Developed Scripts for Analyzing Bigbite MWDCs

Seamus Riordan
Carnegie Mellon University
riordan@jlab.org

May 24, 2006

This documents the scripts that have developed for the multiple wire drift chambers for the BigBite spectrometer detector package. A basic understand of ROOT and C/C++ is assumed for this document.

Contents

1 Drift Chamber Configuration

The information about the drift chamber configuration is held independently in two sections, `mwdc_defs.h` and the ROOT/analyzer database.

1.1 mwdc_defs.h

Most scripts rely on a header file that has been included in the package called `mwdc_defs.h`. This file contains basic information on variables that are frequently used across the scripts, such as the number of planes, the names we give to the planes, pairing of plane types (in a six plane configuration), the naming conventions given to ROOT files (as well as their paths), and limits we put on static arrays. Almost every script requires this header file to function and this should be kept in sync with the database file used to analyze the data. It was designed to keep "hard-coding" of files to a minimum and allow for versatile analyzation from multiple configurations. The file is designed to be quickly modified using intuitive variable names and structures. An example of the header file can be found in Appendix A (page 30).

1.2 ROOT/analyzer Database

Ocassionally a script requires to a working analyzer database file to function. These are usually the cases when it needs specfic information about the geometry of the chamber, such as locations of the planes, wire locations, TDC offset information, etc. In general these scripts look for a database under the same rules a replay scripts do, (i.e. you are required to have a `DB_DIR` environment variable declared pointing to the database).

More information about database structure and the ROOT/analyzer in general can be found at http://hallaweb.jlab.org/root/doc/index.html

1.3 Configuration in the Scripts

Almost all of the scripts have some type of local information stored in them. Ocassionally information regarding the chambers is stored here, such as cell geometry, that cannot be found in the database. For the most part, this information will be static from configuration to configuration, so it will not be necessary to alter. This information usually at the top of the script and is declared using C-style `#define` calls. In almost all cases the scripts also contain information to limit the number of events analyzed, how frequently to update processing information to `stdout`, and probably most importantly, the ranges and binning of relevant histograms. Each script should be scanned for these variables before running.

2 Scripts

There are many scripts for the drift chambers to analyze data from a CODA run. Almost all of these scripts require a working ROOT file containing a tree with the relevant branches for

the code. These are generated from a replay script which takes the raw data given by a CODA run and translating them into a ROOT file for further analysis. Replaying a script produces general information about the wires that were hit, the timing information, reconstructed track information, etc. in the form of a ROOT tree. The structure and access using ROOT for these scripts are beyond the scope of this document, however one may learn more about them from the "ROOT User's Guide" (http://root.cern.ch/root/doc/RootDoc.html).

In each section following we will descript a script, list in expected input to run the script, and then give an example of the output.

2.1 replay.C

Input: Raw data from CODA (.dat file), run number
Output: ROOT file containing a ROOT tree

This is the heart and workhorse of the Hall A ROOT/analyzer. `replay.C` takes the raw data acquired from a run using the CODA system and puts the relevant information into a ROOT tree. Almost all of the scripts following this require this tree to perform their subsequent analysis.

CODA assigns a run number to each data set which the analyzer will also use in the generation of files. The run number for a data set is contained within the structure of the file itself (in a data class THaRun), however it is convenient (and is used as convention with these scripts) to include the run number in the file name itself. For example `bbdc05_1472.dat` specifies run number 1472. The analyzer will then create a ROOT file containing a ROOT tree named `bbdc05_1472.root` with the decoded data. The exact form and path of the input files and output files are contained in the `mwdc_defs.h` file under the definitions `RAWDATA_FORM` for the raw data file and `ROOTFILE_FORM` for the ROOT file. Upon starting `replay.C` a run number must be specified from the keyboard.

It is convenient to also have some variations on the replay scripts depending on the analysis to be done. For example, `replaysmall.C` will only replay a small number of events, `replayall.C` will replay all the events from a run. Since we often would like to keep these special replays seperate from each other, we adopt a ROOT file naming convention with a *-suffix* after the run number. For example, `replaysmall.C` will produce `bbdc05_1472-small.root`. These suffices play a role in determining which run you would like to then analyze with the scripts. When prompted for a run number, one may then enter `1472-small` to use the ROOT file produced by `replaysmall.C`. The names of the files produced are sent to `stdout` during replay.

2.2 Dispatches

The replay scripts now have two options for the method of replaying a script. The traditional option is using the standard THaAnalyzer class included with the Hall A ROOT/analyzer. The newer option developed specifically for the GEn is to use the dispatch facilities of the THaGenAnal class. This allows for arbitrary ordering the execution of decoding of detectors,

tracking, PID, and contains a powerful branching method using the inherent cuts facility. The processing is contained in a dispatch file, an example of which, which approximates the ordering of the standard analyzer is found in Appendix B (page 31).

While the full documentation of the dispatch facility is beyond the scope of this document, the general idea is each line of the dispatch file contains an optional identifier, a necessary command, and an optional argument. Identifying a line for branching purposes is done by the name followed by a colon. For example `BadData:` would identify a line named `BadData`. The command is of the form *name*`.`*command*, the name being the name of the apparatus or spectrometer (`B`, for example), and the command being one of the predetermined command that object can accept. The argument is a string preceded by a space on the same line as the command. For example, the line `B.Decode dc` decodes the dectector `dc` contained within the spectrometer `B`. A special name is `AN` which specifies to call a command to the analyzer itself. In the example provided, `AN.EndStage Decode` ends the stage Decode and evaluates the cuts associated with that stage. See the Hall A ROOT/analyzer documentation for acceptable stage names and formation cuts.

Each line after execution returns some value, OK or NotOK. For example, `AN.EndStage Decode` will return OK if the decoded data passed the cuts and NotOK otherwise. These return values can then be branched upon using the command `brOK` or `brNotOK` which take a line identifier as an argument. They will then branch depending upon the previous line's return value or continue to the next line.

Two other special commands for ending event evaluation are the `Terminate` and `FillTree` commands. `Terminate` ends the processing for the event and does NOT fill the output tree with the data. This is useful in the case that the data does not pass a cut and we wish to discard it. The `FillTree` command ends the processing, but keeps the data we have and puts it into the output tree. This is assumed when the dispatch file reaches a logical end, but should be put in explicitly for clarity.

2.3 showwires.C

Input: ROOT File, run number from keyboard
Output: Graphical display of wire spectra, postscript file `ps/wirehits.ps`

`showwires.C` displays a histogram for each plane binned in such a way that each bin represents one wire filled the number of times that wire was hit. These graphics are then copied into a postscript file for future use. A run number must be specified on running this script, for example `1472` for run 1472 or `1472-all` for 1472 produced by the `replayall.C` script. An example of the output is shown in figure 1.

2.4 showwires.C variations

`showwiresinplane.C`
Input: ROOT File, run number from keyboard, plane name from keyboard
Output: Graphical display of wire spectra for a specific plane, postscript file `ps/wirehits-`*planename*`.ps`.

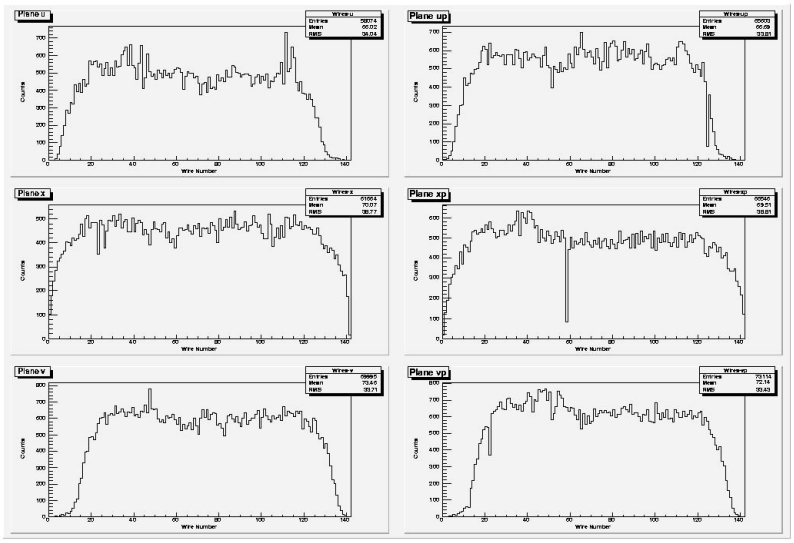

Figure 1: A plot of the wire spectra for each plane

`showwire-single.C`
Input: ROOT File, run number from keyboard
Output: Graphical display of wire spectra of only the first hit on a wire in an event ignoring subseqent hits, postscript file `ps/wirehits-single.ps`.

`showwiresinplane-single.C`
Input: ROOT File, run number from keyboard, plane name from keyboard
Output: Graphical display of wire spectra for a specific plane of only the first hit on a wire in an event ignoring subseqent hits, postscript file `ps/wirehits-`*planename*`-single.ps`.

2.5 findtimes.C

Input: ROOT File, run number from keyboard
Output: Graphical display of time spectra of each plane, postscript file `ps/times.ps`.

`findtimes.C` displays the timing spectra for all planes. An example of output is shown in figure 2.

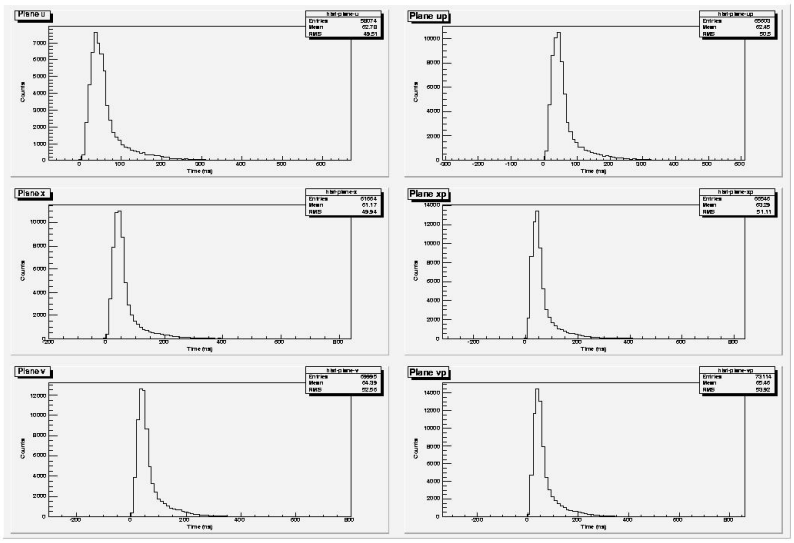

Figure 2: A plot of the time spectra for each plane

2.6 findtimes.C variations

`findtimesinplane.C`
Input: ROOT File, run number from keyboard, plane name from keyboard
Output: Graphical display of time spectra for a specific plane, postscript file `ps/times-`*planename*`.ps`.

`findtimes-single.C`
Input: ROOT File, run number from keyboard
Output: Graphical display of time spectra of only the first hit on a wire in an event ignoring subseqent hits, postscript file `ps/times-single.ps`.

`findtimesinplane-single.C`
Input: ROOT File, run number from keyboard, plane name from keyboard
Output: Graphical display of time spectra for a specific plane of only the first hit on a wire in an event ignoring subseqent hits, postscript file `ps/times-`*planename*`-single.ps`.

`findtimes-singletrack.C`
Input: ROOT File, run number from keyboard
Output: Graphical display of time spectra as specified below, postscript file `ps/times-singletrack.ps`.

Conglomerates the time spectrum for all wires in a plane for each plane and outputs a set of histograms using only first hit for any wire in an event. Only drift times in events where we have one hit per plane are considered. This reduces our data to events that are

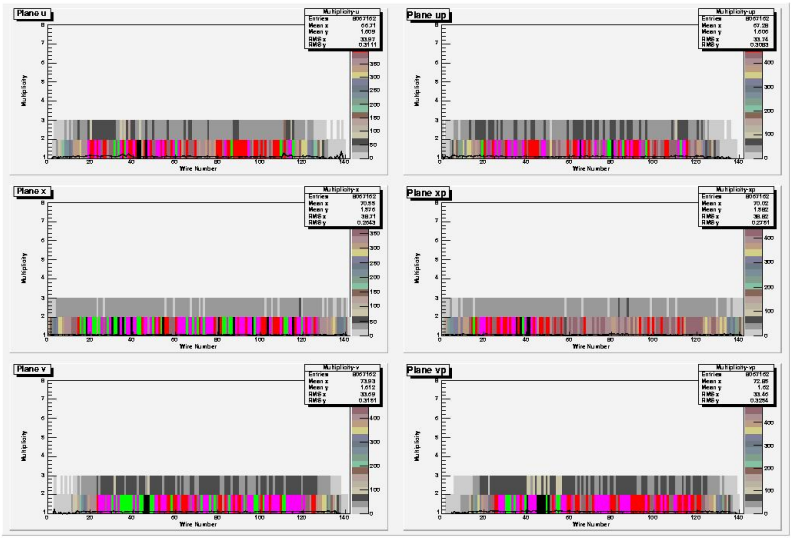

Figure 3: A plot of the multiplicities of all wires in each plane

most likely a single, clean track.

2.7 findmult.C

Input: ROOT File, run number from keyboard
Output: Graphical display of multiplicies for all wires on each plane, postscript file `ps/multipl.ps`

This script calculates the multiplicities of all the wires and outputs them for each plane into a 2D histogram. An example can be found in figure 3.

Variation `findmultinplane.C`
Input: ROOT File, run number from keyboard, plane name from keyboard
Output: Graphical display of multiplicies for all wires in a specific plane, postscript file `ps/multipl-`*planename*`.ps`

2.8 showoccupancy.C

Input: ROOT File, run number from keyboard
Output: Graphical display of occupancies of each plane, postscript file `ps/occupancy-`*run number*`-12.ps`, `ps/occupancy-`*run number*`-3.ps`

This script calculates the occupancies (number of hits per plane) and outputs them for each plane.

Variation `showoccupancy-single.C`
Input: ROOT File, run number
Output: Graphical display of occupancies of each plane, postscript file `ps/occupancy-single-`*run number*`-12.ps`, `ps/occupancy-single-`*run number*`-3.ps`

This script calculates the occupancies (number of hits per plane) and outputs them for each plane, but only uses the first hit on any given wire.

2.9 calct0.C

Input: Set of uncalibrated ROOT files, `db_B.dc.dat` in the directory the script is being run
Output: Text file offsets.dat containing t0 offsets calculated by card, `ps/offsets/offsets-`*plane name*`.ps`

This script calculates TDC offsets from a run on a per card basis. This means that wires are grouped together by their physical connection to the drift chamber by the card they are connected to. The perl script genwiretocard.pl *must* be present in the directory for this script to function. This script determines from the database file `db_B.dc.dat` in the directory the script is being run how these cards are associated with individual wires.

This script generates the TDC offsets by taking the time spectrum for an entire card and then finding the maximum. The script then checks for bins to the left the bin that has 90% the value of the maximum and then the bin with 10% of the maximum. A linear interpolation is then done from these two points to find where this line intersects the x-axis, giving the appropriate t0 offset.

A set of plots is produced for each card, with an individual canvas for each plane, as seen in figure 4. On each plot, the offset position and maximum bin position are represented by red lines. The 10% and 90% marks are represented in blue lines. A green line represents the interpolation between these two points. The offsets are also dumped into the file offsets.dat in a text readable file. The perl script `t0todb.pl` can then be used to put the offsets into the file `db_B.dc.dat-witht0` using `db_B.dc.dat` as a template.

2.10 sett0to0.pl

Input: `db_B.dc.dat`
Output: `db_B.dc.dat.not0`

This is a simple Perl script that takes a database file for BigBite named `db_B.dc.dat` and creates a new database called `db_B.dc.dat.not0` where all the TDC offsets have been set to 0.

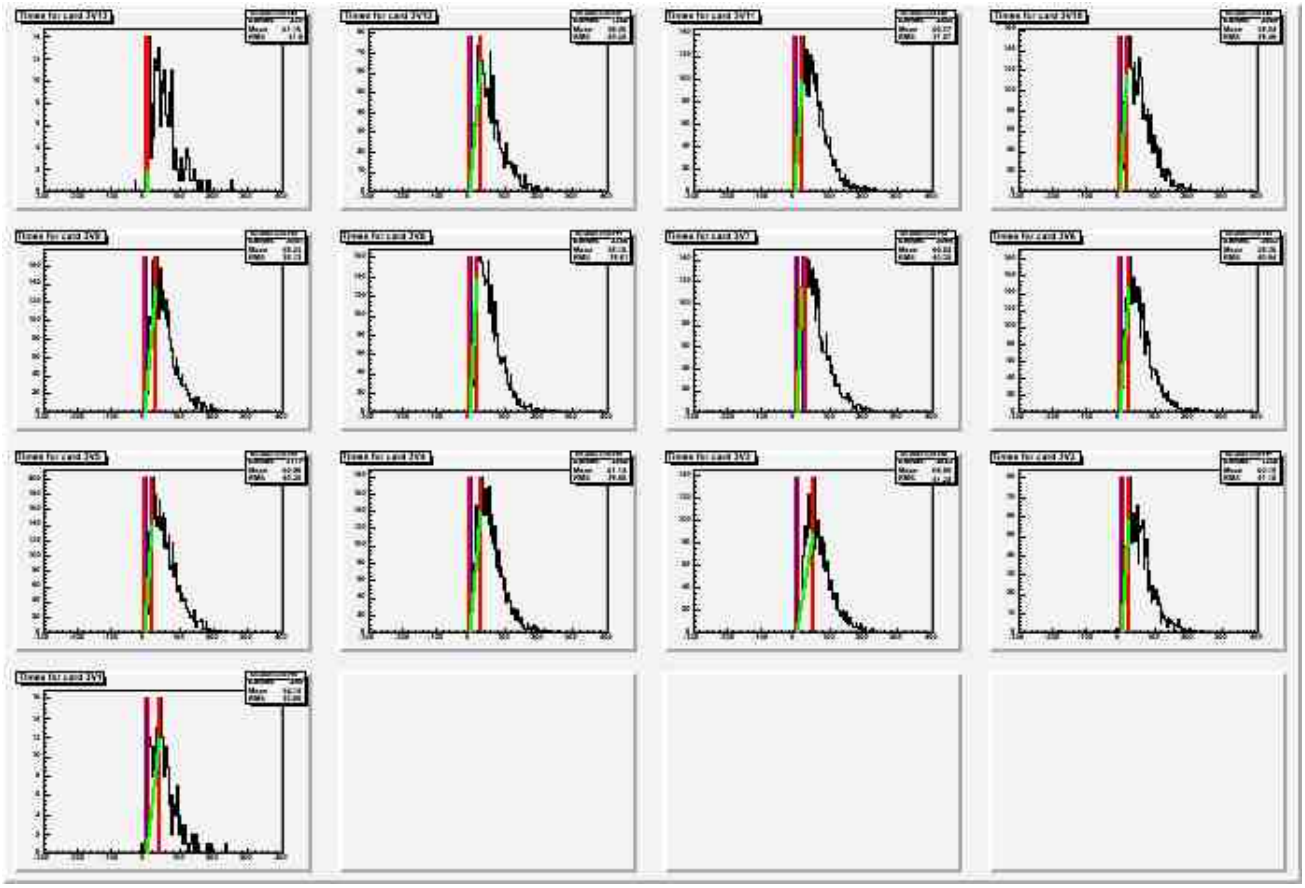

Figure 4: A plot of wire spectrum for a set of cards. The red lines represent the offset position and the maximum bin. The blue lines represent the 10% and 90% marks. The green line is the linear interpolation between the two marks.

2.11 LoadTreeFile.C

Input: ROOT File, run number from keyboard
Output: None

This simply loads a run into the analyzer and opens up the ROOT Tree display

2.12 plott0.C

Input: ROOT File, run number from keyboard
Output: Plot of the TDC offsets and generates a PostScript file `ps/offsets.ps`

This script generates a plot of the TDC offsets (a sample output is given in figure 5). The run number is given simple so it may reference a database file.

2.13 showtrackinfo.C

Input: ROOT file, run number
Output: Histograms of track x, y intercepts at $z = 0$ and slopes x' and y', various PostScript files

This outputs reconstructed track information. The plots and samples are as follows:
`ps/trackx.ps`: Plot of the x intercepts. Figure 6
`ps/tracky.ps`: Plot of the y intercepts. Figure 7
`ps/trackxp.ps`: Plot of the x slopes. Figure 8
`ps/trackyp.ps`: Plot of the y slopes. Figure 9
`ps/trackxvsy.ps`: Plot of the x vs y intercepts. Figure 10
`ps/trackxpvsyp.ps`: Plot of the x vs y slopes. Figure 11
`ps/wiresintracks.ps`: Plot of the wires used in a track. The total wire spectrum is overlayed in red. Figure 12
`ps/numberinrecon.ps`: Plot of the number of planes used in reconstruction. Figure 13

2.14 visualmwdc.C

Input: ROOT file, analyzer database, run number
Output: Visual representation of events

This script is the event viewer for the drift chambers. It allows one to view, event by event, the wires that were hit, their timing information, the reconstructed tracks, and the track information. It also allows one to switch between views of each plane and output the visual to a postscript file. Also defined are a number of cuts that can be turned on and off for the data set.

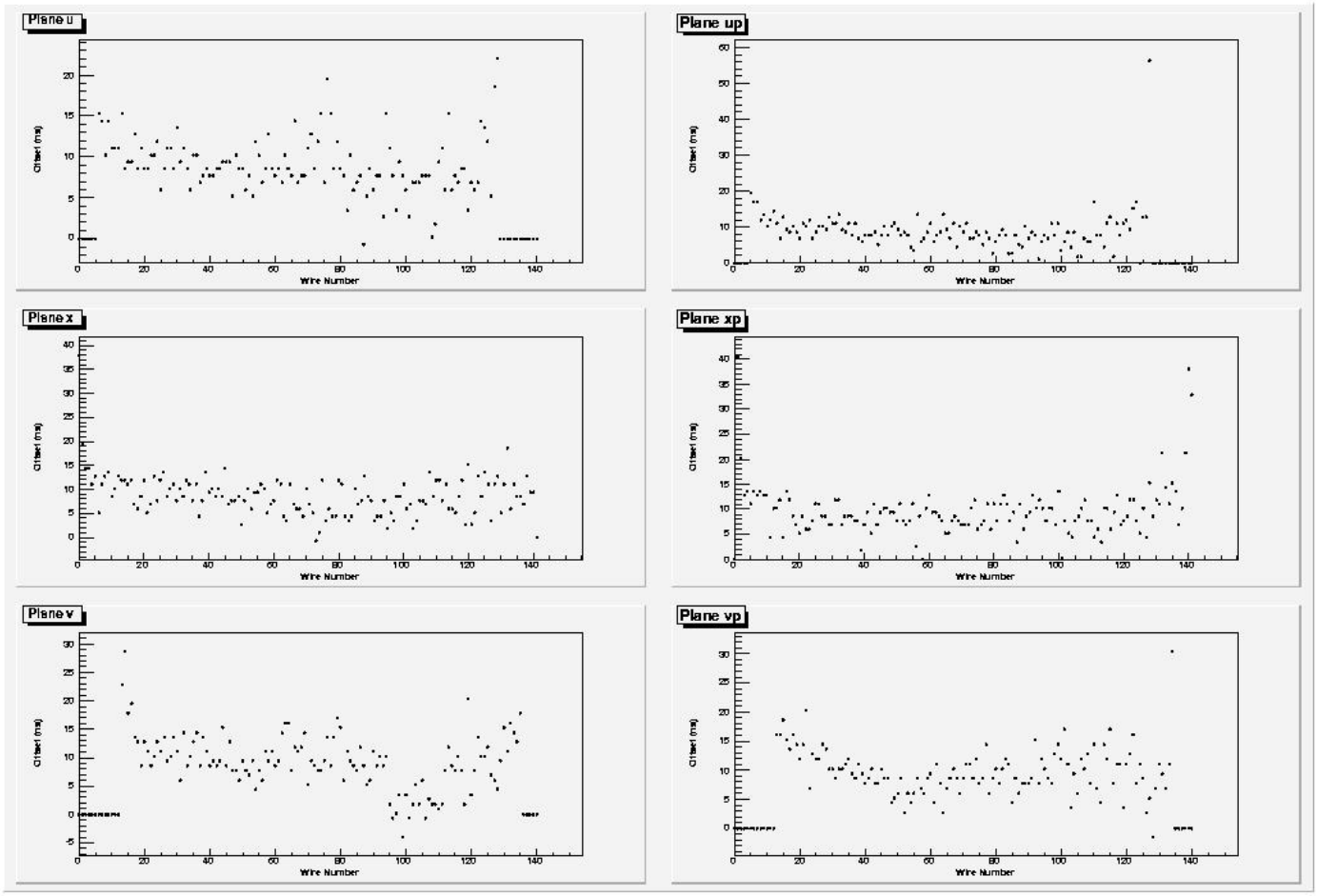

Figure 5: A plot of the t0 offsets for each wire. Each indiviual plane is represented on its own canvas.

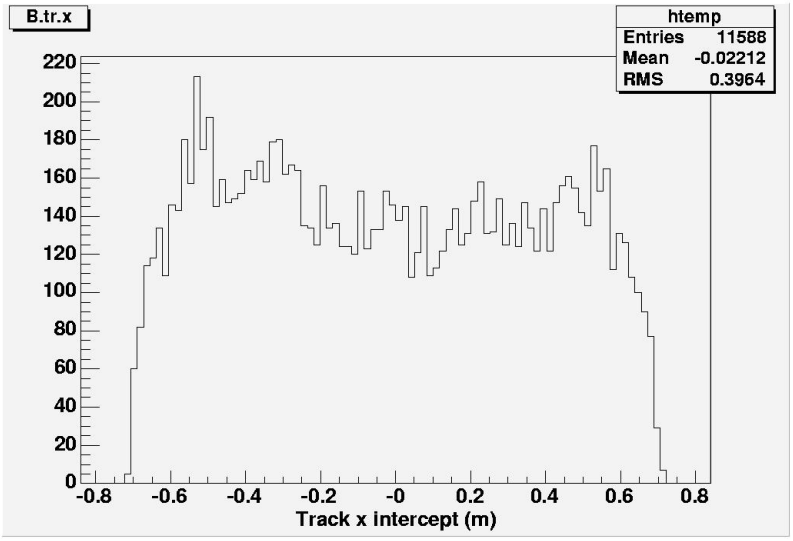

Figure 6: Histogram of the track x intercepts

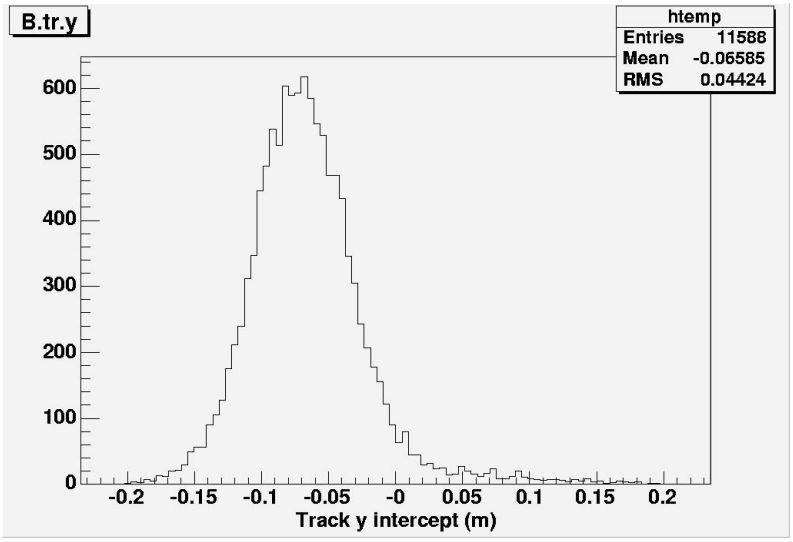

Figure 7: Histogram of the track y intercepts

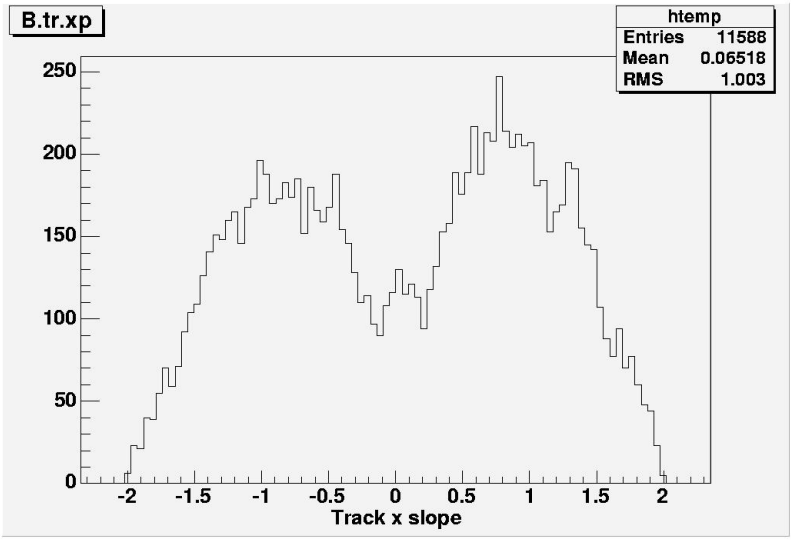

Figure 8: Histogram of the track x slopes

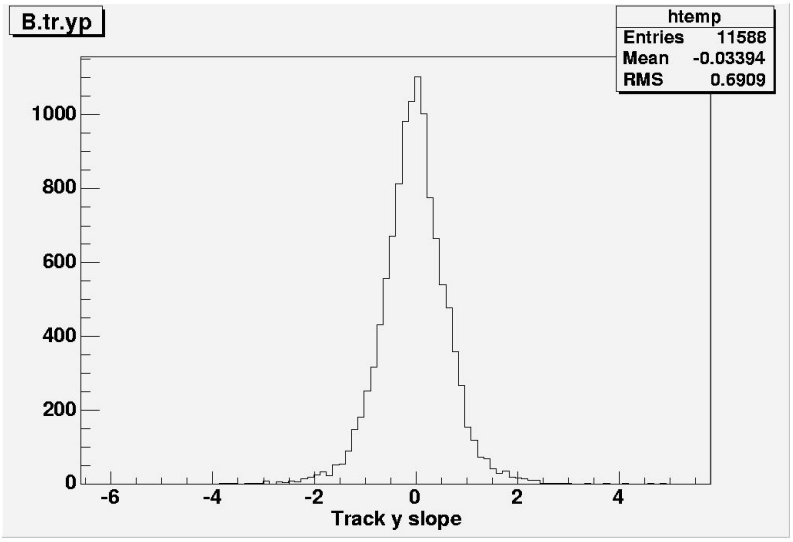

Figure 9: Histogram of the track y slopes

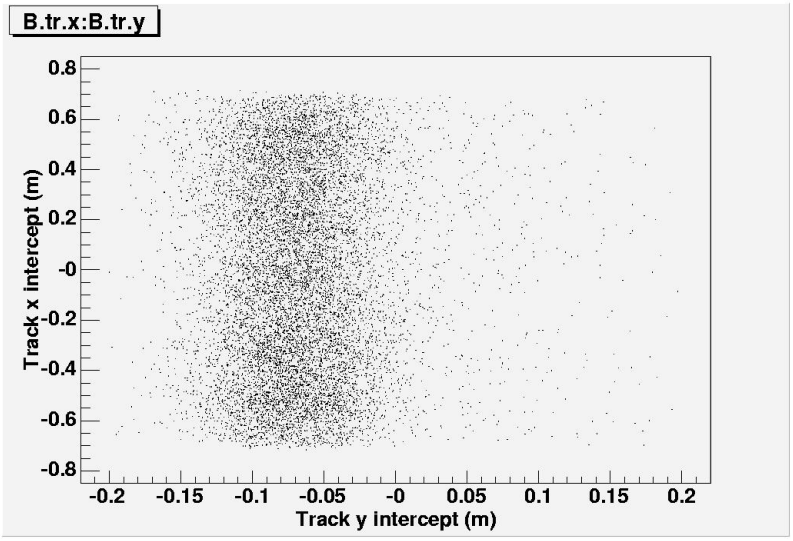

Figure 10: Histogram of the track x intercepts vs y intercepts

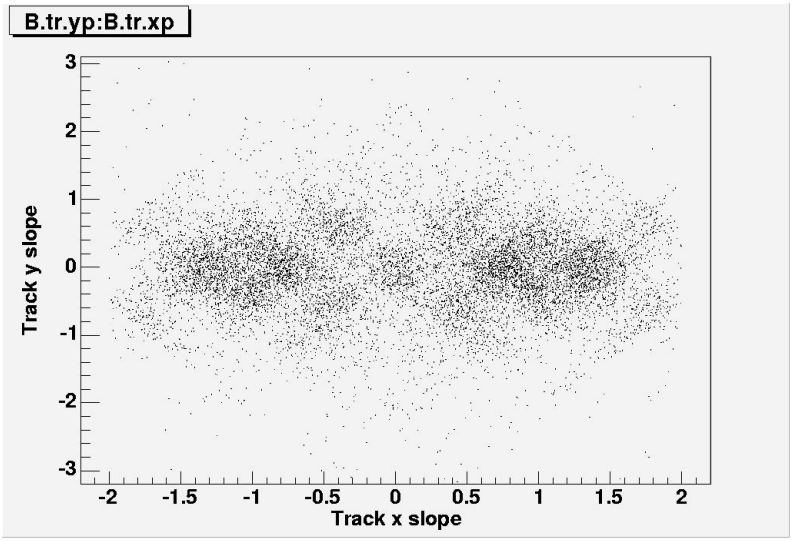

Figure 11: Histogram of the track x slopes vs y slopes

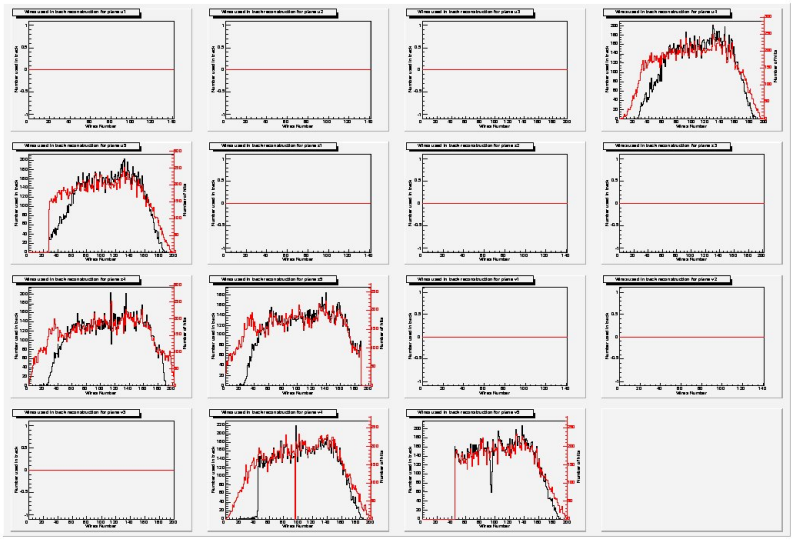

Figure 12: Histogram of the number of times a wire was used in track reconstruction. The total wire spectrum is overlayed in red.

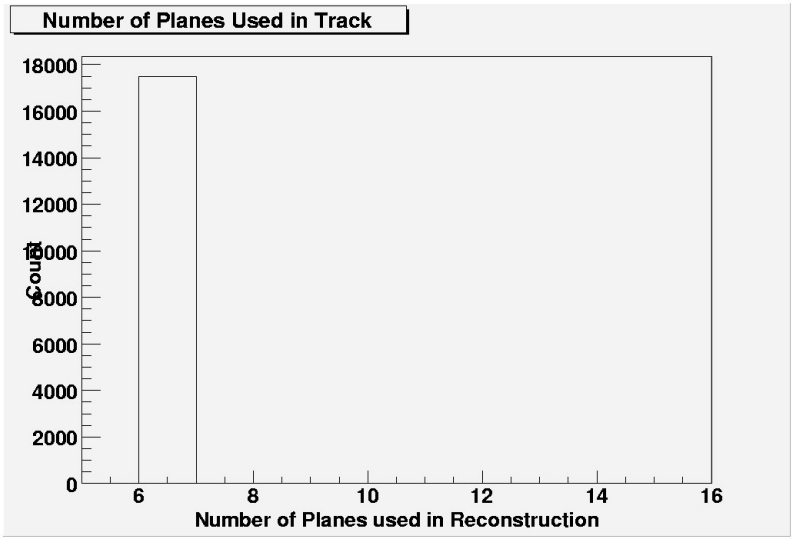

Figure 13: Histogram of the number of planes used in a track reconstruction. In this figure, six and only six planes were used in the reconstruction.

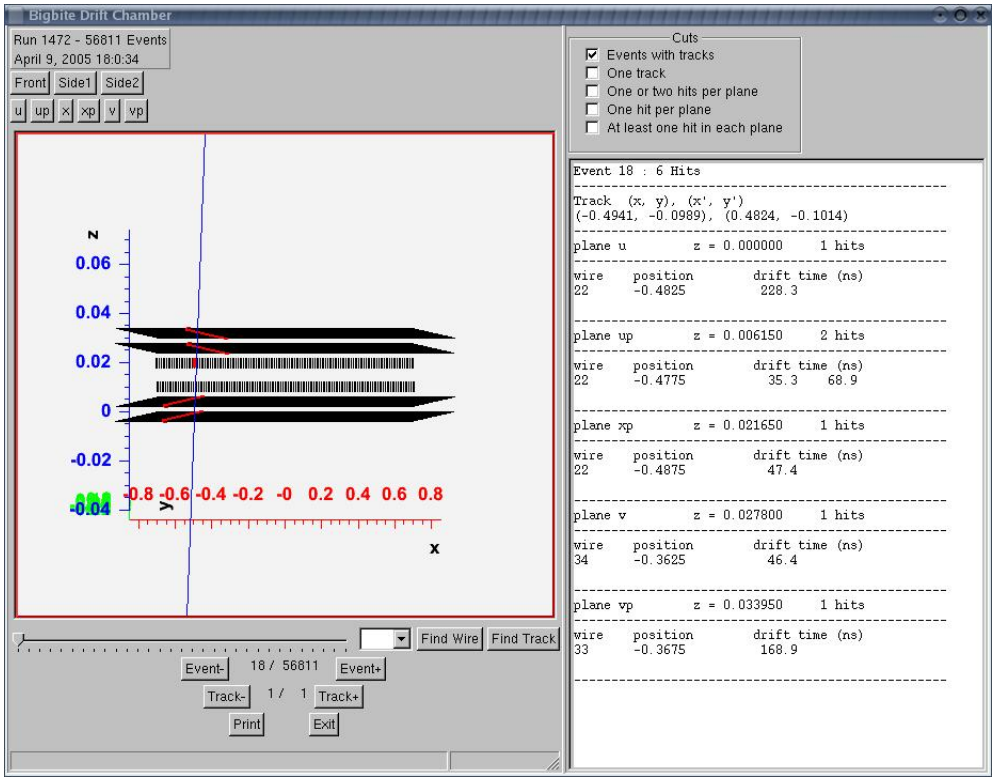

Figure 14: Sample of the event viewer.

In the display, black wire had no hit. Red wires have a hit, but did not pass the shower cuts and therefore are not considered in tracking. Yellow wires have hits and do pass the shower cuts and are used in tracking, but were not found to be associated with any given track. Green wires are wires that were used in a fit track. A sample of the display is given in figure 14.

2.15 findtrackdiffs.C

Input: ROOT file, analyzer database, run number, output suffix
Output: Various outputs involving the residuals of tracks

This script gives information on how well the track reconstruction is working on real data. It mainly involves the residual, which is the difference between the reconstructed track intercept in a plane and the calculated drift distance. This also requests an additional output suffix which is simply appended to the end of all the generated postscript names. This is convenient so that one may replay a run using slight modifications to the reconstruction code and observe the differences. The plots generated and examples of those plots are:
`ps/trackdiffs/residual-`*suffix*`.ps`: Plot of the residuals for each plane (Figure 15)

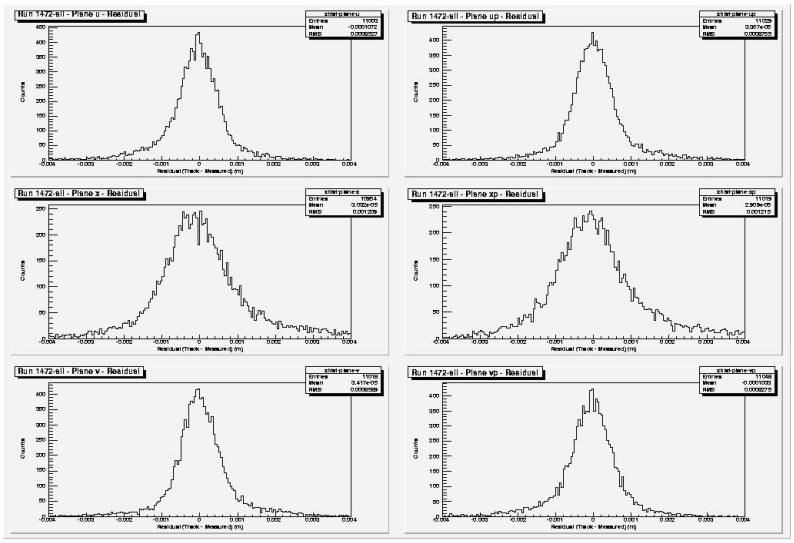

Figure 15: Histogram of residuals for each plane

`ps/trackdiffs/driftdists-`*suffix*`.ps`: Plot of the drift distances used in track reconstruction (Figure 16)
`ps/trackdiffs/trackslopevsresidual-`*suffix*`.ps`: Plot of the track slopes vs residual(Figure 17)
`ps/trackdiffs/drifttimevsresidual-`*suffix*`.ps`: Plot of the drift time vs residual (Figure 18)
`ps/trackdiffs/driftdistvsresidual-`*suffix*`.ps`: Plot of the drift distance vs residual (Figure 19)
`ps/trackdiffs/cham1planevsplane-`*suffix*`.ps`: Plots of plane residuals for first chamber planes against each other (Figure 20)
`ps/trackdiffs/cham1planeplanediff-`*suffix*`.ps`: Plots of the difference between plane residuals for planes of the same type in the first chamber (Figure 21)
`ps/trackdiffs/interceptdiff-`*suffix*`.ps`: Plots the intercept difference for each plane (Figure 22)
`ps/trackdiffs/residualvswire-`*suffix*`.ps`: Plots the residual for each wire in each plane (Figure 23)
`ps/trackdiffs/interceptdiffvswire-`*suffix*`.ps`: Plots the intercept difference for each wire in each plane (Figure 24)

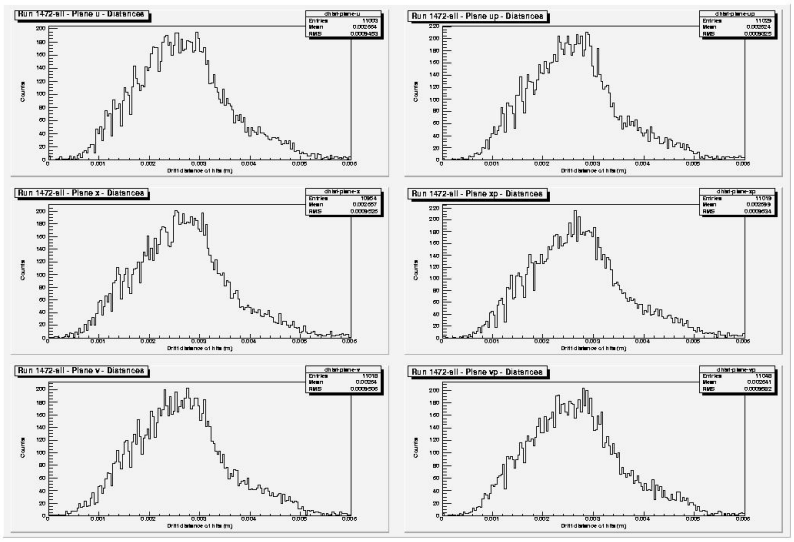

Figure 16: Histogram of the drift distances for each plane

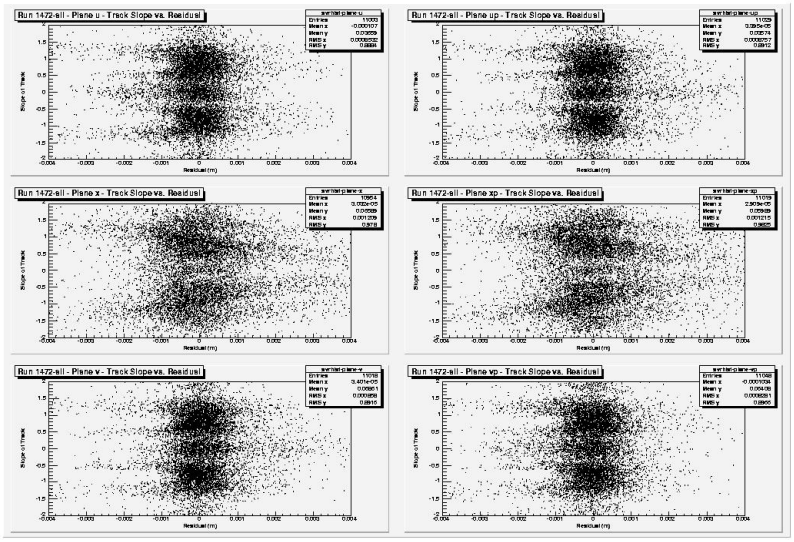

Figure 17: Histogram of the slope of a track vs residual of each hit for a plane

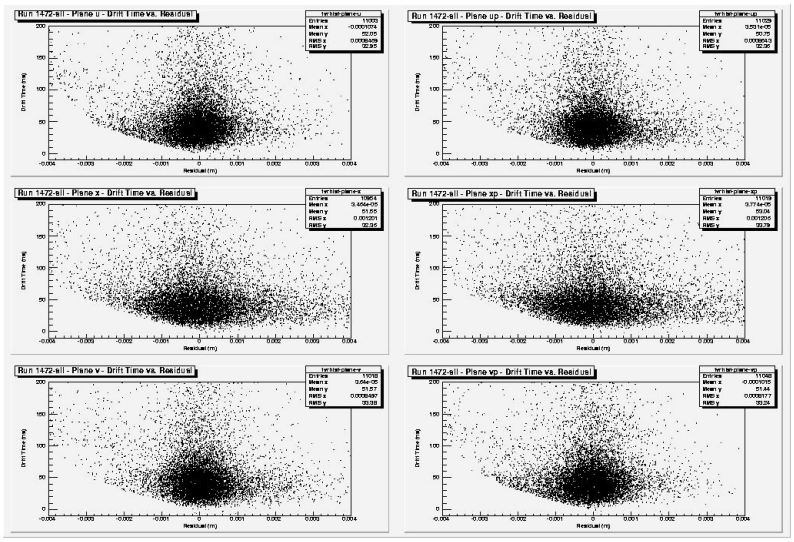

Figure 18: Histogram of the drift time for a hit vs the residual of that hit

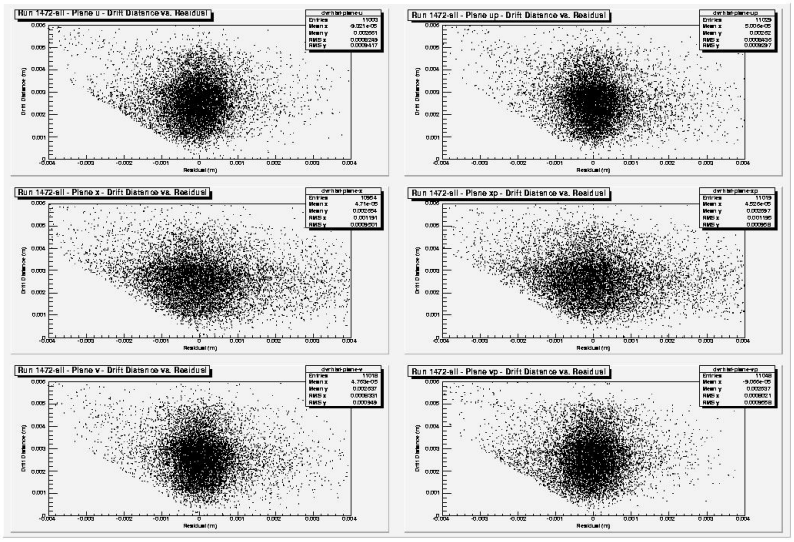

Figure 19: Histogram of the drift time for a hit vs the residual of that hit

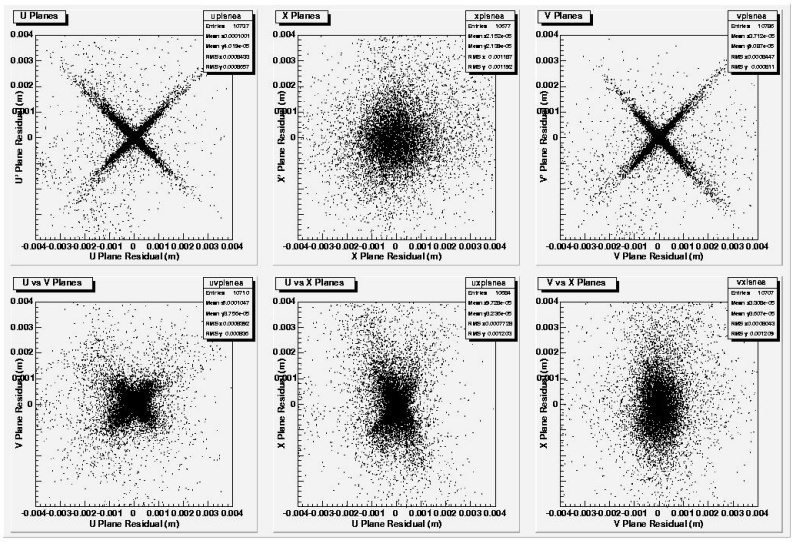

Figure 20: Histogram of the the residuals for first chamber planes plotted against each other

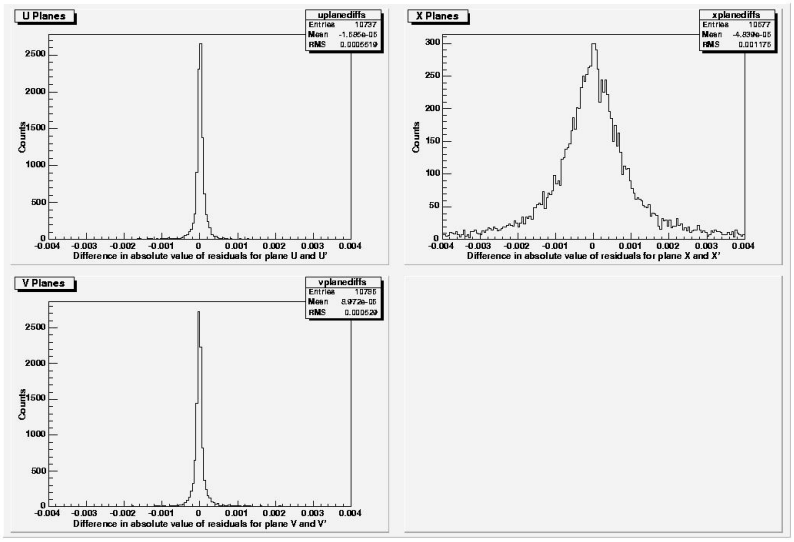

Figure 21: Histogram of the difference of residuals for first chamber planes of the same type

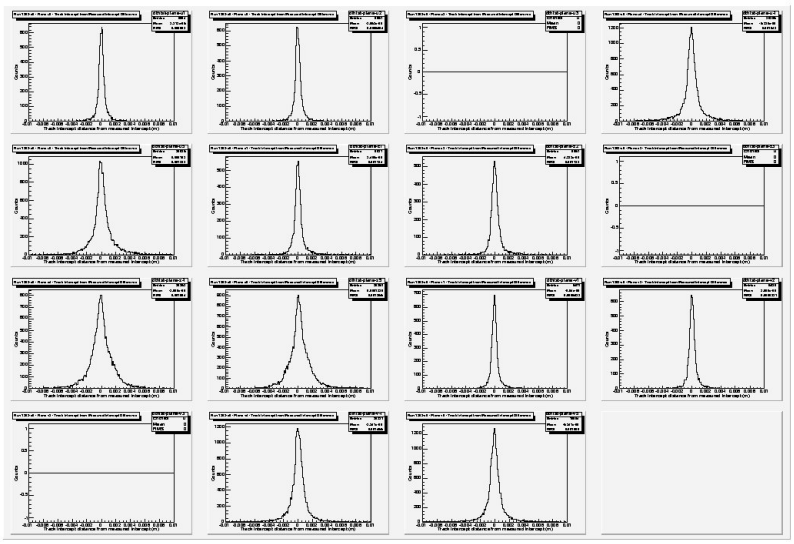

Figure 22: Histogram of the intercept differences (track intercept - measured intercept) for each plane.

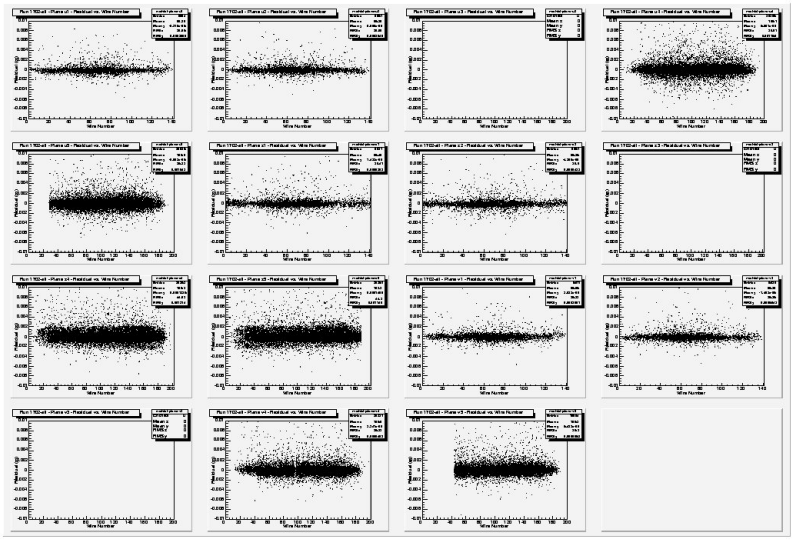

Figure 23: Histogram of the residuals for each wire

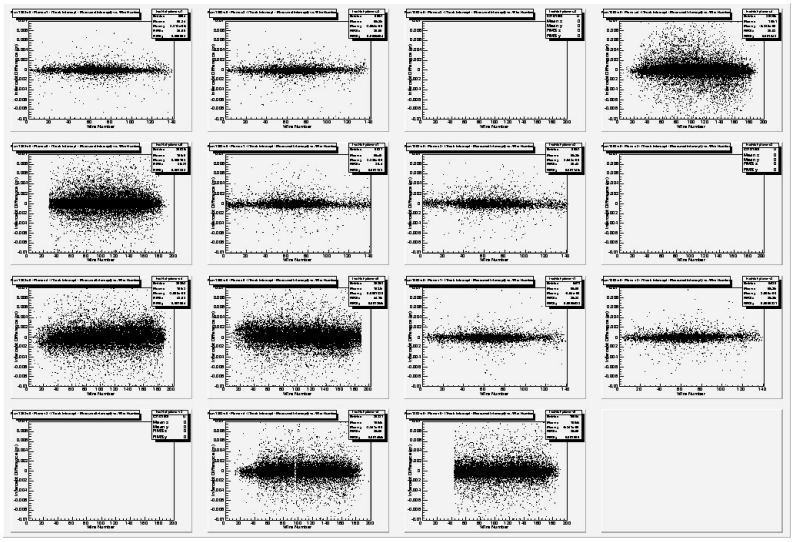

Figure 24: Histogram of the intercept differences for each wire

2.16 showwiresinreconstruct.C

Input: ROOT file, run number
Output: Wires used in reconstruction

This script plots out histograms of which wires were used in the the track reconstruction. See figure 25.

2.17 findeffs.C

Input: ROOT file, run number
Output: Wire Efficiencies, `ps/wireeffs-`*run number*`.ps`

This script plots the efficiencies for bins of wires for all planes. Bins (currently set at 10 wires) are used to reduce the necessary amounts of statistics required to get the efficiency. Cuts are placed on the total shower energy, preshower energy, and χ^2/ Degrees of Freedom which are defined at the top of script. Size of the wire bins are also defined at the top of the script. See figure 26.

2.18 findtrackeff.C

Input: ROOT file, run number
Output: Tracks found for events based on planes passing shower cuts,`ps/trackingeffs-`*run*

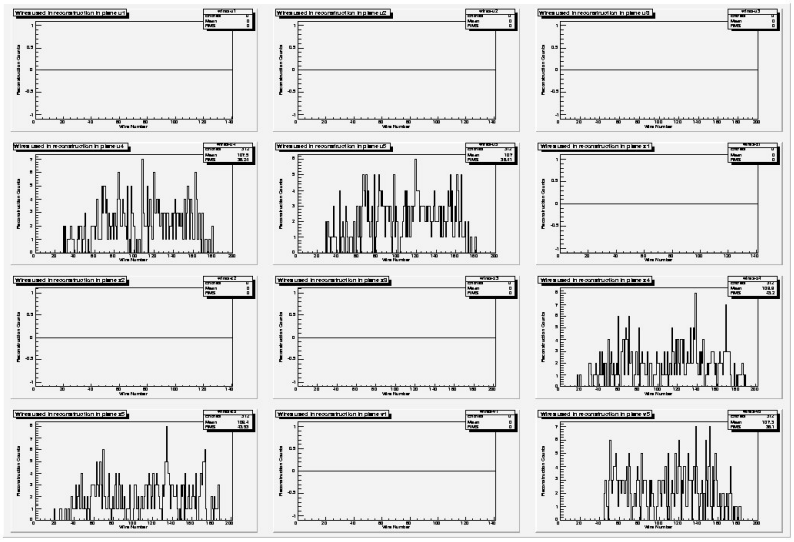

Figure 25: Histogram of the intercept differences for each wire

number`.ps`, `ps/showercut-occupancy-`*run number*`.ps`, `ps/nestgroups-`*run number*`.ps`

This script does not produce a true tracking efficiency.
What is produced is a useful tool for evaluating the tracking code between different runs, but it is very sensitive to noise. What is determined is if a track was found in an event and how many planes had wires passing the shower cuts. Then an graph is produced of the ratio of tracks found for a given number of planes passing the shower cuts and the total number of events that number of planes passed the shower cut. If there were no noise in the system, this would be a measure of the tracking efficiency, but since there may be events where we have a minumum number of planes firing, but not a minimum number of valid planes to reconstruct a track, these numbers are misleading. See figure 27.

Also plotted is the shower cut occupancy, or, a histogram of the number of wires passing a shower cut for a given plane. This also gives you a sense of the amount of noise in the system and an idea of how many wires must be considered in the tracking algorithm. See figure 28.

The estimated number of groups is calculated by, for a given event, taking the number of wires firing on a plane passing the shower cut, and taking the product across all planes (ignore planes without wires passing shower cuts). This roughly gives a sense of how busy an event is and an order of the number of combinations must be considered to do tracking. This plot is useful in helping to determine where to place preemptive tracking cuts in the database to help avoid very busy events. See figure 29.

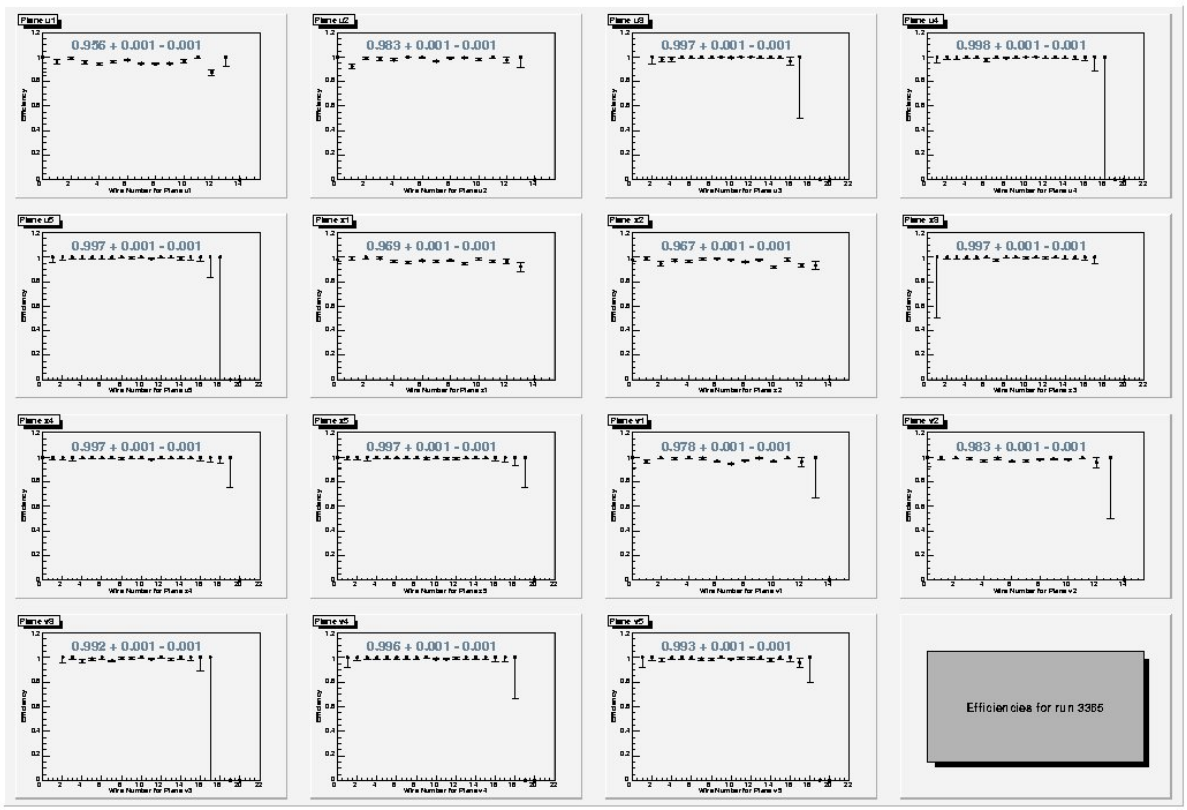

Figure 26: Histogram of the wire efficiencies

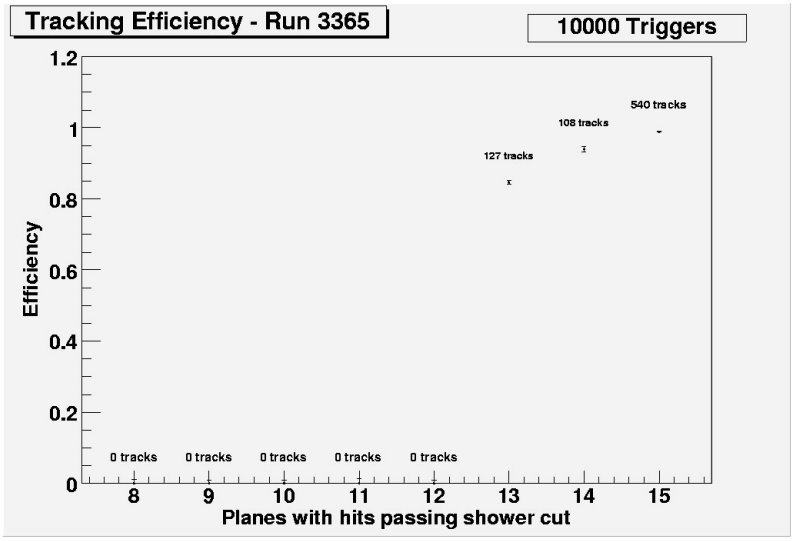

Figure 27: Graph of the percentage of events having tracks given a number of planes firing within the shower cuts.

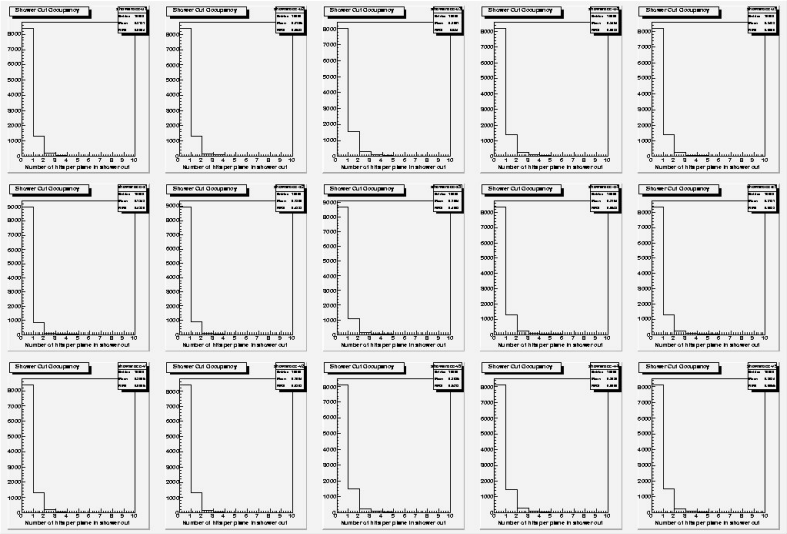

Figure 28: Histogram of number of wires firing within the shower cuts for each plane.

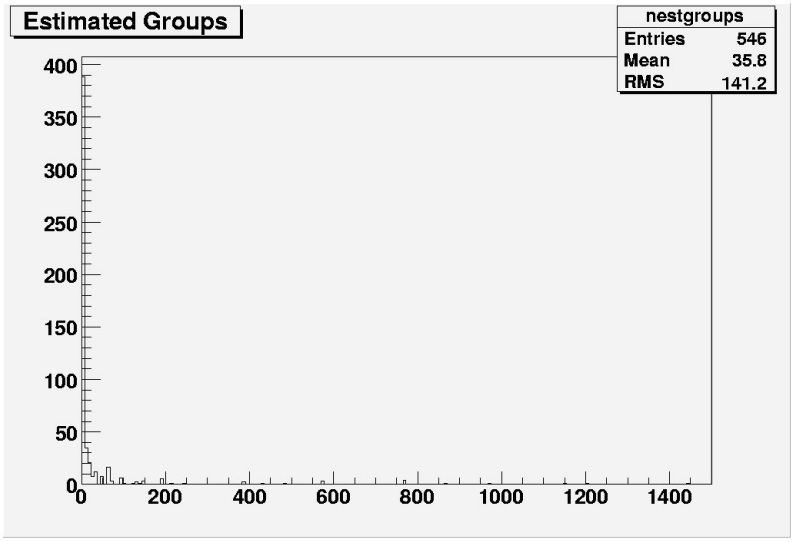

Figure 29: Histogram of number of estimated groups to consider in tracking.

2.19 fitdrift.C

Input: ROOT file, run number
Output: Drift time to distance fit and coefficients, `ps/driftfit-`*run number*`.ps`

Plots the drift distance (found from fitted tracks in tracking code) and plots against the drift time. The polynomial used to fit is:

$$c_1 \tanh(\frac{v_{drift}t}{c_1}) + c_2 * (t - c_3)$$

This polynomial keeps the drift velocity as the leading term. The line for a constant drift velocity model using v_{drift} is shown in a red dashed line. The fit curve is the solid black line. The coefficients as they should be in the database for each plane is shown in the inset in the upper right hand corner. See figure 30.

2.20 findcutarea.C

Input: ROOT file, run number
Output: Distribution of wires around a straight line from the target image to shower cluster, `ps/cutarea-`*run number*`.ps`

This script produces the wire distributions by calculating a line from a given target image (defined at the top of the script in the same coordinate system as accepted by `db_B.dat`) and

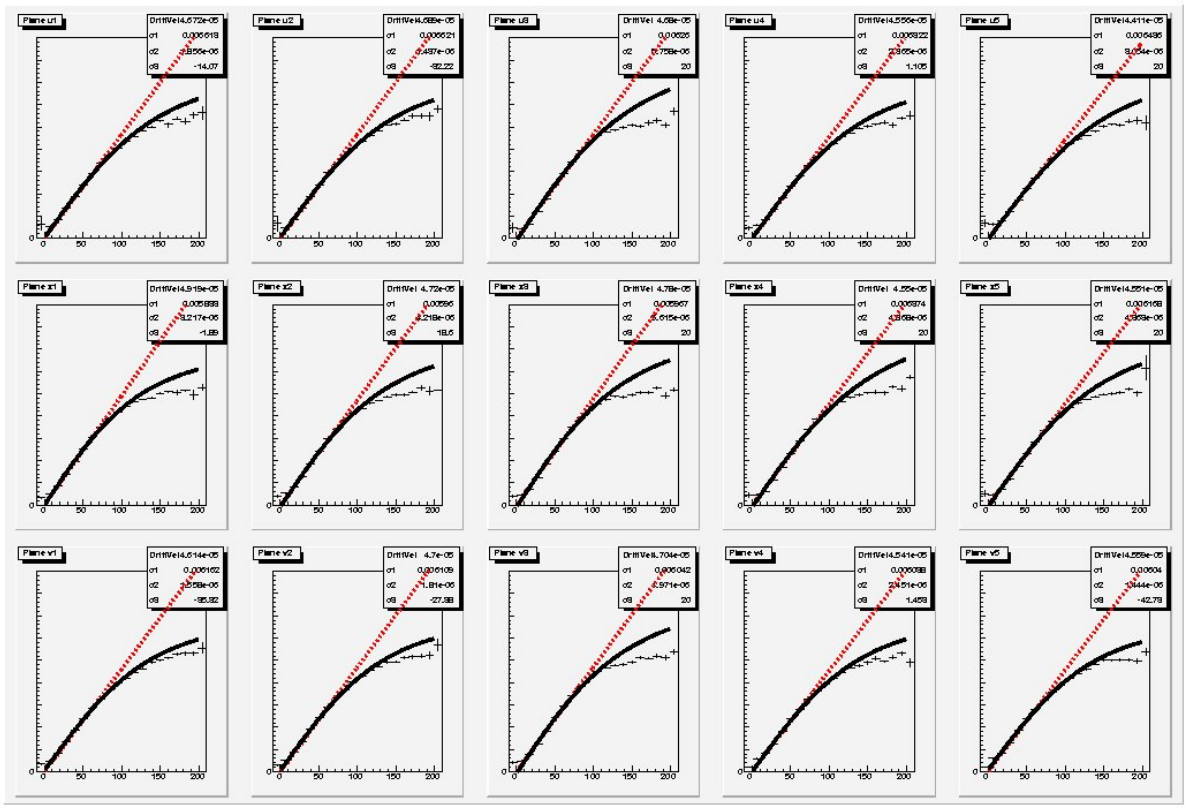

Figure 30: Fit curves to drift distance vs. drift time. The constant velocity model using v_{drift} is shown as a red dashed line. The fit curve is shown as a solid black line.

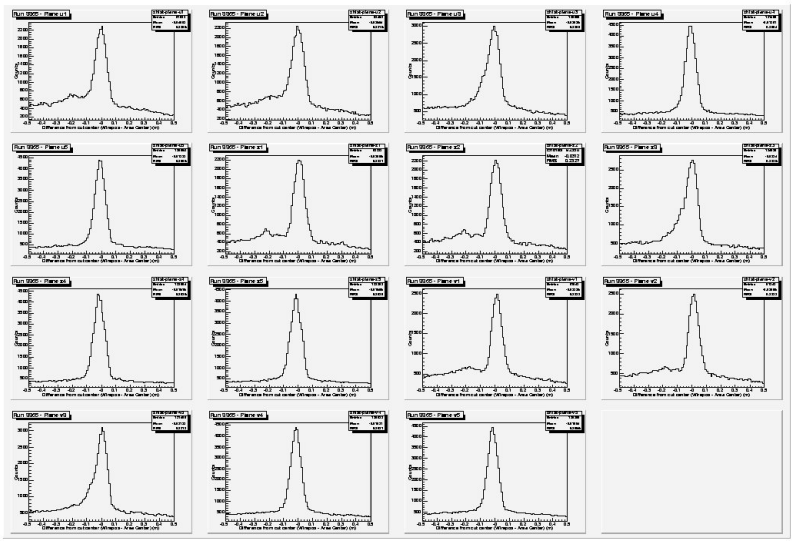

Figure 31: Distribution of wires around a straight line from the target image to shower cluster.

the position of the shower cluster, then finding the intersection of that line with each plane, and then plotting out the distance from all the hit wires from that point. This produces a nice representation of the signal in the drift chambers that caused the shower cluster. The idea of this script is to determine where the location of the target image should be for a given configuration. By varying the position defined at the top of the script in `TARGETOFFSET`, some value will have the center of the signal peaks line up around 0.0 for all planes. This value is what should go in the database `db_B.dat` for the target image position. See figure 31.

2.21 reduceroot.C

Input: ROOT file, run number
Output: ROOT file with a subset of events from input ROOT file

This script is a template to take one ROOT file and strip out a subset of events (for example events that only have tracks). The example provided in the associated tarball of files is not very useful, but serves a template.

2.22 checkwiremap.pl

Input: `db_B.dc.dat` database in the directory with the script
Output: Wire - TDC Information

This script takes the database in the directory and outputs the wiremap in a human readable format. The output is done sorted by wire number and sorted by TDC/channel number.

2.23 t0todb.pl

Input: `db_B.dc.dat` database in the directory with the script, `offsets.dat`
Output: `db_B.dc.dat-witht0`

Takes the data in `offsets.dat` and puts it into the database `db_B.dc.dat-witht0` using `db_B.dc.dat` as a template. The file `offsets.dat` is generated by the script `calct0.C`.

A mwdc_defs.h

This is a sample of the code found in `mwdc_defs.h`.

```
// Definitions file for BigBite Drift Chamber
// Change accordingly

 //Maximum number of wires on any given plane
#define NUMBER_OF_WIRES 210

//Maximum number of hits we could consider
#define MAX_NUMBER_OF_HITS 10000

//Maximum string length
#define STRING_LENGTH 255

//Number of planes we have
#define NUMBER_OF_PLANES 15

//Names of planes we have
enum plane_type { u1, u2, u3, u4, u5, x1, x2, x3, x4, x5, v1, v2, v3, v4, v5 };

// String form of the names of the planes
// The strings must be delimited by the NULL character \0
char plane_type_name[NUMBER_OF_PLANES][STRING_LENGTH] =
  { "u1\0", "u2\0", "u3\0", "u4\0", "u5\0", "x1\0", "x2\0", "x3\0", "x4\0",
    "x5\0", "v1\0", "v2\0", "v3\0", "v4\0", "v5\0" };

int plane_number_of_wires[NUMBER_OF_PLANES] = { 141, 141 ,200, 200, 200,
141, 141, 202, 202, 202,
```

```
141, 141, 200, 200, 200};

#define SIMROOTFILE_FORM "/home/riordan/gen/mwdctlab-all/rootfiles/allbig_%s-sim.root"
#define ROOTFILE_FORM "/home/riordan/gen/mwdctlab-all/rootfiles/allbig_%s.root"
#define ROOTFILEALL_FORM "/home/riordan/gen/mwdctlab-all/rootfiles/allbig_%s-all.root"
#define ROOTFILESMALL_FORM "/home/riordan/gen/mwdctlab-all/rootfiles/allbig_%s-small.roo
#define AGEN_LIB "/home/riordan/gen/hallasoft/agen/libAGen.so"
#define RAWDATA_FORM "/home/riordan/gen/data/AllBigOnes05_%s.dat.0"

#define NULL 0

char *ListPlaneNames()
{
  plane_type plane;
  TString *plane_list_string = new TString("\0") ;
  char final_string[STRING_LENGTH];

  for( plane = 0; plane < NUMBER_OF_PLANES; plane++ )
    {
      plane_list_string->Append( plane_type_name[plane] );
      plane_list_string->Append(" ");
    }

  return plane_list_string->Data();
}
```

B dispatches.dat

This is a sample of the dispatch.dat code used to drive dispatches.

```
#This mimics the standard analyzer for a spectrometer B
#Line name   Command         Arguement
             B.Clear
Decode:      B.Decode
             AN.EndStage     Decode
             brNotOK         MissedCut
Coarse:      B.CoarseProcess NonTracking
             B.CoarseProcess Tracking
Fine:        B.FineProcess   NonTracking
             B.FineProcess   Tracking
             FillTree
MissedCut:   Terminate
```

C offsets.dat

This is a sample of the output generated by `calct0.C`.

```
Plane u
  0 0.0    1 0.0    2 0.0    3 0.0    4 0.0    5 0.0
  6 13.6    7 13.6    8 11.9    9 11.9   10 12.8   11 8.6
 12 13.6   13 12.8   14 10.2   15 12.8   16 12.8   17 13.6
 18 12.8   19 5.2   20 11.9   21 10.2   22 7.7   23 9.4
 24 8.6   25 11.9   26 10.2   27 9.4   28 11.9   29 11.9
 30 11.9   31 7.7   32 11.1   33 12.8   34 10.2   35 12.8
 36 10.2   37 10.2   38 9.4   39 10.2   40 7.7   41 10.2
 42 10.2   43 10.2   44 8.6   45 11.9   46 7.7   47 7.7
 48 9.4   49 6.9   50 -2.4   51 10.2   52 9.4   53 9.4
 54 8.6   55 7.7   56 9.4   57 12.8   58 7.7   59 11.9
 60 4.4   61 11.1   62 12.8   63 10.2   64 10.2   65 12.8
 66 7.7   67 8.6   68 9.4   69 6.9   70 11.1   71 6.0
 72 12.8   73 13.6   74 9.4   75 9.4   76 11.1   77 12.8
 78 10.2   79 7.7   80 9.4   81 12.8   82 5.2   83 7.7
 84 11.1   85 9.4   86 11.1   87 1.0   88 3.5   89 10.2
 90 7.7   91 10.2   92 6.9   93 5.2   94 4.4   95 11.1
 96 2.7   97 12.8   98 9.4   99 7.7  100 -1.5  101 3.5
102 6.9  103 6.9  104 4.4  105 6.0  106 9.4  107 1.8
108 7.7  109 5.2  110 -0.7  111 11.9  112 7.7  113 6.0
114 -2.4  115 6.0  116 6.9  117 2.7  118 6.0  119 1.0
120 7.7  121 4.4  122 9.4  123 1.0  124 8.6  125 5.2
126 7.7  127 1.8  128 4.4  129 0.0  130 0.0  131 0.0
132 0.0  133 0.0  134 0.0  135 0.0  136 0.0  137 0.0
138 0.0  139 0.0  140 0.0  141 0.0

...
```