GMT+2:00. Both angles (altitude and azimuth) were then converted into Right Ascension (RA) and Declination (DEC), relating to polar angle θ and radius r respectively, and so the polar plot of angular distribution could be produced (Figure 6.11).

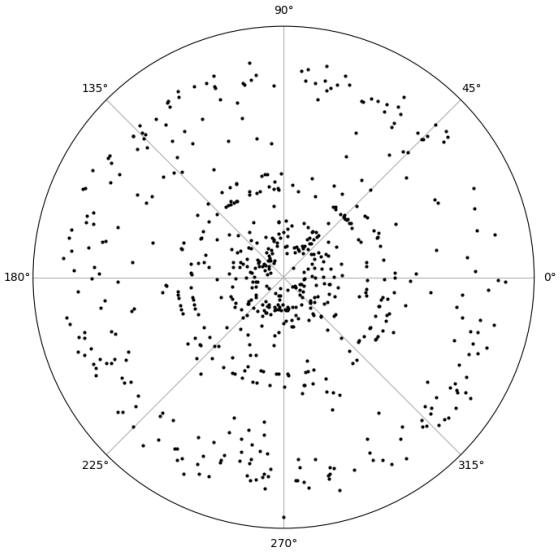


Figure 6.11: A polar plot representing the angular distribution of muons surrounding the TRD. The distribution is more dense towards the center and has fewer events towards the outskirts. There are also concentric circles surrounding the central cluster.

Figure 6.11 had a distribution that was, at least in angular terms, majorly isotropic. The concentric circles could be due to effects of precession [21] - the *wobble* of the Earth as it rotates - and the majority of the distribution was near the centre which is in agreement with the previous year's works.

6.7.3 Conversion to spherical coordinates

The conversions from RA and DEC to spherical coordinates [21] are given by

$$x = \cos(\alpha)\cos(\delta) \tag{6.15}$$

$$y = \sin(\alpha)\cos(\delta) \tag{6.16}$$

$$z = \sin(\delta) \tag{6.17}$$

where α is the Right Ascension and δ is the Declination. Using the same python script, the spherical plot of the angular distribution was produced (Figure 6.12).

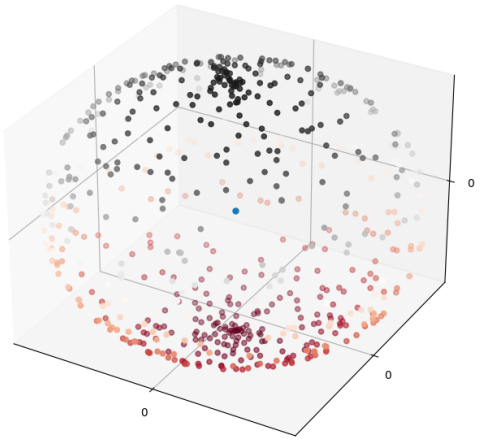


Figure 6.12: The spherical plot of the angular distribution of muons. The blue dot represents the TRD whilst every other dot is an event. Note that the colours of dots are for visual purposes and are not a measure of intensity.

Figure 6.12 illustrated the isotropic nature of cosmic showers, having produced a figure from which the spherical shape could be visualised. There was a slightly larger frequency of muons going directly downward through the TRD as opposed to going horizontally through it and could have been due to the inaccuracy of the second angle's measurement.

6.8 Simulation

6.8.1 Perspective of Cosmic Shower

The first effort was to develop a qualitative perspective on the amount of surface area a prospective cosmic ray shower could cover. This would help to develop a better intuition of the behaviour of cosmic ray showers as well as an understanding of when the detector and scintillation detectors may be reading muons from the same shower. CORSIKA was used to simulate the shower of a 1 PeV proton at normal incidence. The effective surface area covered by muons from the shower reaching sea level was approximately 25 km^2. Figure 6.13 compares this area to the city of Cape Town for a sense of scale. The effective surface area reduces as energy reduces, along with particle flux density. Becoming nearly negligible at energies <100 GeV. It is for this reason that the simulation will keep to energies >100 GeV.

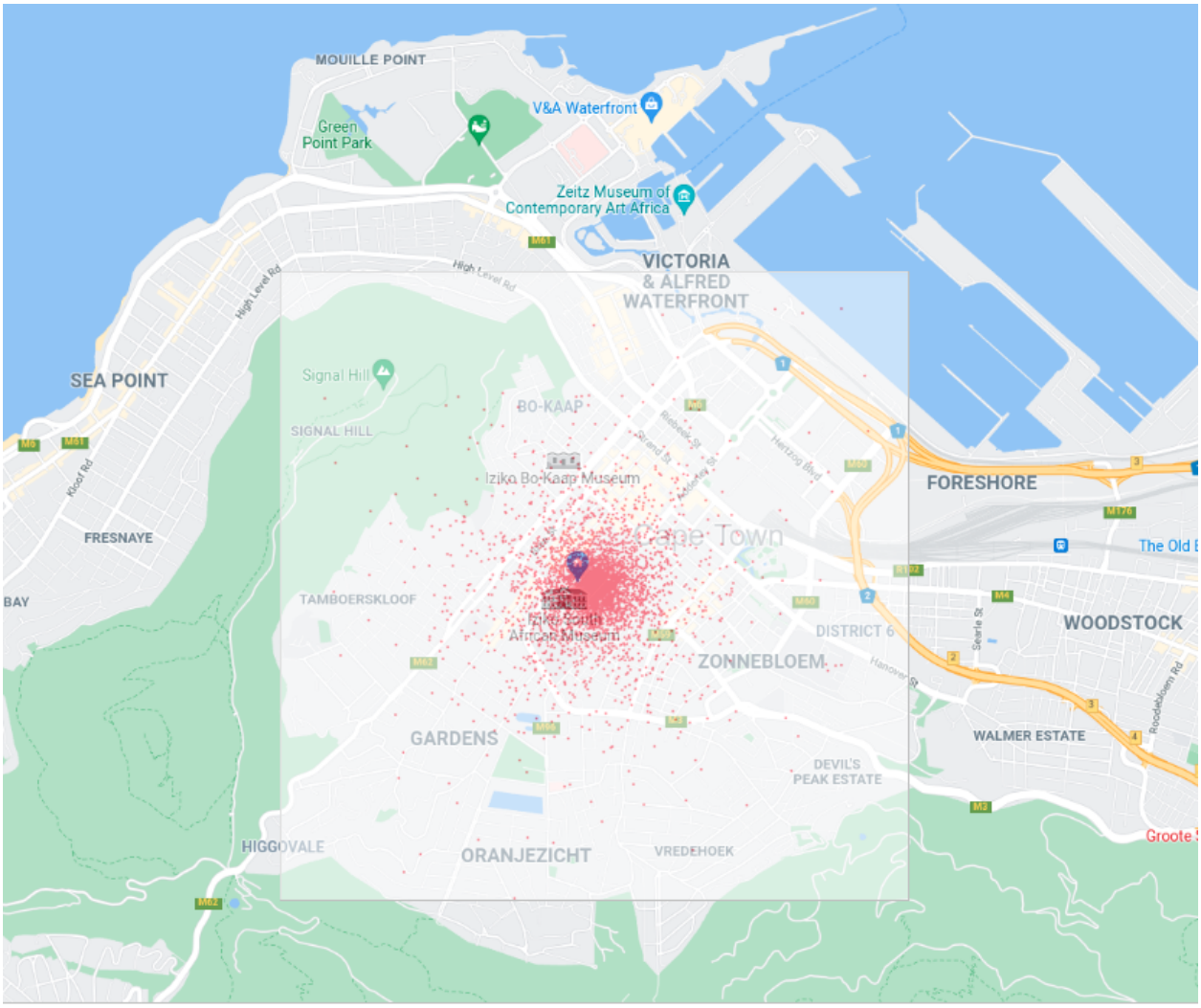


Figure 6.13: Comparison between City of Cape Town and 5 km wide cosmic ray shower. Muons which reach the ground level are indicated as red dots.

6.8.2 Erroneous Endeavours

Initial attempts were made to investigate the relationships between muon count at the detector level with energy and incident angle of the cosmic ray. It was later determined however, that such an analysis would not yield useful results that can be verified experimentally. The rate of incidence of cosmic rays over different zenith angles and energies into the atmosphere is largely unknown and a crucial component to determining a muon flux on the surface of the earth.

Another idea which was considered was to determine the muon count which would be measured by the 1 m^2 TRD for various energies and polar angles of incident cosmic rays with a momentum trajectory towards the centre of the detector. However this would also prove an unhelpful metric to investigate since the proportion of incident cosmic rays moving towards the centre of the detector to every other possible position within the effective area of the detector is very low.

6.8.3 Relative Angular Intensity

A useful result from the simulation would be to determine the relative ratio of intensity between varying angles of incidence. However, it is required to assume that each the cosmic rays are isotropic throughout space. The intensity of the various particles of different energies is indicated in Figure 6.14, in which we note a general differential relationship

between intensity and energy on the order of 10^{-3}GeV. This can thus be used to weight the results of the simulation accordingly as it iterates through several energies and polar angles.

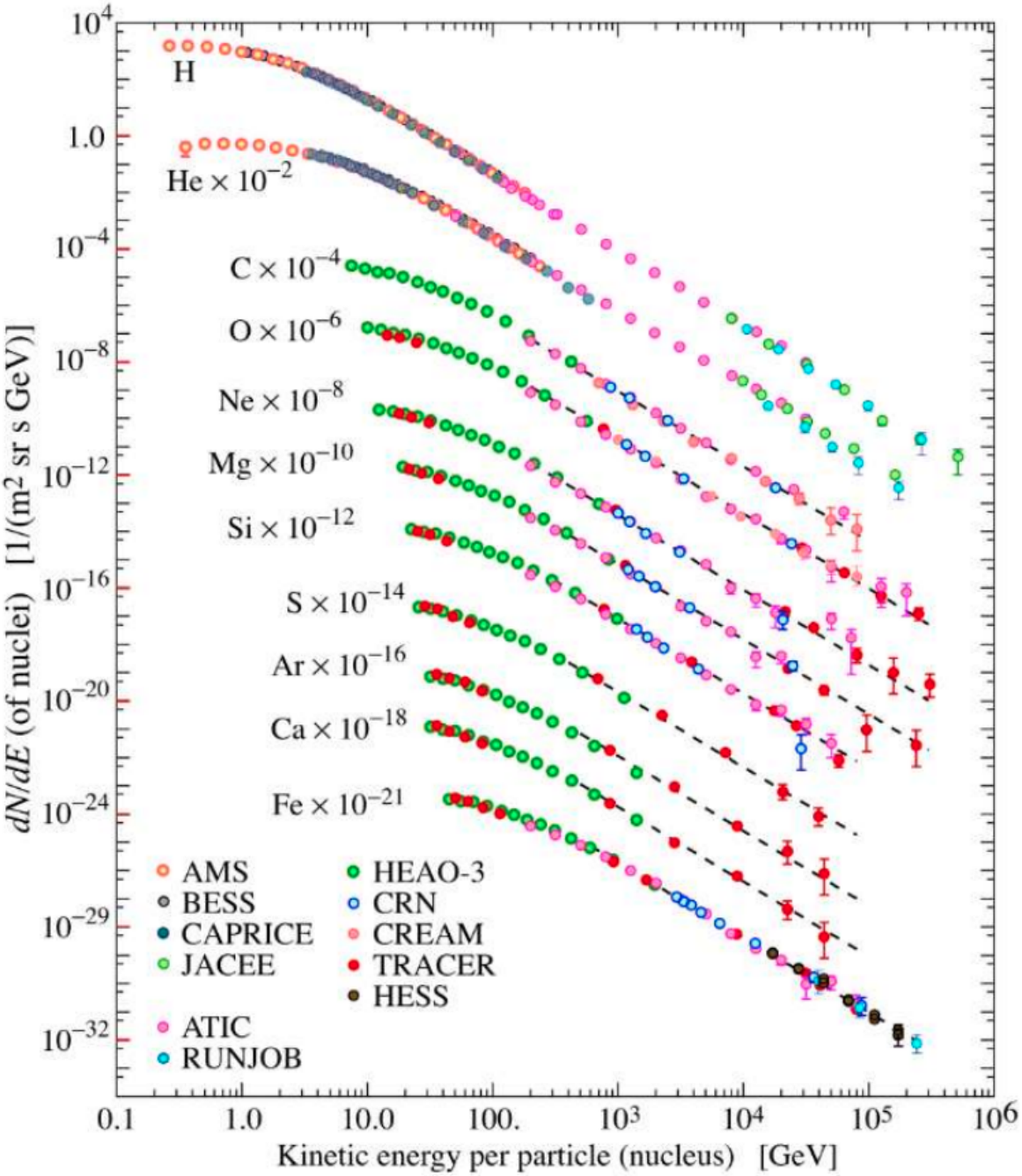


Figure 6.14: Fluence energy distribution of the particles which make up cosmic ray spectrum as a function of energy (Particle Data Group, 2012).

For each angle and energy, 10 showers were simulated to obtain an average muon intensity per shower. The data produced approximately linear relationships which can then have a line of best fit found for each energy. This is demonstrated in Figure 6.15. A linear fit was performed for simplicity due to time restrictions, however a more fitting model is available [22].

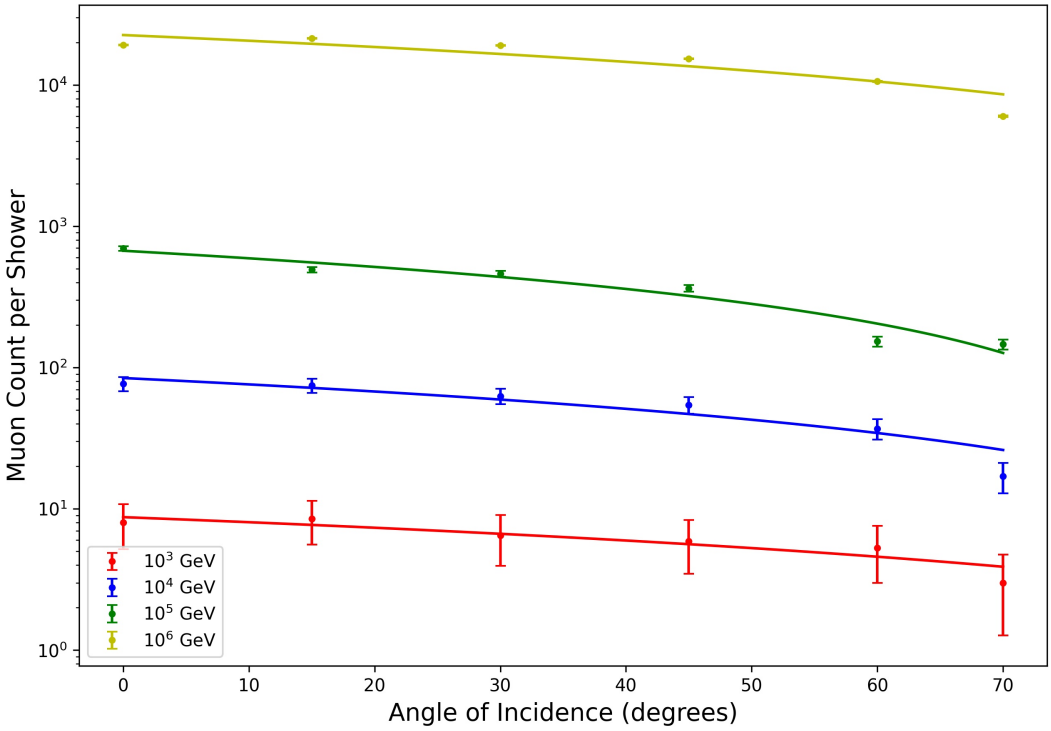


Figure 6.15: Plot of muon intensity at various angles of incidence for energies 10^3 GeV, 10^4 GeV, 10^5 GeV, 10^6 GeV with respective line equations (where N denotes muon count) $N = 1 - 7.92\theta \times 10^{-3}$, $N = 1 - 9.86\theta \times 10^{-3}$, $N = 1 - 11.6\theta \times 10^{-3}$, $N = 1 - 8.85\theta \times 10^{-3}$.

The gradient of the lines is the value of interest as it can be used to find the ratio of muon intensities between two different angles. Since we only want the relative change, each line equation can be normalised by dividing it by its y-intercept. We can then use the data in Figure 6.14 to determine weightings for each of the energy levels, then additionally weigh them by factoring in the muon count for each energies shower. Thus from the weighted means of the gradients of each equation we obtain $m = -7.94 \times 10^{-3}\ \theta^{-1}$. Then, given a certain polar angle θ_1 with intensity I_1, we can find the relative intensity I_{rel}, at angle θ_2 by the equation:

$$I_{\text{rel}} = m(\theta_1 + \theta_2) + I_1 \tag{6.18}$$

It is important to note that the relationships in Figure 6.15 hold only when keeping phi at a constant. If we allow phi to be variable, a rotational degree of symmetry is introduced. This entails an increase in multiplicity of incident rays at a certain polar angle, θ, as θ increases – counteracting the reduction in intensity induced by atmospheric interference. Further analysis is required to take these effects into account quantitatively. The data produced in this section provides substance for comparison against experimental measurements.

6.8.4 Time Distribution

As a brief investigation was performed looking into the time intervals over which muons from the same shower would reach the observation level. These results can assist in determining scintillator time and angular resolutions. Figure 6.16 presents the muon incidence time distribution from a 1 PeV proton shower.

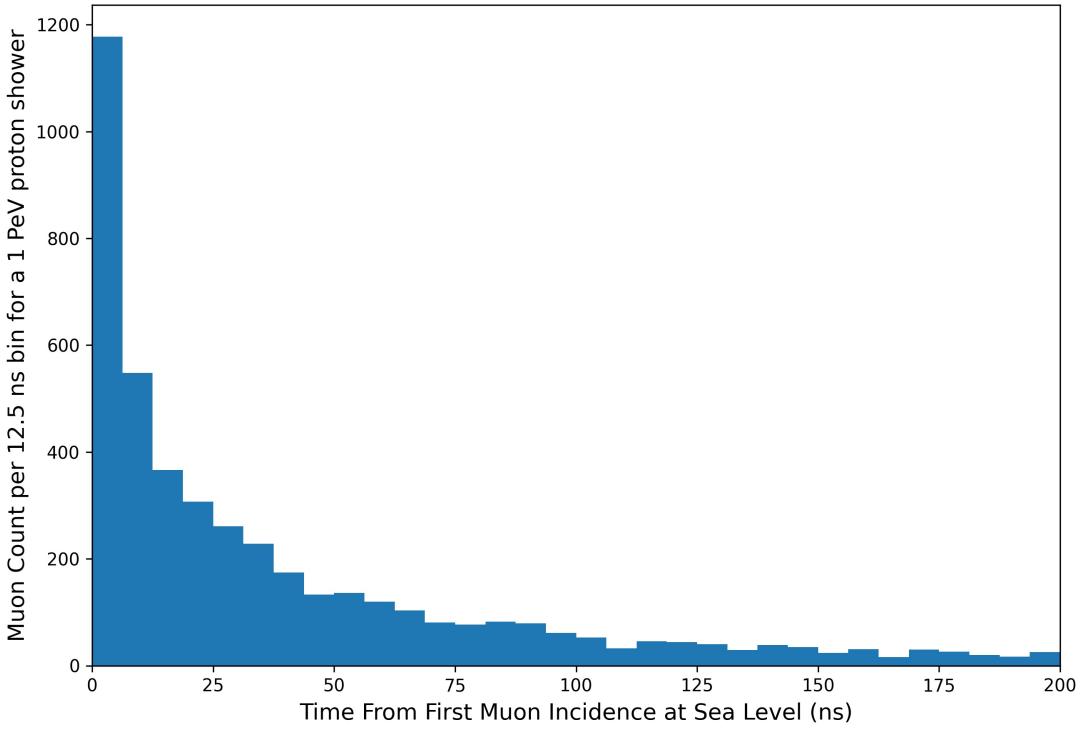


Figure 6.16: Histogram indicating the time spread using counts per 12.5 ns bins over which muons from a 1 PeV proton shower reach sea level.

7 Conclusion

To conclude this report, some highlights must be made of the main results.

The principle objective of this lab which was to consolidate the operations necessary to run the TRD module with full incorporation of the scintillation detectors, was achieved through the implementation of the new DAQ system. This fully integrates waveform data from the scintillation detectors in conjunction with events from the TRD module. (See section 5.) In addition to this, the GUI has allowed the control of the TRDBOX to be greatly simplified for future use, adding further ease to the consolidation of detector operations and data acquisition.

Other highlights include the measurement of the time resolution in section 6.3, the optimisation of the scintillation detector distance in section 6.4 which makes use of a Monte Carlo Simulation, the path reconstruction of particle trajectory in section 6.6, as well as simulations performed using CORSIKA in section 6.8.

An obvious disappointment in this project was that it was not possible to acquire usable new data before the report deadline, despite many valiant efforts by a large portion of the group. This has resulted in only being able to give an outline of the methods by which the angular resolution of the TRD module and angular distribution of cosmic rays in the sky would have been measured. These sections do still contribute to our attempt at consolidating all facets of this project so that future endeavours in this area may benefit and build upon it.

Some suggestions for future endeavours are provided in section 8.

8 Future Endeavours

8.1 Scintillation Detectors

The first improvement that can be made with regards to the scintillation detectors is fixing the problem mentioned in section 4.2, where the data is output in a strange way, breaking the system. We believe the problem is simple to fix but did not have the time to implement a full fix.

We were also not able to fully utilise the scintillation detectors to their fullest ability when it came to the TRD itself. The scintillation detectors can by used to physically exclude some data from the experiment and we only used one configuration, designed to best select muons from muon showers, but other configurations, for other outcomes, can also be explored.

The `scopeRead.py` program in [11] returns data in a `numpy` array. This is not an issue when taking scintillation detector data on its own, but it appears to be a bottleneck when taking data in conjunction with the TRD, as outlined in section 5.3. If the output of the `Reader.getData()` method can be modified to output in hex format, which is how the oscilloscope returns the data in the first place, which can then be processed afterwards, that would improve run time. The problem here arises in the conversion from hex to voltage, which relies on data from the header of the data returned from the oscilloscope. See the programming manual of the oscilloscope for more information on that conversion.

Lastly, the problem discussed in section 3.3 should be addressed as soon as possible in order to begin to properly analyse the angular resolution of the TRD module.

8.2 TRDBOX

Not all of the envisioned commands for the TRDBOX could be implemented in time for this report. The next step would clearly be to finish implementing all of the new commands, including (but not limited to) separate commands for each option contained in the `pre-conf` register, a `reset` command that sets all registers to reasonable default values, and a command that will successfully reset the `pre-cnt` register. This last command would be especially useful when examining how the discriminator threshold settings affect the count rate.

Additionally, the TRD GUI was not fully implemented at the time of writing. While all the necessary foundations have been laid, it was never tested beyond a few simple commands. A fully fleshed-out GUI implementation would make operation of the setup a breeze.

To aid in this, options to bind multiple commands to a single button can be added. Furthermore, the user can be allowed to specify which row the user wants to remove. Lastly, and perhaps most difficult, would be allowing users to save and load sets of configurations, so a configuration can be saved for each possible use case of the GUI, which would eliminate the need to configure the GUI under all normal circumstances.

8.3 Data Acquisition

A full investigation should be done into timing issues with the data acquisition, particularly with regards to the performance of `minidaq.py` and the TRDBOX control settings (see section 5.4 for a detailed discussion on the issues encountered). Either `minidaq.py` or the subevent builder threads should be modified to prevent the stalling discussed in section 5.4.1. The capture of oscilloscope waveform data in the miniDAQ could be sped up significantly using the outline described in section 5.4.2. The `.o32` file format should be fully extended to include oscilloscope waveform data and ADC counts in a single file.

8.4 Analysis

The analysis this year was hindered severely by the amount of time it took to take data, and the other problems with data collection described in section 5.4. It is our hope that the progress made this year will lay down a solid foundation for further analysis in the future, and less time will be wasted on setting up the basic scripts needed for analysis.

One of the main future projects could be to expand on relating analysis of real data to the simulation production from CORSIKA. This could be done by with a larger dataset if you are careful about managing all of the uncertainties. Specifically a good thing to compare is the angular distribution of incident muons. To do this, both the TRD and scintillators must be used together so that zenith and azimuthal angles can both be measured for each event. If you instead setup the TRD and scintillators to measure the same angle, we can calculate the angular resolution of the TRD, as was the original plan for section 6.5.

Using the aforementioned setup, we can reconstruct paths of incident cosmic rays to their location in the sky. Using section 6.7 this can be translated into astronomical coordinates, and further research could be done to try and identify which cosmic objects were the cause of the different cosmic rays.

The code for calculating the regions (`regions.py`) could be expanded to include multi-row regions which are less likely to happen but can provide better angular resolution if they do occur. A more sophisticated algorithm could be used to determine whether a region is comprised of multiple incident particles or just a single particle, and multi-particle regions could either be excluded or disentangled if possible.

8.5 CORSIKA Simulation

CORSIKA was used to simulate various scenarios in order to gain a better understanding over cosmic ray showers, and particularly muon production. A scatter plot of muon positions of incidence at sea level was overlayed onto a map of Cape Town to provide perspective of the effect surface area that a cosmic air shower could cover. Further investigations here could consider the relative surface areas and incident muon densities produced by various energies and incident angles. It could also be looked into what the probabilities of such showers would be that they would send muons through the scintillators and/or TRD.

A more rigourous investigation into the details of Figure 6.14 and its incorporation into a more realistic simulation would be helpful in creating a closer to reality simulation - the goal of which would be to determine what measurements should be expected by the scintillators and TRD.

Lastly, it is worth mentioning that the creators of CORSIKA advised using the python package 'Matrix Cascade Equations' (MCEq) for investigating muon flux. This library interfaces CORSIKA as well as other modules, and would seem to have a steep learning curve.

9 Project Roles

- Tenille Bjerring - Spokesperson, organiser, networking, general herder of cats
- Miles Kidson - Head of scintillation detector subgroup, oscilloscope readout
- Max Matheussen - Time resolution measurements and analysis
- Victor Bantchovski - CORSIKA simulations and theory work
- Matome Modiba -
- Cole Faraday - Data pre-processing, path reconstruction analysis, and analysis technical guide
- Antonia Grindrod - Head of analysis subgroup, Scintillator distance optimisation and TRD angular resolution analysis
- Duncan Torbet - Angular distribution analysis
- Sam Cohen - Monte Carlo simulation analysis, TRD module services
- Nhlanhla Hlengane - TRD Background
- Kamogelo Khareba - TRD Background
- Samson Thornhill - TRDBOX: configuration, implementation of new commands, how-to and commands guide
- Nicholas Yerolemou - TRDBOX: Head of TRDBOX subgroup, implementation of new commands
- Du Toit Spies - TRDBOX: Creation of a GUI to control the TRD
- Frank Smuts - Head of data acquisition subgroup, oscilloscope data, ADC count file output, background data runs, documentation
- Dillon Lewis - Data acquisition, subevent builder integration, cosmic ray data runs, data collection, TRD box wrangling

- Timothy Schlesinger - Communications, data acquisition, extensions to `evdump.py`/`o32reader.py`, solving readout stalling, Github repo, technical guide

10 Technical Guide

This guide serves the purpose of laying out the basic technical skills that will enable the reader to use, understand and modify the available tools. Note that care has been taken to elaborate on the underlying structure, and not just the commands used, since specific implementation details are likely to change between years.

10.1 General Technical Skills

This section lays out the basic skills necessary to work on the codebase and navigate the machine. It is advised that any user has a good grasp of these before working on any code.

10.1.1 Linux basics

To navigate through the Linux system, the following commands will be useful:

- `cd <dir name>` - allows you to change directory
- `ls` - lists the contents of the current directory
- `mkdir <name>` - creates a new directory
- `pwd` - displays the path to the current directory
- `less <file name>` - displays the contents of a file, q to exit
- `rm <name>` - removes a file, use the `-rf` flag for a directory
- `mv <location 1> <location 2>` - moves a file
- `cp <location 1> <location 2>` - copies a file

Some notes:

- Some of the TRD tools will have to be run with `sudo` in front - this means you are invoking root (admin) privileges and should be extra careful.
- You can always use some combination of Ctrl+C and Ctrl+Z to kill a program - this could result in partial execution or errors, but is sometimes necessary e.g. in the case of infinite loops.

10.1.2 Vim

Vim is a software that allows you to edit files in the terminal. To do so, you can open the file with one of the following two commands:

```
vi <file name>
vim <file name>
```

It will open in command mode. You can navigate with the HJKL keys or the arrow keys. You can also use Shift+G and g+g go to the end/beginning of the file respectively while in command mode. To switch to insert mode, which allows you to edit the contents of files, type ‘i’. You should then be able to write your code. Note that many of the python files use four spaces instead of tabs, so be wary of this. To save your changes, you can press Esc to return to command mode. Then type ‘:w’ (followed by Enter) to write the file, ‘:q’ to quit, or ‘:wq’ to do both (:q! will quit without saving). In command mode, you can also undo with ‘u’ and search by typing ‘/’ followed by the search string. If you search for a word that appears multiple times using ‘/’, you can use ‘n’ to go to the next instance of the search string.

10.1.3 `ssh` and `sftp`

`ssh` is a tool that allows a user to access a machine remotely. The command used to access the TRD is:

```
ssh trd@alicetrd.phy.uct.ac.za
```

To be able to use this (without the password for the machine), you will need to setup an ssh key on your GitHub and send your username to the machine admin (Tom Dietel). This ssh key can be setup under User - Settings - SSH and GPG keys. This may also require some local configuration.
Once `ssh` is setup, it is also possible to use `sftp`. This allows you to transfer files between the remote machine and your local machine. It can be used with:

```
sftp trd@alicetrd.phy.uct.ac.za
get <filename>
put <filename>
```

Where `get` will download a file, and `put` will upload a file. To do this with directories, add the `-r` flag, or zip them first. To use commands locally (like `cd` and `ls`), preface them with `l` e.g. `lcd` and `lls`. This allows you to navigate both remotely and locally.

10.1.4 Virtual environments

Virtual environments (`venv`) can be thought of as isolated Python environments where libraries can be temporarily installed without being available everywhere on the machine. It will often be necessary for a user to activate a `venv` and install the requirements before running a tool or command. Usually, there will be an associated README explaining how to do so, but in general the process for this will be:

```
python3 -m venv venv
. venv/bin/activate
pip install -r requirements.txt
pip install -e .
```

Sometimes, it is necessary to run:

```
pip install ---upgrade pip
```

To upgrade pip, so that it can recognise all of the packages correctly.
Once the `venv` is installed you may find that you want to run a different version of the program you are currently running. Navigating to the location of this new program and running will result in the same old version of your program being run. In order to change which version of the program is being run you should navigate to where you activated your `venv` session, place the new version of the program within a sub directory and run

```
pip install -e .
```

This will rebuild your `venv` with the new version of the program you wish to run.

10.1.5 Git usage

Git is a version control system that allows you to keep track of different versions of files and allows different people to work on the same files and then “merge" their work. Work on this project will require a mix of local and remote git usage. The recommendation is for someone in the group (preferably someone who has used and understands GitHub or at least git) to setup a repo on GitHub. They can then give everyone developer access to this repo and everyone can work locally and upload their changes to this repo.

Setup
To set up a local repo to track this remote repo, make a new folder with `mkdir` (it works quite well if everyone works under a folder with their name) and run:

```
git init
git remote add origin <link to repo>
```

Where the link will look like `https://github.com/TenilleLori/ALICE-Project/`.
You can run:

```
git remote -v
```

To check that the origin remote is correctly setup.
Then run:

```
git pull upstream <branch name>
```

Where the only branch is probably 'master' (or 'main'). If the repo is not public, you may need to enter a username and access token.

Local management
The command:

```
git branch
```

Shows you the branches you have, and marks the currently selected one with an asterisk.
To create a new branch, you can run:

```
git branch <name>
```

It is usually good practice to use your own name as the branch name. To then select this branch, use:

```
git checkout <name>
```

To delete a branch, use:

```
git branch -D <name>
```

Once you make changes to files, you will need to stage and commit them. To do this, you can start by running:

```
git status
```

This will tell you the state of each file in your repo, including files that have been created, deleted or modified. To select some of these files to include in your push to the remote repo, use:

```
git add <file 1> <file 2> ... <file N>
```

Remember that files in subdirectories must be referenced with a path that indicates this. You can run `git status` again to check that the correct files have been added, then run:

```
git commit -m "<message>"
```

To commit these files. This gives you a version of the repo that has a unique hash and that you can always revert to if you need (`git revert <hash>` where the hash can be found from `git log`). After a few commits, you will want to push your work to the remote repo so others can pull it and work on it.

Pushing changes to remote repo
To be able to push your changes, you need to set up a git access token on your account (and ensure that your account has the correct privileges on the remote repo). To set this up, go to User > Settings > Developer settings > Personal access tokens. Make a new access token, give it at least all repo permissions. Ensure that you copy its hash to a textfile and don't lose this! (Although you can always create another token if you do.) Now, you can run:

```
git push upstream <branch name>
```

Where branch name should be the name of the branch you are currently on (`git branch` to check this). This will prompt you to enter a username and password. Enter your username, and enter the hash of your git access token as a password. This should then push your changes to a branch on the remote repo.
The final step is to create a pull request. Visit your remote repo, and go to pull requests. Create a new one from the branch to which you just pushed to into master, then merge it. Now, anyone who pulls from master should see your

changes.

Dealing with merge conflicts
Contact someone in your group who knows git the best. Pray to whichever god you believe in that you don't come across many of these. To avoid them, try not to work on the same parts of a file at the same time. These come in to play when you both change a certain line in your file and then try to push your changes. If you are regularly pushing and pulling your work, you are less likely to come across major conflicts. If it comes down to it, don't be afraid to use '–force' - but remember that this is a last resort.

General workflow practice

- Work in your own name folder, on a branch with your name
- Before you start working, pull from upstream (master)
- Do your work, with regular stages and commits
- Push to upstream when done, to a branch with your name
- Create a pull request from that branch to master
- Tell someone to check your changes and merge your pull request

10.1.6 `tmux`

`tmux` allows multiple users to work in the same workspace, or a single user to save the state of a terminal in between work sessions, as well as allowing multiple terminals to be opened in the same window.
Some useful commands:

- `tmux new -s <name>` - creates a new named tmux session. It is a good idea to use your own name here
- `tmux ls` - lists available sessions
- `tmux attach -t <name>` - enters the session specified
- `tmux kill-session -t <name>` - deletes a session permanently

Some shortcuts inside a tmux session:

- Ctrl+B then " - splits into two terminals above each other
- Ctrl+B then % - splits into two terminals next to each other
- Ctrl+B then arrow key - moves between terminals
- Ctrl+B+arrow key - resizes terminals (note: here, the Ctrl+B is still held when the arrow key is pressed)
- Ctrl+B then D - exits session
- Ctrl+D - closes a terminal
- Ctrl+B then [- scroll mode (exit with q)
- Ctrl+B then { or } - reorders terminals

10.2 Project-Specific Skills

This section lays out some general skills and libraries that are of special interest and applicability in this project.

10.2.1 Bit-wise Operators

There are 4 main bit-wise operators of specific interest:

&
The & (and) operator works on binary strings as follows: it matches the strings up from start to end, and works on each position by giving an output of 1 if both strings have a 1 in that position and 0 otherwise.
00101100 & 10101001 outputs 00101000
0101 & 1001 outputs 0001

|
The | (or) operator works on binary strings as follows: it matches the strings up from start to end, and works on each position by giving an output of 1 if either of the strings have a 1 in that position and 0 otherwise.
00101100 & 10101001 outputs 10101101
0101 & 1001 outputs 1101

»
The » (right-shift) operator works on a binary string and integer n as follows: it takes the string and shifts everything to the right n places, then pads the string with 0s to ensure it has the same length as when it started.
00101100 » 2 outputs 00001011
0101 » 1 outputs 0010

«
The « (left-shift) operator works on a binary string and integer n as follows: it takes the string and shifts everything to the left n places, then pads the string with 0s to ensure it has the same length as when it started.
00101100 « 2 outputs 10110000
0101 « 1 outputs 1010

Hex and binary
Most of the binary strings you deal with will actually be hex strings. Each hex character corresponds to 4 binary bits, since $16 = 2^4$.
0x10AE4502 = 0001 0000 1010 1110 0100 0101 0000 0010
To see how bitwise operators apply to hex strings, first convert the string to binary, apply the operator, then convert back.

10.2.2 `click`

`click` is a library that allows you to call functions inside your python files from command line. To do this, you define a click group with the decorator:

```
@click.group()
```

And define click methods with the decorator:

```
@<group name>.command()
```

Where group name is the name of your `click` group. You can also allow `click` to capture flags and arguments with:

```
@click.argument('<name>', default=<default>)
@click.option('--<name>', '-<initial>', is_flag=True, help=<help>)
```

e.g.

```
@click.argument('timestamp', default=datetime.now())
@click.option('--othermode', '-o', is_flag=True, help='other mode')
```

Now you can call click methods in command line. E.g. if your group is called 'minidaq' and your method is called 'read' then you can run:

```
minidaq read
```

Note that an underscore in a function name becomes a dash in a command line call.

10.2.3 ZeroMQ

ZeroMQ is a message-sending library that can be used cross-language (e.g. C++ to Python). Most of the tools we use have 'REQUEST' sockets that send a request to another socket that then replies with a response. This send/receive pattern is used in minidaq, for example.

```
self.context = zmq.Context() # ZMQ context
self.sfp0 = self.context.socket(zmq.REQ) # New request socket
self.sfp0.connect('tcp://localhost:7750') # Connect to port
ctx.obj.sfp0.send_string("read") # Send request
data = ctx.obj.sfp0.recv() # Receive response
```

10.3 Project Operation

This section covers some of the specific commands that can be used to invoke the currently available tools. Note that these often change between years, and may no longer have the same syntax.

10.3.1 Normal usage

To take data (the standard use case), the following commands should be run, in different terminals:
1. The trdbox thread:

```
sudo /usr/local/sbin/trdboxd
```

2. The subevent builder thread:

```
/usr/local/sbin/subevd --sfp0-enable=true --sfp1-enable=true
```

3. The actual data acquisition thread, minidaq. Before running this, ensure you have activated the venv, as in section 10.1.4. Then you can run either of the following commands, depending on what measurements you desire:

```
minidaq trigger-read -n <num events>
minidaq background-read -n <num events>
```

Note: before running these commands, ensure there is a folder named 'data' in your current directory.
You can also, instead of steps 1 and 2, just run:

```
minidaq setup
```

This data will be saved to the data folder. Then, it can be read with:

```
evdump <filename>
```

The output of this should look something like:

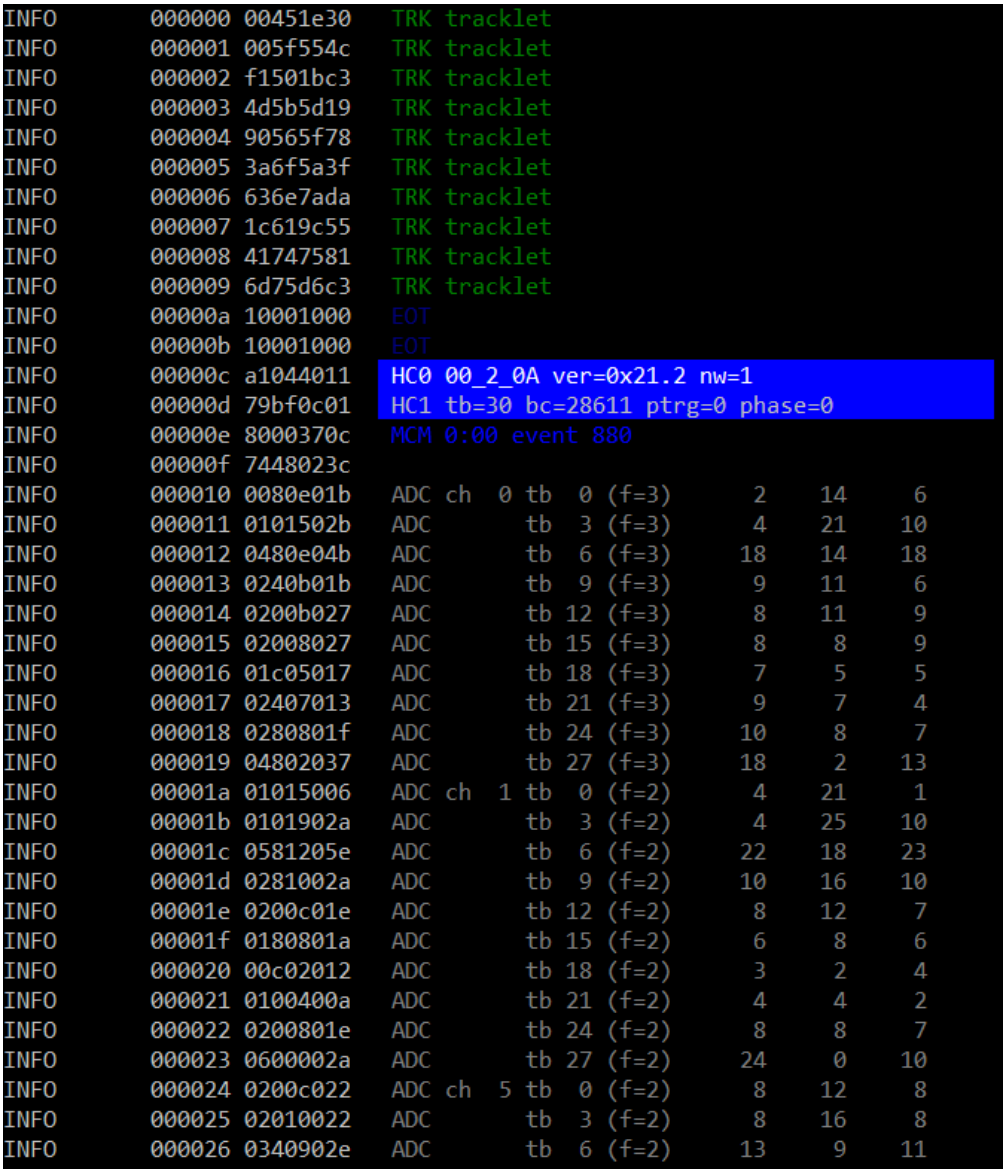


Figure 10.1: This is the data that is output by the `evdump` command. What you don't want to see is many red lines saying '`SKP ... skip parsing ...`'. This probably means the TRD has been configured with `100` instead of `101`.

To better see this, run:

```
evdump <filename> 2>&1 | less -R
```

10.3.2 `trdmon` and `smmon`

It is often the case that `minidaq` will stall because the trdbox is not correctly configured. To check this, there are two commands that you can run (you will probably want to run these in separate terminals so you can watch them as you run other commands). Note that both of these can only be run in an activated `venv`. The first is:

```
trdmon
```

Which should show something like Figure 10.2.

Figure 10.2: This is the display that comes up after the `trdmon` command is run. This green 'CONFIGURED' state is what you will usually seek.

The second command is:

`smmon`

Which will show something like Figure 10.3.

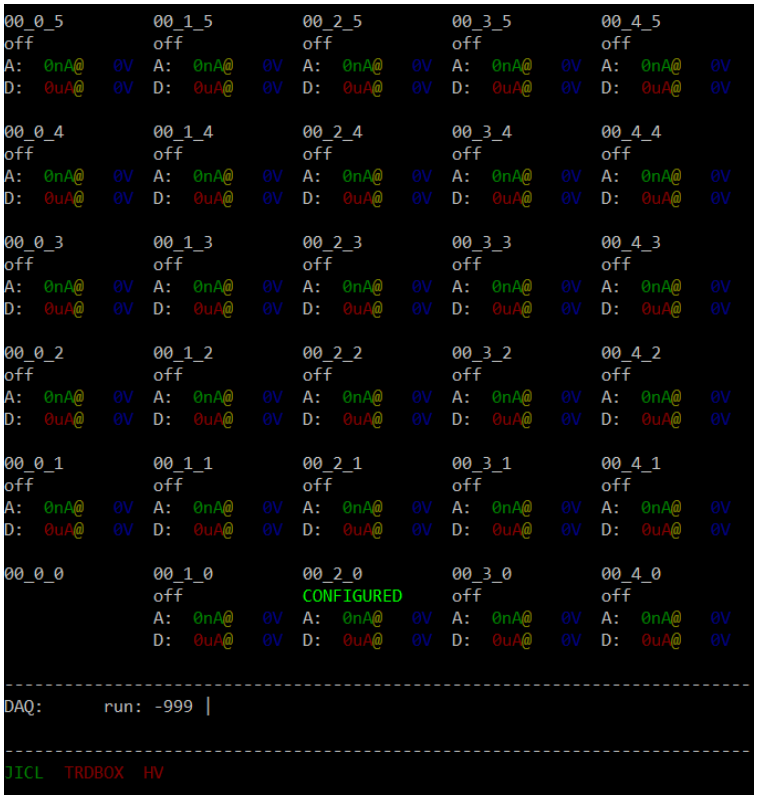


Figure 10.3: This is the display that comes up after the `smmon` command is run. As with `trdmon`, this green 'CONFIGURED' state is what you will usually seek.

10.3.3 `dimcoco` and `nginject`

Often, `trdmon` and `smmon` will not show you the magic green 'CONFIGURED' state. Then, you will have to use nginject to communicate with the TRD. In general, this will involve calling:

```
nginject all <code>
```

Where the code specifies the action for the TRDBOX to take. Note that you need to have an activated `venv` to run nginject. To see the possible codes you can run this with, run:

```
wing_tags ls
```

This should show the following tags:

```
   10  CFINIT    initialize                          r5842
   20  CFUNCFG   initialize                          r5842
   30  CFSTDBY   shutdown                            r5842
   40  CFRESET   shutdown                            r5842
   80  CFCMND    fxs_query                           r5842
  100  CFCFG     cf_p_nozs_tb30_trk_autotrg          r5775
  101  CFCFG     cf_p_zs-s16-deh_tb30_trk_autotrg    r5775
  110  CFCFG     cf_p_nozs_tb30_trk_autotrg          r5775
  200  CFCFG     cf2_trkltp                          r5869-02
  201  CFCFG     cf2_default                         r5869-02
  210  CFCFG     cf2_default                         r5918
  211  CFCFG     cf2_trkl-3q                         r5918
  212  CFCFG     cf2_trkl-tp                         r5918
  220  CFCFG     cf2_krypton_tb30                    r5940.07.3d9da439
  221  CFCFG     cf2_krypton_tb60                    r5940.07.3d9da439
  222  CFCFG     cf2_krypton_tb63                    r5940.07.3d9da439
 9000  CFTEST    bridge_test                         r5842
 9001  CFTEST    reset_test                          r5842
 9002  CFTEST    shutdown_test                       r5842
 9003  CFTEST    ori_test                            r5842
 9004  CFTEST    niscsn                              r5842
 9005  CFTEST    niscsn_fast                         r5842
 9006  CFTEST    dmm_test                            r5842
 9007  CFTEST    imm_test                            r5842
 9008  CFTEST    ddd_test                            r5842
 9009  CFTEST    read_laser_ids                      r5842
 9100  CFCMND    adcstat                             r5842
```

To set up for the `minidaq` to read events, the correct sequence is:

```
nginject all 20 30 10 101
```

Note that configuring with `100` instead of `101` will result in output that is unreadable by `evdump`.
If either `wing_tags` or `nginject` is non-responsive, there is probably a problem with `dimcoco`. This process needs to run in the background for both to work, and is known to stall. To sort this out, you need to kill the process and restart it. To do so, run:

```
ps -C dimcoco
kill <process id>
dimcoco
```

Make sure for the last command that you are in an activated `venv`. If this problem with `nginject` persists, or if there is load-shedding, the following sequence of commands can restore a working version of `dimcoco`:

```
sudo systemctl stop firewalld
dim_send_command pwr/lv off
dim_send_command pwr/lv on
sudo systemctl stop oracle-xe-18c
sudo systemctl start oracle-xe-18c
```

```
sudo systemctl stop oracle-xe-18c
sudo systemctl start oracle-xe-18c
dimcoco
dim_send_command pwr/nim on
```

Which involves power cycling the low voltage and restarting an Oracle database that dimcoco communicates with. The database is not connected to a backup power supply, and will likely be an issue after going through loadshedding. There may also be cases where `nginject` commands complete, but have no effect on the status of the TRDBOX as displayed on `trdmon` or `smmon`. In this case, running `nginject 40` may fix the issue.

10.3.4 TRDBOX Commands and How-To

The TRDBOX commands are all stored in the file `alicetrd-python/src/dcs/trdbox.py`. Each command is defined as a Python function that makes use of Click, a command line interface tool for making new commands available on the command line. All of the currently available commands can be found by typing `trdbox` into the command line, or alternatively `trdbox -help`. All of the current working commands are listed below.

```
dis-conf
dis-thr
pretrigger
reg-read
reg-write
set-dgg
set-dis-conf
set-pre-conf
set-pre-dgg
sfp
status
unblock
```

The most important ones have already been described in Table 4.1, but for completeness all other commands have been listed. The command `reg-write` allow one to manually change registers (and this is in fact the mechanism that most other commands, such as `set-dgg`, use), but it is unintuitive and was phased out as much as possible.

If one needs to manually adjust registers via the `trdbox reg-write` command, one needs to know first the register addresses (and also the current value they are set to). Both can be determined via the `trdbox status` command. An example of the output of `trdbox status` is shown below.

```
pre_conf   [0x100]: 0x00000805 = 2053
pre_dgg    [0x101]: 0x10121012 = 269619218
pre_cnt    [0x102]: 0x0000a7da = 42970
pre_stat   [0x103]: 0x00000001 = 1
dis_conf   [0x28d]: 0x0000000c = 12
dis_freq0  [0x280]: 0x000000d6 = 214
dis_freq1  [0x281]: 0x00000052 = 82
dis_time0  [0x284]: 0x1e440001 = 507772929
dis_time1  [0x285]: 0xcda00008 = 3449815048
```

The bracketed hexadecimal numbers, for instance `[0x100]` for the `pre-conf` register, define the address of each register. This is the address one must use when writing to these registers. The eight digit hexadecimal numbers, preceded by `0x`, give the current value written to the respective register. The far right hand side is merely this hexadecimal number converted to decimal. In all registers except for `pre-cnt`, the decimal number is irrelevant. In the case of `pre-cnt`, however, the decimal number is the number of total triggers up to that point.

An example of writing a new value to the `pre-conf` register is given below.

```
trdbox reg-write 0x100 0x12345678
```

Upon running `trdbox status` after performing this command[1], one should see

```
pre_conf   [0x100]: 0x12345678 = 305419896
pre_dgg    [0x101]: 0x10121012 = 269619218
pre_cnt    [0x102]: 0x0000a7da = 42970
pre_stat   [0x103]: 0x00000001 = 1
dis_conf   [0x28d]: 0x0000000c = 12
dis_freq0  [0x280]: 0x000000d6 = 214
dis_freq1  [0x281]: 0x00000052 = 82
dis_time0  [0x284]: 0x1e440001 = 507772929
dis_time1  [0x285]: 0xcda00008 = 3449815048
```

confirming that the command successfully changed the desired setting.

To make new commands accessible from the command line, the recipe is fairly straightforward. An example for making a dedicated command to write a user-supplied value to one register is given below.

```
@trdbox.command()
@click.argument('argument', callback=lambda c,p,x: int(x,0))
@click.pass_context
def example(ctx, argument):
    ctx.obj.exec(f"write {register address} {argument}")
```

Of course, the function contents itself can have more information than simply writing to one register. For instance, the user could supply four different values instead of just one, with the function's job being to concatenate these into one eight digit hexadecimal number that can then be written to a specific register. Bitwise operators come extremely handy when making new functions due to the requirement of the argument values being hexadecimal. See section 10.2.1 for more information about this, and also look at functions such as `dis_thr` in (for the case of the Git repository specific to this project) `ALICE-Project/src/dcs/trdbox.py` to see these operators implemented [14]. As a last note, functions within `trdbox.py` require underscores, however when used in the command line these are replaced with hyphens, i.e. '-'.

10.4 Analysis Guide

This section details the flow of the analysis for the different sections. The options and switches for each script used in the analysis section are detailed in Table 10.1.

10.4.1 Pre-processing of the data

Only one script is needed for pre-processing of the data which is `raw2npy.py`. This script is straight forward to use and has multiple options (detailed in Table 10.1) which can be used to generate different processed sets of data based on what it will be used for (looking for broken region of detector, use in path reconstruction, etc.).

10.4.2 Background noise

In this section, the script `background_noise.py` is predominantly used. There are two workflows which are commonly used in this section:

1. **Finding broken regions of the detector:** Run `background_noise.py` on nonzero suppressed data (see Table 10.1). This will output a noise plot and the areas with very high average ADC count correspond to broken areas of the detector which must be zero suppressed.

2. **Finding background:** Run `background_noise.py` on zero suppressed data (see Table 10.1) to see a noise plot, and to find the average background ADC count.

[1] Note: the actual setting supplied here to `pre-conf` is just an example; if you tried to set `pre-conf` to this value you would never get any data taken from the TRD chamber.

10.4.3 Path reconstruction

The basic flow of the analysis for section 6.6 is described here. The basic workflow is:

1. Get zero suppressed data from `background_noise.py`
2. Pass that file to `roi.py`, experiment with different values for `THRESHOLD` to see what produces a reasonable amount of regions. Ideally this value should be above 1000, but this may not be feasible depending on the data.
3. Pass both the raw data Numpy file and the regions Numpy file to `path_reconstruct.py` to generate the distribution of incidence angles. The `--plot` switch can be set to a region number to plot the ADC values (3d path) and the line of regression for the particle path.

10.4.4 Analysis scripts

The following table can be used as a reference guide for the analysis scripts.

Table 10.1: Analysis cheat sheet. Description of the options used for the scripts in the data Analysis section. Bold entries are required options, and *output* is the file and format that the script outputs.

Script name	Argument	Var. Type	Description
`raw2npy.py`	**filename**	Path	The TRD raw data file (`.o32`) to process
	filename_out	Path	The name of the `.npy` file to produce
	`--n/--nevents`	Integer	Limit the number of events to process from the file
	`--s/--suppress`	Boolean	Zero suppress the region of the detector which is broken
	`--p/--progress`	Integer	print event number every `p` events
	`DATA_EXCLUDE_MASK`	Constant Numpy Arr.	The regions of the detector which are broken. Same shape as *output*
	output	Numpy array file	Outputs a Numpy array file with dimensions “event" × “pad row" × “pad col." × “time bin"
`background_noise.py`	**data**	Path	The processed, zero-supressed .npy file (output from `raw2npy.py`)
`roi.py`	**filename**	Path	The processed, zero-supressed .npy file (output from `raw2npy.py`)
	out_file	Path	The output .npy file to save the regions to
	`-n/--nevents`	Integer	The processed, zero-supressed .npy file (output from `raw2npy.py`)
	output	Numpy array file	Outputs a Numpy array file with dimensions “event" × “row" × “col. start" × “col. end" × “time bin"
`path_reconstruct.py`	**data**	Path	The raw data numpy array file (`.npy`) file generated by `raw2npy.py`
	regions	Path	The regions numpy array file generated by `roi.py`
	`--plot`	Nonnegative Integer	Region number (zero indexed) to plot
	`BACKGROUND`	Constant float	Backround ADC count, found with `background_noise.py`

11 CORSIKA User Guide

11.1 Installation and Setup

This section is designed to assist in understanding the User Guide supplied with the CORSIKA software, where details and explanations may not be clear. The goal is to provide a basic orientation whereafter the user can then investigate the software and its possibilities themselves.

It is advised a UNIX or Linux system be used to run CORSIKA. However, it is highly recommended Windows is avoided due to many compatibility issues, most prominently with attempting to install a Fortran compiler (Fortran is the language that CORSIKA is written in). To obtain the software visit www.iap.kit.edu/corsika/79.php and follow the instructions. Once downloaded and extracted. One can set up the software by executing

```
./coconut
```

via the terminal in the primary CORSIKA directory (Note that './' is used to run programs from the terminal). From there various setup options will be presented which govern various modalities and outputs of the simulation. It is advised initially to keep things to the default setting in order to get a feel for how the software works. Running ./coconut again at any time will allow one to adjust one's set up parameters. Further details on the options can be found in Chapter 3 of the User Guide pdf.

11.2 Steering Keywords

Steering is a term used to refer to the parameterisation and setup of the cosmic ray showers to be simulated. Chapter 4 goes into detail about the various steering keywords and their effects. There are several notable keywords worth mentioning. The supplied input files in the run folder can be viewed for formatting examples, what is important is that at least two spaces be left between each keyword and subsequent arguments:

- RUNNR: Run number is the run ID which is used to name the DAT file. Note that CORSIKA will not run if a DAT file already exists with the input run number. Hence repeated runs will the same run number require that the corresponding DAT file be deleted.
- NSHOW: Number of showers to simulate with the input parameters. Higher energy showers can take exponentially longer, and so in retrieving simulation data it is necessary to consider this and correspondingly adjust NSHOW to adapt to time constraints.
- PRMPAR: Primary particle of the cosmic shower, the initial cosmic ray which enters the atmosphere. A list of the particles and their ID's can be found in Chapter 8. The particles of interest are protons (14) and muons (5).
- ERANGE: Provide the upper and lower bounds for the primary particle energy. Energies are selected at random between this range. If upper = lower, then the energy is fixed.
- ESLOPE: Set exponent of differential primary energy spectrum from which random primary particle energies will be sampled. Defualt value is -2.7.
- THETAP: Provide the upper and lower bound for the zenith angle of the primary particle. CORSIKA randomly samples angles from a probability distribution which respects particle fluxes from all solid angle elements of the sky and a registration by a horizontal flat detector arrangement.
- PHIP: Provide the upper and lower bound for the azimuthal angles, selected over uniform distribution due to assumed azimuthal symmetry.
- THIN: Worth investigating when requiring to run larger simulations. Requires to be initialised as an option in setup. Creates various data-thinning conditions which accelerate the simulation process, yet using statistical algorithms provides nearly identical results.

11.3 Running software

Once the software has been installed the terminal can be opened in the run directory and the corsika/ program run with arguments for input keyword file and output file:

```
./corsika77410Linux_EPOS_urqmd <input_file> output.txt
```

The input file should match the options selected in the setup. For example, this is not necessarily clear fortunately examples are provided in the run directory of input files corresponding to various setup options. For example if EPOS (the default) is selected as the particle interaction model, then the necessary keywords for utilising this model must be called in the steering card (keyword file). The output file receives only certain relevant data regarding the simulation. However, the important simulation results are sent to a file named DATnnnnnn, where the last six characters describe the run number (or run ID).

11.4 Additional Details

Coast libraries offer a more advanced set of analysis tools. There is the Coast directory within the main folder which contains a 3D imaging tool that may be of use. The README's in the respective coast directories provide details for the installation and use. In the `src/utils/coast/CorsikaRead` directory one can run the makefile by simply calling

```
make
```

from the terminal in order to compile CoriskaReader and CorsikaPlotter tools - note that the latter requires having ROOT installed. CorsikaReader is used to extract the binary data from a DATnnnnnn file and can be output to text file for analysis using

```
./CorsikaReader DAT000001 > output.txt
```

CorsikaPlotter uses the ROOT interface to generate plots of the data contained in the DATnnnnnn file. This is perhaps of limited use but may provide insights into the structuring of the data. First enter into terminal

```
./CorsikaPlotter DATnnnnnn
```

which will produce a .root file. This file should then be entered as a root argument

```
root DATnnnnnn_1.root
```

which can then be opened from ROOT with the command

```
.x fileopen.C
```

to navigate the ROOT browser in which various plots are stored.

References

[1] Home.cern. 2021. Home | CERN. [online] Available at: <https://home.cern/> [Accessed 23 August 2021].

[2] Alice-collaboration.web.cern.ch. 2021. ALICE | ALICE Collaboration. [online] Available at: <https://alice-collaboration.web.cern.ch> [Accessed 9 October 2021].

[3] 2001. Technical Design Report of the Transition Radiation Detector. 2nd ed. CERN: CERN.

[4] Wulff, E. S., 2009. Position resolution and Zero Suppression of the ALICE TRD. Diplomarbeit. Westfälische Wilhelms-Universität Münster.

[5] Bjerring, T., 2021. Position Resolution of the ALICE Transition Radiation Detector. Cape Town.

[6] Ginzburg, V. and Frank, I., 1945. Radiation of an uniformly moving electron due to its transition from one medium to another. Journal of Physics, Moscow, (9(5):353-362).

[7] Barreiros, A. Barrella, J. Batik, T. Bohra, J. Camroodien, A. Govender, S. Lees, R. Rughubar, R. Skosana, V. Tladi, M. Wilkinson, J. Zeeman, W. (2018). 'Detecting cosmic ray muons with an ALICE TRD Readout Chamber and commissioning a HiSPARC detector using electronic pulse processing'. UCT.

[8] Beetar, C. Blaauw, C. Catzel, R. Clayton, H. Davis, C. Diana, D. Geen, U. Grimbly, S. Johnson, D. Jackelman, T. McGregor, A. Moiloa, K. Oelgeschläger, T. Perin, R. Rudner, B. (2019), 'An Investigation Into the UCT-ALICE Transition Radiation Detector', UCT.

[9] HiSPARC Project. Available at https://www.hisparc.nl/en/ (2003)

[10] Knoll, G. F. (2010). *Radiation Detection and Measurement*. 4th ed. New York: Wiley.

[11] GitHub - mkidson/ALICE_LAB_2021: The programs used to handle the scintillation detectors for the ALICE TRD lab 2021. [online] Available at https://github.com/mkidson/ALICE_LAB_2021.

[12] GitHub - tdietel/alicetrd-python: DCS utilities for the ALICE TRD, implemented in Python. [online] Available at: https://github.com/tdietel/alicetrd-python.

[13] GitHub - OpenWave-GW/OpenWave-1KB: A simple python program that can get image or raw data from digital storage oscilloscope (GDS-1000B) via the USB port. [online] Available at https://github.com/OpenWave-GW/OpenWave-1KB.

[14] GitHub - TenilleLori/ALICE-Project: PHY3004W Group Project. [online] Available at: https://github.com/TenilleLori/ALICE-Project.

[15] Arlow, H. Faul, S. Nathanson, N. Passmore, J. Pillay, K. Ramcharan, C. Roussos, G. Scannel, O. Thring, A. Wagener, J. Zimper, S. (2020). 'Detecting and Tracking Cosmic Ray Muons using a Readout Chamber of the ALICE TRD'. UCT.

[16] Viljoen, J. "Detecting cosmic ray trajectories in the Southern Hemisphere, using an ALICE TRD".

[17] GitHub - ColeFaraday/AliceAnalysis: The programs used to to analyse data for the ALICE TRD lab 2021. [online] Available at https://github.com/ColeFaraday/AliceAnalysis.

[18] Liu, L. "The Speed and Lifetime of Cosmic Ray Muons". (2007)

[19] W. Blum, W. Riegler, and L. Rolandi, "Particle detection with drift chambers", 2. ed. Berlin Heidelberg: Springer, 2008.

[20] GitHub - CalleyRamcharan/ALICE2020Public: Some analysis programs were modelled from this repoy. [online] Available at: https://github.com/CalleyRamcharan/ALICE2020Public.

[21] 5. Coordinate Transforms (2021) Faraday.uwyo.edu. Available at: http://faraday.uwyo.edu/~admyers/ASTR5160/handouts/51605.pdf (Accessed: 1 November 2021).

[22] Grieder, P. K. F. Cosmic rays at earth: Researcher's reference, manual and data book. Elsevier, Amsterdam, 2001.