Some Analysis of Vertical Drift Chamber Data Using ROOT

Lloyd Joseph Snow

August 5, 2009

Abstract

Dr. David Armstrong and several others have been making and testing vertical drift chambers (VDCs) for use at Jefferson Lab. This summer I am analyzing and interpreting some of the data from one of the VDCs using ROOT, a software package developed at CERN specifically for data analysis. The result of my work is a ROOT macro which automates a few tedious analysis tasks which had previously been done by hand or not at all.

1 Background

1.1 The rest of the project

1.1.1 Vertical Drift Chambers

A vertical drift chamber is a container housing a gas mixture and many regularly spaced conducting wires. The VDCs of Dr. Armstrong's group presently use a gas mixture of 65% Argon and 35% ethane and have 280 wires separated into two layers sandwiched between foils kept at 4000 volts. A VDC detects particles passing through it when they ionize the gas in the chamber. The ionized gas and the liberated electrons then flow based on their charge and other factors. Liberated electrons flow to the nearest wire in the chamber causing an electrical signal upon contact.

The data harvested from the chamber are these signals in the form of recorded time values for each wire in the chamber. There are trigger mechanisms involved, two plastic scintillators, that start and stop the recording of the time values for the wires. One start followed by a stop is a single *event* for the chamber and the data related to that event is a *time value*, in picoseconds, for every wire in the chamber. If a wire never signaled during the event, then it's time value is by default zero. Thus the *nonzero* time values are of wires which signaled because a passing particle ionized gas whose liberated electrons eventually drifted towards that one wire.

The purpose of the start signal is to tell the hardware to start actually recording any incoming signals. The time value for a given wire is the time from the first signal from the wire after the start signal until the moment that the stop signal is given. This means that the wires with the largest time value are in fact the wires that were hit earliest after the start signal. The actual time value of the time from the start signal to the time of the wire's signal is not the actual data that will be recorded, but it could be inferred if desired by subtracting the time value corresponding to the earliest moment after the start signal. That "earliest moment" time value is reliably calculable from the data and is referred to as the *t-naught* value for a given wire.

A *track* in the VDC is the calculated path of the passing particle. Tracks are reconstructed from the run data.

1.1.2 Cosmic Ray Tests

Presently the VDC in production by Dr. Armstrong's group is being tested cosmic rays to trigger the start mechanisms, and thus for a given run of the chamber, each event should involve the passing of a cosmic ray. This data is compiled automatically into a ROOT file in a certain format which my macro is written to read and analyze. A *run* is the ROOT file collecting the events and their sets of time values from one continuous operation of the VDC.

1.2 Software

1.2.1 ROOT

ROOT, Roots Object Oriented Technologies, is a C++ library or program that is developed at CERN and used widely in many physics projects. ROOT can be included in a C++ project as a software library or any ROOT distribution comes with binaries that can be executed to use ROOT in an interactive program mode that utilizes CINT to code dynamically from a command line or also dynamically from user-written macros loaded from files.

The primary virtue of a macro is that they can generally be run without portability concerns on any system that has ROOT installed. A macro is loosely C++ code, but need not be enough to compile independent of its usage in CINT.

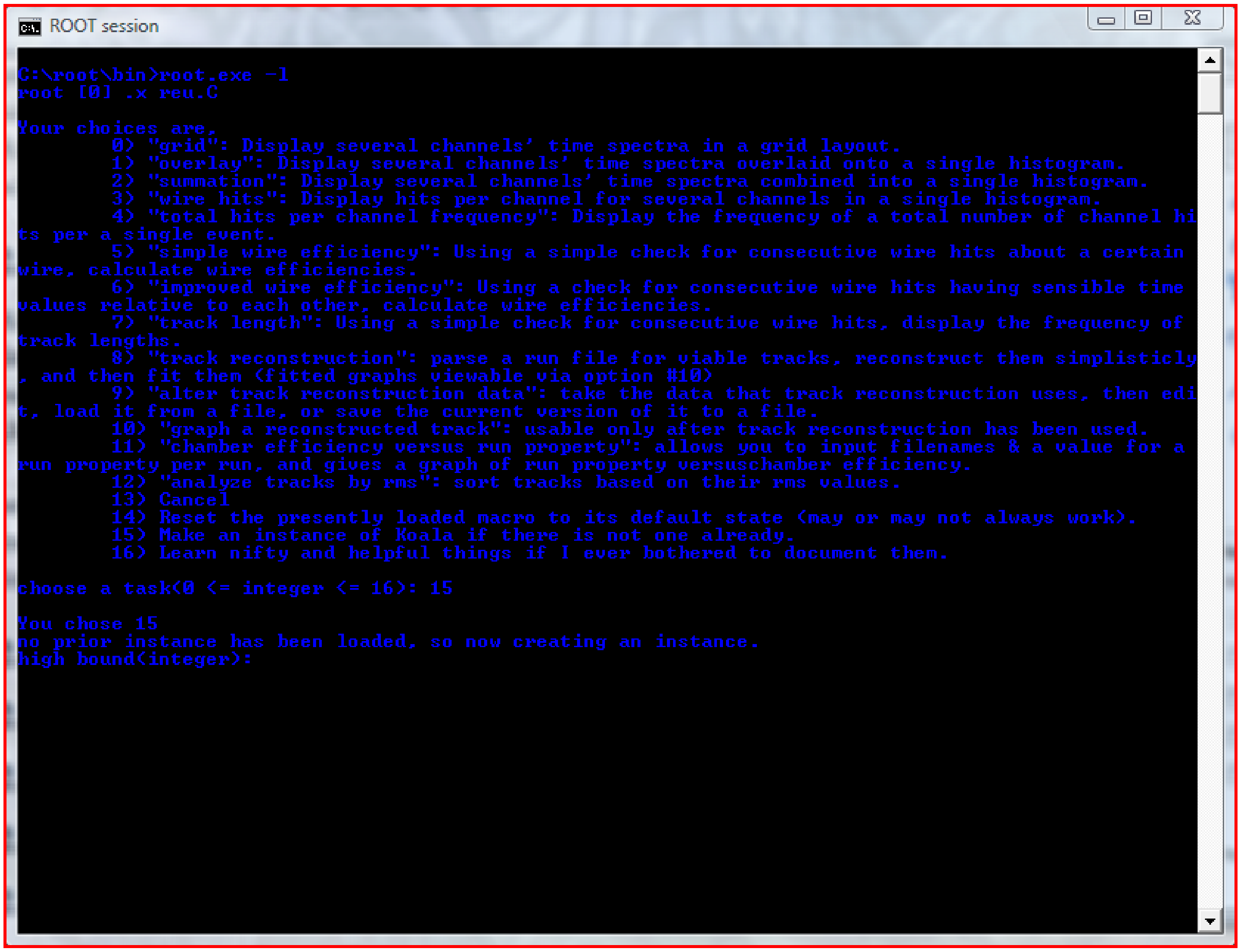
1.2.2 ROOT uses CINT

CINT, the C Interpreter, is supposed to be a C++ interpreter that can take C++ code and interpret it dynamically much like many other interpreted-by-design programming languages such as Lua. CINT does not perfectly support the present C++ standard, and anyone used to the C++ that would work on modern compilers will periodically find themselves confronted by deficiencies between CINT and those same modern compilers in what code does and does not compile. In short, knowing C++ beforehand can work against you just as much as it can work for you when it comes to CINT.

2 Results

The result of my industrious toil is a file named "reu.C". In the interests of being portable, "reu.C" is a ROOT macro, and presents the user with a text-based interface. It gathers together the common code necessary for all the various analysis tasks that I was instructed to automate. From the main menu it shows, the user is able to select from the various tasks. After a task is executed, the program returns to the ROOT prompt, but the data from the last run of the macro is still in scope and accessible and can be used again by calling the function "reu()" from the ROOT prompt.

Many of the figures to follow show output gleaned from C775_1197.root which is a ROOT file representing a certain run with the VDC at 4050 Volts and the threshold voltage at 1.0 Volts.



```
C:\root\bin>root.exe -l
root [0] .x reu.C

Your choices are,
        0) "grid": Display several channels' time spectra in a grid layout.
        1) "overlay": Display several channels' time spectra overlaid onto a single histogram.
        2) "summation": Display several channels' time spectra combined into a single histogram.
        3) "wire hits": Display hits per channel for several channels in a single histogram.
        4) "total hits per channel frequency": Display the frequency of a total number of channel hi
ts per a single event.
        5) "simple wire efficiency": Using a simple check for consecutive wire hits about a certain
wire, calculate wire efficiencies.
        6) "improved wire efficiency": Using a check for consecutive wire hits having sensible time
values relative to each other, calculate wire efficiencies.
        7) "track length": Using a simple check for consecutive wire hits, display the frequency of
track lengths.
        8) "track reconstruction": parse a run file for viable tracks, reconstruct them simplisticly
, and then fit them (fitted graphs viewable via option #10)
        9) "alter track reconstruction data": take the data that track reconstruction uses, then edi
t, load it from a file, or save the current version of it to a file.
        10) "graph a reconstructed track": usable only after track reconstruction has been used.
        11) "chamber efficiency versus run property": allows you to input filenames & a value for a
run property per run, and gives a graph of run property versuschamber efficiency.
        12) "analyze tracks by rms": sort tracks based on their rms values.
        13) Cancel
        14) Reset the presently loaded macro to its default state (may or may not always work).
        15) Make an instance of Koala if there is not one already.
        16) Learn nifty and helpful things if I ever bothered to document them.

choose a task(0 <= integer <= 16): 15

You chose 15
no prior instance has been loaded, so now creating an instance.
high bound(integer):
```

Figure 1: Main menu of reu.C

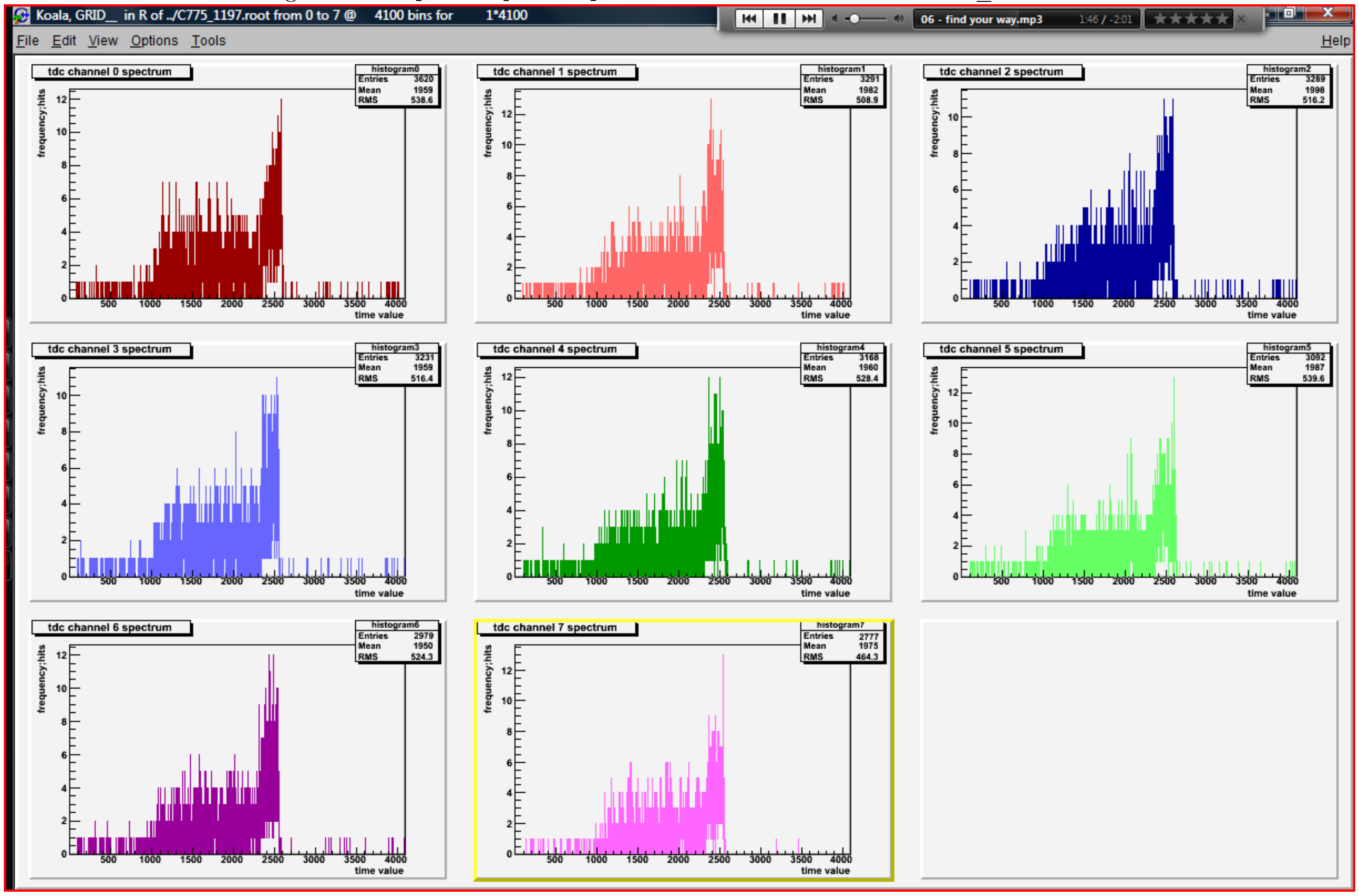
2.1 Visualize the frequency of occurrence of specific time values for a given run

As mentioned before, an event as found in the ROOT file representing a run of data from the chamber is a collection of time values garnered from the vertical drift chamber. The "time spectrum" of a run is the histogram showing the frequency of occurence of time values throughout the run. For a working vertical drift chamber, the time spectra are expected to have a characteristic shape. The characteristic shape of a time spectrum stems from the pattern of acceleration that the freed electrons should be experiencing as they approach the wire. Aside from background noise, the spectrum should resemble a leading hump or plataeu followed by and conjoined to a much higher hump. The first part are time values from free electrons that were freed further away from the wire, and the second part are time values from free elections that were freed much closer to the wire. Bear in mind that these are the recorded time values only and so the largest time values are from the wires that signalled earliest.

The first task I had to do was to make graphs of the characteristic time spectra. Three modes of display were requested: One mode displaying time spectra of time values for individual channels in separate histograms arranged across the screen as a grid, a second mode displaying time spectra of time values for individual channels all overlaid on the same histogram, and a third mode displaying the time spectrum of all the channels together. These modes are options 0, 1, and 2 from the main menu.

The second and third modes provide simple ways to visually verify that all the time spectra for all the wires are sharing the same or similar characteristics.

Figure 2: Output of option 0 performed on channels 0-7 of C775_1197.root



	Entries	Mean	RMS
histogram0	3620	1959	538.6
histogram1	3291	1982	508.9
histogram2	3289	1998	516.2
histogram3	3231	1959	516.4
histogram4	3168	1960	528.4
histogram5	3092	1987	539.6
histogram6	2979	1950	524.3
histogram7	2777	1975	464.3

Figure 3: Output of option 1 performed on channels 0-7 of C775_1197.root

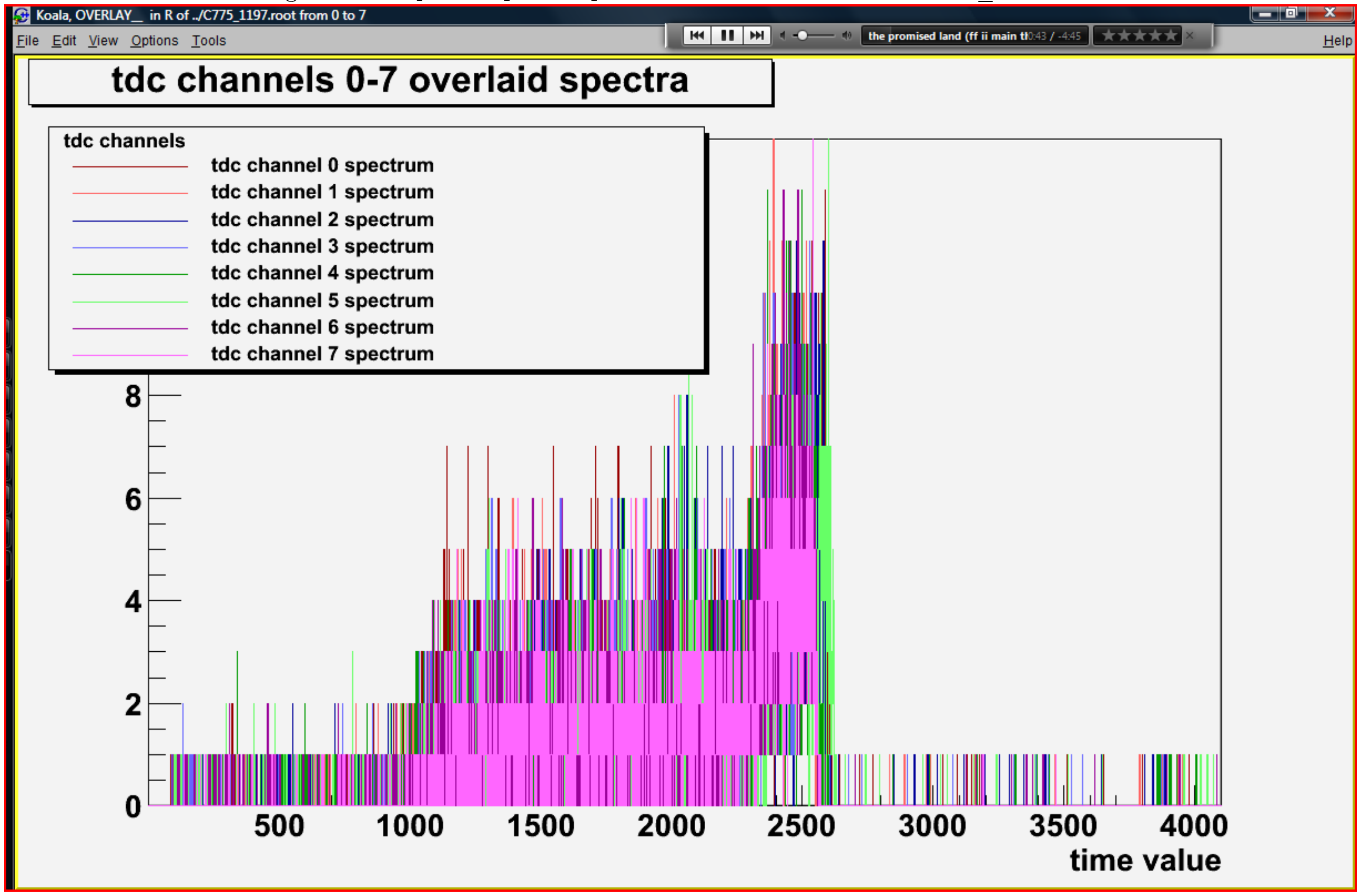
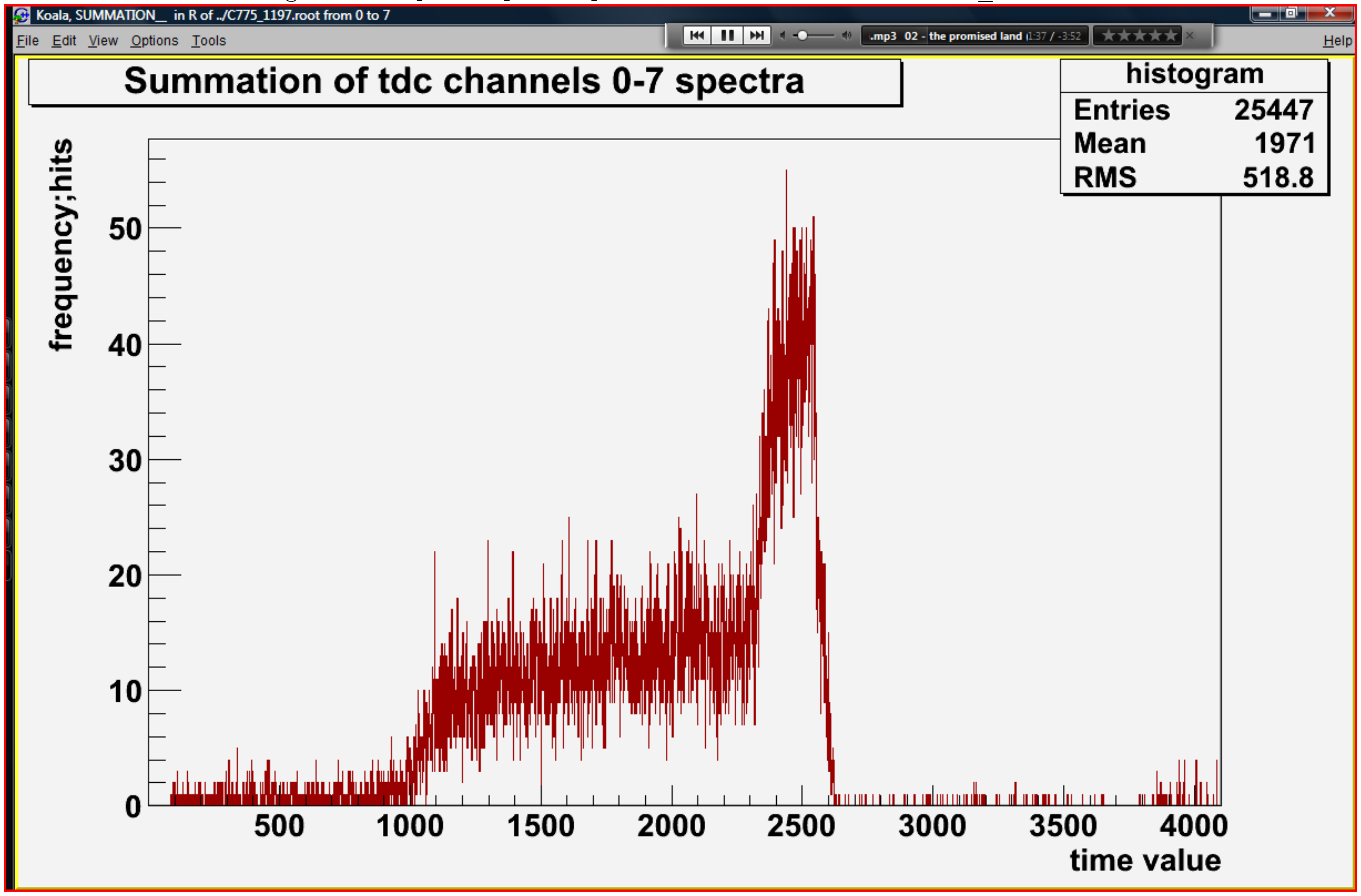


Figure 4: Output of option 2 performed on channels 0-7 of C775_1197.root



histogram	
Entries	25447
Mean	1971
RMS	518.8

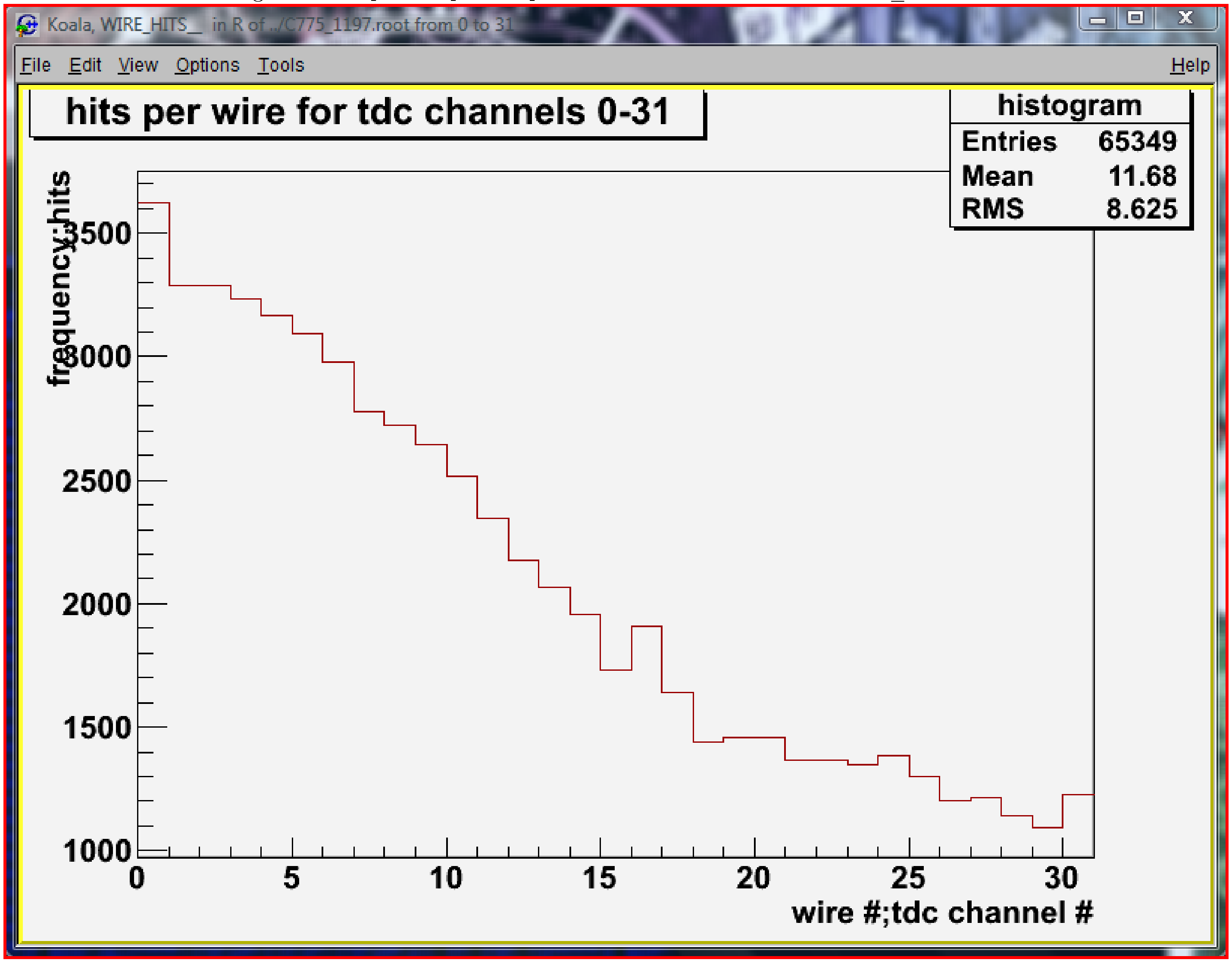
```
namespace root_has_clunky_class_designs
{

    class Koala
    {
        void TeaGrid__( );
        void TeaOverlay__( );
        void TeaSummation__( );
    };

};
```

Figure 5: Major functions pertinent to time spectra

2.2 Visualize the frequency of hits per wire in a run

A wire in the chamber is considered "hit" in a given event of the given run if and only if its time value is nonzero. It is of interest to compare the amount of times wires are hit in a run. To this end, option 3 draws a histogram of frequency of hits versus wire number. This histogram should normally show a consistent slope (this is related to the position of the scintillators during the run) and any significant abberations from this slope reflect possible defects.

Figure 6: Output of option 3 performed on channels 0-31 of C775_1197.root



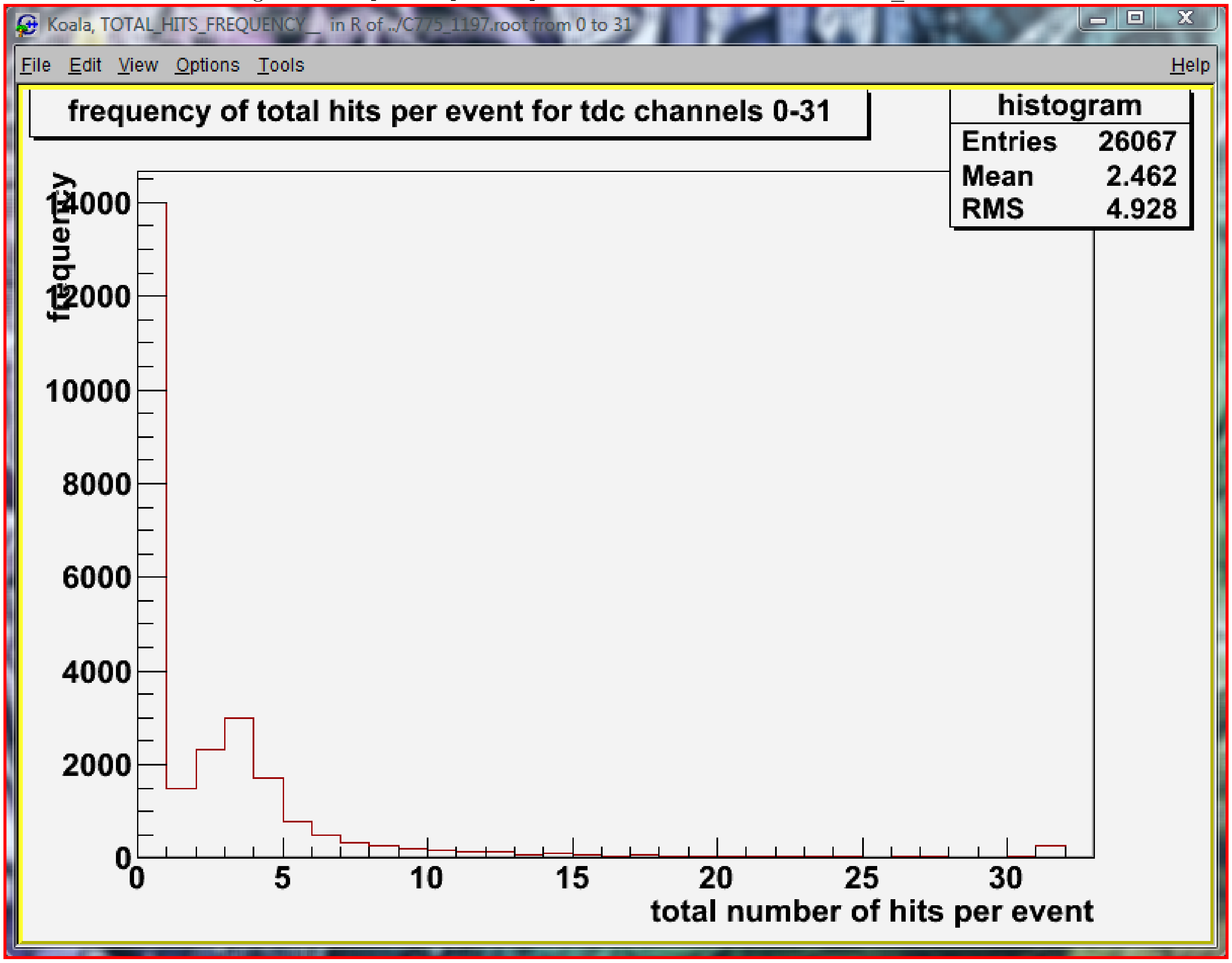
```
namespace root_has_clunky_class_designs
{

    class Koala
    {
        void TeaWire_Hits__( );
    };

};
```

Figure 7: Major functions pertinent to hits per wire

2.3 Visualize the frequency of total hits per event

It is also of interest to examine the frequency with which events in the run have a certain total number of hits in their wires. Option 4 draws a histogram of frequency versus total hits per an event. Since most events happen to represent at most one track and nothing else, the results on this digram loosely correlate to track length in a sense and are useful check because track length corresponds to angle. A passing particle incident perpendicular to the plane of the wires would leave a very short track, as in perhaps one nonzero timevalue in an event. Particles making less of an angle with the plane of the wires will leave longer tracks. However, the the later task in section 2.5 is better for this purpose.

Figure 8: Output of option 4 performed on channels 0-31 of C775_1197.root



```
namespace root_has_clunky_class_designs
{

    class Koala
    {
        void TeaTotal_Hits_Frequency__( );
    };

};
```

Figure 9: Major functions pertinent to total hits per event

2.4 Calculate efficiencies for wires and for the chamber given a run of data

One of the main purposes of my analysis was to show that the chamber was actually working efficiently and as expected. To this end, options 5 and 6 calculate the wire efficiencies and the corresponding chamber efficiency. In both options, the efficiency of a given wire is the amount of actual "hits" upon that wire divided by the amount of "triggers" about that wire. The two options differ only in their criteria for a "hit" or a "trigger."

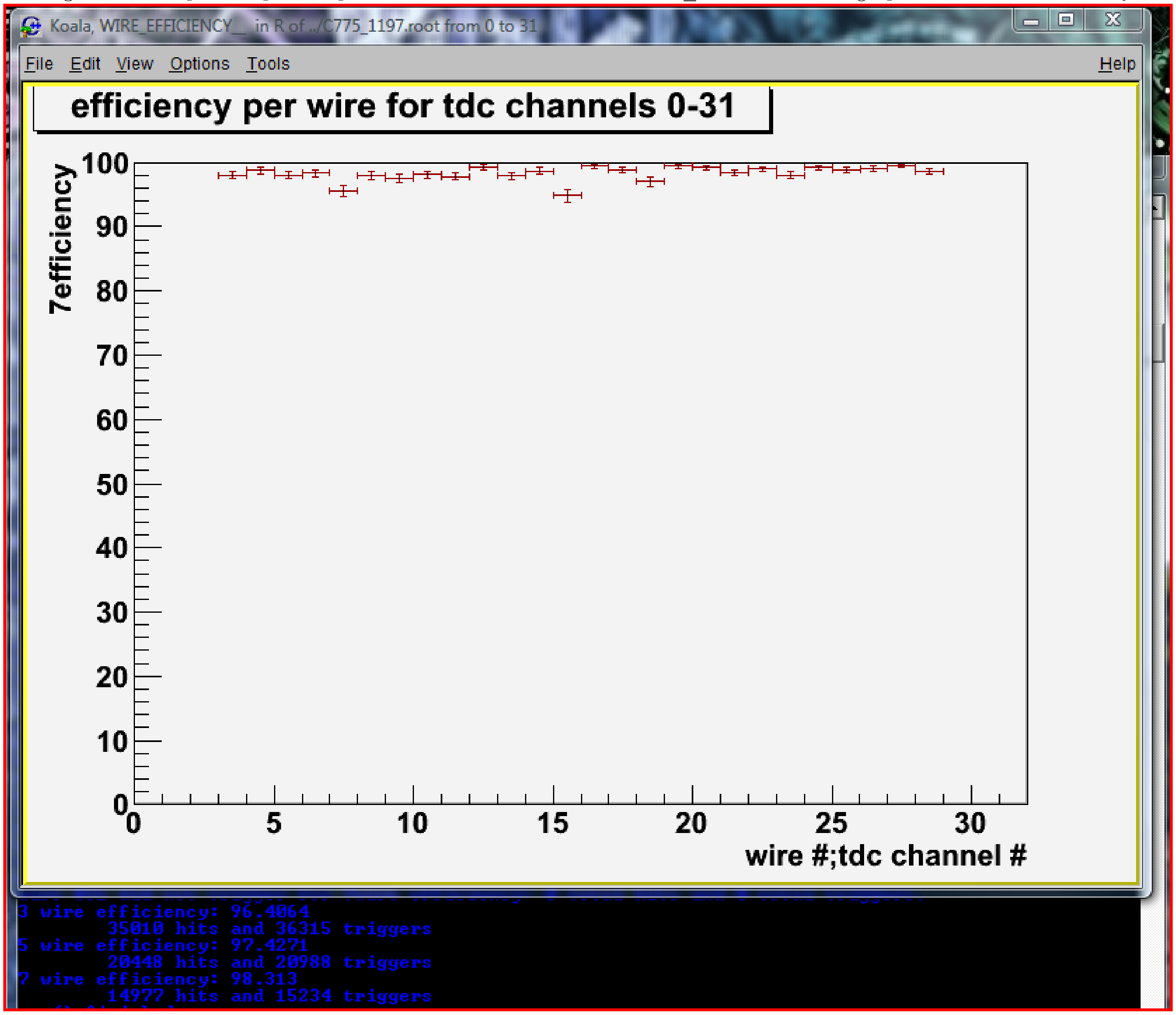
Option 5 is the simple wire efficiency option wherein a trigger involving a certain wire is when enough wires to either side of that wire had nonzero time values. Regardless of whether or not the certain wire in question also had a nonzero time value, a trigger only involves the surrounding wires. A hit is a trigger where the certain wire in question also had a nonzero time value.

Option 6 is the improved wire efficiency option which has all the requirements of option 5, but also requires that the time values in a trigger reflect an expected pattern. This additional criteria throws out many false triggers and gives better results. Since the larger time values reflect the actual time values from where the particle passed closer to the wire, the time values from each wire in a track should first grow larger and the smaller with the largest values towards the center of the track, and thus this expect pattern helps cull many false tracks from consideration.

The names used to refer to the wire efficiencies in option 5 are *3 wire efficiency*, *5 wire efficiency*, and *7 wire efficiency*. These refer to guaging efficiency by comparing triggers and hits the correspond to tracks of the length mentioned in the efficiency name.

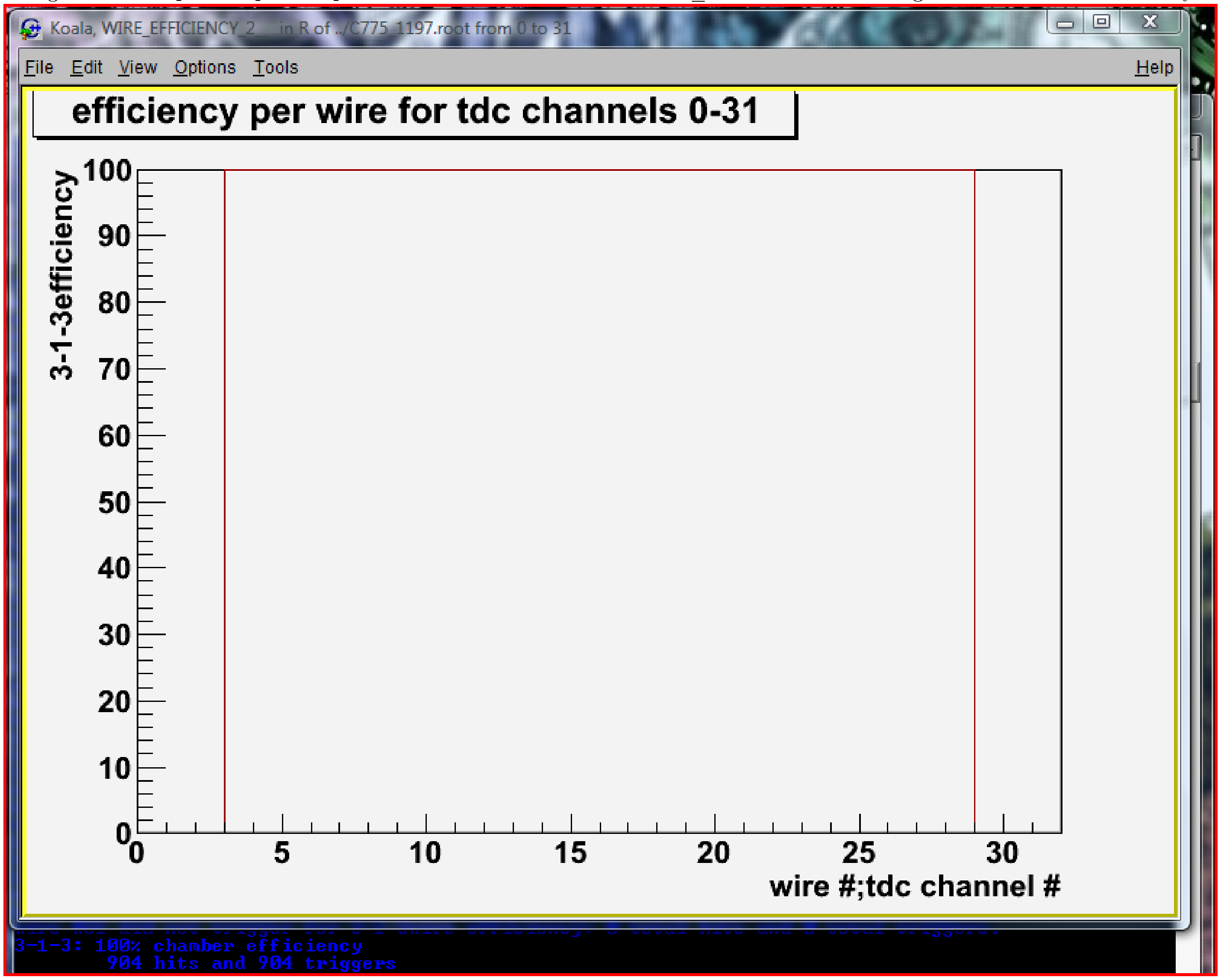
The names used to refer to the wire efficiencies in option 6 are of the form *a-b-c wire efficiency* where a, b, and c are integers representing the amount of wires to the left of the wires to be tested, the amount of wires in the center, and the amount of wires to the right of the wires to be tested respectively. For example, 3-1-3 wire efficiency tests the same length tracks that 7 wire efficiency tests, but because of the required pattern of the time values, many more triggers are found for 7 wire efficiency than for 3-1-3 wire efficiency.

Figure 10: Output of option 5 performed on channels 0-31 of C775_1197.root witha graph of the 7 wire efficiency


Koala, WIRE_EFFICIENCY__ in R of ../C775_1197.root from 0 to 31
File
Edit
View
Options
Tools
Help
efficiency per wire for tdc channels 0-31
7efficiency
wire #;tdc channel #
0
10
20
30
40
50
60
70
80
90
100
0
5
10
15
20
25
30

```
3 wire efficiency: 96.4064
        35010 hits and 36315 triggers
5 wire efficiency: 97.4271
        20448 hits and 20988 triggers
7 wire efficiency: 98.313
        14977 hits and 15234 triggers
```

Figure 11: Output of option 6 performed on channels 0-31 of C775_1197.root and looking at the 3-1-3 wire efficiency


Koala, WIRE_EFFICIENCY_2 in R of ../C775_1197.root from 0 to 31
File
Edit
View
Options
Tools
Help
efficiency per wire for tdc channels 0-31
3-1-3efficiency
wire #;tdc channel #

```
3-1-3: 100% chamber efficiency
        904 hits and 904 triggers
```

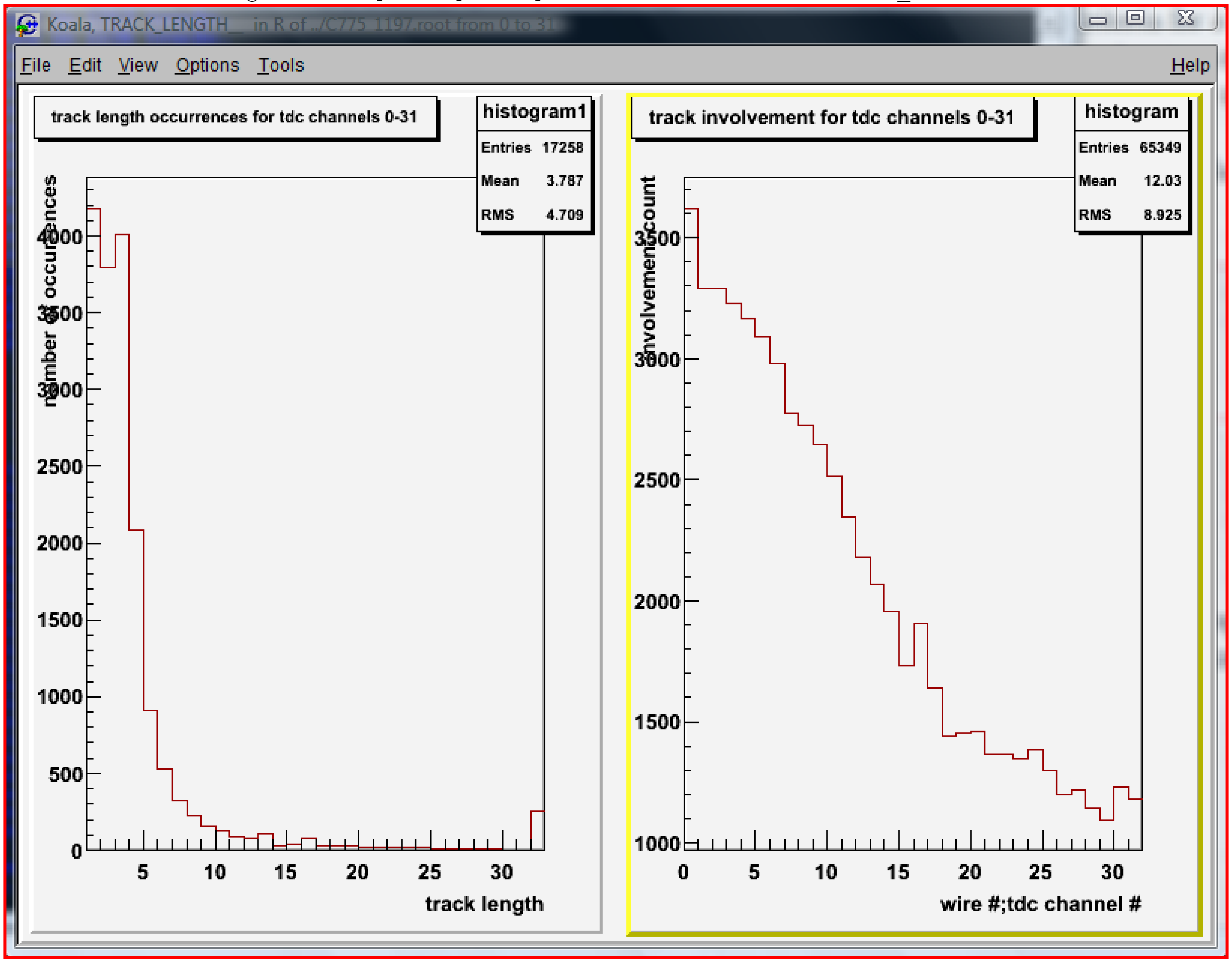
```
namespace root_has_clunky_class_designs
{

    class Koala
    {
        wingedness__ classify_wire_wingedness(
            UInt_t start ,TeaChannel_Data& given
            ,UInt_t left_wing_amount ,UInt_t center_amount
            ,UInt_t right_wing_amount
            ) const;
        symmetry__ classify_wire_symmetry(
            UInt_t start ,TeaChannel_Data& given
            ,bool odd ,UInt_t wing_size
            ,UInt_t wire_amount
            ) const;
        Double_t TeaWire_Efficiency_2__invisible(
            UInt_t left_wing_amount ,UInt_t center_amount
            ,UInt_t right_wing_amount
            );
        Double_t TeaWire_Efficiency_2__(
            UInt_t left_wing_amount = 3 ,UInt_t center_amount = 1
            ,UInt_t right_wing_amount = 3 ,bool show_hits_ = false
            ,bool show_triggers_ = true ,bool show_faux_triggers_ = false
            ,bool draw_it = true
            );
        TeaWire_Efficiency__( UInt_t draw_eff = 3 );
    };

};
```

Figure 12: Major functions pertinent to wire efficiencies

2.5 Visualize the frequency of track size for a given run of data

It is also of interest to examine the frequency with which tracks of a certain size occur in the run and the frequency with which wires are involved in them. Option 7 draws a histogram of frequency versus track size and one of frequency of involvement versus wire. It was only requested to have the simplistic criteria for a track, meaning that a track here is just several consecutive nonzero time values in an event. Because track length corresponds to angle, these histograms provide a good means to guage the ballpark angles of the tracks that the chamber is presently recording. A passing particle incident perpendicular to the plane of the wires would leave a very short track, as in perhaps one nonzero timevalue in an event. Particles making less of an angle with the plane of the wires will leave longer tracks.

Figure 13: Output of option 7 performed on channels 0-31 of C775_1197.root



```
namespace root_has_clunky_class_designs
{

    class Koala
    {
        void TeaTrack_Length__( );
    };

};
```

Figure 14: Major functions pertinent to track size

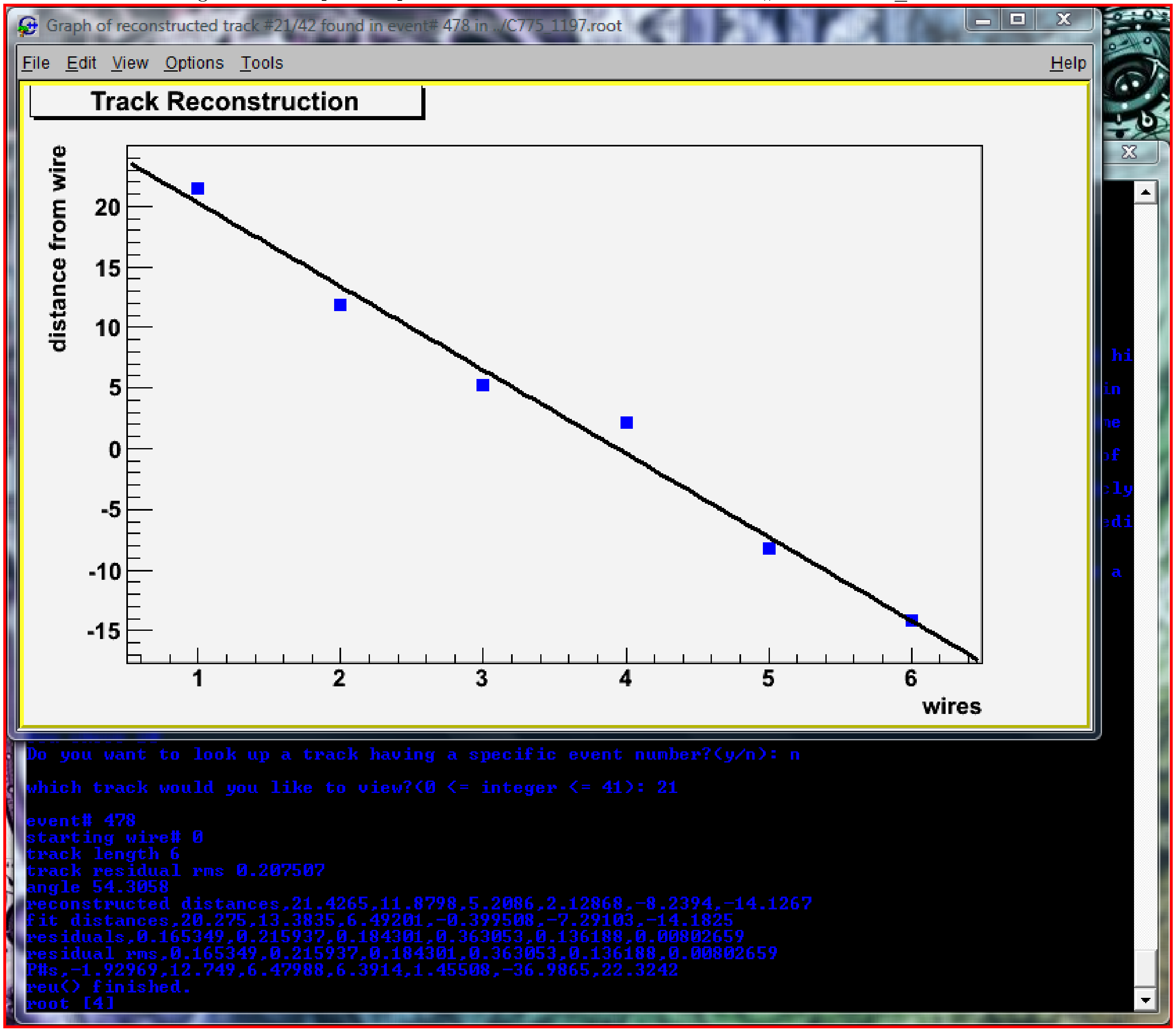
2.6 Find, Reconstruct, and Fit tracks according to a simplified model

One of the most tangible results from this project was macro option 8 which searches event by event for potential tracks of cosmic rays, reconstructs the distance from the wire of the ray for each wire, and then fits a simple line to the distances. Although future work on track reconstruction will be done differently, this was at least another good check that the chamber is actually producing intelligible results.

Macro option 10 actually allows one to graph the reconstructed distances of the track and see the linear fit superimposed on the graph, and macro option 12 allows one to separate the reconstructed graphs based on whether or not the rms of their linear fit is above or below a certain value. Macro option 9 allows creation, viewing, and editting of the "reconstruction_data.txt" config file that contains many of the constants and numbers that the track reconstruction needs but cannot find in the ROOT file. Many of these numbers are related to the function that was derived from a different simulation, called Garfield, that characterized much of the physics involved in track reconstruction. The macro only uses the set of equations produced from Garfield to find the reconstructed distances of the track from the wire. Additionally, reconstruction is an iterative process with each iteration producing a result closer to the actual track itself. This is why the reconstruction option in the macro takes considerably longer than any other option to run to completion.

In any of the outputs, the text describing a track has the number of the event within the ROOT file, the number of the starting wire, the length of the total track, the reconstructed distances of the track from each wire, the distances of the linear fit to the reconstructed distances from each wire, root mean squared values (called residuals) of the the distance between the reconstructed distances and fit itself, an average root mean squared value, the calculated angle of the track, and the final values of the terms in the equations used to reconstruct the track distance.

Figure 15: Output of option 10 to look at the track in event #478 of C775_1197.root


Graph of reconstructed track #21/42 found in event# 478 in ../C775_1197.root
File Edit View Options Tools Help
Track Reconstruction
distance from wire
20
15
10
5
0
-5
-10
-15
1
2
3
4
5
6
wires

```
Do you want to look up a track having a specific event number?(y/n): n

which track would you like to view?(0 <= integer <= 41): 21

event# 478
starting wire# 0
track length 6
track residual rms 0.207507
angle 54.3058
reconstructed distances,21.4265,11.8798,5.2086,2.12868,-8.2394,-14.1267
fit distances,20.275,13.3835,6.49201,-0.399508,-7.29103,-14.1825
residuals,0.165349,0.215937,0.184301,0.363053,0.136188,0.00802659
residual rms,0.165349,0.215937,0.184301,0.363053,0.136188,0.00802659
P#s,-1.92969,12.749,6.47988,6.3914,1.45508,-36.9865,22.3242
reu() finished.
root [4]
```

```
namespace root_has_clunky_class_designs
{

    class Koala
    {
        struct TeaTrack_Reconstruction_Data
        {
            void alter( );
        };
        struct TeaTrack
        {
            void graph( bool fit = true );
        };
        void TeaTrack_Reconstruction__(
            UInt_t event_low_ ,UInt_t event_high_
            ,std::ostream& out ,bool legible = false
            ,UInt_t lower_length = 5 ,UInt_t upper_length = 8
            );
    };

};
bool main_menu( bool looping = false );
```

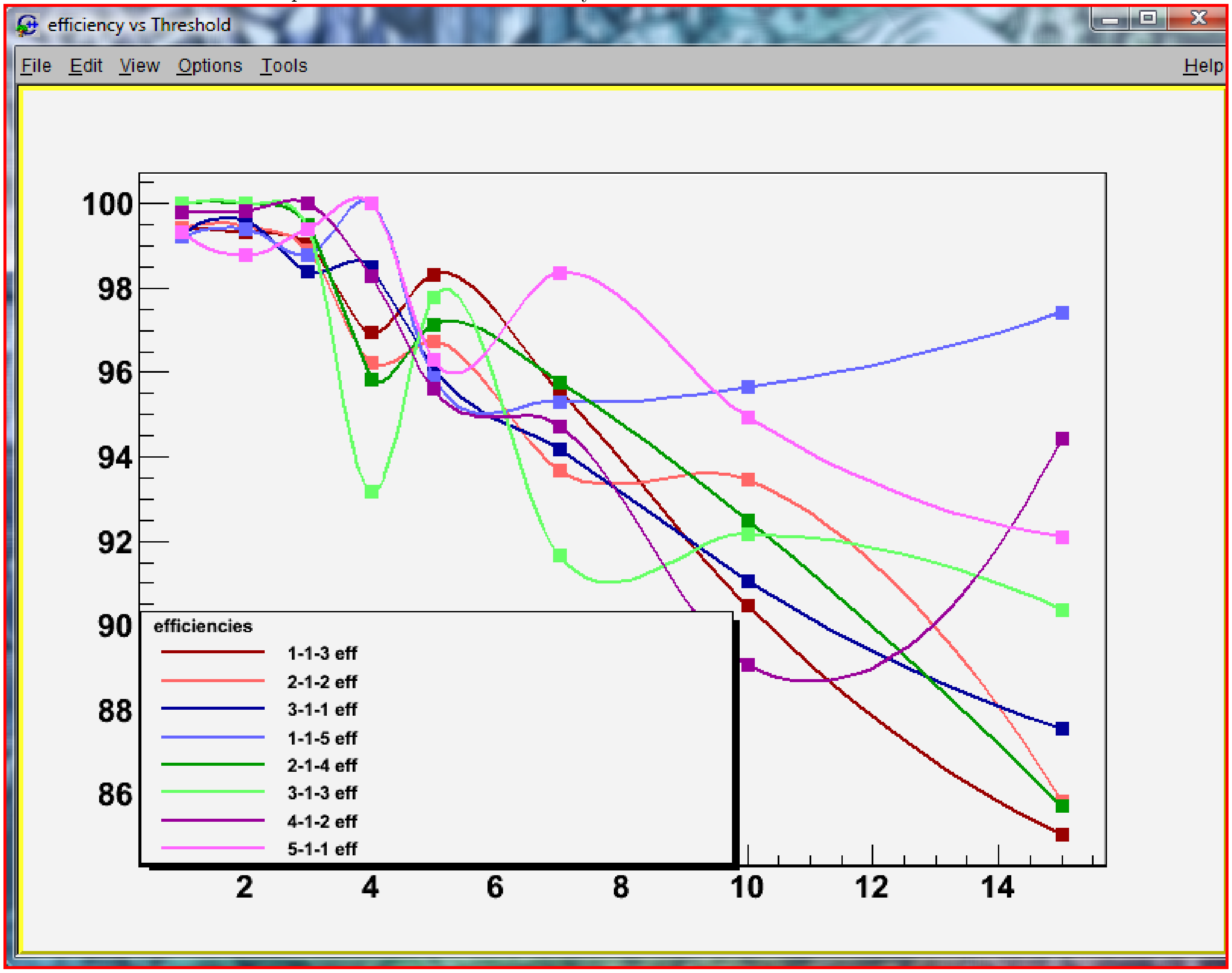
Figure 16: Major functions pertinent to track reconstruction

2.7 Graph chamber efficiency versus a run property

The last task the macro had to complete was to perform a comparison of the chamber efficiency in several runs having different values of some arbitrary value. Option 11 does this. This is option can be used as a check to see that adjusting certain run parameters is having the expected result for a vertical drift chamber.

The intended use for this option was to confirm the effects of threshold voltage and chamber voltage on efficiency, but there was only data enough to run the analysis for the differing threshold voltages since all of the latest runs have been at the same chamber efficiency. However, the threshold voltage analysis did confirm that as the threshold voltage decreases, the chamber grows more efficient up to a certain point where the gains in efficiency begin to level off.

Figure 17: Graph output of option 11 showing the effects of threshold voltage on the chamber efficiency for 8 distinct runs. The different colored lines represent the chamber efficiency for different efficiencies.



```
namespace root_has_clunky_class_designs
{

    void eff_vs_run_property_analysis(
        const std::string& run_property_name
        ,const TeaSequence<froot<Double_t> >& files
        );

};
```

Figure 18: Major functions pertinent to chamber efficiency versus a run property.

3 Conclusion

3.1 Shortfalls

3.1.1 T-naught

T-naught values are per wire values that represent the smallest actual time value that is possible for that wire. They do not vary by run, but rather vary with the conditions the chamber is setup in such that it would be nice to automatically analyze the time spectra for a run of the chamber to determine the t-naught values for each wire in a run. The t-naught values are in short the time intercept as extrapolated from a linear fit to the far side of the major hump in the spectrum.

It is tedious but doable for a human to look at graphs like those in option 0 and roughly approximate this value, and although it would have been nice if the macro could automate this task, I was too unexperienced with the matter of characterizing and finding specific parts of graphs or histograms. After roughly a week's worth of time wasted on attempting to both find previous solutions online and trying to code a good solution myself, I decided to abandon that task in favor of finishing the other objectives. In retrospect, skipping this was probably a good idea, but I would like to note that the problem of finding these t-naught values automatically is certainly something that is doable, but perhaps it is a problem better posed to someone with a broader experience in dealing with graphs and data via software.

3.2 Recommended Future Work

3.2.1 Portability versus Speed

As mentioned before, a root macro represents portable semi-C++ code that any ROOT installation can interpret dynamically. The reason I chose to keep the code as a macro is because of the limited time available for development and the fact that I have no prior experience with developing a program to run on multiple platforms.

Because the macro is interpreted rather than compiled, it is certainly slower than it could be, and most notably track reconstruction for large runs has taken a prohibitively long time to run to completion. I would recommend future versions of the macro actually be honestly compiled C++ programs that incorporate ROOT only as a third party library, and although it might be tempting to some to just develop the application in the same linux flavor that tycho runs, I would recommend against committing the application to any specific platform. Although some details will differ between compiling the application on different platforms, as long as the functionality used is available on all the platforms, the majority of the code will not need to be changed to port it to a different application. Using large and common C++ libraries such ROOT and Boost would be a good idea for this.

At least with these allowances, the largest problem in keeping the application viable for multiple platforms will be in setting up the project to compile correctly on each platform. This can be time consuming and frustrating to no end, but a cross platform IDE like code::blocks that is designed specifically to provide a common development environment regardless of platform could mitigate this trouble significantly. I would be more inclined to recommend a collaboration with the computer science department if this is pursued further as a healthy amount of prior experience drastically facilitates the matter.

3.2.2 Code Design

The macro has poor code design. It was largely built up major function by major function as each task was presented and later realized. Many of the tasks share much common code already, but it is likely that further redundant code could be eliminated and replaced by more common functions or classes. Furthermore, the code should be restructured into seperate classes somehow and divided up into multiple source files as at just above 3000 lines the one file it occupies has become rather crowded. Additionally, because of many bad experiences with CINT, the code does not use as many other classes, such as the incredibly useful classes in the Boost libraries, as it could. It is possible to use those other classes with CINT, but the approach necessary is different than what an actual C++ compiler would demand.

Since the project started so late, after many initial problems with CINT and trying to follow the normal pattern of code-reuse that I am used to, I basically gave up on trying to use any third party libraries and almost all of the standard template library. I did not deem it worthwhile to write code based on useful classes only to later find that CINT was not configured properly to deal with them like a normal (competent) C++ compiler, and I decided it would be more time efficient just to write quick classes, like TeaSequence, to provide the functionality I desired rather than to pray that I could resolve each and every CINT conflict in a reasonable amount of time.

Developing the same code as a standalone C++ application that uses ROOT as a third party library will almost completely do away with all the limitations in the code design due to CINT's unique support for C++, and without these limitations, the code can be completely revolutionized by actually reusing on demand as development progresses well tested and well designed classes already written by someone else without either the impediment of having to figure out how to get CINT to compile normal C++ code or the impediment of having to just write the class yourself.

3.3 Summary

The "reu.C" macro now automates many tedious analysis chores, and room for further development of the macro's design and capabilities is wide open. Using this macro, tests of the VDCs as they are built should be able to happen in faster times than before.