TWaveform User Manual

University of California, Santa Barbara
High Energy Physics Group
David Stuart Group
Carlos A. Osorio[*]
2017-01-25
Version 1.0

Abstract

This document describes the capabilities of the `TWaveform` utility library for `C++` . `TWaveform` serves as a powerful tool for fitting and analyzing waveforms. It is able to recognize different types of pulses within a waveform, and can help to uncover properties of a certain type of signal. In this text we will unearth some of the applications for this class, as well as how to use it effectively. `TWaveform` works primarily through the data analysis framework, ROOT v.6.06/04.

Contents

[*] carlososoriov@hotmail.com

1 Introduction

1.1 TWaveform Description

The `TWaveform` package contains a C++ class focused on analyzing waveform signals. It was designed to be used through ROOT, CERN's data analysis framework. Within the `TWaveform` tool set there are also a number of non-member functions that were designed to broaden the class's functionality. Listed below are some of the processes the `TWaveform` class, in conjunction with its non-member functions, is capable of doing.

→ Chapter 2.2

1. Read and plot waveforms from CSV and txt files. It can also be adjusted to read data from other file formats.

→ Chapter 2.3

2. Find the integral and slope of a waveform. Also capable of finding the signal's standard deviation when there is noise.
3. Recognize waveforms that are pure noise.
4. Find the peak position and height of a single pulse. Also finds a pulse's rise position.
5. Compare waveforms to each other. It can determine if two waveforms have similar shapes by finding the lowest χ^2 possible between the two after scaling them and shifting their positions. This fit can easily be plotted using one of TWaveform's member functions.

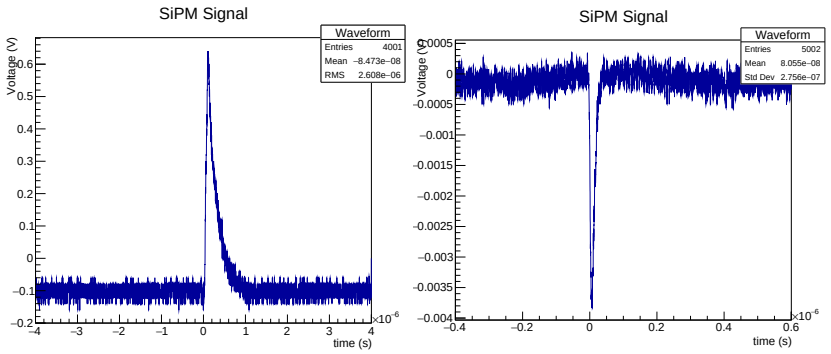
→ Chapter 2.4

6. Create templates for different shapes of pulses within a waveform set. It does so by analyzing a large number of waveforms and getting an average of the pulses with similar shape and/or size.
7. Find a waveform's best fitting template. It is also capable of fitting multiple templates to a single waveform. This allows TWaveform to do the following processes:
   - Recognize if a waveform has multiple pulses
   - Record the position and height of the different pulses' peaks.
   - Record the rise position of the separate pulses.
   - Find the relative strength of distinct pulses within a waveform.

→ Chapter 2.5

8. Create TTrees and TChains[1] with information on the properties of a large set of waveforms, such as:
   - Rise and peak positions.
   - Peak heights.
   - Best fitting templates.
   - Area under waveform.
   - Signal's standard deviation.
   - Uncertainty values.
9. Create and plot 1D and 2D histograms of the time distribution of pulses and their peak heights.

[1] ROOT objects that can store large blocks of structured data



(a) A plot of a single pulse waveform from the SiPM particle detector.

(b) A plot of a single pulse waveform from the PMT detector.

Figure 1: These plots show a visual representation of single pulse waveforms acquired by both particle detectors.

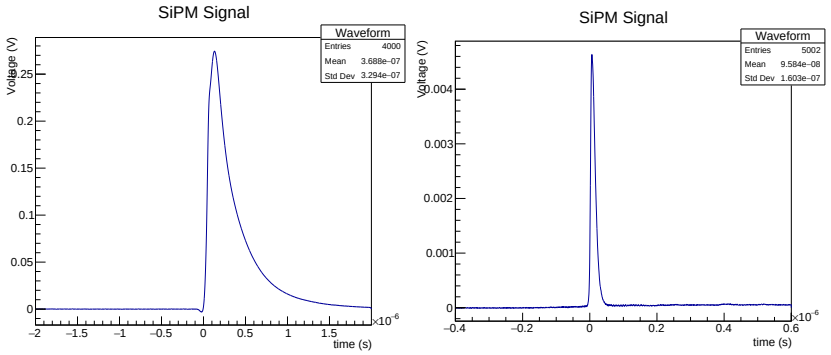
1.2 Overview

As you read through this user guide, you will find that the `TWaveform` utility library can be used for a number of different tasks, such as those described in the section above. It is worth noting, nonethelss, that this powerful tool was designed with a specific task in mind; analyzing and uncovering properties of large sets of waveforms. You may be asking yourself, what exactly is meant by a "waveform" in this context, and what properties do these "waveforms" need to have in order for `TWaveform` to work its magic on them? A waveform, as defined by various dictionaries, is just the graphic representation of the shape of a wave at a given time. While similar, this is not exactly what we mean by waveform in this context. Instead, a waveform for us will be any type of signal that has a variable with a defined value changing with respect to a second variable (usually time). An example of such is any type of electrical signal, whose voltage value changes with time.

Now, with regards to the second question, since `TWaveform` relies on a collection of template pulses in order to fit and analyze the waveforms, it will work best towards signals that are composed by discrete and defined types of pulses. That is, waveforms whose single pulses resemble a finite set of recurring shapes. If this is not clear enough, don't worry! After reading this section you will gain a better understanding of the types of waveforms that can be analyzed and what can be achieved with the `TWaveform` package at hand! As it is almost always the case, this task will be easier to accomplish with a series of examples.

For this purpose we will use two different sets of data. The first one of these, whose waveform representation can be seen in Fig.1a, is the data extracted from a a scintillating fiber and Silicon Photomultiplier (SiPM) based particle detector. Essentially, how this device works is; when an incoming particle hits the detector, a collection of photons are released. These are received by the SiPM, which in turn generates a voltage pulse correlated to the number of photons received, and whose data is later collected by an oscilloscope.

Knowing the details of how this detector works is not necessary for our discussion. However, it is worth noting that the signal extracted from this detector is composed of pulses that correspond to the number of photons



(a) A SiPM detector's waveform template.

(b) A PMT detector's waveform template.

Figure 2: These plots show some of the pulse templates generated from the waveforms of both particle detectors.

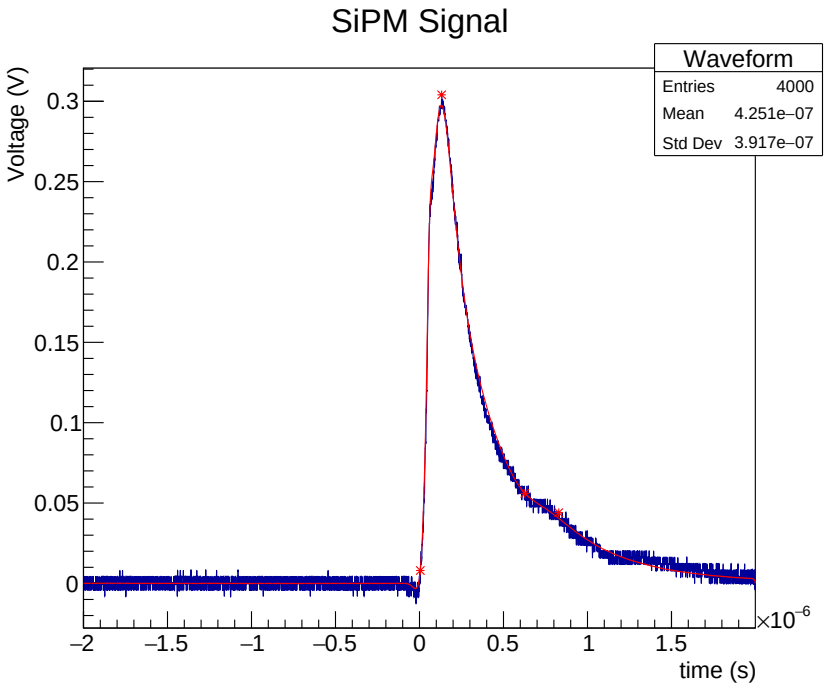
received by the SiPM. As such, the shape of the pulses vary very slightly, and noticeable changes between them are usually related to the height of the pulses.

With that being said, a total of 5,922 waveforms were collected from this detector. The first step in analyzing these waveforms is to create a set of pulse templates that can be used for fitting. To do this, and to retrieve the different shapes of the pulses contained in these waveforms, `TWaveform's` template generation methods can be applied.[2] These methods work on the basis that waveforms with comparable area beneath them will have a similar shape, and thus creates the templates according to their corresponding areas. A result of one of the templates generated through this process can be seen in Fig.2a. This template represents just one of the pulse shapes found, and generated, from the SiPM's waveforms.

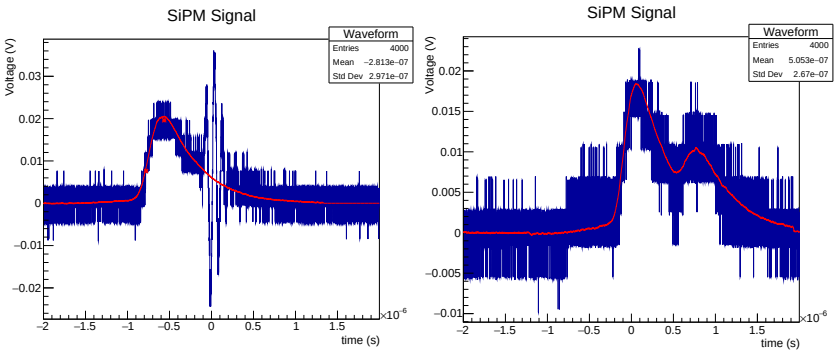
Now that we have the template pulses for the SiPM waveforms, we can proceed to analyze any of these waveforms with a very simple method called `AnalyzeWave()`. This function will fit the chosen waveform to the templates generated, and will then use that fit to determine the number and position of the pulses contained inside it. A few plots of some of these fitted templates are provided in Fig.3, where the blue curves represent the signal, and the red ones the best fit. Also, there are a number of red marks within the plots that represent each pulse's peak and rise position, as calculated by `TWaveform`. The first of these images, Fig.3a, displays the power contained inside of `TWaveform's` algorithms, as it is able to calculate in a precise way, the position of even a subtle pulse inside the waveform. The other two plots (Figs.3b and 3c) showcase how the fitting algorithm can perform even under waveforms with high levels of noise. Incidentally, information on what type of pulses are fitted, and their corresponding height and position, is saved into each of the `TWaveform` instances fitted.

It is also possible, with `TWaveform's` methods, to analyze in this way all the waveforms in a given directory. Information on all of these waves, such as the number, position and height of pulses, can be saved into a series of data set objects called TTrees.[3] These objects are saved into .root files, and can be quickly and easily accessed at any time.

[2] See Section 2.4 for more on these methods.
[3] For a detailed definition of TTrees and how to use them, go to Section.2.5



(a) A waveform with two pulses, one large and the other very subtle.



(b) A single pulse waveform with noise being fitted by `TWaveform's` methods.

(c) A double-pulse waveform with high noise fitted with `TWaveform's` tools.

Figure 3: These plots show some instances of the fitted waveforms from the SiPM detector. The blue curve represents the signal, and the red curve its best fit. The red marks represent the peak and rise position of each pulse.

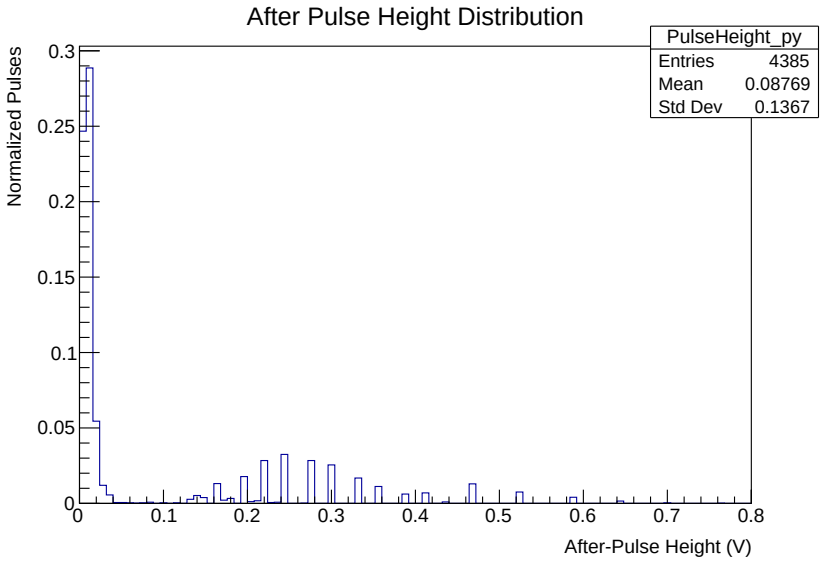


Figure 4: A histogram of the after-pulse height distribution of the SiPM waveforms analyzed. This histogram was built using `TWaveformSpace's` methods, and the data was acquired from a TChain of the SiPM waveforms.

`TWaveform` also includes a set of non-member functions that are capable of reading these data structures and creating 1D and 2D histograms of the data within it. Examples of such are shown in Figures 4, 5 and 7, where histograms were made for both the SiPM data, and PMT data that we will introduce later on. The first one of these plots (Fig.4) is a representation of the height distribution of after-pulses found in the SiPM waveforms. By after-pulses we of course mean the second pulse to appear and be fitted in a waveform. By taking a glance at this graph, we can recognize there is a significant surge of after-pulses with a height of around 10mV.

The substantial amount of after-pulses with this height (almost 30% of the total after-pulses) tell us that either a recurring physical process is happening inside the SiPM detector that creates after-pulses with this height, or that `TWaveform` is actually over-fitting the waveforms, and we might need to adjust the parameters with which the fits are being made.

To gain a better understanding of the effects behind this spike in pulse heights, we might want to look at some of the waveforms creating this surge, or otherwise, build another histogram to analyze the data further. We will carry on to do the latter, and build a histogram that compares the after-pulses' height with their time distribution (calculated as the difference in time between the first and second pulse), displayed in Fig.5.

Before we move on to analyze this new histogram, notice that there seems to be another apparently significant rise of after-pulses, centered at a height of around 250mV. The effect accounting for this group of pulses seems to be very different in nature to the first spike we encountered.

Lets carry on and evaluate what Figure 5 can tell us about these two events. First off, we can see that the huge surge of after-pulses with a 10 mV height have a seemingly equal distribution in time, for the exception of a large spike at about 30ns. The rest of the distribution with this

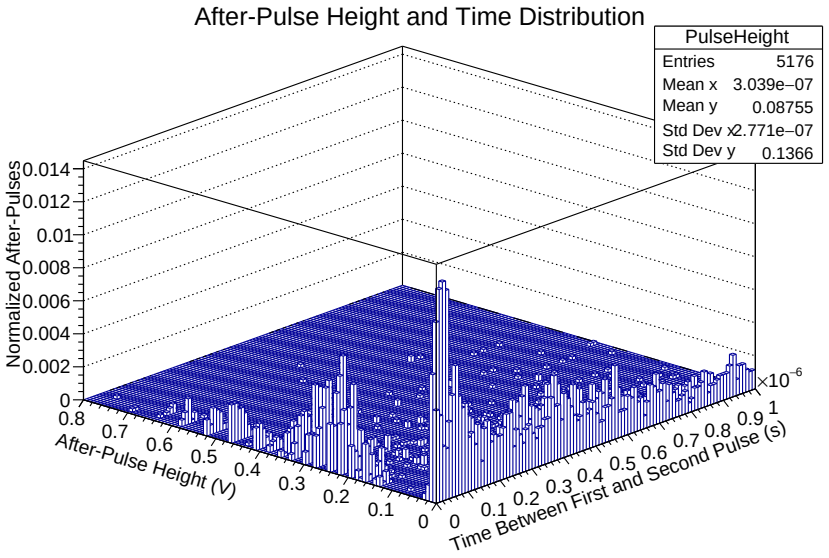


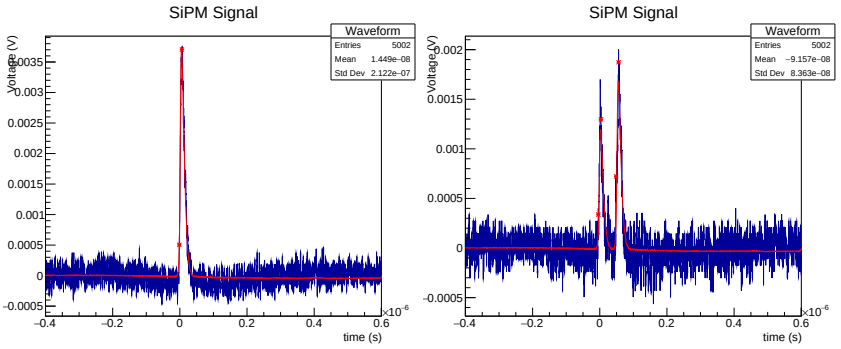
Figure 5: A 2D histogram of the after-pulse height and time distribution for the SiPM waveforms.

height has a few oscillations here and there, but nothing very significant, to say the least. From this data we can therefore infer that most of the after-pulses recorded with a 10mV height are probably not the product of any specific event related to the first pulse. These may, instead, be the product of other different phenomena, such as gamma rays hitting the detector and creating a pulse, or, as we mentioned earlier, `TWaveform` over-fitting some of the waveforms with imaginary pulses.

It is also evident from this plot that the 250mV after-pulses are caused by a different phenomenon than the first. After closer inspection, we can witness that these pulses follow a somewhat apparent Gaussian distribution, centered at 45ns. This centered distribution shows evidence that these collection of after-pulses might be related, in some nature, to their waveform's first pulse. In order to study the relationship between these pulses, we could draw another histogram that would compare the height of the first and second pulses. However, we will not delve into this rabbit hole, and now that you have seen what can be accomplished with `TWaveform`, will stop the SiPM's waveform analysis here.

The next set of waveforms we are going to briefly analyze come from a similar particle detector as the SiPM one described above. The main difference between these two is that, instead of having a Silicon Photomultiplier receiving the incoming photons, a Photomultiplier Tube (PMT) is used. Since these two devices react to the photons slightly differently, the signal they create differ in shape and size. Another aspect in which these two data sets diverge from each other, is that the PMT detector's scintillating fiber's side, facing the PMT, was covered with a reflective material that had been pierced with a couple of very small holes. In effect, these allowed very few of the incoming photons to reach the PMT, and thus the signal received from this detector is composed mostly of one-photon pulses. One such pulse can be seen in Fig.1b.

For the PMT detector we were able to collect a number of 7659 waveforms. Following the previous procedure, we used these collection of waveforms to create a set of templates. One of these templates is shown



(a) A single-pulse waveform with its best fit

(b) A waveform with two pulses and their respective fits.

Figure 6: Plots of PMT waveforms being fitted by the generated templates. Once again, the blue curve represents the PMT's signal and the red one its best fit. The waveform's calculated peaks and rise positions are also denoted by red asterisks.

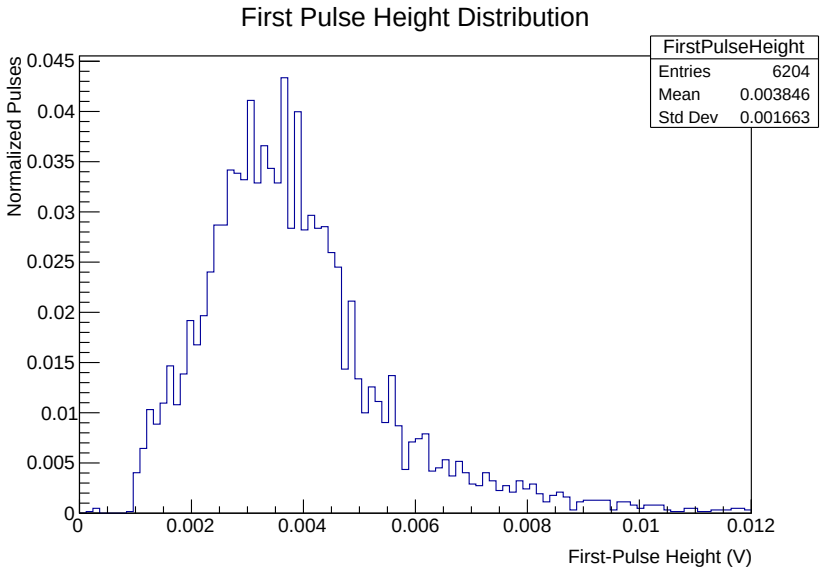


Figure 7: A plot of the height distribution for the PMT's waveforms' first pulse.

in Fig.2b. In this plot we can clearly see the difference between the shapes of the SiPM and PMT pulses. Note also how the PMT template has a positive signal, while its original waveforms (Fig.1b) have negative voltages. This is caused by a `TWaveform` method that reverses the sign of a waveform's signal if it is found to be negative.[4] As such, this demonstrates how `TWaveform` is capable of handling both negative and positive waveforms.

In Fig.6 you can see how the pulse templates generated are used to fit PMT's waveforms. As with the SiPM data, `TWaveform's AnalyzeWave()` method was capable of generating a good fit for both single and double pulse waveforms. We also proceeded to analyze all the waveforms in the PMT directory, and saved their data into TTrees and a TChain.

By using `TWaveformSpace's` histogram building methods, we were able to group all this data, and represent it in the plot on Fig.7. This graph illustrates the height distribution of the PMT waveforms' first pulse. Once again, we encounter what seems like a normal distribution. In this case, however, the distribution is positively skewed. The results of this histogram come as no surprise; since the PMT detector allowed mostly single photons to pass at a time through the PMT, we expected the height of the single pulses to be similar, and actually represent the energy distribution of the incoming photons. This last distribution would follow a gaussian curve, but since we have a lower end limit for the height of the pulses (and thus their energies), this distribution is skewed to the right.

This section meant to give you a sense of the capabilities of the `TWaveform` package, and how you could use it to analyze large sets of waveforms you may have in hand. Detailed descriptions of most of the `TWaveform` methods are given in Chapter 2, and in Chapter 3 we will actually guide you through the process of implementing these methods to analyze the PMT waveforms presented in this section.

1.3 Source Code

The `TWaveform` package includes the `TWaveform` source code and header, as well as a LinkDef header and a Makefile (TWaveform.cxx, TWaveform.h, MyLinkDef.h, and Makefile respectively). Following are the instructions for downloading, compiling and loading the source code into ROOT's interactive shell.[5]

Download

Compile As stated above, a Makefile is included within the `TWaveform` package. This file compiles the source code and creates a shared object of the `TWaveform` source code by making use of ROOT's rootcint program. It is important to note that this Makefile was designed to work with macOS 10.12.1, and may need to be modified when used on other operating systems. Once the file has been modified to work with the device's operating system, all that is needed to compile and link the source code is to open the TWaveform package directory through terminal and input:

```
make all
```

If done correctly, the following output should be seen:

[4] Look at `SetSignalPositive()` in source code

[5] TWaveform works best when used with ROOT v.6.06/04. For instructions on how to use and download ROOT check ROOT's user guide.

```
Linking TWaveform.so ...
Done
```

Load Now that the `TWaveform` shared object has been linked, it is ready to be downloaded into ROOT's interactive shell. Below is an example of how to do this through the computer's terminal:

```
This-MacBook:TWaveform user$ root -l
root [0] gSystem->Load("TWaveform.so");
root [1] TWaveform wave;
```

After downloading the shared object into ROOT's interactive shell, all of `TWaveform's` member and non-member functions are available for use. In the following chapters we will go through some of the functions included inside the `TWaveform` package and how to use them.

1.4 Quick Start

This section has the purpose of guiding you through a set of very simple steps, so that you can experience, first-hand, the capabilities and functionality of the `TWaveform` tool set. To be able to go through these steps, you should make sure that you have downloaded a version of ROOT (preferably v.6.06) and that within your `TWaveform` package is a directory named Waveforms, which should contain data waveforms from both the PMT and SiPM detectors. In this section we will specifically be using the data files in the Run125 folder. This contains a small number of txt files (25 to be exact) that hold data from the PMT detector's signal, described in Section 1.2. You should also check that there is a file in the "Templates" directory called "TemplateTXT.root".

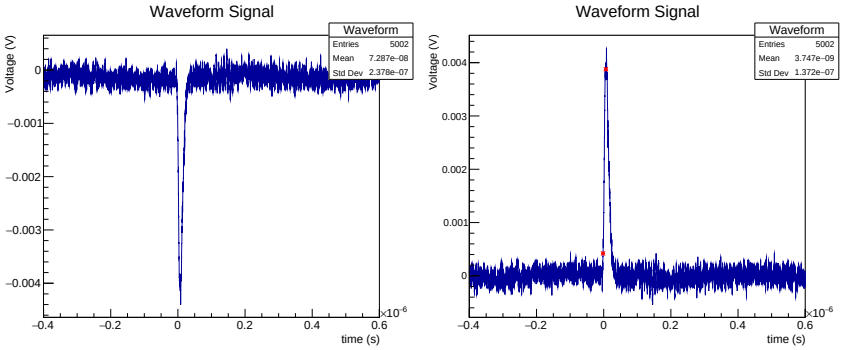
Once you have verified that you have all of these requisites, you can jump right in and start using `TWaveform` for this small tutorial. First of all, lets read one of the waveforms in the txt files, and plot it using `TWaveform`;

```
root [0] gSystem->Load("TWaveform.so");
root [1] TWaveform wave;
root [2] wave.ReadFile(wave.GetFilename(20,125));
root [3] wave.Draw();
```

After running these commands, a canvas window should pop-up with a plot of the PMT's signal. It ought to look something like the plot in Fig.8a.

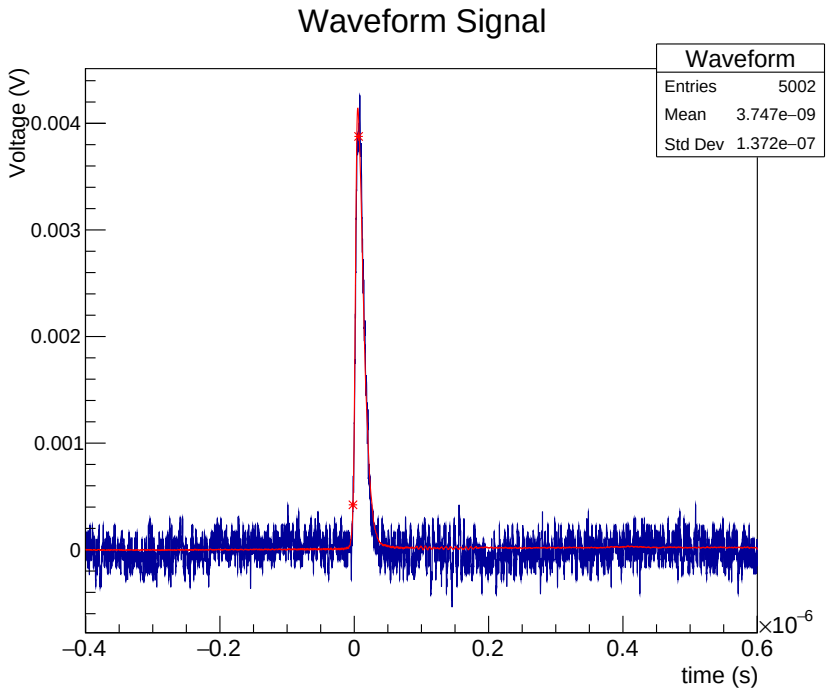
Lets now proceed to do a small processing of the waveforms. This method will turn the signal to positive (if it is negative), apply an offset so that it rests on the x-axis, and calculate a series of properties from the waveform that are needed for incoming analysis. Among these properties, the most significant are the integral of the waveform (saved in the array data member: vIAreaA), and a rough calculation of the waveform's first pulse's peak and rise position. In some cases, such as the one displayed here, this method will yield seemingly correct values for the peak and rise positions of a pulse. Nonetheless, since the algorithm used to calculate these points is far less accurate than the template fitting method we will show later on, you should try not rely on this mechanism for any meticulous analysis. Lets now see this process in action:

```
root [4] wave.PreDataProcessing();
root [5] wave.Draw();
```



(a) The plot of the waveform's raw data.

(b) The plot of the waveform after being pre-processed.



(c) A plot of the waveform along with its best fitted template after being completely analyzed. The blue curve represents the signal, and the red curve its best fit.

Figure 8: These are plots of one of the data waveforms included in the Run125 directory of the `TWaveform` package. The red asterisk in (b) and (c) represent the peak and rise position of each pulse.

```
root [6] wave.MarkRise();
```

Once again, the canvas window will pop-up, but this time with the waveform having a positive signal (Fig.8b). If you run the `MarkRise()` command after, as done in here, the peak and rise positions calculated will be marked into the plot with red asterisks.

Now, its time to take the big guns out! The method we will run next is the heart of the `TWaveform` tool-set. With it, the waveform will be fitted to the templates provided inside the "TemplateTXT.root" file. This will allow for a much more precise calculation of all the pulses' peak and rise positions, along with the total number of pulses found in the waveform, among many other things. Lets get to it then:

```
root [7] wave.AnalyzeWave();
root [8] wave.DrawBestFit();
root [9] wave.MarkRise();
```

Just like that, we have analyzed the entire waveform, and have gained valuable information about its pulses. The `DrawBestFit()` function also plots the waveform along its best fitted template, as illustrated by Fig.8c. It is worth clarifying that when running `AnalyzeWave()` on any waveform, the `PreDataProcessing()` method is called automatically.

Since to create templates and TTrees, a large number of waveforms are recommended, we are not gonna run these processes in here. If you want to learn how to do use these for your own data, you can check Sections 3.6 and 3.8. We have, however, included the TTrees and TChain of all the PMT waveforms in Run125 that were used to create the histograms in Section 1.2. You can use said trees and chain to recreate any of the PMT histograms, and more, by using `TWaveformSpace's` data analysis functions. You can run these simply by passing it the file-name of the tree or chain you want to analyze, just as:

```
root [10] TH2F *hist =
    TWaveformSpace::ChiSquaredAndPulseHeightHist(
    "WaveChains/WaveChain125.root");
root [11] hist->Draw("surf2");
```

Finally, we have also included a couple of data files from the SiPM detector, inside the Waveforms directory, so you can explore `TWaveform's` tools with more complicated waveforms. These waves will also demonstrate even further the potential behind the `Twaveform` package. This time, we will leave it for yourself to discover how to access and analyze these waveforms. For a guide on what steps to take, go through Section 3. A few hints were also left in TWaveformConstants.cxx as to the values you may need to modify to successfully use these waveforms.

2 How To Use

Throughout this chapter we will learn about some of the main methods contained inside the `TWaveform` class and how to use them effectively. We will start by looking at the basics, such as reading files and plotting waveforms, and will proceed to more complicated functions, like creating pulse templates, finding the peaks and rise position of pulses on a

waveform, and so on.[6]

2.1 Class, Structs and Environment

Before we dive in straight into `TWaveform's` member functions, it might be useful to take a moment to explore some of the class's preeminent properties and environment settings. By expanding on these subjects, the user will be able to achieve a higher level of understanding on how to make the best out of the tools provided in the `TWaveform` package.

One of the main attributes of the `TWaveform` class is that it is derived from the ROOT object, TNamed, which in turn inherits from the common base class: TObject. This inheritance allows a `TWaveform` object to have access to all of TNamed's properties. For this reason, every instance of `TWaveform` can be given a key name and title that can be used to reference the object's instance with ease. As we will see further on in Section 2.4, this ability permits `TWaveform` objects to be conveniently stored and read from .root files, among other things.

Data Members

Lets now proceed to take a quick look at `TWaveform's` data members and their main purposes:

- std::string directoryAddress: Refers to the address of the directory that contains the data files that will be read onto the `TWaveform` objects.
- std::string genericFileName: A string that contains the generic file name given to the data files that hold each waveform's information.
- std::string fileType: This data member saves in a string the type of files that contain the waveforms' data.[7]
- int nSamples: Saves the number of data points that are read to create the waveform.
- float t[8192]: An array of type float that holds the time (x-axis)[8] values of each data point in the waveform.
- float vA[8192]: A float type array which saves the voltage (y-axis) values of the waveform's data points.
- float tBinSize: This data point saves the time difference between consecutive data points.
- double vAreaA[8192]: An array of type double that contains the area under the waveform between consecutive data points.
- double vIAreaA[8192]: Similar to vAreaA, this array of type double saves the area below the waveform from its start, and up to the current sample point.
- float vOffsetA: A float data member that records the voltage (y-axis) offset applied to the waveform.
- float stDevA: This float value refers to the waveform's standard deviation.

[6] For a complete look at all of `TWaveform's` member functions and data members, refer to the TWaveform header and/or source file ("TWaveform.h" and "TWaveform.cxx" respectively)

[7] For a more detailed explanation on how these file data members work go to 3.1.

[8] Throughout this manual, time will represent the waveform's x-axis values, while voltage will be used to reference the y-axis values.

- int risePosA[4096]: This int array holds the position of each pulse's rise position in the waveform.
- int peaksA[4096]: An int type array that carries the each pulse's peak position in the waveform.
- int nPeaksA: Records the number of pulses found in the waveform.
- bool noiseA: A boolean data member that determines if the current waveform is composed of some kind of pure periodic noise.
- std::vector<std::string>bestTemp: A vector holding a set of strings that contain the name of the templates that fit the waveform the best.
- std::vector<int>bestTempOffset: An int type vector containing the time offset applied to each template to give the best fit with the current waveform.
- std::vector<double>bestTempScale: Similar to bestTempOffset above, this vector of type double records the voltage scale applied to each template as a best fit parameter.
- double bestTempChi: This data member holds the normalized χ^2 between the current waveform and its best fitted templates.

Above is just an overview on what each of `TWaveform's` data members is used for. A more detailed explanation on how to access these data members and the information they hold will be given throughout the following chapters.

Associated Structs

The `TWaveform` source code also introduces a set of structs that will be shown to be useful towards the implementation of the class. In total, there are three of these structs;

- fitParam: Used to save a single template's best fitting parameters. It contains bestOffset, bestScale, and chiSquared as data members.
- chiParam: Similar to fitParam, but instead holds the fitting parameters for a whole set of templates into the vectors: bestTemp, bestOffset, and bestScale. It also includes a data member for χ^2 named chiNor.
- treeData: This struct contains data members that hold all the information extracted from a single entry of a waveform tree. We will expand more on trees and the struct treeData in Section 2.5.

As you start utilizing the `TWaveform` package, you will find it great convenience to make use of these structs. Also, as will be seen throughout this chapter, there are a number of functions that will either use or return these structs. For this reason, it might be a good idea to take a quick look at their definitions inside the TWaveform.h header file.

TWaveform Constants

Apart from the structs just mentioned, the `TWaveform` package also includes a set of environment variables that will make it easier to make changes to how some of `TWaveform's` methods function. All these variables are saved inside the TWaveformConstants.cxx file included with the package, and are under the TWaveformConstants namespace. By accessing this file and changing the variables' values, you will be able to control the environment setting within which the `TWaveform` class will operate. For a quick explanation on what each variable means, you can check the

TWaveformConstants header file. We will also study how each of these variables affect the `TWaveform` environment in the next few chapters.

2.2 Basic Methods

Now that we have uncovered some of `TWaveform's` properties and environment, we have the sufficient tools to dive into some of the class' member functions. The first step in doing so will be to learn how to declare an instance of a `TWaveform` object.

Declaration

With this being said, there are four main ways in which a `TWaveform` object can be declared; with a key name and title, with the file data types specified, with both or without either.

```
root [1] TWaveform wave1
(TWaveform &) Name: Title:
root [2] TWaveform wave2("wave2","A Waveform")
(TWaveform &) Name: wave2 Title: A Waveform
root [3] TWaveform wave3("data/waveforms","/waveform", ".txt")
(TWaveform &) Name: Title:
root [4] TWaveform wave2("wave4","Another Waveform",
    "data/waveforms", "/waveform", ".txt")
(TWaveform &) Name: wave4 Title: Another Waveform
```

For the first instance of `TWaveform` (wave1), we declared the object without passing it any variables. As such, wave1 is then initialized without a key name nor title, and its file data types (directoryAddress, genericFileName, and fileType) are set to their default values as defined in TWaveformConstants.cxx. For wave2 we give the `TWaveform` object a key name and title, while in wave3 we initialize the object's file data types (as defined above) to the passed values, respectively. The fourth `TWaveform` instance is just a combination of these last two descriptions.

Reading Files

Now that we know how to declare a `TWaveform` object, we need a function that is able to read and save the data from files into a `TWaveform` object. Fortunately, the `TWaveform` class comes with a built-in function, `ReadFile(std::string fileName)`, that can read files and save the necessary data into the `TWaveform` object. Currently, this built-in function only works for reading off files with one of either two formats. Below is shown an example of the format used for .CSV files, through a method called `ReadCSV()`. The other format, used for .txt files, is recorded in the TWaveform.cxx file under `ReadTXT()`'s function definition.

```
Memory Length,4000,
Trigger Level,-2.40000e+01,
Source,CH2,
Probe,1.0X,
Vertical Units,V,
Vertical Scale,5.00000e-02,
Vertical Position,1.04000e-01,
Horizontal Units,S,
Horizontal Scale,1.00000e-03,
Horizontal Position,4.0000000e-05,
Horizontal Mode,Main,
Sampling Period,4.00000e-06,
Firmware,V1.09,
Time, ,
Mode,Detail or Fast,
Waveform Data,
```

```
//   What follows depends on whether it is detail mode,
//   or fast mode.
//   For detail mode it is:
time[1],v[1],
time[2],v[2],
time[3],v[3],
...
time[n],v[n],
//   For fast mode it is:
v[1],
v[2],
v[3],
...
v[n],
```

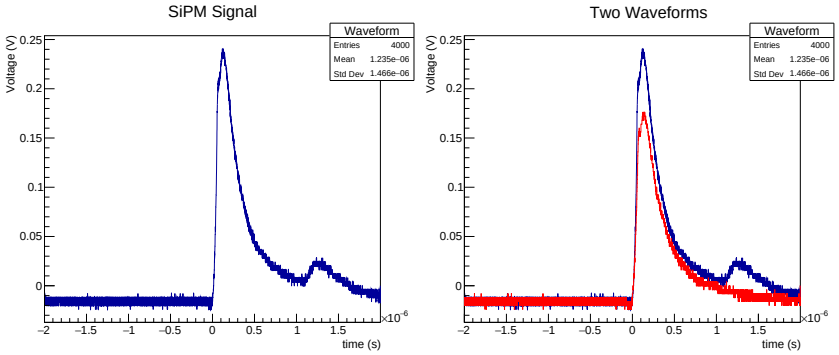
If a format different from these two is used in the data files that need to be read, it is necessary to create a new `Read()` method into the `TWaveform` class that is capable of reading these types of files. Afterwards, you will also have to modify the built-in `ReadFile()` method to include the newly written function. If confused with this process, don't worry! We will drive you through the necessary steps to do so with an example in Section 3.2.

To use `ReadFile()`, you just need to pass it a string with the name of the file that you want to read. The format that will be used to read the file is defined by the `TWaveform's` data member: fileType. If, for example, fileType = ".txt", then the `ReadTXT()` method will be used to read the file. You can define this data member when declaring it, as explained above, or change it any point by invoking the `SetFileName(std::string dirAddress, std::string fName, std::string fType)` member function.

`TWaveform` also contains a method that is capable of generating strings with the names of files that have a predetermined naming system, and only differ by their file and/or directory number; `GetFilename(int waveNum, int dirNum)`. This method is specially useful when reading a series of numbered files from different directories. Similar to the `ReadFile()` method above, `GetFilename()` can choose between different naming formats for each specific file type (these are shown in `GetTXTFilename()` and `GetCSVFilename()` function definition). Again, the format used in this method is defined by the `TWaveform's` fileType data member. If none of the available formats fit the naming system of your files, you will need to create a method that does so, and then modify `GetFilename()` to include it. `This step is crucial!` The `GetFilename()` function will be used in many of the waveform analyzing methods that we will see later on.

Next on is an example on how to use the methods described above:

```
root [1] TWaveform wave("/Users/Documents/Waveforms", "/DS",
    ".CSV");
root [2] wave.ReadFile("/Users/Documents/Waveforms20/DS234.CSV");
root [3] wave.vA[3]
(float) 0.274337f
root [4] wave.Clear();
root [5] wave.ReadCSV(wave.GetFilename(234,20));
root [7] wave.vA[3]
(float) 0.274337f
root [6] wave.GetFilename(234,20);
```



(a) One waveform (b) Two waveforms

Figure 9: These plots were created by the piece of code in the Draw section above. These figures show how one and two waveforms are plotted by passing different variables to `TWaveform's` Draw() method.

```
(std::string) "/Users/Documents/Waveforms20/DS234.CSV"
```

Draw After reading the waveform data into the `TWaveform` object, it might be important to plot it. Luckily, `TWaveform` has a built-in method precisely for this; `Draw(std::string histName, Option_t* option, Color_t color)`[9]. The first variable, histName, is just the key name given to the histogram, while the other two variables are options given to ROOT's TH1 class to specify how the histogram is plotted. Following is an example on how to use this function.

```
root [1] TWaveform wave1, wave2;
root [2] wave1.ReadCSV((wave.GetFilename(234,20));
root [3] wave1.Draw();
root [4] wave2.ReadCSV((wave.GetFilename(452,20));
root [5] wave2.Draw("wave2", "hist same", kRed);
root [6] TH1F* hist=((TH1F*)(gROOT->FindObject("Waveform")));
(TH1F *) 0x7fce66b00000
root [7] hist->SetTitle("Two Waveforms");
```

We can see the plots created by this piece of code in Figure 9. In Fig. 9a, only one waveform is plotted by using the `Draw()` method with its predetermined variables. On the other hand, Fig. 9b shows a plot of two waveforms by making use of the "same" Option_t variable passed to `Draw()`. Note that through the piece of code in line [6] we were able to retrieve the histogram that was being plotted, and so change some of its properties, such as the plot's title.

Two other basic functions that may be worth looking into are the `Shift(int offset, double scale)` and `SmoothOut(int nPoints)` member functions. As their names suggest, the `Shift()` method is capable of shifting and scaling a waveform by the values passed to it. Similarly, `SmoothOut()` can smoothen out waveforms and reduce their visible noise by averaging the signal at each point with nPoints surrounding it. We won't show how to use these methods here, but you are encouraged to try them out for yourself.

[9] If no variables are passed to Draw(), they take the following values: histName = "Waveform", option="C Hist", color=kBlue+2.

2.3 Analyze Waveforms

Now that we know how to read CSV files with the `TWaveform` class, and how to use some of its basic member functions, it is time to look at some methods included inside the class that can be used to analyze different properties of a waveform. These include functions that are able to calculate the average, the standard deviation, and the integral of a waveform, among others.

Finding the Average

Lets first take a quick look at how the `FindAvg()` method works:

```
float FindAvg(int pos,
              std::string data,
              int nPoints)
```

This method will return the float value of the waveform's average signal calculated from its position (pos) in the x-axis, and up to nPoints after that. Lets look at what the passing variables represent;

- pos: The index of the position in the x-axis at which the waveform's average will start to be calculated. If pos is set to 0, the function will start averaging the waveform from its start.
- data: Defines the data for which the average will be calculated. If data = "Waveform", the function will calculate the original waveform's average signal. On the other hand, if data = "Integral", the function will return the average signal of the waveform's integral.
- nPoints: Sets the amount of data points, after pos, that will be used to calculate the average. If pos + nPoints is greater than the total amount of data points (nSamples), then the average will be calculated until the end of the waveform.

Area and Integral

Another set of useful member functions in `TWaveform` are the `FindArea()`, and the `FindIntegral()` functions:

```
void FindArea()
void FindIntegral()
```

`FindArea()` is a function that calculates the area under the waveform between each data point and assigns it to TWaveform's data type: vAreaA. This means that, after using `FindArea()`, vAreaA[3] saves the value for the area below the waveform signal between t[3] and t[4].

On the other hand, `FindIntegral()`, as its name suggests, calculates the integral of a waveform, and saves it into `TWaveform's` data type: vIAreaA. This way, vIArea[3], for example, will contain the total area under the waveform from data points t[0] to t[3].

Calculating Slopes

Next on we will examine how to calculate the slope of a waveform by making use of the `FindSlope()` method.

```
float FindSlope(int pos,
                std::string data,
                int nPoints)
```

The main purpose of this method is to find the slope of a waveform at the position (pos) defined. This function's passed variables work similar to `FindAvg()`;

- pos: The position/index of the waveform where the slope will be calculated. pos is such that t[pos] gives the time position of the slope calculated.
- data: As with `FindAvg()`, data defines from which data member will the slope be calculated for. If data = "Waveform", the function will calculate the slope of the original waveform's signal (vA). Otherwise, if data = "Integral", the function will return the slope of the waveform's integral (vIAreaA).
- nPoints: Represents the number of data points that will be averaged out to calculate the slope of the waveform. If, for instance, nPoints = 4, then the slope will be calculated as the difference between the returned value of `FindAvg(pos, data, nPoints)` and `FindAvg(pos - nPoints, data, nPoints)`.

Applying an Offset

Another useful method found in the `TWaveform` aresenal is `ApplyOffset()`:

```
void ApplyOffset(std::string data)
```

This function is capable of finding if a waveform has a voltage offset (y-axis), and if so, displaces the waveform so that there is no offset. In this instance, the variable "data" refers to which data set is used by the method's algorithm to find the waveform's offset. It is greatly recommended that data is always set to "Integral", as it was tested to employ a much more effective algorithm to calculate and apply the offset.[10] Once `ApplyOffset()` is called, the voltage value by which the waveform was displaced is saved in vOffsetA. If desired, the offset can be calculated without applying it by using `FindOffset()` in the same way as `ApplyOffset()` would be used.

Noise and Standard Deviation

When analyzing any type of waveform, it is extremely useful to be able to determine if a waveform is pure noise, or in fact the signal has a pulse. For this purpose, a method was integrated into the `TWaveform` class, named `NoiseFilter()`, that is capable of signaling if a waveform is pure noise or not. Another important function used for error calculations is `FindStdev()`, which calculates the standard deviation of a waveform's signal.

```
void NoiseFilter()
void FindStdev()
```

For both of these methods to work properly, `ApplyOffset()` must first be called. After this, if `NoiseFilter()` is called, `TWaveform's` noiseA data member is changed to true if the waveform is pure noise, and false otherwise. Similarly, after calling `FindStdev()`, `TWaveform's` data member, stDevA, is set to the calculated standard deviation value.

[10] This will change in a future update, where the use of the data variable will be removed.

2.4 Template Fitting

Now we move on to look at one of the most powerful tools contained within `TWaveform`; template fitting. As expressed in the introduction, by creating and fitting a set of templates using the following methods, you will be able to identify multiple pulses inside of a waveform, as well as recording their properties, such as pulse height, pulse position, and more. In this section we will look at how to create templates using `TWaveform's` member functions.

Create Templates

First off, we need to know how to create the waveform templates that will be used for fitting. The main method used for doing this is called `AvgAreaTemplates()`, and it has two slightly different versions;

```
std::vector<TWaveform> AvgAreaTemplates(int waveDir,
                        const char *fName,
                        double minArea,
                        double maxArea,
                        int numTemps,
                        const char *tempName,
                        int numWaves)
```

and

```
std::vector<TWaveform> AvgAreaTemplates(int waveDir,
                        const char *fName,
                        double *areaRange,
                        int numTemps,
                        const char *tempName,
                        int numWaves)
```

What this function does, basically, is it takes a set of waveforms in the directory given by `GetFilename(n, waveDir)`[11], and averages all the waveforms that have areas within the given ranges to create and save the templates into a root file. After templates are generated, the function returns a vector containing all of the templates just created.

As for the variables that are passed to the first version of this function;

- waveDir references the number of the directory where the waveforms are.
- fName is the name given to the root file where the templates will be saved.
- minArea and maxArea determine the area limits for which the templates will be calculated
- numTemps is the number of templates the user wants to create.
- tempName is just the generic name given to each template[12].
- numWaves gives reference to the total number of waveforms that wish to be included in the template creation. Note that if numWaves is set to 0, all the waveforms in the directories are used for the template generation.

[11] Here n is any integer. Check Section 2.2 for more details

[12] If tempName = "Template", the templates will be named: "Template0", "Template1", "Template2", etc.

The way this first version works, is that it takes the area range given to it[13], and divides it into equal steps, for a total number of numTemps ranges. Each template is then assigned to one of these area ranges, from which waveforms that have areas within the range will be averaged. Lets see how to implement this method:

```
root [1] TWaveform wave;
root [2] std::vector<TWaveform> temps;
root [3] temps = wave.AvgAreaTemplates(20,
    "Templates_20.root", 9e-9, 1.7e-7, 7, "Template_", 0);
root [4] temps.at(1).Draw();
root [5] wave.SaveTemplates(temps, "Templates_Smooth.root",
    "TempSmooth_", true, 20);
root [6] TFile *tempFile = new
    TFile("Templates_21.root","Read");
root [7] .ls
TFile**   Templates_20.root
 TFile*   Templates_20.root
   KEY: TWaveform Template_0;1
   KEY: TWaveform Template_1;1
   KEY: TWaveform Template_2;1
   KEY: TWaveform Template_3;1
   KEY: TWaveform Template_4;1
   KEY: TWaveform Template_5;1
   KEY: TWaveform Template_6;1
root [8] tempFile->Close();
root [9] wave.ReadTemp("TempSmooth_1","Template_Smooth.root");
root [10] wave.Draw();
```

The example above showcases how to create a set of templates out of the CSV files found at the waveform directory #21. There are a few things we should look at in this block of code.

- Line [5]: notice the `SaveTemplates()` function used. This method saves a vector of waveforms passed to it into a root file. When this method is passed a `true` value for its 3^{rd} parameter, the templates will be smoothed out, as seen in Fig. 10b.
- Line [7]: We can see here the contents of the newly created template file.
- Line [9]: `ReadTemp()` reads the waveform specified by its parameters.

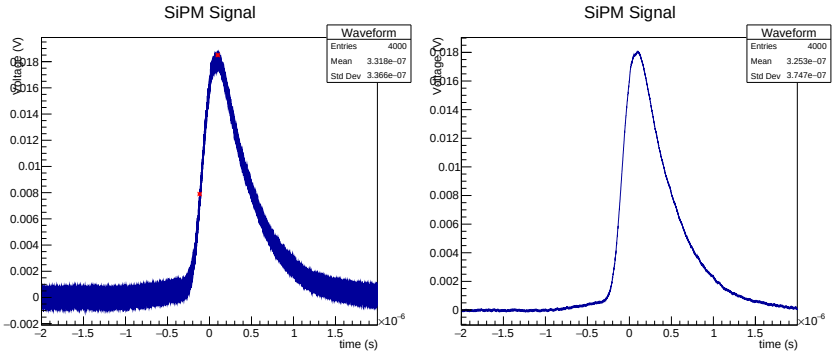
The only change in the second version of the `AvgAreaTemplates()` method is that, instead of passing it a minimum and maximum area values for the area ranges, a double array, named areaRange, is passed with the limits of the area range for each template to be created. This second method is specially useful when the waveforms have different shape-changing rates with respect to their areas.

Fitting Templates

Now that we have created a set of pulse templates, it is time to have a look at the template fitting methods contained in `TWaveform`. The main function that will do this is `FindBestTemps()`:

```
bool FindBestTemps(const char *tempFile,
                   const char *tempName,
```

[13] maxArea - minArea



(a) A plot of "Template_1" created in the template creation example.

(b) A plot of "Template_1" after being smoothed out.

Figure 10: These plots were created by the piece of code in the Create Templates section above. These figures show the results of using the template generation method `AvgAreaTemplates()`, and the difference between the original and smoothed out versions of the same template.

```
int tempNum,
int pulseNum,
int iterations,
double minChi,
int tLim,
double scaleLim,
int tStep,
double scaleStep)
```

This method, essentialy, finds the set of templates, along with their fitting parameters, that, when combined, best fit the waveform in question. The names of the best templates, and their respective fitting parameters, are saved into the waveform's data members; bestTemp, bestTempOffset, bestTempScale, and bestTempChi accordingly. The variables that are passed to this function are defined as follows:

- tempFile: the name of the root file were the set of templates that are going to be fitted are saved.
- tempName: the generic name given to the templates saved inside the root file tempName.
- tempNum: the number of templates that are saved inside the root file tempName.
- iterations: the number of times that you wish the function loops over the fitting algorithm for multiple pulses. As the number of iterations increases, so does the precision of the multiple pulse fit.
- minChi: the minimum normalized χ^2 for the best template fit to be considered as a good fit. If the χ^2 between the waveform and templates being fitted is smaller than minChi, the function will return false, and the waveform's data members won't be modified.
- tLim: the maximum number of data points the templates will be shifted to try to fit the waveform.[14]

[14] This shift is in the time axis of the waveform

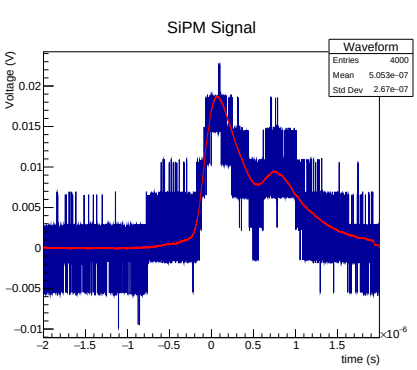
- scaleLim: the maximum multiplier that will be used to scale the templates and find a fit with the waveform.
- tStep: the number of integer steps at which the templates are shifted and χ^2 is calculated for the template fit.
- scaleStep: the multiplier steps at which the templates are scaled and χ^2 is calculated for the template fit.

After succesfully finding the best fit templates for a waveform, it is very easy to plot it by using the `DrawBestFit()` member function. We will explore how to use both of these methods in the piece of code that follows:

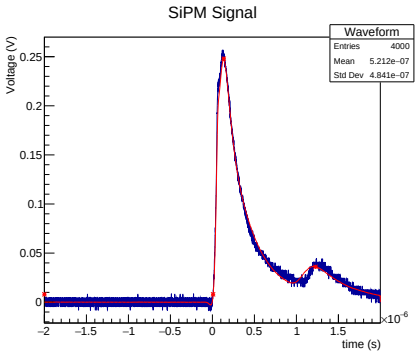
```
root [1] TWaveform wave, waveMark;
root [2] wave.ReadCSV(wave.GetFilename(423,20));
root [3] wave.FindBestTemps("Templates_Smooth.root",
    "TempSmooth_", 7, 4, 10, 1000, 3, 8, 0.1);
root [4] wave.bestTemp
(std::vector<std::string> &) { "TempSmooth_2", "TempSmooth_1" }
root [5] wave.bestTempScale
(std::vector<double> &) { 0.800000, 0.300000 }
root [6] wave.bestTempOffset
(std::vector<int> &) { 60, -730 }
root [7] wave.bestTempChi
(double) 0.998188
root [8] wave.DrawBestFit("Templates_Smooth.root");
root [9] wave.FindPeaksTemp("Templates_Smooth.root");
root [10] wave.nPeaksA
(int) 2
root [11] wave.peaksA[0]
(int) 2068
root [12] wave.peaksA[1]
(int) 2831
root [13] waveMark.ReadCSV(wave.GetFilename(421,20));
root [14] waveMark.FindBestTemps("Templates_Smooth.root",
    "TempSmooth_", 7, 4, 10, 1000, 3, 8, 0.1);
root [15] waveMark.DrawBestFit("Templates_Smooth.root");
root [16] waveMark.FindPeaksTemp("Templates_Smooth.root");
root [17] waveMark.vA[waveMark.peaksA[0]]
(float) 0.248021f
root [18] waveMark.t[waveMark.peaksA[1]]
(float) 1.22700e-06f
root [19] waveMark.MarkRise();
```

Lets take a closer look at what was done in this piece of code.

- Lines 3 & 14: `FindBestTemps()` was used to find the best fit templates for the `TWaveforms` wave and waveMark.
- Lines 4-7: We look at the parameters that were recorded to the `TWaveform` wave's data members after using `FindBestTemps()`.
- Lines 8 & 15: The waveforms are plotted, along with their best fit template, by using `DrawBestFit()`. These plots are shown in Fig. 11.
- Lines 9 & 16: `TWaveform's FindPeaksTemp()` is used to calculate and record the position of the pulses' peaks in each of the waveforms.
- Lines 10-12: We look at the number of pulses inside the `TWaveform` wave, and their peaks' position.



(a) A plot of the `TWaveform` wave with its best fit.



(b) A plot of waveMark with its best fit. The red asteriks mark the waveform's peak and rise positions.

Figure 11: These plots were created by the piece of code in the Fitting Templates section above. These figures show the results of using the template fitting method; `FindBestTemps()`, by displaying both the waveforms' signal (blue), and their calculated best fit (red).

- Line 17: The height (in Volts) of waveMark's first pulse's peak is shown.
- Line 18: We see the time position (in seconds) of waveMark's second pulse's peak.
- Line 19: The `TWaveform` member function, `MarkRise()` is used to mark the position of waveMark's pulse peaks, and first pulse's rise position in the current plot. The result of this is seen in Fig. 11b.

Just as that, we have been able to fit the templates into the given waveforms with a remarkable degree of precision. Not only this, but we were able to discover some properties of the pulses, like their peak heights and time positions, with a fairly simple piece of code. It is with these methods, and the relative ease with which they can be used, that the `TWaveform` class gains its power. In the next section we will look at some non-member functions included inside the `TWaveform` source code that are able to analyze a large set of waveforms and save their properties into data sets called TTrees.

2.5 TTrees

When analyzing a large set of waveforms, it may take a significant amount of time for all the waveforms to be fitted and analyzed. For this reason, it is beneficial to have an object that can store the properties of all the waveforms that need to be analyzed. This way, the large set of waveforms don't have to be fitted and processed every time this information is wished to be accessed. ROOT's TTree and TChain objects do precisely this. They are capable of storing large blocks of structured data that can later be accessed quickly. In this section we will look at a few non-member functions included in `TWaveform` that are used to analyze and save data in TTrees and TChains, as well as reading data off of them.

Create trees

`TWaveform` contains two main methods for creating trees, and one for connecting trees into a TChain. The two TTrees generating methods are

named; `CreateTree()`, and `CreateTrees()`. The only difference between these two is that the first method only creates one TTree with all the needed data, while the second method creates a set of TTrees, each with the information of 400 waveforms maximum[15]. This second method is very useful when a large set of waveforms are being processed. This is due to the fact that sometimes individual trees can become corrupted for whatever reason. Therefore, by saving the data into multiple trees, if one of the trees becomes corrupted, only a smaller set of waveforms will be needed to be processed again, instead of the whole, larger set.

When `CreateTrees()` is used, all the TTrees created by this method can later be easily linked to create a TChain by using the `CreateChain()` method. If you wish to skip this middle step, the `CreateTreeChain()` method both creates a set of TTrees with the same data as `CreateTrees()`, and then links them up into a TChain automatically. This last method is the one we advise to use the most for its relative simplicity and safeguard against corrupted files, and it is the one we will explore in this section. Nonetheless, there might be cases were the other methods will come in handy, so we encourage you to try them out if you can!

```
void CreateTreeChain(int fileNum,
                     const char *fName,
                     const char *treeName,
                     const char *chainName,
                     const char *chainFile,
                     const char *treeTitle,
                     const char *chainTitle,
                     const char *tempFile,
                     const char *tempName,
                     int tempNum,
                     int pulseNum,
                     int iterations,
                     double minChi,
                     int tLim,
                     double scaleLim,
                     int tStep,
                     double scaleStep)
```

This function may look a bit overwhelming by its large set of parameters, but as we go through them you will see that using it is a lot easier than it seems.

- fileNum: the number of the directory where the waveforms to be processed are found using `GetFilenam(n, fileNum)`.
- fName: the generic address name that the TTrees will be saved to. If, for example, fName = "WaveTrees/WaveTree", then the trees created by this method will be saved in the root files with address: "WaveTrees/WaveTree0.root", "WaveTrees/WaveTree1.root", and so on.
- treeName: the key name that will be given to the TTrees saved in the root files.
- chainName: the key name that will be given to the TChain of all the TTrees.

[15] This number can easily be modified in the source code if so desired.

- chainFile: the generic address name that the TChain will be saved to. If, for example, chainFile = "WaveChains/WaveChain", then the TChain created by this method will be saved into the root file with address: "WaveChains/WaveChain(fileNum).root".
- treeTitle & chainTitle: the titles given to the TTrees and the TChain object, respectively.
- Remaining variables: these parameters are the ones that will be used by the `FindBestTemps()` method to process and fit the waveforms[16].

Now that we know the meaning behind the parameters passed to this function, lets see an example on how to use it, and check the variables that were saved into the TTrees.

```
root [1] TWaveform wave;
root [2] CreateTreeChain(19, "WaveTrees/WaveTree19/WaveTree",
    "WaveTree", "WaveChain", "WaveChains/WaveChain", "Waveform
    Tree", "A Waveform Chain", "Templates_Smooth.root",
    "TempSmooth_", 7, 4, 10, 1000, 3, 8, 0.1);
0
1
...
953
954
Last file to be added: WaveTrees/WaveTree19/WaveTree2.root
root [3] TFile *file = new
    TFile("WaveChains/WaveChain19.root", "Read");
root [4] .ls
TFile**   WaveChains/WaveChain19.root
  TFile*     WaveChains/WaveChain19.root
    KEY: TChain WaveChain;1 A Waveform Chain
root [5] TChain *chain =
    ((TChain*)(gROOT->FindObject("WaveChain")));
root [6] chain->Print();
...
root [7] chain->Scan();
...
```

Lets try to decompose this piece of code:

- Line 2: The given variables are used in conjunction with the `CreateTreeChain()` method to create a TChain of the TTrees created for the waveforms found in directory # 19. The numbers displayed below represent the number of the waveforms that are being processed.
- Line 3 & 4: We open the root file recently created, and see it contains the TChain generated.
- Line 5: We retrieve the TChain found in "WaveChains/WaveChain19.root" and assign it to the variable; chain.
- Line 6: We print the newly generated TChain's content. This output can be seen in Fig. 12.
- Line 7: We scan the data of some of the first entries in the TChain. This output can be seen in Fig. 13.

[16] For a reminder on how to use these, look at section 2.4

It is now useful to take a look at the information saved into the TTrees, and consequently, into the TChains. These data points can be seen in both Fig. 12 and Fig. 13. Lets take a closer look at each of these variables:

- Noise: A boolean value that records true if the waveform was determined to be pure noise by the member function; `NoiseFilter1()`.
- RisePosition: A double data type that files the rise position of the waveform's first pulse.[17]
- StandardDeviation: A float data type that documents the standard deviation of the noise in each waveform, as calculated by the member function; `FindStdev()`.
- Area: A double data type the records the total area below the waveform. This value is calculated after using the member function `Integral()`.
- BestTemps: A vector of type `std::string` that saves the name of the best fitting templates for the pulses in each waveform. They are sorted by order of appearance in each waveform[18].
- PulseStrength: A type double vector that records the height of each of the individual pulses in the waveform. This looks at the peak height (in volts) of the individual templates when scaled to their best fit value.
- PeakPosition: An int type vector that saves the position in the waveform where the pulse's peaks are found. By multiplying these values by the TimeBins value, you are able to calculate the time position (in seconds) of each peak.
- SignalStrength: A vector of type double that documents the height of the waveform's signal (in volts) at the position of each of its pulses' peak.
- TimeBins: A float data type that records the difference in time (in seconds) between each data point in the waveform.
- TimeUncertainty: A vector of type int that saves the uncertainty at which each pulses' peak and rise position is determined. This value is calculated by use of the function `GetTimeUncertainty()`.
- PulseHeightUncertainty: A vector of type double that saves the uncertainty at which each pulses' peak height is determined. This value is calculated by use of the function `GetPulseUncertainty()`.
- WaveformNumber: An int data type that saves the number of the waveform in its respective directory.
- DirectoryNumber: An int data type that saves the number of the directory in which the waveform's CSV file is saved.

You will find that inside `TWaveformSpace` there are a collection of data analysis and histogram generating functions. We will not go through them individually, as there are a big number of them, and chances are you won't use them. However, if you find yourself needing to plot some of the data saved in your TTrees and TChains, take a quick look at these functions' definitions inside the TWaveform.cxx file. You may find

[17] Work is being done to include the rise position of all the pulses.
[18] The first string in the vector refers to the first pulse in the waveform, and so on

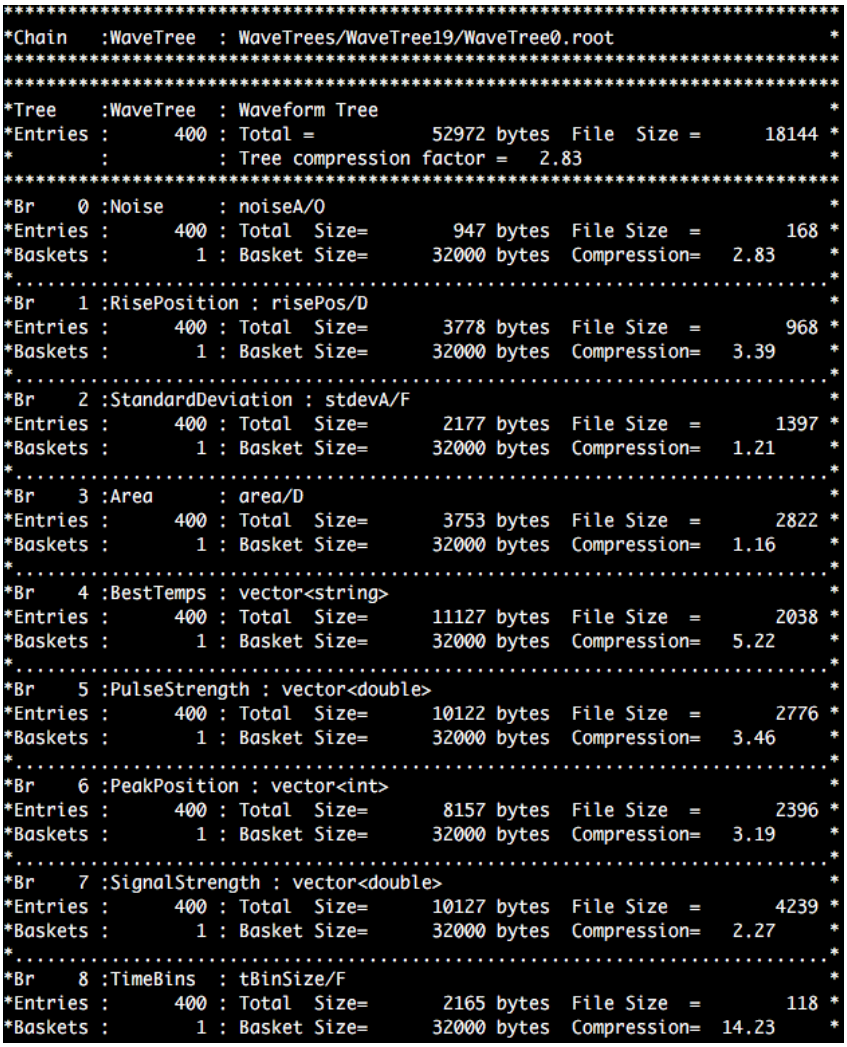


Figure 12: A screenshot taken of the output of line 6 in the Create Trees piece of code above (chain->Print()). Here we get to see the contents of the first TTree linked into the TChain.

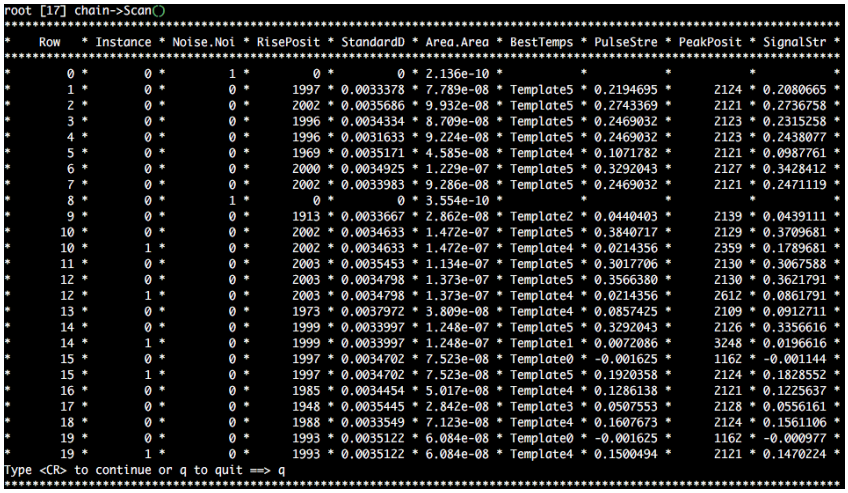


Figure 13: A screenshot taken of the output of line 7 in the Create Trees piece of code above (`chain->Scan()`). Here we get to see some of the first data entries in the TChain generated.

in here a method that can already and easily provide you with the histogram you need. If you don't find what you need within these methods, `TWaveformSpace` also provides you with a function that can easily retrieve a TTree's and TChain's entries:

```
treeData GetTreeData(int entry, TChain* chain)
```

By just passing it the chain (or tree) that has the data, and the number of the entry you want to retrieve, this function will return a treeData object (as defined in Section 2.1) with all the information saved in that chain's (or tree's) entry. You can therefore use this method to build the histogram of your choice, or for that case, any visual representation of your data.

That is all for `TWaveform's` and `TWaveformSpace's` function descriptions. Following, we will guide you through a real-life example of how to use all these methods effectively towards the analysis of a collection of waveform signals.

3 Example

In this chapter we will take a step by step approach on how to set up everything in the `TWaveform` package to make it capable of processing different types of data waveforms. We will also go through the implementation of the Template and Tree creation methods, and everything that is needed for `TWaveform` to run smoothly with your data. Moreover, we will guide you through the exact steps taken to analyze the PMT waveforms from Section 1.2 with the `TWaveform` package. Here is a quick overview of the steps we will carry out through this process:

1. Change file address parameters in the TWaveformConstants.cxx file.
2. Create a Read[Format] method for your specific data files' format, and add it to `TWaveform's ReadFile()` method.

3. Develop a Get[FileType]Filename method that associates to your data files' specific naming system and add it to `TWaveform's GetFilename()` member function.
4. Fine tune slopeSigma in TWaveformConstants.cxx to calculate the rise positions of the waveforms.
5. Create pulse templates.
6. Check a few fitted waveforms. If not fitting correctly, change fitting parameters in TWaveformConstants.cxx.
7. (optional) Modify minimumPulseHeight and noise filter parameters in the TWaveformConstants.cxx file.
8. Run the TTrees and TChain generation methods for your waveforms.
9. (optional) Modify Tree Hist parameters in TWaveformConstants, and create desired histograms.

3.1 Setting File Address Parameters

Inside the `TWaveform` package, you will find that there is a file called TWaveformConstants.cxx. Inside this file are a number of constant variables, under the TWaveformConstants namespace, that define key settings on how `TWaveform` will perform certain tasks. It is crucial to modify some of these variables to relate to the environment in which you will be running `TWaveform`. Remember, also, that any time you change any of the `TWaveform` files, it is necessary to run the 'make all' command on the terminal once the change has been saved.

As displayed in the list above, the first variables in TWaveformConstants that we will change are the file address parameters. These parameters help point the `TWaveform` objects to the data files containing the waveforms' signal information. To change these, first open TWaveformConstants.cxx with any text editor. You will see that there are three variables under the file address parameters; directoryAddress, fileType, and fileName. Lets see how to modify these values with an example.

Say you have a set of directories containing the waveform data files, and they are all saved with the same naming scheme, as follows:

```
// Waveform files in Run125
"/Users/Documents/Run125/data/C2wave0000.txt"
"/Users/Documents/Run125/data/C2wave0001.txt"
...
"/Users/Documents/Run125/data/C2wave7659.txt"

// Waveform files in Run134
"/Users/Documents/Run134/data/C2wave0000.txt"
"/Users/Documents/Run134/data/C2wave0001.txt"
...
"/Users/Documents/Run134/data/C2wave6534.txt"
```

We now set the values of the file address parameters as:

```
extern const std::string directoryAddress = "/Users/Documents/Run"
extern const std::string fileName = "/data/C2wave"
extern const std::string fileType = ".txt"
```

Notice here how we didn't include the numbers of the parent directory, nor the file names, in the parameters. In short, this is so that we can iterate through the files and directory numbers. This will be easier to understand once we create the GetFilename method in Section 3.3.

3.2 Creating Read Method

There are a large number of ways in which data files containing the waveform information can be formatted. For this reason, before we start using the `TWaveform` class, we need to make sure that we are able to read the data files into `TWaveform` objects. To do this, we need to add a file reading method into the `TWaveform` class that is compatible with the file formats of our data files. For our specific example, the txt files are formatted as:

```
LECROYWR104MXi-A 52137 Waveform
Segments 1 SegmentSize 5002
Segment TrigTime TimeSinceSegment1
#1 03-Apr-2017 14:53:22 0
Time Ampl
time[0] v[0]
time[1] v[1]
time[2] v[2]
...
time[n] v[n]
```

As such, we now implement a new method into `TWaveform's` class definition that is capable of reading this type of files, and assigning their data to a `TWaveform` object's data members. We named this function `ReadTXT()`:[19]

```
1  bool TWaveform::ReadTXT(std::string fName1, bool debug){
2    float tIn, vIn;
3    std::string line;
4    int nChar(0);
5    std::ifstream file(fName1.c_str(), std::ifstream::in);
6    if (!file.good()){
7      return false;
8    } else{
9      std::getline(file, line);
10     std::getline(file, line);
11     std::getline(file, line);
12     std::getline(file, line);
13     std::getline(file, line);
14     int i(0);
15     while(std::getline(file, line)){
16       nChar = sscanf(line.c_str(), "%f %f", &tIn, &vIn);
17       this->vA[i] = vIn;
18       this->t[i] = tIn;
19       i++;
20     }
21     this->nSamples = i;
22     this->tBinSize = t[1] - t[0];
23   }
24   file.close();
25   return true;
```

[19] The ReadTXT() method shown here is a simplification of the one actually implemented and shows the essential parts of the function. For the complete method, check the TWaveform.cxx file.

```
26  }
```

Let's now decompose this method into its essential parts:

- lines 5-8: Creates a stream object for the file and checks if the file exists/was read successfully.
- lines 9-13: Reads and discards the header of the txt files, which in this case don't contain data that will be read into the `TWaveform` object.
- lines 15-20: Reads the data lines of the txt files and assigns the values read to the `TWaveform` object's t and vA data members accordingly.
- lines 21-22: Sets the `TWaveform` object's nSamples and tBinSize data members to their respective values.
- lines 24-26: Closes the file and returns true after file has been successfully read.

While the method you create doesn't have to follow the same syntax or form, it should be able produce the same results highlighted in the points above. After you have done so, it is necessary to modify the `ReadFile()` method to include our new member function, as:

```
1
2   bool TWaveform::ReadFile(std::string fName){
3     bool read;
4     if(this->fileType == ".CSV"){
5       read = this->ReadCSV(fName);
6     }
7     else if(this->fileType == ".txt"){
8       read = this->ReadTXT(fName);
9     }
10    return read;
11  }
```

The purpose of this last step is to make `ReadFile()` be able to read your specific format, as specified by the `TWaveform` object's fileType data member (in this case set to ".txt"). This step is crucial as many of `TWaveform's` methods will make use of this `ReadFile()` method.

3.3 Developing GetFilename Method

Now that we can read waveform data from txt files with `TWaveform`, we need to work on implementing a method that can be used as a shortcut to iterate through various file and directory numbers. Inside `TWaveform`, a method, called `GetCSVFilename(int waveNum, int dirNum)`, is included. This function returns a string with the following structure:

```
"[directoryAddress][dirNum][fileName][waveNum][fileType]"
```

which, using the address parameters of our txt example, would translate to:

```
"/Users/Documents/Run[dirNum]/data/C2wave[waveNum].txt"
// And setting dirNum to 125 and waveNum to 8:
"/Users/Documents/Run125/data/C2wave8.txt"
```

This method looks like it is working fine. However, if you have a different file-naming system, this method will not work well for you. An example of this mishap, if you can remember from Section 3.1, are our txt file-names, which have a slightly different syntax from the file-names shown above. The only difference between them is that, in our case, fileName is always followed by four integers. That means, that if we set waveNum to 8, we should get the following file address instead:

```
"/Users/Documents/Run125/data/C2wave0008.txt"
```

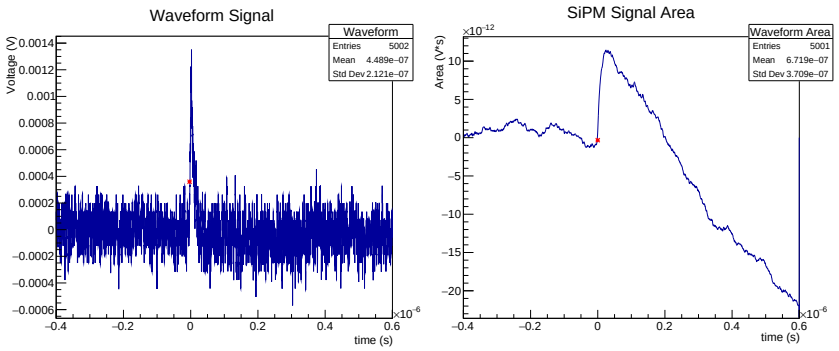
It should now be evident that we need create a getFilename method that can adjust to our file naming system. We will call this new method `getTXTFilename()`, and implement it as follows:

```
1
2  std::string TWaveform::GetTXTFilename(int waveNum, int dirNum){
3    std::ostringstream sDir, sWave;
4    sDir << dirNum;
5    sWave << waveNum;
6    std::string direc, fileName, dirNumStr(sDir.str()),
         waveNumStr(sWave.str()), zeros;
7    direc = this->directoryAddress;
8    for (int i = 0; i < 5 - waveNumStr.length(); ++i){
9       zeros += "0";
10    }
11   if(dirNum == -1){
12     dirNumStr = "";
13   }
14   fileName = direc + dirNumStr + this->genericFileName + zeros +
         waveNumStr + this->fileType;
15   return fileName;
16 }
```

The basic difference between this new method and `getCSVFilename()` is that, in here, a set of zeros are added before the waveNum, and so the total number of integers in the file-name is equal to four. Note also that when dirNum is given as -1, the function won't attach any number between the directory address and the filename. As with the reading method before, you can create your getFilename method in any way you want, as long as it returns a string pointing to your files.

To finish implementing our new getFilename method, we need to add it into `getFilename()`'s function definition. This is done as follows:

```
1
2  std::string TWaveform::GetFilename(int waveNum, int dirNum){
3    std::string filename;
4    if(this->genericFileName == "/DS"){
5      filename = this->GetCSVFilename(waveNum, dirNum);
6    } else if(this->genericFileName == "/data/C2wave" ||
         this->genericFileName == "/data/C1wave"){
7      filename = this->GetTXTFilename(waveNum, dirNum);
8    } else{
9      filename = this->GetCSVFilename(waveNum, dirNum);
10   }
11   return filename;
12 }
```



(a) PMT waveform. (b) PMT waveform's integral

Figure 14: The plot of a PMT waveform (a), and its integral (b). The red marks on both plots refer to the waveform's pulse rise position.

As you can see from the snippet of code above, the getFilename method that is called when using `GetFilename()` is decided by the `TWaveform` object's genericFileName data member. Then, if the `TWaveform` object we are using has its genericFileName set to "/data/C2wave", `GetFilename()` will proceed to use our newly created `GetTXTFilename()` method. It is absolutely necessary to have a getFilename method that refers to your data files, as this function is used in several other of `TWaveform's` methods.

3.4 Tuning slopeSigma

At this point, we are now capable of reading data files easily into our `TWaveform` objects. It is therefore time to start tuning everything that we need to create templates and analyze our waveforms. Fortunately, to be able to create the pulse templates, we only need to tune the slopeSigma variable in TWaveformConstants.cxx. The reason we have to do this is that different signals usually come with different voltage scales (y-axis). As such, we need to set the minimum change in a waveform's integral's slope that alludes to the rise of a new pulse.

To understand this more clearly, lets take a look at one of the PMT's waveform, and a plot of its integral. As you can see in Fig.14, there is a visible slope change in Fig.14b when the pulse starts to rise, visible in Fig.14a. We thus have to set slopeSigma to be the minimum slope change that is used to identify a pulse's rise.

The most recommended way to do this is to use `TWaveform` to look at a couple of waveform's in your data, and check the change in their integral's slope when their pulses start to rise. This is made easier by making use of `TWaveform's FindSlope()` and `PlotIntegral()` methods, described in Section 2.3 and the source code, respectively. You can thus use `FindSlope()` to calculate the difference between the slope of the waveform's integral before and after the pulse has started to rise. After deciding on a value, head to the TWaveformConstants.cxx file and plug your value for the slopeSigma variable. Do not worry about getting a very precise slopeSigma value. As you will see in the next section, the main purpose of this task is to calculate the rise position of the template pulses. As such, you will only know if slopeSigma was set correctly after generating the templates.

3.5 Generating Pulse Templates

The next step in our voyage to analyze our data files is to create a set of template pulses. With the steps that we have taken, it should be fairly easy to start using `TWaveform's` template generation methods. To understand this step fully, it might be beneficial to check how these methods work in Section 2.4. Above all, you should try to understand the variables that are passed to these functions.

After going through the implementation of the `AvgAreaTemplates()`, you might be a bit puzzled as to what values to use for the minArea and maxArea variables. Lucky for you, there is a function inside the `TWaveformSpace` namespace that will help you get a sense of what values to use. This function is called `GetIntegralLimits()`, and its purpose is to return a vector with the minimum and maximum definite integral values found in the waveforms of the directory passed to it. In other words, it will return the areas below both the smallest and largest waveforms in the directory.[20] These limits will give you a sense of the range of areas found in your waveforms. From it, you can decide what range of areas you want to create your templates from. If you wanted to include all your waveforms in the template generation process, you should simply use these values as the minArea and maxArea variable, like this:

```
root [0] gSystem->Load("TWaveform.so");
root [1] TWaveform wave;
root [2] std::vector<double> integralLim =
    TWaveformSpace::GetIntegralLimits(125);
root [3] std::vector<TWaveform> templates =
    AvgAreaTemplates(125, "Template/TemplateTXT.root",
    integralLim[0], integralLim[1], 4, "Template");
```

In essence, this is all that needs to be done to get a set of pulse templates from a collection of waveforms. In this example, we created 4 sets of templates from our directory #125, with a common name of "Template" (so they are named "Template0", "Template1", etc.). All these templates were then assigned to the `TWaveform` vector, templates, as well as saved to the root file "TemplateTXT.root". Once you have saved the templates into the root file, it is very simple to retrieve them by just using `TWaveform's ReadTemp()` method.[21]

In order to simplify the fitting process that will be shown in the next section, it is highly recommended to change the values of the "Template Generation Parameters" in TWaveformConstants to reflect the file address, template name, and number of templates you chose in the template creation process above.

We now have our pulse templates, but it is not yet time to call victory! It is very important to check if the templates were created correctly, and if the slopeSigma we defined in the last section has a correct choice of value. To do this just simply draw the templates and look at their rise and peak positions;

```
root[4] templates[0].Draw();
root[5] templates[0].MarkRise();
```

[20] integralLim[0] will be the minimum area, while integralLim[1] will be the maximum.
[21] To learn how to use `ReadTemp()` go to Section 2.4

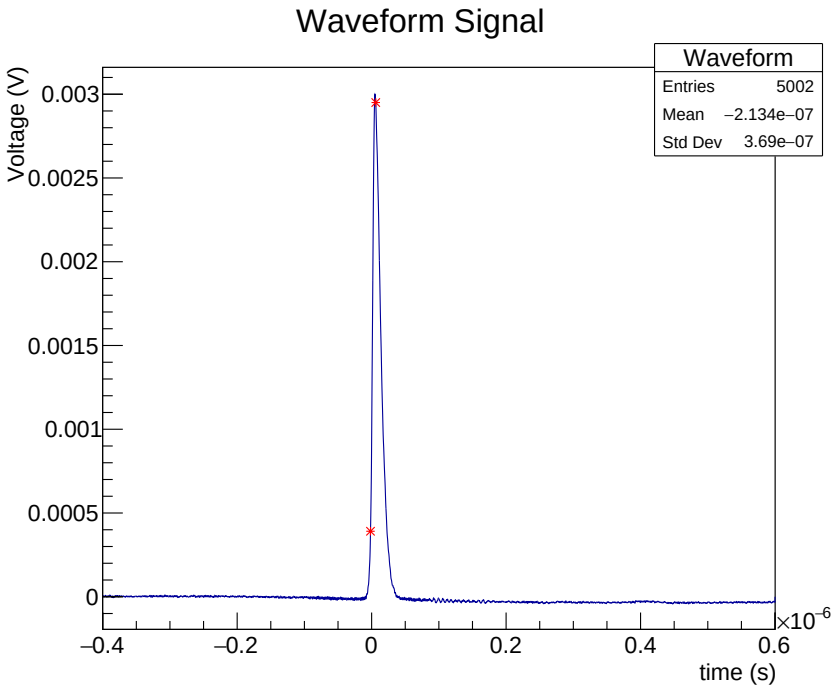


Figure 15: A plot of the first pulse template created from the txt files in directory 125. The red asterisks mark the peak and rising positions of the pulse.

After inserting this piece of code, you should get a plot of the first of your generated templates, reminiscent to that of Figure 15. Notice here that the pulse's rise position has a small margin of error. This is, however, good enough of a calculation for our purposes. On the other hand, if the rise position of your templates was not calculated, or you are not pleased with its precision, you might have to re-adjust your slopeSigma constant.

3.6 Fitting Waveforms

Once you are satisfied with the template pulses generated through the previous section's process, it is time to fit some waveforms! This can be easily done with `TWaveform` in just a few lines of code. We simply need to read a data file into a `TWaveform` object and call the `AnalyzeWave()` member function, such as:

```
root [0] gSystem->Load("TWaveform.so");
root [1] TWaveform wave;
root [2] wave.ReadFile(wave.GetFilename(20,125));
root [3] wave.AnalyzeWave();
root [4] wave.DrawBestFit();
root [5] wave.MarkRise();
```

In here we also use `DrawBestFit()` to plot the waveform along with its best calculated fit. The results of this snippet of code are shown in Fig.16. By looking at this plot, `TWaveform's` capabilities start to become more apparent. Not only was the waveform fitted with a great degree of accuracy, its peak and rise position were also calculated and recorded with relative precision.

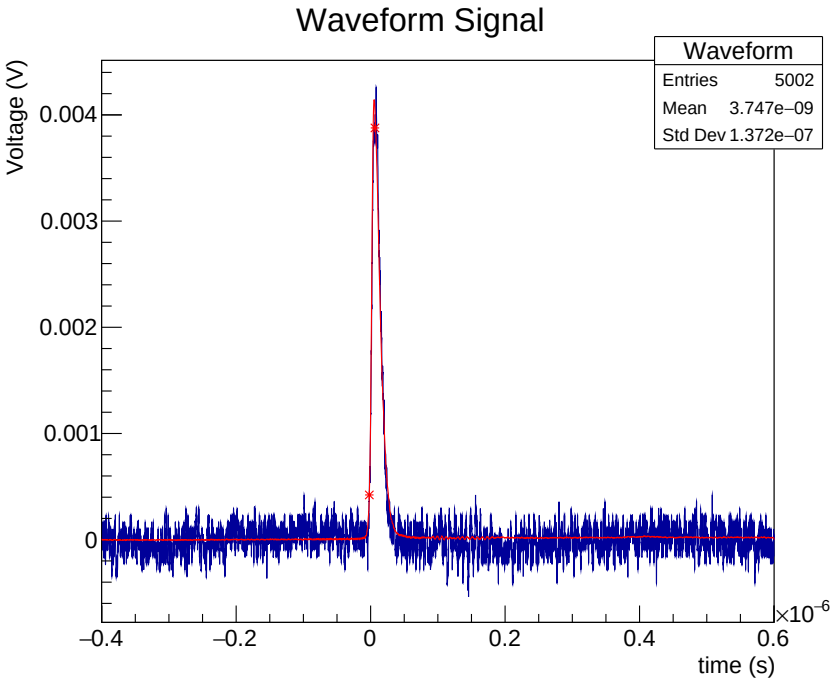


Figure 16: A plot of waveform #20 in our txt data files being fitted with the pulse templates in the "TemplateTXT.root" file. Once again, the red asterisks mark the peak and rising positions of the pulse.

Needless to say, we got a bit lucky with our previous fit, and not all waveforms will be fitted with the same level of accuracy. Nonetheless, if you find, after checking a couple of fitted waveforms, that they are not being fitted with the level of precision desired, you can try to change some of the "Fitting Parameters" in TWaveformConstants to improve the fits. Lets take a quick look at some of these parameters and what they represent:

- scaleResolution: The precision with which templates are scaled to fit the waveforms. If, for example, this value is set to 1, `TWaveform` will calculate the fits for each template with at least an integer difference in how they are scaled. The lower the value given to the resolution variables, the higher the precision of the fit, and the more time it will take for a the fits to be evaluated.

- timeResolution: Similar to scaleResolution, it defines the precision by which each template is offseted (in the x-axis) to calculate a waveform's best fit. Contrary to what its name suggests, this value doesn't refer to the difference in time between each consecutive offset given to the templates. Instead, it refers to the minimum number of samples by which a template is shifted to calculate a its fit with a waveform.

- timeStepDivider and scaleStepDivider: Makes reference to the number of χ^2 calculations that will be made for each template fit (with different scales and time offsets). By increasing these values, the precision of the fits will increase accordingly. However, an increase in said values will also increase the processing time for each fit.

- timeFittingLim and scaleFittingLim: The limits to which each template will be scaled and offset to fit the waveforms.
- maxPulses: As its name suggests, the maximum number of pulses that `TWaveform` will try to fit into its waveforms.
- maxChiNorm: The maximum normalized χ^2 value for a calculated fit to be deemed as a 'good fit'.

As seen from the description of the parameters above, there is a trade-off between fitting precision and processing time. When fitting your waveforms, it is a good idea to experiment with different fitting parameter values until you reach a preferred balance between precision and processing time.

3.7 Noise Filter Parameters (Optional)

After being able to fit our waveforms with the template pulses, we now want to start setting everything up to create a set of TTrees with our waveforms' information. In this section we will look at what parameters we need to adjust in order for the `AnalyzeWave()` method to determine if a waveform is pure noise, and if the pulses being fitted are actual pulses, or just small fluctuations in the signal that are being over-fitted.

As you can see from this section's title, this part is completely optional, and you can skip ahead to the next section without problem. However, if you want to customize `TWaveform` to tackle your data to the fullest and maximize the time efficiency for the TTree creation methods, this step is highly recommended.

Lets tackle each one of these parameters one by one:

- maxAvgForNoise: The maximum value that the total average of a waveform can have, and still be considered as pure noise.
- maxAreaForNoise: Akin to maxAvgForNoise, this value sets the maximum area that can exist below a waveform for it to still be considered as noise.
- noiseFilterMethod: This parameter defines how waveforms will be processed to determine if they are noise. There are only two options for this variable; "Area" and "Avg". If "Area" is chosen, `TWaveform's` analyzing methods will calculate the waveforms' areas and compare them to maxAreaForNoise to determine if they are pure noise. If "Avg" is used instead, the total average of the waveforms will be calculated and compared to maxAreaForNoise.
- minimumPulseHeight: This value sets the minimum height a pulse should have for it to be considered as a real pulse in the fitting method, instead of just a fluctuation of the waveform's noise. If this value is set to 0, the minimum height for a pulse to be considered 'real' will be calculated as two standard deviations of the waveform's signal.

3.8 TTree and TChain Generation

In this section we will provide you with the easiest way to start creating trees of your waveform data. First on, you may want to modify some of the "Tree and Chain Generation Parameters" inside of TWaveformConstants. Lets take closer look at these:

- treeSaveStep: This value refers to the maximum number of waveforms that are going to be assigned to a single tree. The reason for having this setting is that, sometimes, you may want to create a TTree, or TChain, of a very large number of waveforms, which could thus take a considerable amount of time to process. By having a limited amount of waveforms for a single tree, we avoid the risk of losing all the processed data if an error, or any other difficulty, arises. If such a thing happens, then, only one tree worth of data would be lost (or corrupted), and you wouldn't have to process all the waveforms again, but just the ones contained in that single tree.

- treeDirectoryAddress: Simply put, the address of the directory in which the TTrees' root files will be saved.

- treeName: The name given to the TTrees created. This name is also used by some of the TTree generation methods as the file-name for the root files containing the trees. In these cases, the root files will have their file-addresses in the form of: "[treeDirectoryAddress][dirNum]/[treeName][treeNum].root", where dirNum represents the directory number of the waveforms that are being analyzed, and treeNum is just the index of the TTree being generated.

- chainDirectoryAddres: As its name suggests, the directory address of where the TChains' root files will be saved.

- chainName: Just as treeName, represents the name give to the TChains and their root file-names. As such, TChains created by the method that will be shown below are given the following file addressess: "[chainDirectoryAddress]/[chainName][dirNum].root".

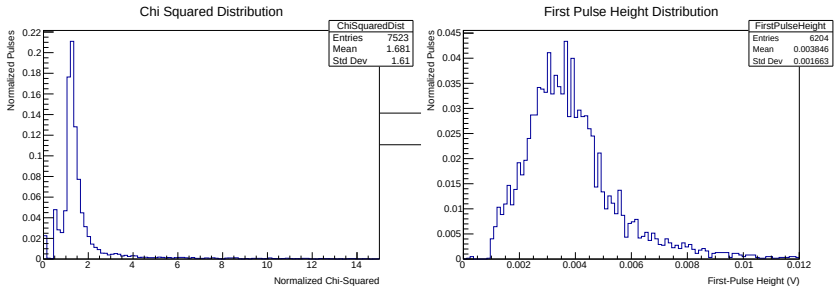
We are now finally ready to start creating some TTrees and TChains of our waveforms! This process is made particularly simple by making use of `TWaveformSpace's` function called `CreateTreeChain()`. This method will generate a number of TTrees containing useful information about your waveforms,[22] saves them into your tree directory, and then creates and saves a TChain that links all these TTrees together. All you need for this processing magic to happen is to run the following command;

```
root [2] TWaveformSpace::CreateTreeChain(dirNum);
```

where, as before, dirNum refers to the directory number of the waveforms you want to analyze. The TTrees will then start to be filled and saved to their respective root files, whose path-names are described in the treeName parameter above. Lets try to run this procedure with the example we have been using throughout this chapter:

```
root [2] TWaveformSpace::CreateTreeChain(125);
wave: 0 - Done
wave: 1 - Done
...
wave: 7659 - Done
Tree number: 19 created
```

[22] Check Section 2.5 for an in-depth explanation about all the data saved in these TTrees.



(a) Histogram of the χ^2 distribution for all the fitted waveforms

(b) Distribution of the waveforms' first pulse height

Figure 17: PLots of the histograms built by the piece of code on Section 3.9, using the the data from the PMT TChains.

Just like that, we have our TTrees and TChain for all the 7659 waveforms contained in the Run125 directory. We can find the TTrees in the directory "WaveTrees/WaveTree125", with file-names: "WaveTree0.root", "WaveTree1.root", etc. We can also locate our newly created TChain with the following address: "WaveChains/WaveChain125.root".[23] It is important to add that, in order for the TTrees and TChains to be generated successfully, their respective directories have to already exist.

3.9 Data Analysis Histograms

Having already created a TChain with useful information about all our waveforms, we might want to build a visual representation of our data. In `TWaveformSpace` you will find that there are a myriad of different histogram building methods that compare different properties from your set of data. There are functions for histograms that show the time distribution of after-pulses, others that compare the first pulse's height to the waveform's χ^2 fit, and so on. If you, otherwise, want to create your own histograms, `TWaveformSpace` provides a method to facilitate this process called `GetTreeData()`.[24]

Lets plot a couple of histograms for the TChain we just generated by making use of `TWaveformSpace's` available methods:

```
root [2] TH1F *chiHist = TWaveformSpace::ChiSquaredHist(
    "WaveChains/WaveChain125.root");
root [3] TH1F *firstPulseHeight =
    TWaveformSpace::FirstPulseHeight(
    "WaveChains/WaveChain125.root");
root [4] TH2F *chiAndHeight =
    TWaveformSpace::ChiSquaredAndPulseHeightHist(
    "WaveChains/WaveChain125.root");
root [5] chiHist->Draw("hist");
root [6] firstPulseHeight->Draw("hist");
root [7] chiAndHeight->Draw("surf2");
```

The plots in Figs. 17 and 18 display the histograms created in the snippet of code above. These histograms show but just a small chunk of the information you can attain and represent through `TWaveform's` methods. We now pass on the torch to you, the now pronounced `TWaveform` master

[23] To learn how to access these TTrees and TChains individually, go to Section 2.5
[24] Go to Section 2.5 for more information on this function.

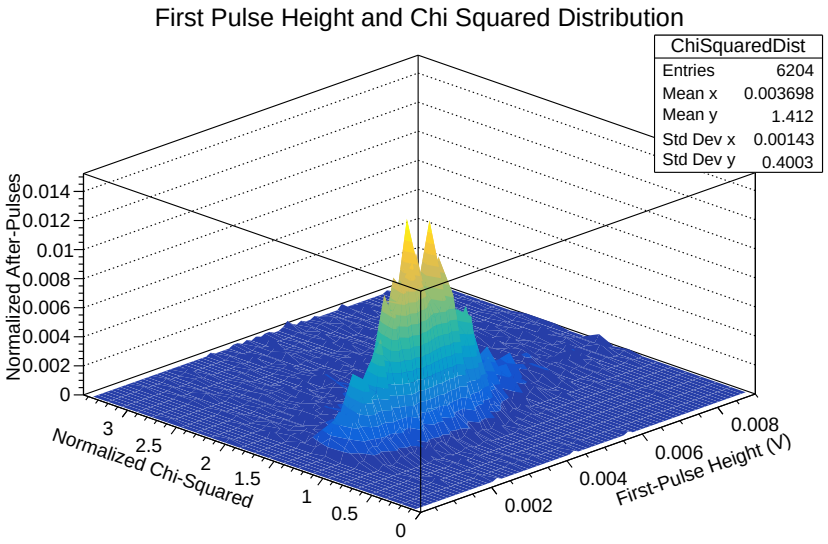


Figure 18: A plot of the 2D histogram built in Section 3.9, which compares the normalized χ^2 of all of the PMT's fitted waveforms with their first pulse's height.

analyst, to get creative with this powerful tool, and use it to uncover the evasive secrets of nature you ought to encounter.

Remember. It's dangerous to go alone! Take this.

4 Concluding Remarks

The `TWaveform` software is the result of an almost one-year project that started as an approach to analyze the signal outputted from particle detector designs based on scintillating fibers and silicon photomultipliers. This software was created with the specific intent of studying these particle detector designs for their implementation into CERN's CMS experiment. In particular, `TWaveform` had the main objective of identifying multiple pulses within a waveform, and use it to study the distribution of after-pulses that were generated by the SiPM's interaction photons.

All the work contained inside the `TWaveform` package, including this guide, was written solely by Carlos A. Osorio, while working at an internship with UCSB professor, Dr. David Stuart, at the university's High Energy Physics Group. Dr. Stuart supervised the project in its entirety, and helped shape the vision of what the `TWaveform` software would become. He guided me through the process of turning `TWaveform` into a more expansive tool that would be able to analyze the signal from a myriad of different devices, and also aided me when confronting major hurdles. There is no doubt that without his help, building the `TWaveform` package would have not been possible.

The implementation of the `TWaveform` class, and all its associated modules, were started with very little prior programming experience. As such, you will find that the structure of the software and its algorithms, in many turns, does not follow industry standards. You may also evidence an improvement in the implementation of the methods included within these modules as the project advanced. This is evidence of the learning curve experienced by the developer (me) as more time was spent

expanding the software's functionality. Moreover, there are still a lot of adjustments that, if time permitted, are yearned to be made.

A lot of work and dedication went into this project. I hope that it will be of great use to you in whatever journey lies ahead!

Index