ALICE TRD Lab 2021

Tenille Bjerring et al.

3 November 2021

Abstract

I (Tenille) will write this at the end.

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 ultrarelativistic 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 the 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]

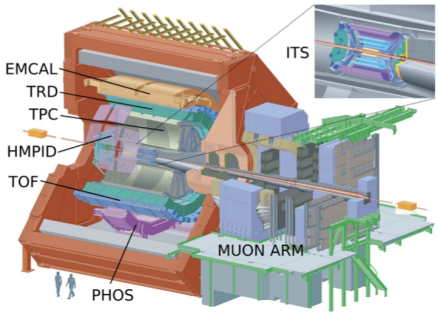

Figure 1.1: The ALICE detector consisting of: the ElectroMagnetic CALorimeter (EMCAL), the Transition Radiation Detector (TRD), the Time Projection Chamber (TPC), the High Momentum Particle Identification detector(HMPID), the Time of Flight detector (TOF), the Photon Spectrometer (PHOS), the Inner Tracking 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 aim of this project is to consolidate and document all operations necessary to run the TRD module and scintillation detectors, acquire data from both and then analyse this to identify the origins of the 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 8.

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, or simply a proton, which is the most abundant element in the universe. accounts for 89% of the cosmic rays incident to the earth's atmosphere. Then a further 9% consists of helium nuclei. The remaining fraction is composed of heavier element nuclei and electrons. For our purposes we only consider 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 themselves 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 also 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 this process lies in the production and subsequent detection of muons.

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 specifically fine tuned 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.

2.2 Detector Theory

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 ψ 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]

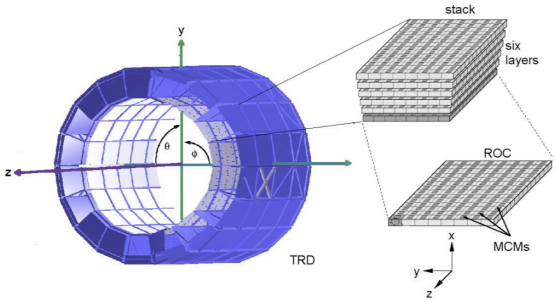

Figure 2.1: TRD barrel with 18 supermodules, showing a zoomed in view of one stack with 6 layers as well as one ROC

One module consists of a multi-wire proportional readout chamber filled with Argon 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. Transition radiation occurs when an ultrarelativistic particle crosses the interface between materials with differing dielectric constants. This effect is however not measureable in this experiment as the momenta of the cosmic rays are typically not sufficient for transition radiation to occur and the Argon does not have a high enough X-ray photoabsorption probability.

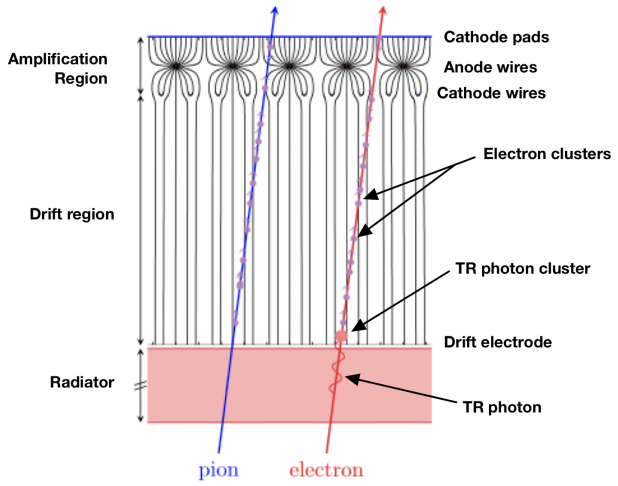

Figure 2.2: Cross section through one ROC with tracks of a pion and electron in order to illustrate the typical ionisation when using the Xe/CO_2 gas mixture as used at CERN. For the ROC used in this project, transition radiation is not measured. The ionisation is represented by so called 'clusters'.

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). Two scintillation detectors are used as a triggering device for the TRD module.

2.3 Scintillation Detector Theory

The two scintillation detectors used in this experiment were constructed in 2018 and 2019 respectively [8][9]. Their design is based on the detectors used in the HiSPARC project [7], 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 $0.5\,\mathrm{m}^2$.

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, but 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 PMT's can be found in [6] and details on the construction of the detectors can be found in [8] and [9].

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 subsection 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 subsection 3.1.

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. (Just waiting to take a nice picture of this, and then will replace it.)

3 Experimental Set-up

3.1 Scintillation Detectors

3.1.1 Set-up of Scintillation Detectors for TRD operation

The reasoning for choosing the set-up as we did is discussed further in subsection 6.5, but Figure 3.1 shows the set-up chosen. Ideally the two scintillation detectors would be at the same height as the TRD chamber, but due to space constraints of the room housing the TRD chamber, they had to be placed 30 cm below the TRD chamber. Both scintillation detectors were at the same height.

The two scintillation detectors sent their output signal to both the oscilloscope and the TRD box. Inside the TRD box, as described in subsection 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 chamber. The oscilloscope was set to hold the data when it received a trigger, so that the data could be collected.

In order to connect the scintillation detectors to the oscilloscope and TRD box, coaxial cables were used. These ran from the detector to a splitter which took the signal to the oscilloscope as well as the TRD box. The large diameter cables were used up to the splitter, and then the smaller, brown cables were used from the splitter to the TRD box. 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 chamber.

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.

3.1.2 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

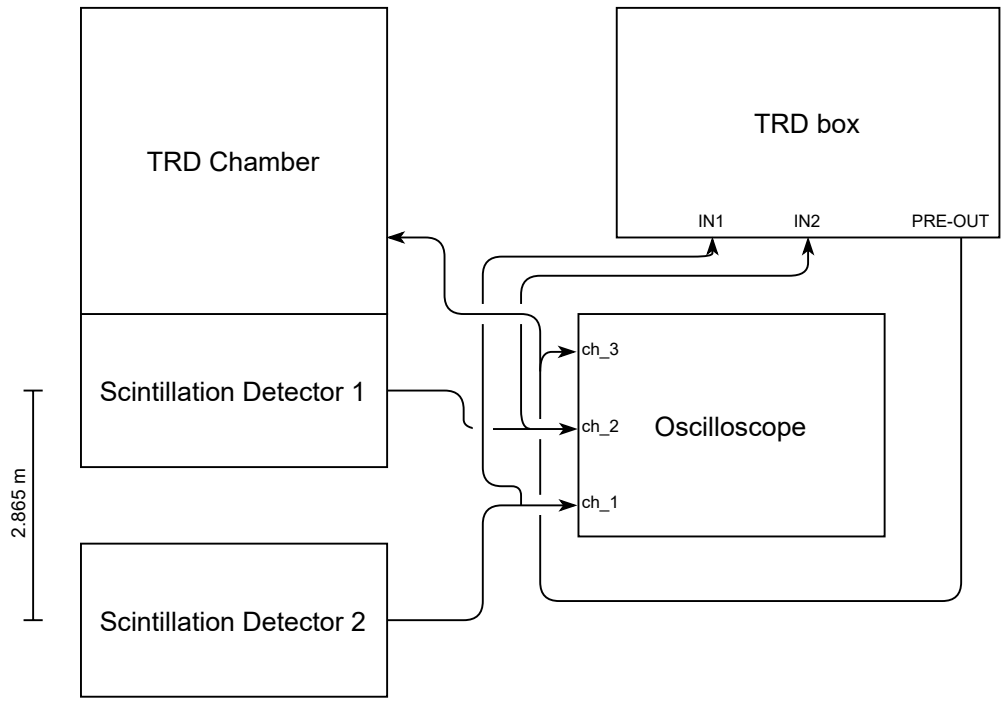

Figure 3.1: Simplified diagram of the set-up of the scintillation detectors with respect to the TRD chamber. Some details of the entire set-up have been left out, but this includes all the information relevant to the operation of the scintillation detectors as the triggers for the TRD chamber. Note that the TRD box output labelled "PRE-OUT" is the output that sends the trigger signal. Lengths are not to scale.

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.

4 TRD Operation

4.1 TRDBOX

4.1.1 TRDBOX Operation

The TRDBOX is the link between the TRD chamber, 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 chamber 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.

In the current setup, the TRDBOX takes in independent signals from both scintillation detectors Table 4.2, these signals are passed through the two discriminators in the TRDBOX. The discriminators act as a gateway, preventing signals which are too weak from being passed on to the TRD chamber. 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, considering the particle coincident if they fall within a user set time frame of one another.

The setting 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.

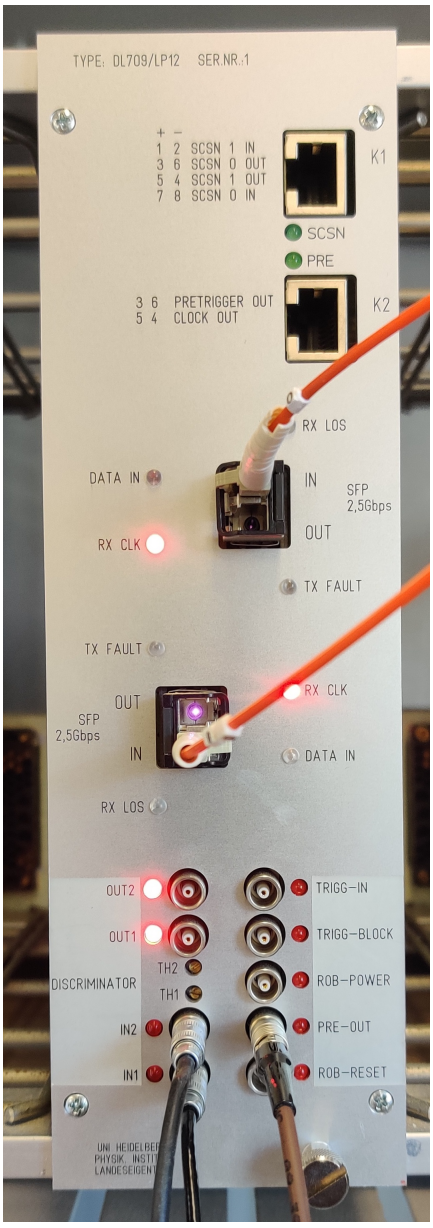

Figure 4.1: Image of the TRDBOX face, where the nine input pins are shown in the bottom of the image. OUT1 and OUT2 flash each time a signal passes into IN1 and IN2, respectively. The scintillation detectors are connected to IN1 and IN2. PRE-OUT is the pretrigger output, and it is connected to the TRD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

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 subsubsection 9.1.3), 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` (cite Tom's GitHub here). Once the `venv` is activated, then any TRDBOX command can be executed from anywhere else on the control computer. See subsubsection 9.1.4 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 chamber, as well as controlling the pretrigger (which is the signal that is actually sent to the TRD chamber). It also controls whether or not autoblock is on, which is where the TRDBOX will not send further trigger signals to the TRD chamber 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 chamber while the TRD chamber is actively taking data from an event.

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 chamber.
`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-pre-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 `ADDRESS` is a string name for the register, as displayed when when `STATUS` is run.
`reg-write [ADDRESS] [DATA]`	Writes `DATA` to the specified address where `ADDRESS` is written as described above.

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 [11] program was used. The version used can be found at [12], 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 [12]. 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 filepath 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 subsubsection 9.1.4), 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` in the linux command line after plugging the oscilloscope into the TRD computer via USB, then looking for something that looks like this:

```
[125714.945441] usb 3-8: Product: IDS-1074B
[125714.945443] usb 3-8: Manufacturer: RS
[125714.945444] usb 3-8: SerialNumber: 631D108G1
[125714.956492] cdc_acm 3-8:2.0: ttyACM2: USB ACM device
```

`IDS-1074B` is the oscilloscope, and `ttyAMC2` is the name needed. 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 line break 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 chamber, 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 filepath.

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 angular resolution measurements, the details of which are in subsection 6.3 and subsection 6.5. This was done using the programs in [12]. The second implementation is discussed in subsection 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 TRD-Box was designed. This was done to ensure more intuitive control over the TRD box, 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 be able to be 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.

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. 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.

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.

4.4 Services: High and Low Voltage and Gas Supply

Sam C needs to write this section

5 Data Acquisition

A new data acquisition system, based on the ZeroMQ messaging library, was constructed 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 [13].

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 TRD box: events may take place in all the detectors simultaneously as a result of cosmic ray showers, and to trigger events.

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 TRD box 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 unblock the hardware trigger and dump the event buffers before taking a new reading. A 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 (`ctx.obj.sfp0.send_string("read")` is used for the first sub-detector) - 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 in integer arrays in `subevent_t` objects, that also store information on the equipment type and ID used to take the measurements. The `subevent_t` objects are aggregated in `event_t` objects, that are passed to an `eventToFile()` method to store on disk.

The raw data is stored in a standard `.o32` format. 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 (two, for two subevents). 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. This format was necessary to decode the data with an o32 file reader (`o32reader.py`). 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

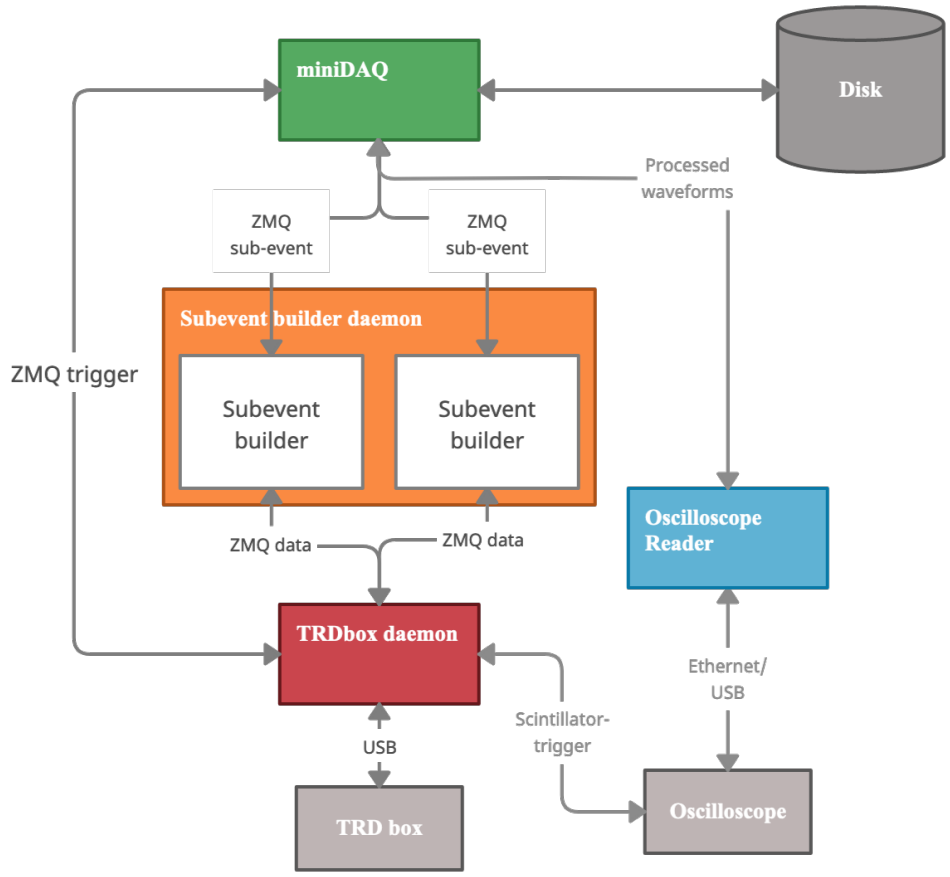

Figure 5.1: A diagram of the layout for the data acquisition system. The miniDAQ program initiates data collection runs, and stores the recorded ADC counts and oscilloscope waveforms to disk for each event. Data collection for an event may be initiated by a manual ZMQ trigger in the miniDAQ, or by a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

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 ADC counts with MCM headers, 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`.

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 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 [8] 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. Also important was the mode of the TRDbox when taking the background measurements - the chamber should be set to take non-zero-suppressed readings by means of the `nginject all 100` command.

Cosmic ray detection measurements utilise the scintillation detectors to detect events and send a trigger for recording data from the TRD. As discussed in subsection 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 subsection 4.2 and subsection 4.1.

The data runs for cosmic ray measurements were carried out in the `trigger_event()` method of the miniDAQ. `trigger_event()` 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()` was called. Upon completion of the `readevent()` method, the hardware trigger for the TRDbox was unblocked. The chamber 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 9.3 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 timeframe that each event was recorded. The implementation used the `oscilloscopeRead` package (see subsection 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.

The collection of the oscilloscope data was found to be a noticeable bottleneck for the speed of the `readevent()` method. This 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. This would involve implementing the `readRawDataFile()` method in the `dso1kb` module as well as modifying `getRawData` from the same module.

5.3.1 Extending o32 Files to Store Oscilloscope 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 Stalling

A persistent stalling issue was encountered when implementing data runs, 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.

5.5 Additional Functionality in the miniDAQ

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

6 Results and Analysis

6.1 Data pre-processing

The data that comes directly form 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 [10], the scripts `o32reader.py`, and `adcarray.py` were both used and were written by Jan-Albert Viljoen for his Honours project in 2016[]. The details of how these scripts work are not important for this report, and their functionality is implemented by a script `raw2npy.py`[], 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 [10] 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 TRD Background

As mentioned in subsection 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 in `https://bitbucket.org/uctalice/pytrd/`. Using the background readings and measuring the noise in units of counts per time bin, the mean and standard deviations

for each pixel 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 a the obtained background data.

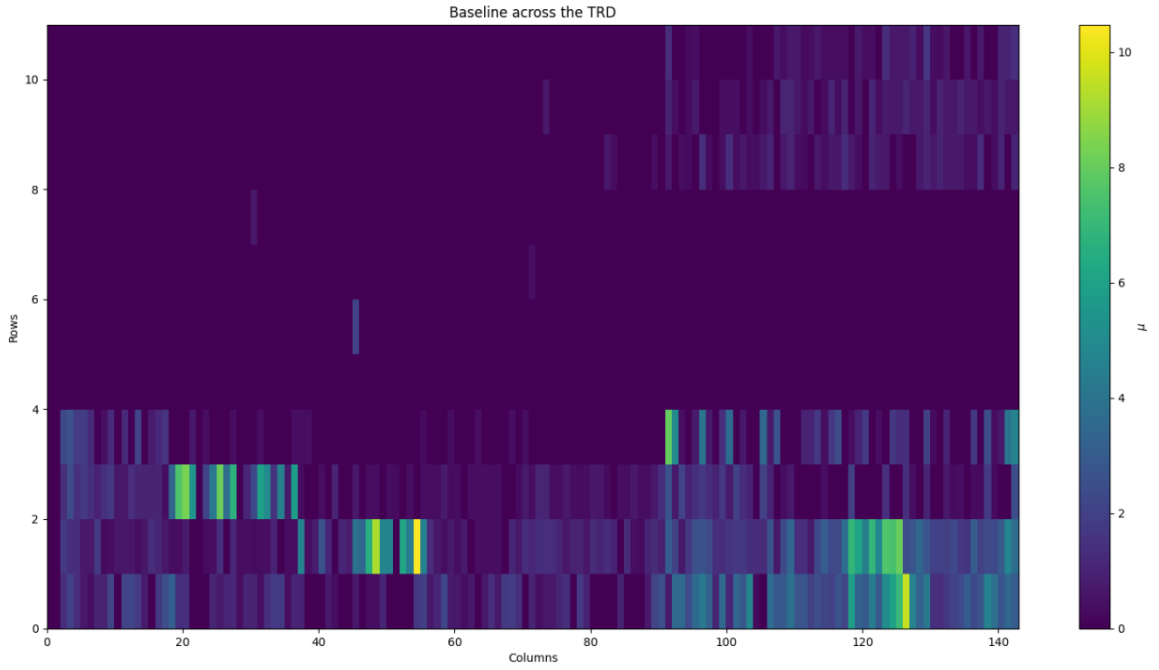

Figure 6.1: Caption

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 above 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 also the electronics and construction of the scintillation detector itself. The time resolution is an essential measurement to make 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 subsubsection 3.1.2.

6.3.1 Time Difference Between Two Oscilloscope events

To analyse the time between the two oscilloscope events we used the analysis described in [10] 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 6.2. 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

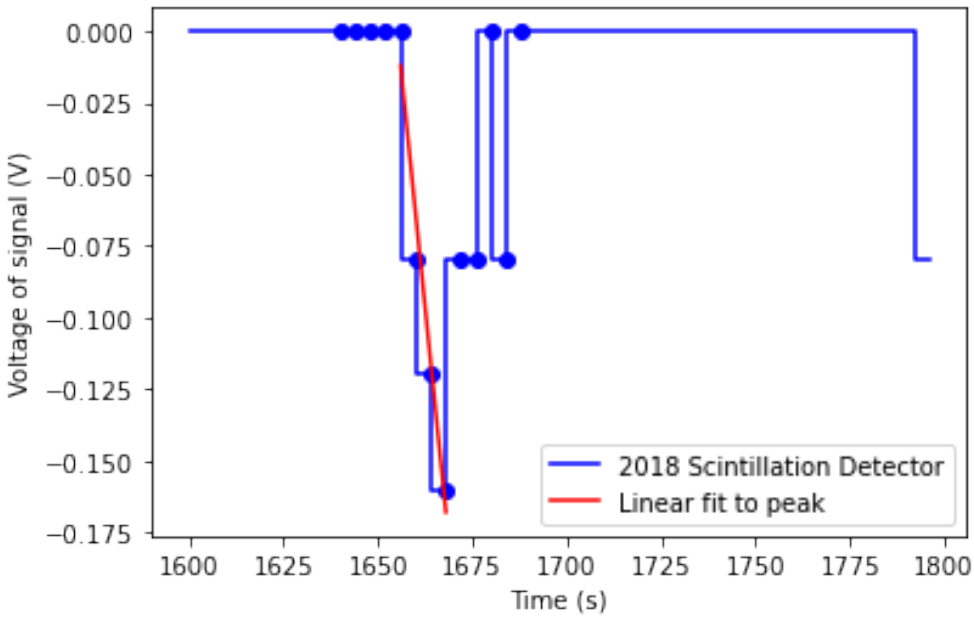

Figure 6.2: Pulse isolated from the 1000th data set obtained for the configuration with the 2018 scintillation detector on top. The linear fit is also plotted alongside the peak. Only the data from the 2019 detector is plotted to improve readability.[This plot is just a placeholder.I will be trying to create a more detailed one]

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 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 `filenamehere.py` and can be found in [13].

6.4 Optimising the Distance Between the Scintillation Detectors

6.4.1 Muon Shower Incidence Angle Determined by the Scintillation Detectors

Once the time resolution of the scintillation detectors had been accurately determined, the angular resolution of the system of the two scintillation detectors, when separated by a certain distance, could be found. A series of 4 runs 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 with the detectors at different separation distances.

The 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). With the further assumption that the two muons travel at exactly the same speed, taken to be $v = 0.994c$, a difference in the time of detection is expected between the two detectors, which depends on the angle of incidence. This is illustrated in Figure 6.3.

The time difference between the scintillation detectors for a single muon shower event was found, in each case, using the method outlined in the time resolution section (i.e. employing the `filenamehere.py` script, found in

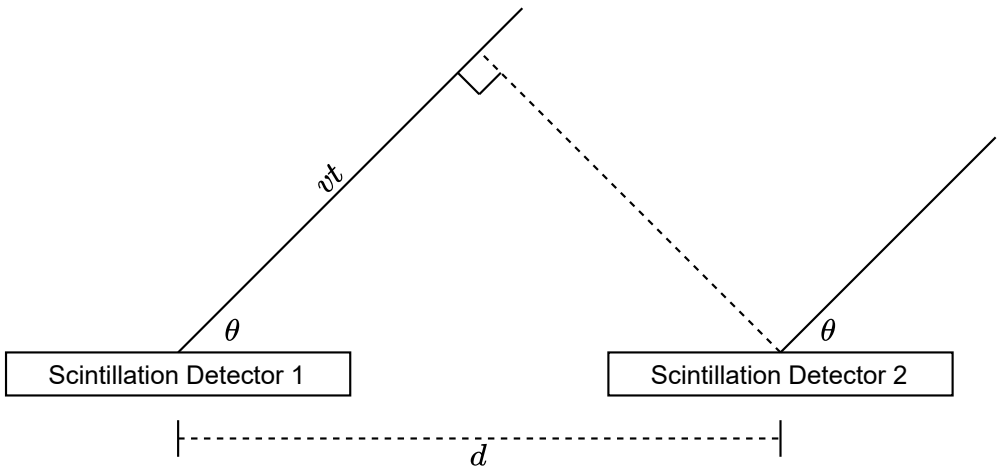

Figure 6.3: The geometry of the two-scintillation detector system (for a given separation distance, d), used to measure θ, the angle of incidence, in the plane depicted, of a muon shower. In the case where the angle was between 0 and 90 degrees, a muon was expected to arrive at the right-hand detector (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.

[13]). The uncertainty on this values was found via propagation, according to

$$u(t_1 - t_2) = \sqrt{[u(t_1)]^2 + [u(t_2)]^2} \tag{6.1}$$

where the values of $u(t_1)$ and $u(t_2)$ were both equal to the time resolution as determined in subsection 6.3.

The time difference, t between the detectors was used, by simple trigonometry, to determine the cosine incidence angle for each event, given the known separation distance, d, according to:

$$\cos\theta = \frac{vt}{d} \tag{6.2}$$

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

$$u(\cos\theta) = \sqrt{\left[\frac{v(u(t))}{vt}\right]^2 + \left[\frac{u(d)}{d}\right]^2} \tag{6.3}$$

The incidence angle, θ found from the cosine, had an associated uncertainty determined by

$$u(\theta) = \frac{u(\cos\theta)}{\sin\theta} \tag{6.4}$$

where the uncertainty on the distance between the scintillation detectors, $u(d) = 0.0003\,\mathrm{m}$, was found from the uncertainties on the positions of the midpoints as measured with a tape measure, via similar propagation to that for the time differences.

The process of determining the incidence angle and associated uncertainty, for each event, was achieved using the *link python code here*. Note that angles could not be found for certain events, in cases where the value of $\cos(\theta)$ exceeded 1. These data points were omitted when the angular resolution of the system at each distance was being determined. The table below shows the number of events captured during the run at the given separation distance, and the number of these events for which angles could not be determined.

Separation Distance (m)	Total Events Detected	Number of omitted events
2.082	100	40
2.865	100	51
3.625	100	39
7.500	50	26

Table 6.1: Number of detected versus omitted events for different separation distances. The difference gives the number of events used to find the angular resolution.

6.4.2 Angular Resolution of the Two-detector System and Optimisation Criteria for the Separation Distance

For each run at a given separation distance, the mean of the uncertainties on the incidence angle obtained for each event was taken to be the angular resolution of the two-detector setup at the given separation distance.

To determine the optimal separation distance between the scintillation detectors, from those used for each run, the coincident event rate and angular resolution was determined, for each distance. The angular resolution of the two-detector system was expected to increase with separation distance, but at further distances there would be a lower event rate (at further separation distances fewer coincident events are expected), which would not be practical for data acquisition. Hence, a trade-off between the event rate and the angular resolution had to be made, which constituted an optimisation of the two-detector system. The optimal separation distance of the scintillation detectors (and thus the ideal setup for the rest of the experiment), was determined at the point where a reasonable angular resolution had been achieved and the event rate was still high enough to be practical. It was determined that the optimal separation distance between the two scintillation detectors, was at a distance of 2.865 m. The event rate and angular resolution for the other separation distances are shown in the table below.

Separation Distance (m)	Event Rate (s^{-1})	Angular Resolution (°)
2.082	0.140	2.569
2.865	0.089	2.146
3.625	0.067	1.851
7.500	0.031	1.263

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

6.5 Angular Resolution of the TRD

TRD and scintillation detector data had been obtained for a relative orientation of the TRD and scintillation detectors, which allowed both to measure the same angle. The difference in the angle, for each muon shower event, as measured by the TRD and by the scintillation detectors, was then found. This difference was assumed to follow a normal distribution, and so the uncertainty on the angular difference was taken to be the standard deviation of the angular difference data. *insert gaussian plot of angular difference and quote uncertainty obtained*

Using this uncertainty and that previously determined for the angular resolution of the scintillation detector system at the optimal separation distance (i.e. the angular resolution of the two-detector system) a value for the angular resolution of the TRD, could be found via uncertainty propagation:

$$u(\theta_{\text{TRD}}) = \sqrt{[u(\theta_{\text{TRD}} - \theta_{\text{Scintillators}})]^2 - [u(\theta_{\text{Scintillators}})]^2} \tag{6.5}$$

Quote result of uncertainty (i.e. angular resolution) of the TRD

6.6 Path reconstruction

6.6.1 Background theory

In this section data from the TRD will be used to reconstruct the path of particles passing through the detector. The TRD can only precisely reconstruct the angle of the incoming particle along the column dimension as the column width is much smaller than the row width, meaning that particles usually stay in one row of the TRD. 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. The depth value z can be reconstruction from the time dimension since the electron drift velocity is constant.

We can actually reconstruct the position of a hit to within a single pad using methods described in Wulff's thesis[4]. 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 6.6[14] can be used to describe *position resolution function* which is defined for every pad as $Q_{\text{pad}}/Q_{\text{tot}}$ and can be parametrised as a Gaussian distribution as shown.

$$\text{PRF}(y) = \frac{Q_{\text{pad}}}{Q_{\text{tot}}} = \frac{Q_i}{Q_{i-1} + Q_i + Q_{i+1}} = Ae^{-\frac{y^2}{2\sigma^2}} \tag{6.6}$$

Applying Equation 6.6 to a set of adjacent pads leads us to Equation 6.7 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 $W = 7.55$ mm (TODO: Cite theory sec.) is the width of the pad (in the column direction). Importantly, this equation for the position depends only on known quantities and the standard deviation σ is no longer present in the equation.

$$y = \frac{W}{2} \frac{\ln(Q_{i+1}/Q_{i-1})}{\ln\left(Q_i^2/Q_{i-1}Q_{i+1}\right)} \tag{6.7}$$

6.6.2 Implementation of path reconstruction

The zero-suppressed, pre-processed data output by `raw2npy.py` is used as an input to `roi.py` both of which were adapted from CALLEY'S GITHUB (TODO: Cite). 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. Regions of interest of length one are discarded since they cannot be used to reconstruct a path.

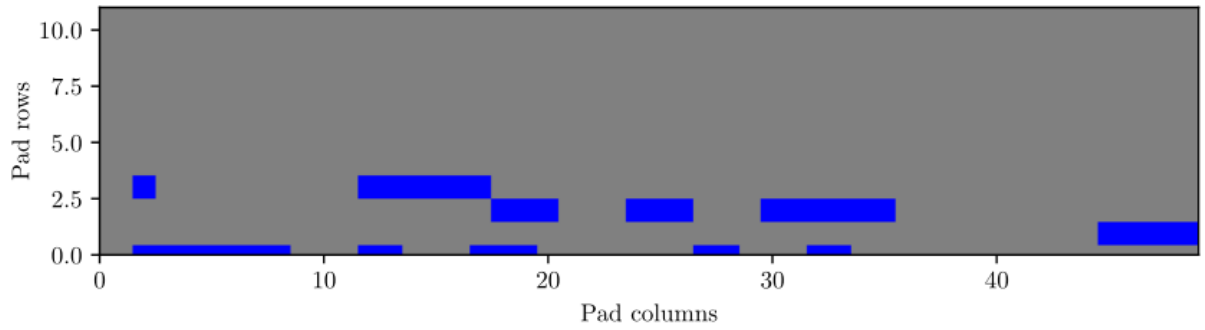

Figure 6.4: Illustration of the regions produced by `roi.py` for a specific event (event 2 from dataset FILL IN). The blue shaded pixels correspond to regions, different regions are regions which are separated by at least one column or are in different rows as previously described. (TODO: this is just the rough figure, I will make it look prettier later)

The script `path_reconstruct.py` takes in the regions of interest array from `roi.py` and uses Equation 6.7 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 $x-z$ plane).

TODO: Full section on path reconstruction

6.7 Simulation

6.7.1 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 6.5 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.

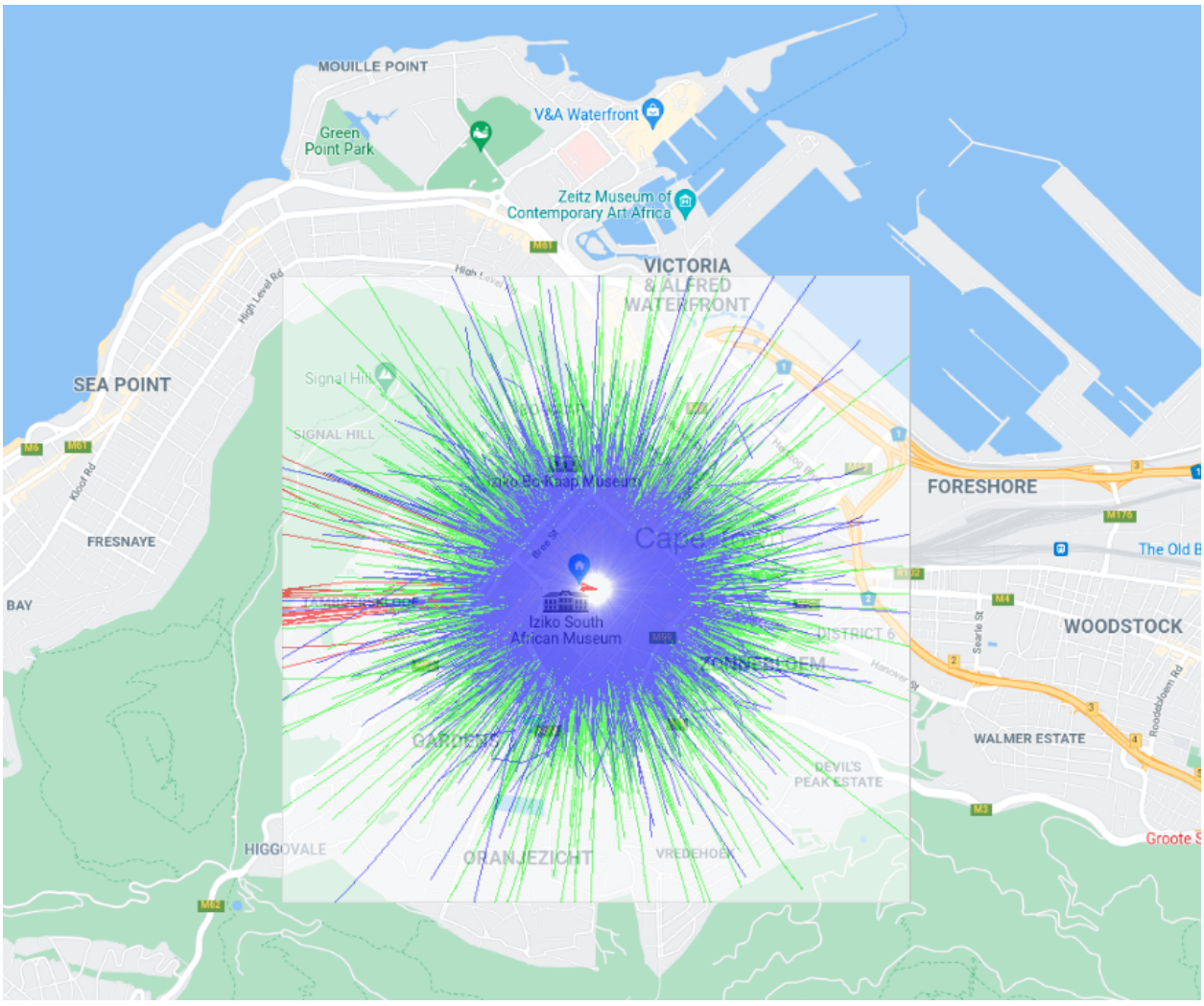

Figure 6.5: Comparison between City of Cape Town and 5 km wide cosmic ray shower. Hadrons are in blue and muons in green.

6.7.2 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 $1\,\mathrm{m}^2$ 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.

6.7.3 Relative Angular Intensity

A useful result from the simulation would be to determine the relative ratio of intensity between varying angles of incidence. However, several assumptions are required. Firstly, we assume that the distributions of energies and trajectories of the cosmic rays are isotropic throughout space[]. This leaves only the atmosphere's interaction with the cosmic rays to be the only free parameter to investigate. We thus iterate through several energies and polar angles.

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 6.6. A linear fit was performed for simplicity due to time restrictions, however a more rigourous analysis of the behavioural model is available[].

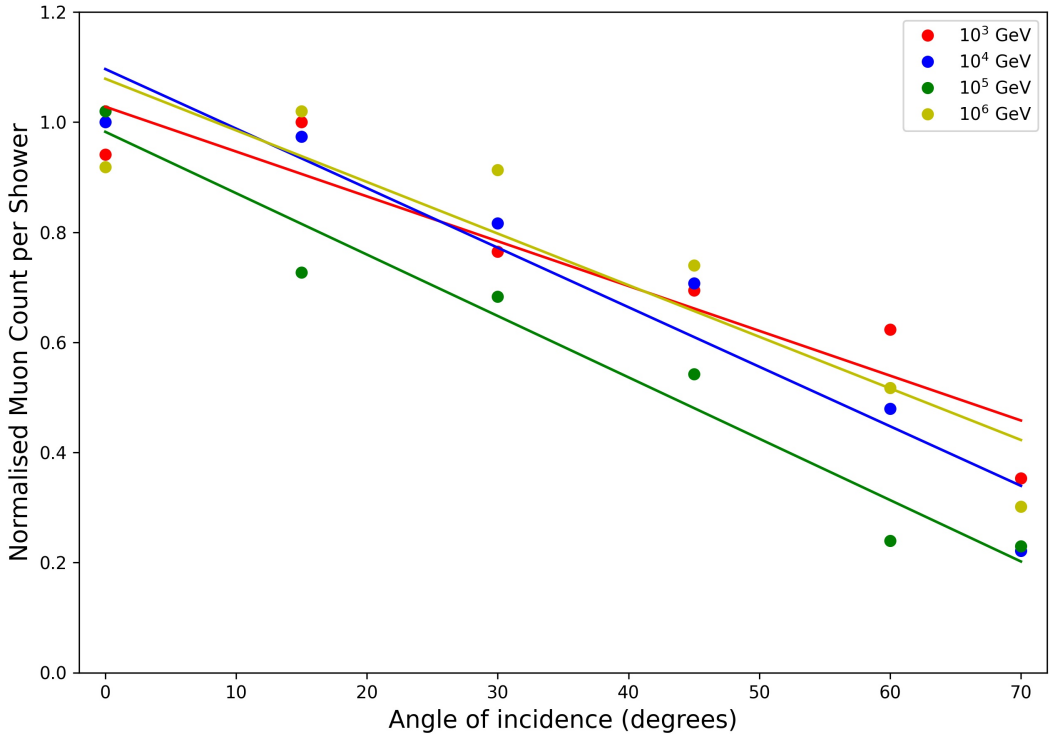

Figure 6.6: Plot of muon intensity at various angles of incidence for several energies.

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. Due to our assumption of isotropy this relationship can be taken to hold for all energies, thus offering an insight into total relative intensity based on the incident ray's polar angle. The mean gradient is found to be $m = 0.0095\ \ \theta^{-1}$. Thus given a certain polar angle θ with intensity I, we can find the relative intensity I_{rel}, at angle θ_2 by the equation:

$$N_2 = m(\theta_1 + \theta_2) + N_1 \tag{6.8}$$

It is important to note that the relationships in Fig. 8 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. This can be taken into account by considering a solid angle integration (...). The data produced in this section provides substance for comparison against experimental measurements.

7 Conclusion

8 Project Roles

9 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.

9.1 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.

9.1.1 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.

9.1.2 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. To switch to edit mode, type ‘i’. You can navigate with arrow keys (Shift+G and G+G go to the end/beginning of the file respectively). You should then be able to write your code. Note that many of the python files use 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.

9.1.3 `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.

9.1.4 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.

9.1.5 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

9.1.6 `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

9.2 Project-Specific Skills

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

9.2.1 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.

9.2.2 `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.

9.2.3 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
```

9.3 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.

9.3.1 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
```

3. The actual data acquisition thread, minidaq. Before running this, ensure you have activated the venv, as in section 9.1.4. 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:

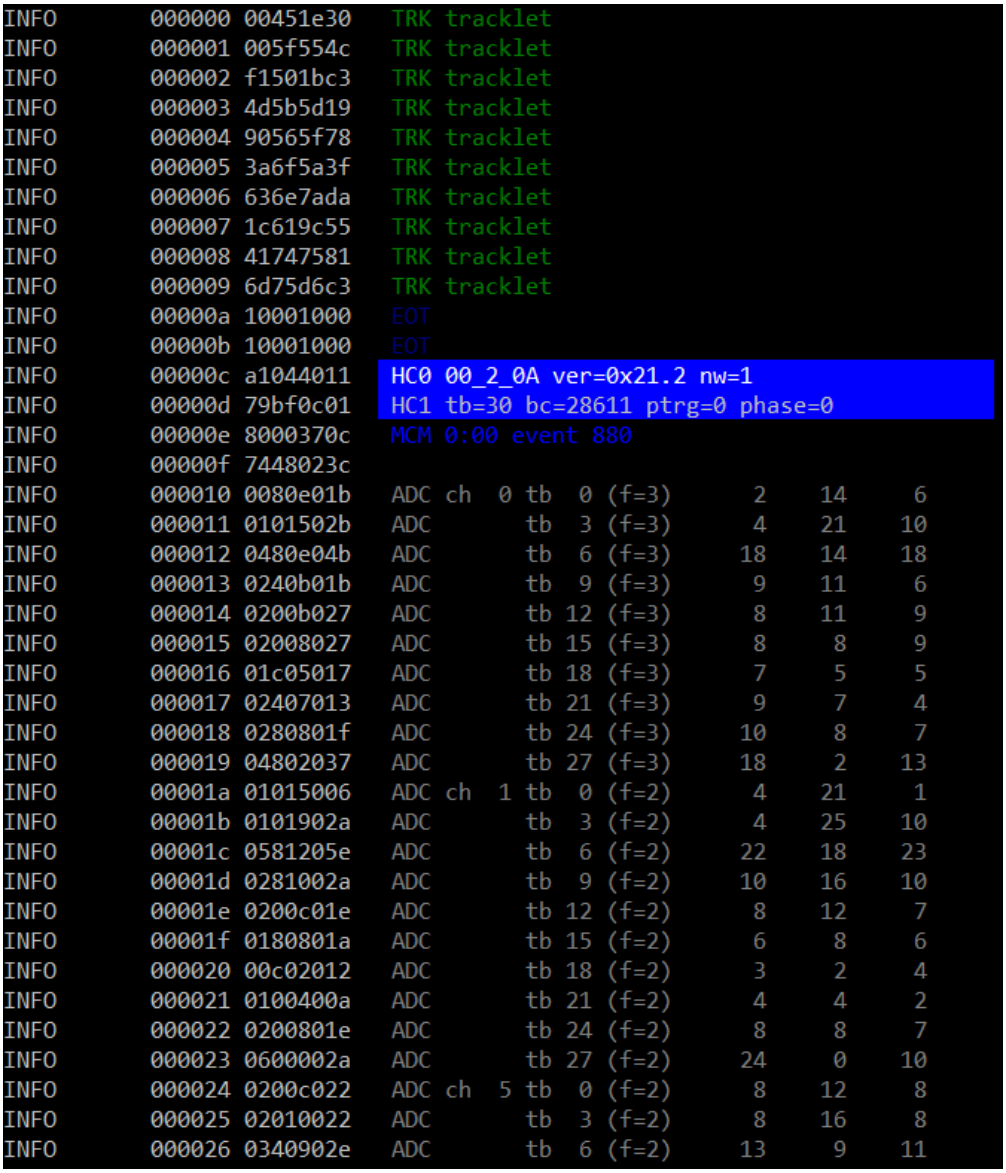

Figure 9.1: This is the data that is output by the `evdump` command. What you don't want to see is many red lines saying '`SKP ... skip parsing ...`'. This probably means the trd has been configured with `100` instead of `101`.

To better see this, run:

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

9.3.2 `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 9.2.

Figure 9.2: 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 9.3.

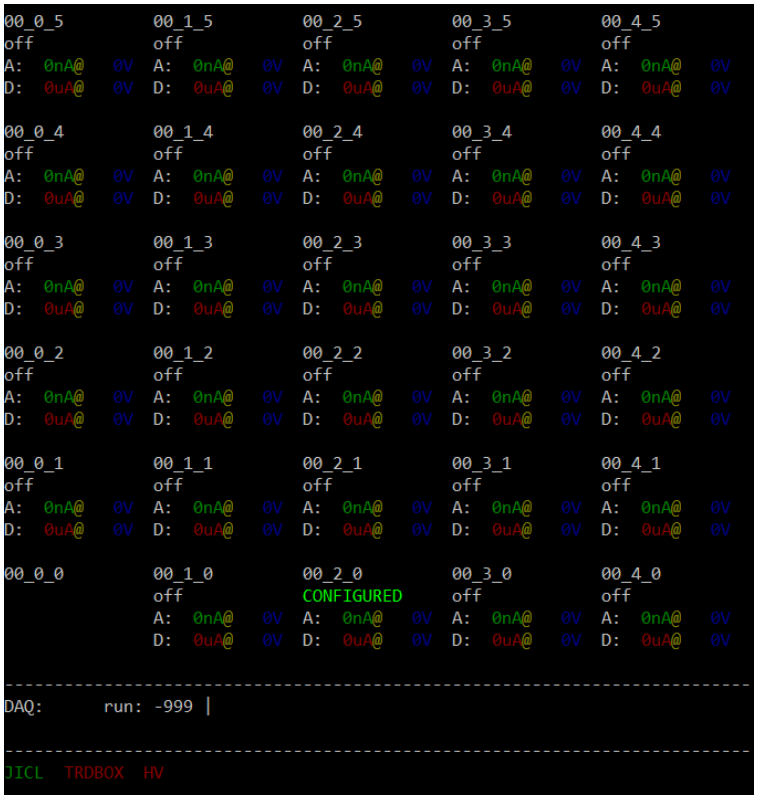

Figure 9.3: This is the display that comes up after the `smmon` command is run. As with `trdmon`, this green 'CONFIGURED' state is what you will usually seek.

9.3.3 `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 CORSIKA User Guide

10.1 Installation and Setup

It is advised a linux system be used to run CORSIKA. MacOS may work as well. However, it is highly recommended Windows is avoided. To obtain the software, visit [site] 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. From there various options will be presented. 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.

10.2 Options

10.3 Steering Keywords

10.4 Running software

10.5 Additional Details

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] 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).

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

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