HrbDecoder documentation

JINR Summer Student Programme project report

Author: Rafał Bielski

Project title: Data processing and particle track reconstruction for a hexagonal wire chamber for the MPD testbeam

Supervisors: Vadim Babkin, Mikhail Rumyantsev, Vyatcheslav Golovatyuk

Contents

1. Introduction

The aim of this project was to develop a program allowing to decode raw output data sent by front-end electronics of a hexagonal wire chamber detector and to reconstruct particle tracks from this data. The detector is being developed for the needs of a testbeam facility for the future MPD experiment. It comprises six planes, each consisting of 96 wires and attached to a custom readout board HRB6ASD (its documentation can be found online at `http://afi.jinr.ru/HRB6ASD`). An object-oriented C++ application was created, which consists of two classes, one providing tools for data readout and decoding and second providing a track reconstruction. Moreover, it comprises a header file providing tools for drawing an event display and an executable program combining and using all these tools. The application was named after one of the classes – HrbDecoder. The project is based on the STL and ROOT libraries.

This report is supposed to help potential users to understand the operation of the program and the used algorithms. It introduces the quantities used throughout the project, describes the structure of the program and its operation step by step. Instructions on usage of the application are provided, as well as results of its operation on a test dataset collected with cosmic rays. The project can be understood best, when both – the source code and the report – are read in parallel.

2. Input data

Six data files – one for each board/plane – were recorded during a short period of cosmic particle data taking and they are used as input to the program. Such a data file consists of a set of events encoded in a binary format described at `http://afi.jinr.ru/DataFormatHrb`. During the detector operation, the HRB collects a signal from the wires with a certain sampling interval, Δt, and records a certain number of samples, N_s, for each event. In the dataset most extensively used during the program development, $\Delta t = 8$ ns and $N_\mathrm{s} = 42$, but a different dataset with $N_\mathrm{s} = 37$ was also tested. The data from an event is encoded in the HRB Raw Data Format and after a set of header information, contains N_s 128-bit words – one for each time sample. Each of the first 96 bits corresponds to a wire, whereas the last 32 bits are always empty.

3. Quick start

In order to simply compile and run the program in a bash shell in Linux without changing any settings, only two commands are needed. Assuming the ROOT libraries are set up properly, the compilation is done by executing the command `make` in the program directory. The program is run by simply typing `./runHrbDecoder` in the command line, followed by six arguments being the locations of the input data files. Assuming there is a subdirectory `data` which contains six `.dat` files with the HRB data, the program may be run with the command `./runHrbDecoder data/*.dat`.

4. Definitions

The hexagonal wire chamber comprises six planes, each rotated by an angle of 60° with respect to the preceding plane. The planes are labeled starting from the top: X1, YL1, YL2, X2, YR1, YR2. Each consists of 96 wires numbered clockwise, as seen in Figure 1. Distance between wires in one plane is equal to $d_{\mathrm{w}} = 2.5$ mm, whereas the vertical distance between neighbouring planes equals $d_{\mathrm{z}} = 10.0$ mm. Parallel planes are shifted with respect to each other by half of this width. A right-handed Cartesian coordinate system is defined with the z-axis pointing upwards, perpendicular to the planes and $z = 0$ aligned with the X1 plane. The point $(x, y) = (0, 0)$ is located in the middle between 47th wires of all planes, as seen in Figure 1b. The x-axis is perpendicular to the wires of the X1 and X2 planes and oriented in the direction of growing number of X2 wires. The y-axis is parallel to the X1 and X2 wires.

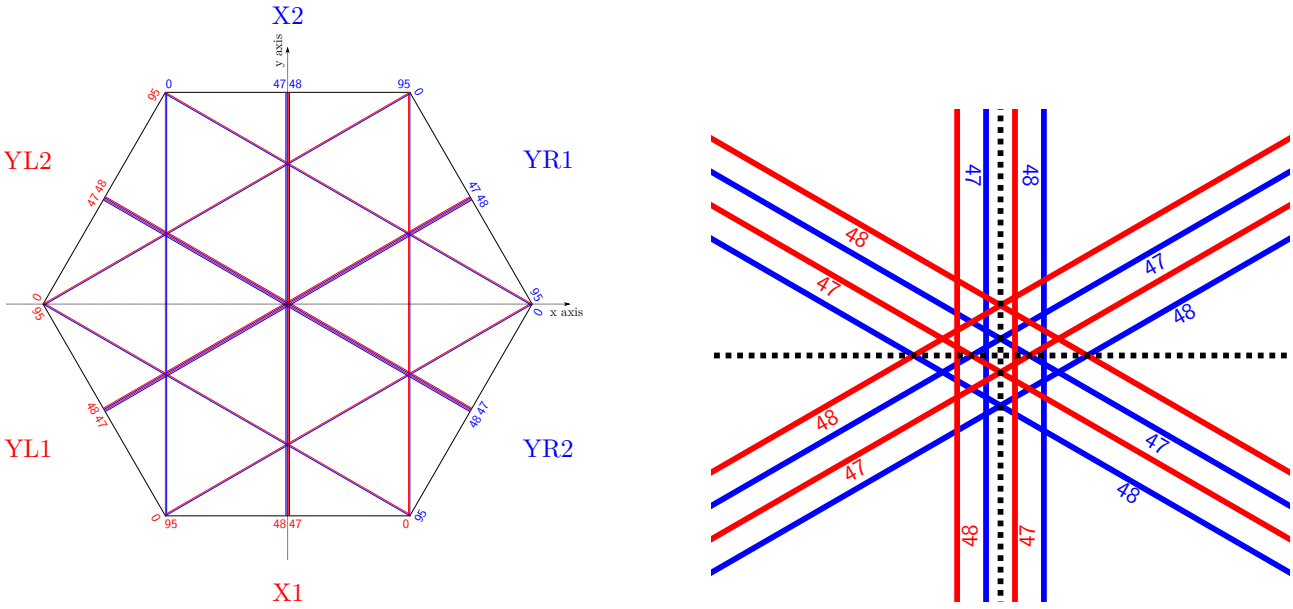


Figure 1: (a) Definition of the coordinate system with respect to the wire positions. Wire numbers are shown around the detector. (b) Close-up of the central region.

Two auxiliary axes were defined – $y1$ perpendicular to YL1 and YR1 wires and $y2$ perpendicular to YL2 and YR2 wires, as shown in Figure 2. With these definitions, wire coordinates in each plane can be expressed along the corresponding axis. With n_{P} being the wire number in plane P, they are defined as follows:

$$
\begin{aligned}
\text{X1}: &\quad x = (46.75 - n_{\mathrm{X1}}) \cdot d_{\mathrm{w}} \\
\text{YL1}: &\quad y1 = (n_{\mathrm{YL1}} - 46.75) \cdot d_{\mathrm{w}} \\
\text{YL2}: &\quad y2 = (n_{\mathrm{YL2}} - 46.75) \cdot d_{\mathrm{w}} \\
\text{X2}: &\quad x = (n_{\mathrm{X2}} - 46.75) \cdot d_{\mathrm{w}} \\
\text{YR1}: &\quad y1 = (46.75 - n_{\mathrm{YR1}}) \cdot d_{\mathrm{w}} \\
\text{YR2}: &\quad y2 = (46.75 - n_{\mathrm{YR2}}) \cdot d_{\mathrm{w}}
\end{aligned}
$$

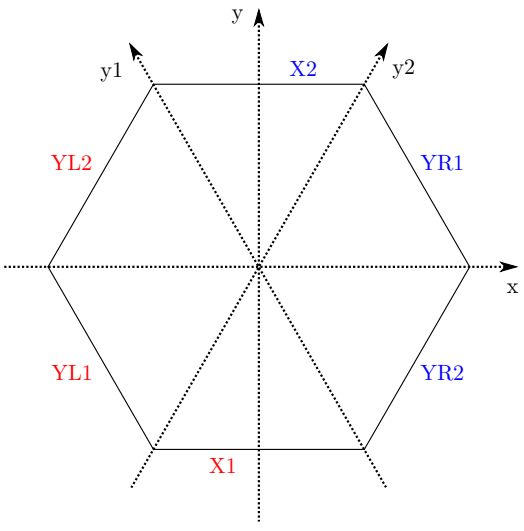


Figure 2: Definition of auxiliary axes $y1$ and $y2$

The primary goal of this application is to decode the raw output data sent by the front-end electronics of each plane and reconstruct a single particle track in each event. In the absence of magnetic field, the tracks are expected to be straight lines. One of the possible definitions of a three-dimensional straight line is by a point and a direction vector, another one comprises a point and two angles - azimuthal and polar. The program uses both, delivering information about a point where the track crosses the $z = 0$ plane (i.e. the X1 plane), $P_0 = (x_0,\ y_0,\ 0)$, azimuthal angle, ϕ, polar angle, θ, and three coordinates of normalised particle momentum vector, $\mathbf{p} = [p_x,\ p_y,\ p_z]$. Length of this vector is always equal to unity. The vertical coordinate p_z is assumed to be always negative. Definition of the parameters is presented in Figure 3. Relations between them are as follows:

$$
\begin{aligned}
p_x &= \cos\phi\ |\sin\theta| \\
p_y &= \sin\phi\ |\sin\theta| \\
p_z &= -|\cos\theta| = -\sqrt{1 - p_x^2 - p_y^2}
\end{aligned}
$$

$$
\begin{aligned}
p_x,\ p_y &\in [-1;\ 1] \\
p_z &\in [-1;\ 0] \\
\phi &\in [0;\ 2\pi) \\
\theta &\in [0;\ \pi/2]
\end{aligned}
$$

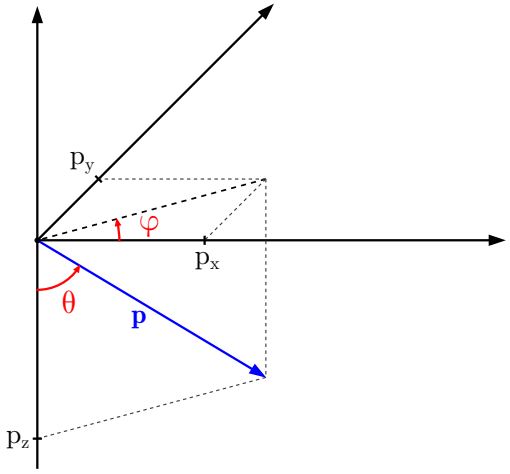


Figure 3: Definition of track parameters

Please note that the polar angle is defined in the opposite direction than usually. In this case, with the assumption of negative p_z, the angle θ remains in the range $[0;\ \pi/2]$. Although $\sin\theta$, $\cos\theta$ and $\tan\theta$ are always non-negative in this range, absolute values are still used throughout this report in order to emphasize this fact. Other useful relations allow to calculate x, y, $y1$ and $y2$ coordinates of the track at any given z:

$$
\begin{aligned}
x(z) &= x_0 + z\tfrac{p_x}{p_z} & \qquad y1(z) &= -\tfrac{1}{2}x(z) + \tfrac{\sqrt{3}}{2}y(z) \\
y(z) &= y_0 + z\tfrac{p_y}{p_z} & \qquad y2(z) &= \tfrac{1}{2}x(z) + \tfrac{\sqrt{3}}{2}y(z)
\end{aligned}
$$

5. Structure of the program

The whole project is based on two classes, `HrbDecoder` and `HrbEvent`, and an executable program `runHrbDecoder`. Additional functions allowing to create event displays are collected in a separate header file. A basic Makefile was created in order to simplify the compilation of the program. During the development of the project, also a simple simulation has been performed, which helped to ensure correct track reconstruction. However, due to a large extent of simplification, the simulation did not include crucial features of real data. Hence, it is not described in this report. The project uses extensively the STL and ROOT (v5.34) libraries. Their documentation can be found online at `http://cplusplus.com` and `http://root.cern.ch` respectively.

5.1. Class HrbDecoder

The class `HrbDecoder`, defined in the file `HrbDecoder.h` and implemented in the file `HrbDecoder.cpp`, serves as a tool for decoding raw HRB data and extracting information about events and hits. It provides methods for reading, checking and printing all header information, as well as the event data. At the construction of an object of this class, a number of files is opened, using the STL `ifstream` class (one file per one HRB board / detector plane). The readout methods of this class read subsequent words of a chosen file. The user has to know the structure of the file (see Section 2) and understand the location of the readout pointer in order to control what information is read. The most important method of this class is `ReadHits`, which reads all the event data (N_s 128-bit words) and retrieves the hit information, returning it as a list of 3-element vectors including the plane number, the time sample number and the channel (wire) number. All members and methods are briefly described in Appendix A.

5.2. Class HrbEvent

The class `HrbEvent`, defined in the file `HrbEvent.h` and implemented in the file `HrbEvent.cpp`, stores all information relevant to a single event. At the construction of an object of this class, track reconstruction is performed, which is described in details in Section 6.4. If the reconstruction succeeds, the method `HrbEvent::IsGoodEvent()` returns `kTRUE` and all the track parameters can be retrieved using dedicated methods. All members and methods are briefly described in Appendix B.

5.3. Executable program runHrbDecoder

The executable program implemented in the file `runHrbDecoder.cpp` is an easy-to-use application allowing to obtain full information about events and particle tracks, having only raw data files as an input. Moreover it enables the user to obtain event displays or to print any relevant information to the standard output. Six raw datafiles should be provided as an input in the command line arguments, and the rest is fully automatised. Some flags may be changed in the source code, controlling the behaviour of the program, as described in Section 6.1. Also commenting/uncommenting certain parts of code may become useful in its usage.

5.4. EventDisplay

The file `EventDisplay.h` provides several functions allowing to draw an event display. The file is included in the source of `runHrbDecoder`. In order to generate an event display, it is sufficient to call the function `EventDisplay(HrbEvent*)`. It will generate a PDF file `eventDisplay.pdf` in the current directory containing four projections of the detector – ZX, ZY1, ZY2, YX – with the hits and the track and pause the program execution. Please note that the YX projection shows only the averaged good hit location in each plane, while all the other projections show all good hits in green and all bad hits in red. The functions provided in this header file may be also used to generate a ROOT file containing all the event displays. A block of code doing this is provided in the file `runHrbDecoder.cpp` and commented out. Please expect this procedure to take a significant amount of time and generate a big ROOT file (a few hundred MegaBytes).

6. The program operation step by step

6.1. Initialisation

At first, the printing level is set in the program. One can control the amount of information printed to the standard output by changing the value of `gErrorIgnoreLevel` to `kInfo`, `kWarning`, `kError` or any other value defined in `TError.h`. Next, one can set the flag `displayEbyE`. If its value is `kTRUE`, the program draws all events one by one and pauses after each. If the flag `displayOnlyGood` is also set to `kTRUE`, only good events will be drawn. The flag `printRecoInfo` may be set if one wants to print a detailed information during event reconstruction even if `gErrorIgnoreLevel` is high. Afterwards, the program checks whether the data types used to read 32- and 64-bit words from the files are indeed 32- and 64-bit sized.

After this first setup, an array of filenames is created from command line arguments and used to initialise a `HrbDecoder` object. After that, plane numbers and names have to be assigned to serial numbers of the HRB boards. It is important to fill the ID and name vectors in the order from the top to bottom plane, as the vector index is then used as the plane number.

In the next step, two ROOT trees are created – one which will be used to sort the events and second which will store information of all good events and will be written to the output file.

6.2. Data readout

The data readout is performed in a while-loop, which ends when the readout pointer in any of the data files encounters a problem or the end of the file, or if a bad Sync word[1] is read. Each turn of the loop reads one event from each plane. In a perfect case, all data from HRB boards should be well synchronised and the same event should be read from all planes. However, the data used as input to this program, was sent from the HRBs via the TCP/IP protocol and might have lost the synchronisation. Due to that fact, an event sorting procedure is applied, which is described in the next section. In order to read one event from each file, a for-loop is performed over the files. All subsequent words of the data frame are read from a file and printed, if the printing level is set to `kInfo` or lower. The event timestamp and a list of hits from a single plane are written to the sorting tree (for each plane separately).

6.3. Event sorting

After the data readout, the sorting tree contains about $N_{\text{planes}} \cdot N_{\text{events}}$ entries. Each entry contains a timestamp and a list of hits recorded in one plane at this time. The entries are sorted by timestamp, and hit lists corresponding to one timestamp are merged into one list, which is used as an input to the `HrbEvent` object constructor. Every new event is added to an array of events (`HrbEvent` objects).

[1] Sync word is the first word in each data frame, which is supposed to be "*P". For details, see `http://afi.jinr.ru/M-Link%20Data%20Layer`.

6.4. Event reconstruction

The event reconstruction is performed in the constructor of an `HrbEvent` object. At first, the raw hit list provided as an input is sorted by plane numbes using the `SortHits()` method. This procedure is not necessary in the current implementation of the reconstruction, but it may become useful in case of any modifications. Hits of a single plane in a single event are ordered by the time sample without any need of sorting, as they are sent by the HRB in a single data frame.

In the first step of event reconstruction, all hits are marked as good or bad hits. This is done for each plane separately. The first (in time) hit in a plane is always marked as good. Subsequent hits are good only if the time difference with respect to the first hit equals two samples ($2 \cdot \Delta t$) or less. This procedure rejects problematic hits which are seen in the data, emerging in the same or neighbouring wire as the first hit, but after a relatively long time (e.g. five samples, which corresponds to 40 ns in case of $\Delta t = 8$ ns). All good hits are stored in a new container (a vector of vectors). Having the good hits array (wire numbers), hit coordinates along axes corresponding to each plane are calculated (x, $y1$ or $y2$), as described in Section 4. Also an average hit location in each plane is calculated, taking only good hits into account.

Afterwards, each plane is marked as a good or bad plane. A plane is good only if the number of good hits in it, N_{gh}, equals 4 or less. The difference between wire numbers associated with any two good hits has to be equal to N_{gh} or less. This condition leaves a space for only one unfired wire in a hit cluster and defines the maximal cluster size to five wires. At this stage, an event may be marked as bad if the number of good planes, N_{gp}, is lower than 4 or if $N_{\mathrm{gp}} = 4$, but two parallel planes are bad (X1&X2 or YL1&YR1 or YL2&YR2). In both cases there is too little information for unequivocal track determination. Thus, if the event is marked as bad at this stage, particle track reconstruction will not be performed.

The reconstruction of a particle track in this program is defined as a least squares problem. A 4-argument function $S(\phi, \theta, x_0, y_0)$, calculated as a sum of squared distances between the track and all good hits in good planes, is minimised using a numerical minimisation algorithm. A track is assumed to be a three-dimensional straight line defined by four parameters, as described in Section 4. In general, S can be written as:

$$S(\phi, \theta, x_0, y_0) = \sum_{ij} d_{ij}^2,$$

where $i \in \{\mathrm{X1}, \mathrm{YL1}, \mathrm{YL2}, \mathrm{X2}, \mathrm{YR1}, \mathrm{YR2}\}$ is the plane index, $j = 1, 2, 3, ...$ is the hit index in a particular plane and d_{ij} is a distance between the track and the j-th hit in the i-th plane. The track-hit distances in a particular plane are calculated along the axis perpendicular to its wires (e.g. along x for X1 or along $y1$ for YL1), at the same value of z. For example, if the first wire fired in the YR1 plane (at $z = -d_{\mathrm{z}}$) was the 21st, then:

$$\begin{aligned} y1_{\mathrm{YL1},1} &= (21 - 46.75) \cdot d_{\mathrm{w}} = -64.375 \text{ mm} \\ d_{\mathrm{YL1},1} &= |y1_{\mathrm{YL1},1} + \tfrac{1}{2}(x_0 - d_{\mathrm{z}}\tfrac{p_x}{p_z}) - \tfrac{\sqrt{3}}{2}(y_0 - d_{\mathrm{z}}\tfrac{p_y}{p_z})| \\ &= |y1_{\mathrm{YL1},1} + \tfrac{1}{2}(x_0 + d_{\mathrm{z}}\cos\phi|\tan\theta|) - \tfrac{\sqrt{3}}{2}(y_0 + d_{\mathrm{z}}\sin\phi|\tan\theta|)| \end{aligned}$$

The function S is minimised using the `MIGRAD` algorithm implemented in the Minuit2 minimiser framework implemented in ROOT. Its documentation may be found at `http://root.cern.ch/drupal/content/minuit2-manual-534`. All parameters of the fitted track may be obtained from the `HrbEvent` object using dedicated methods (see the reference in Appendix B). Although a simple usage of the minimiser, as described above, gives quite satisfactory results, several rare problems emerged during the project development. Some of them have been identified and two additional procedures were applied – a choice of ϕ-angle range and a few additional requirements for a good event.

During the work on this project, a simulation was developed, which was rather oversimplified but helped to resolve all major problems with the implementation of reconstruction algorithm. It has also shown, that in some cases the minimiser is not capable of finding a correct value of the ϕ-angle. In order to minimise errors arising from this behaviour, dependence of hit locations on the true ϕ was studied and an algorithm constraining the angle values was developed. Three auxiliary variables were employed, defined as differences between average good hit locations in parallel planes:

$$\begin{aligned} \Delta x &= \langle x \rangle^{\text{good}}_{\text{X2}} - \langle x \rangle^{\text{good}}_{\text{X1}} \\ \Delta y1 &= \langle y1 \rangle^{\text{good}}_{\text{YR1}} - \langle y1 \rangle^{\text{good}}_{\text{YL1}} \\ \Delta y2 &= \langle y2 \rangle^{\text{good}}_{\text{YR2}} - \langle y2 \rangle^{\text{good}}_{\text{YL2}} \end{aligned}$$

which are closely related to the 'true' values calculated for a track:

$$\begin{aligned} (\Delta x)_{\text{t}} &= x(-3d_{\text{z}}) - x(0) &&= 3d_{\text{z}} |\tan\theta| \ \cos\phi \\ (\Delta y1)_{\text{t}} &= y1(-4d_{\text{z}}) - y1(-d_{\text{z}}) &&= 3d_{\text{z}} |\tan\theta| \ \cos(\phi - \tfrac{2\pi}{3}) \\ (\Delta y2)_{\text{t}} &= y2(-5d_{\text{z}}) - y2(-2d_{\text{z}}) &&= 3d_{\text{z}} |\tan\theta| \ \cos(\phi - \tfrac{\pi}{3}) \end{aligned}$$

Using the above relations, one can extract some information about ϕ knowing only the sign of Δx, $\Delta y1$ and $\Delta y2$. This may be clearly seen in Figure 4.

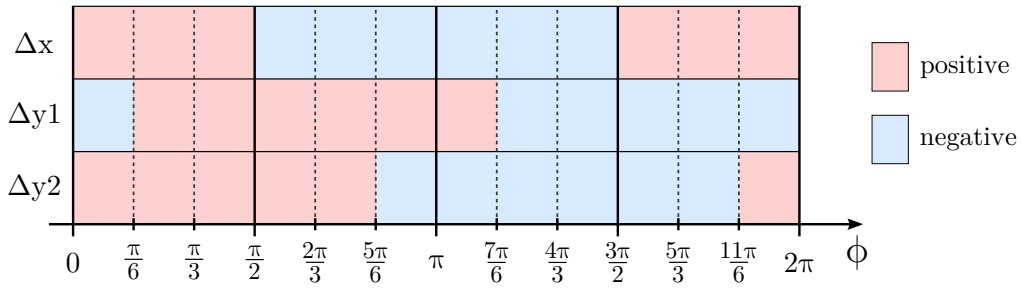


Figure 4: Sign of $(\Delta x)_{\text{t}}$, $(\Delta y1)_{\text{t}}$ and $(\Delta y2)_{\text{t}}$ depending on the value of ϕ

The relations were used to determine a range of ϕ which was used as a constraint in the minimiser. Each of the differences calculated from a true track can equal zero only in a few single points, i.e. almost never. Nevertheless, a zero value happens relatively often for the differences calculated from real hits due to discrete location of the wires, and the fact that particles usually cross the detector at small value of θ. Because of that, all the cases with zeros had to be resolved separately and the ranges were determined basing on the simulation. In such a case, the singular value of ϕ is smeared with a

Gaussian-like distribution of a certain width σ due to the mentioned detector geometry effects. A $\pm 2\sigma$ range was used to constrain the minimiser. All possible combinations an the corresponding ϕ ranges are listed in Table 1. As it may be seen, there are two cases which are impossible to obtain with true track calculation, but sometimes occur in the data: $(++-)$ and $(--+)$. This may happen, if a true Δi ($i \in \{x,\ y1,\ y2\}$) is close to zero and in one plane, the wire closest to the track remains unfired, while the next one is fired. If such a combination is found and one of Δi values is close to zero ($\Delta i < d_{\mathrm{w}}$), it is treated as zero and the range is then redetermined. However, impossible combinations with high Δi have been also spotted in the data. Such events are treated as bad and rejected, as a track indeed cannot be fitted to the hits. In the (0 0 0) case, ϕ is undetermined and irrelevant, as $\theta = 0$.

Δx	$\Delta y1$	$\Delta y2$	ϕ	Δx	$\Delta y1$	$\Delta y2$	ϕ	Δx	$\Delta y1$	$\Delta y2$	ϕ
0	0	0	undetermined	+	0	0	0 ± 0.6	−	0	0	$\pi \pm 0.6$
0	0	+	$\pi/3 \pm 0.6$	+	0	+	$\pi/6 \pm 0.1$	−	0	+	$[\pi/2;\ 5\pi/6]$*
0	0	−	$4\pi/3 \pm 0.6$	+	0	−	$[3\pi/2;\ 11\pi/6]$*	−	0	−	$7\pi/6 \pm 0.1$
0	+	0	$2\pi/3 \pm 0.6$	+	+	0	$\pi/3 \pm 0.6$	−	+	0	$5\pi/6 \pm 0.1$
0	+	+	$\pi/2 \pm 0.1$	+	+	+	$[\pi/6;\ \pi/2]$	−	+	+	$[\pi/2;\ 5\pi/6]$
0	+	−	$\pi \pm 0.3$	+	+	−	impossible	−	+	−	$[5\pi/6;\ 7\pi/6]$
0	−	0	$5\pi/3 \pm 0.6$	+	−	0	$11\pi/6 \pm 0.1$	−	−	0	$4\pi/3 \pm 0.3$
0	−	+	0 ± 0.3	+	−	+	$[-\pi/6;\ \pi/6]$	−	−	+	impossible
0	−	−	$3\pi/2 \pm 0.1$	+	−	−	$[3\pi/2;\ 11\pi/6]$	−	−	−	$[7\pi/6;\ 3\pi/2]$

Table 1: All 27 possible Δx, $\Delta y1$, $\Delta y2$ sign combinations and the corresponding ϕ range in case of hits recorded in six planes. One σ width is given, where a central value is noted. The minimiser constraint uses a $\pm 2\sigma$ range instead. The combinations marked with * are extremely rare and the exact central value could not have been determined. Only approximate range is used in these cases.

The whole above discussion applies to a case, when all three Δi values are determined, i.e. there are six good planes in the event. If one of the planes is marked as bad, only two of the Δi values may be calculated and one remains undetermined. All such cases have been collected in Table 2. Unlike in six-plane events, a $\pm 3\sigma$ range is used if the central value and the width are given.

In the events with only four good planes, ϕ cannot be constrained and the full range $[0;\ 2\pi]$ is used in the minimiser.

Δx	$\Delta y1$	$\Delta y2$	ϕ
x	0	0	$[0;\ 2\pi]$
x	0	+	$\pi/6 \pm 0.15$
x	0	−	$7\pi/6 \pm 0.15$
x	+	0	$5\pi/6 \pm 0.15$
x	+	+	$[\pi/6;\ 5\pi/6]$
x	+	−	$[5\pi/6;\ \pi/6]$
x	−	0	$11\pi/6 \pm 0.15$
x	−	+	$[-\pi/6;\ \pi/6]$
x	−	−	$[7\pi/6;\ \pi/6]$

Δx	$\Delta y1$	$\Delta y2$	ϕ
0	x	0	$[0;\ 2\pi]$
0	x	+	$\pi/2 \pm 0.15$
0	x	−	$3\pi/2 \pm 0.15$
+	x	0	$11\pi/6 \pm 0.15$
+	x	+	$[-\pi/6;\ \pi/2]$
+	x	−	$[3\pi/2;\ 11\pi/6]$
−	x	0	$5\pi/6 \pm 0.15$
−	x	+	$[\pi/2;\ 5\pi/3]$
−	x	−	$[5\pi/3;\ 3\pi/2]$

Δx	$\Delta y1$	$\Delta y2$	ϕ
0	0	x	$[0;\ 2\pi]$
0	+	x	$\pi/2 \pm 0.15$
0	−	x	$3\pi/2 \pm 0.15$
+	0	x	$\pi/6 \pm 0.15$
+	+	x	$[\pi/6;\ \pi/2]$
+	−	x	$[-\pi/2;\ \pi/6]$
−	0	x	$7\pi/6 \pm 0.15$
−	+	x	$[\pi/2;\ 7\pi/6]$
−	−	x	$[7\pi/6;\ 3\pi/2]$

Table 2: All 27 possible Δx, $\Delta y1$, $\Delta y2$ sign combinations and the corresponding ϕ range in case of hits recorded in five planes. Missing (undetermined) Δi value is denoted with x. One σ width is given, where a central value is noted. The minimiser constraint uses a $\pm 3\sigma$ range instead.

After determining the constrained ϕ range and running the minimiser algorithm, a set of additional requirements is applied to the event. Another auxiliary variable is used at this stage, which is defined as square root of S at the minimum divided by the number of good hits in good planes, $N_{\mathrm{gh}}^{\mathrm{gp}}$ (i.e. the number of hits contributing to S):

$$d'_{\mathrm{rms}} = \frac{\sqrt{S}}{N_{\mathrm{gh}}^{\mathrm{gp}}} = \frac{\mathrm{RMS}(d_{ij})}{\sqrt{N_{\mathrm{gh}}^{\mathrm{gp}}}}$$

It has been determined in the tested dataset, that a transition between well and badly reconstructed events occurs between $d'_{\mathrm{rms}} = d_{\mathrm{w}}$ and $d'_{\mathrm{rms}} = 2d_{\mathrm{w}}$.

At first, the minimiser status is checked. If the minimiser did not succeed[2], but $d'_{\mathrm{rms}} < d_{\mathrm{w}}$, the event is still considered good. Otherwise, a minimisation is repeated with a looser strategy and precision. If the second minimisation succeeds, the event is good. If it does not, again the $d'_{\mathrm{rms}} < d_{\mathrm{w}}$ requirement decides whether to accept the event or not. If either the first or the second minimisation succeeds, but $d'_{\mathrm{rms}} > 2d_{\mathrm{w}}$, the event is rejected. A tighter cut is imposed on events, where one or more planes were rejected despite having hits (i.e. rejected due to too many or too scattered hits). In this case, events with $d'_{\mathrm{rms}} > d_{\mathrm{w}}$ are rejected.

An additional check is made on the x_0 and y_0 values. If they do not exceed the detector dimensions, the event is accepted. In the dataset used in the project development, no event was rejected at this stage. After fulfilling all the above requirements, the event is ultimately accepted and all the track parameters are saved in the object member values. These can be later retrieved using dedicated 'getter' methods.

[2] A sucessful minimisation returns a status=0. For the possible errors and corresponding status values, please see the Minuit2 documentation.

6.5. Writing the output

All events, good and bad, may be displayed just after the reconstruction (in the same loop which creates `HrbEvent` objects), if the `displayEbyE` flag is `kTRUE` and `displayOnlyGood` is `kFALSE`. The event display will be written to the file `eventDisplay.pdf` in the current directory and the program will be paused. After pressing Enter, next event will be reconstructed and displayed. If the flag `printRecoInfo` is also `kTRUE` each `HrbEvent` constructor will print very detailed information about the event to the standard output. It may be then redirected to a log file.

After the event reconstruction loop ends, another loop is initialised which goes through the events array. In this loop, every good event information is saved to the ROOT tree (named `HrbTree`) which will be written to the output file. If `displayEbyE=kTRUE` and `displayOnlyGood=kTRUE`, the good events are displayed at this stage in the same way as described above. In the last step, the tree is written to a file `output.root` in the current directory and the program ends.

An additional feature is available, which is commented out in the file `runHrbDecoder.cpp`. Uncommenting the large code block at the end of the file will result in writing all event displays to a separate ROOT file. The procedure takes a significant amount of time (several minutes) and the resulting file is large (a few hundred MegaBytes).

7. Usage

The program was developed and tested only in Linux (Mint 16, 64-bit), but it does not use any system- or architecture-specific features and should be easily usable on other platforms. It depends only on the STL and ROOT libraries, thus it is crucial to set up the latter properly. Especially, the dynamic linker has to know the location of ROOT libraries, which is achieved by sourcing the script `thisroot.sh` which is available in the ROOT installation directory, in the `bin` subdirectory.

The program can be easily compiled by just executing the command `make` in the application directory. Then, it may be run with the command `./runHrbDecoder` followed by six arguments being locations of six input datafiles. In case of having a subdirectory `data` containing only six `.dat` files, one can type `./runHrbDecoder data/*.dat`. Settings are changed by making changes in the source code, especially by changing flags in `runHrbDecoder.cpp`. Typical combinations of the flags are:

Option	**Com. 1**	**Com. 2**	**Com. 3**
gErrorIgnoreLevel	kError	kWarning	kWarning
displayEbyE	kFALSE	kTRUE	kFALSE
displayOnlyGood	kFALSE	kFALSE	kFALSE
printRecoInfo	kFALSE	kTRUE	kTRUE

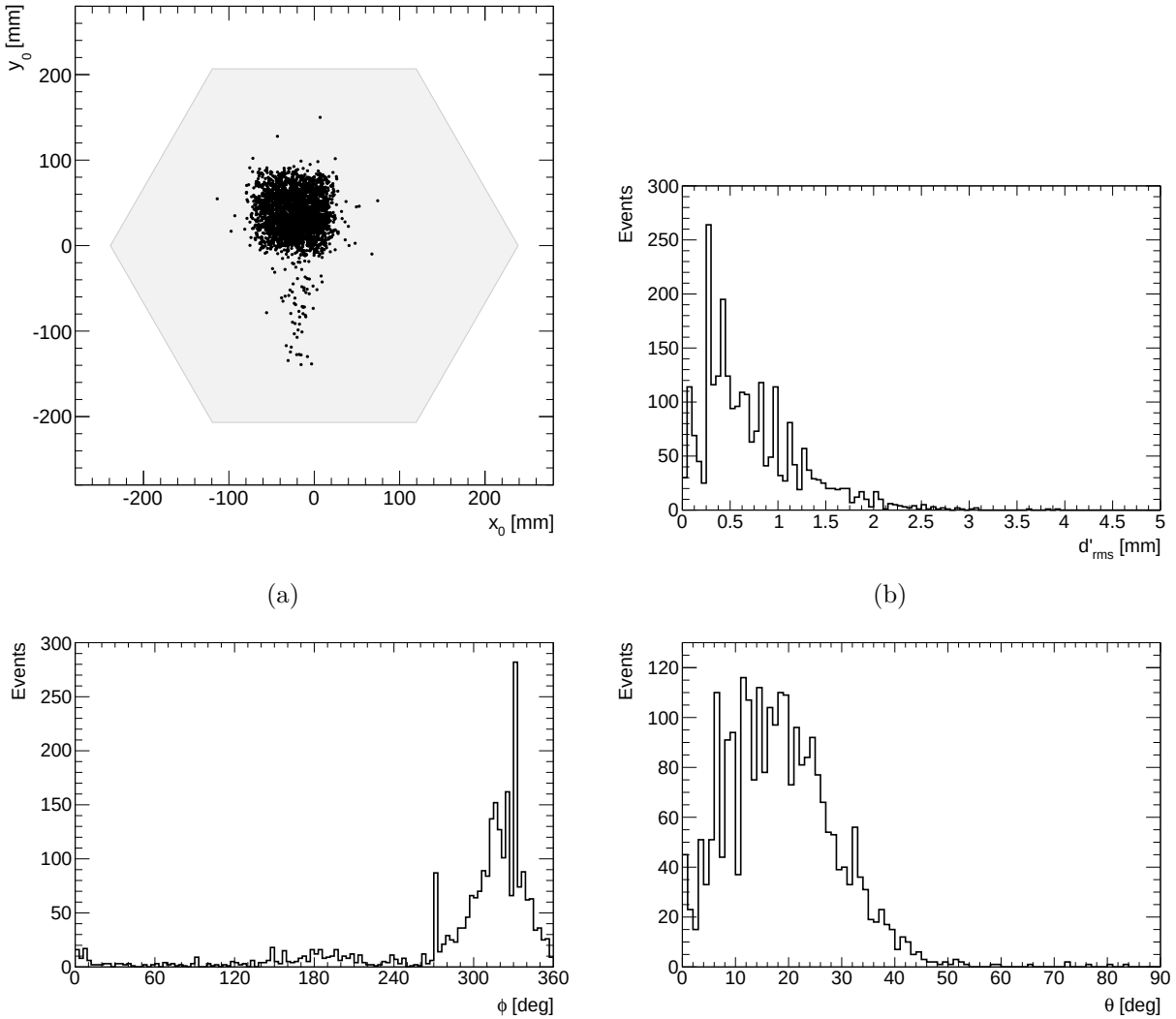
In the first case, the program execution will be the fastest and only minimal information will be printed: the number of entries in the sorting tree (i.e. number of one-plane events read from the data), the total number of events and the number of good events. The second case is used to display event by event along with the reconstruction information. It is advised to open the file `eventDisplay.pdf` in a PDF reader which refreshes the view, when the file is updated (e.g. Evince), run the HrbDecoder in a terminal window and just switch subsequent events by pressing Enter. The third flag combination prevents drawing event displays and prints all the reconstruction information without pausing the program execution. It is useful for writing a log file by redirecting the standard output.

8. Results

The tested dataset contained 19551 one-plane events, which were combined into 3258 full events after sorting. The reconstruction resulted in 2570 good events. Log information and event displays of the rejected events show, that most of them contain multiple hit clusters arising clearly from two particles crossing the detector. Some events were rejected due to having many hits spread around a big area or due to having too little hits. Both situations may be related to noise effects (the former being a real particle plus noise and the latter being just sheer noise). 198 events (6% of all events) were rejected due to impossible $\Delta x \Delta y 1 \Delta y 2$ combination, which could not be corrected by treating one of Δi as zero. A reason for this behaviour has not yet been identified.

Figure 5 shows the collective results of event reconstruction in the dataset. The location of trigger scintillators can be clearly seen in Figure 5a along with the location of light guide, which can also sometimes trigger an event. The d'_{rms}distribution correctly peaks and low values and falls more or less exponentially, reaching negligible values around $d_{\mathrm{w}} = 2.5$ mm. The angular distributions of the tracks reflect the alignment of the two trigger scintillators. Despite relatively low statistics in this dataset, as for testing purposes, the results appear satisfactory. However, some strong and apparently statistics-independent non-linearities can be spotted in the obtained distributions (e.g. the peaks at $\phi = 270^\circ$ and $\phi = 330^\circ$). It has not yet been resolved whether they are related to the detector geometry or caused by features of the reconstruction algorithm.

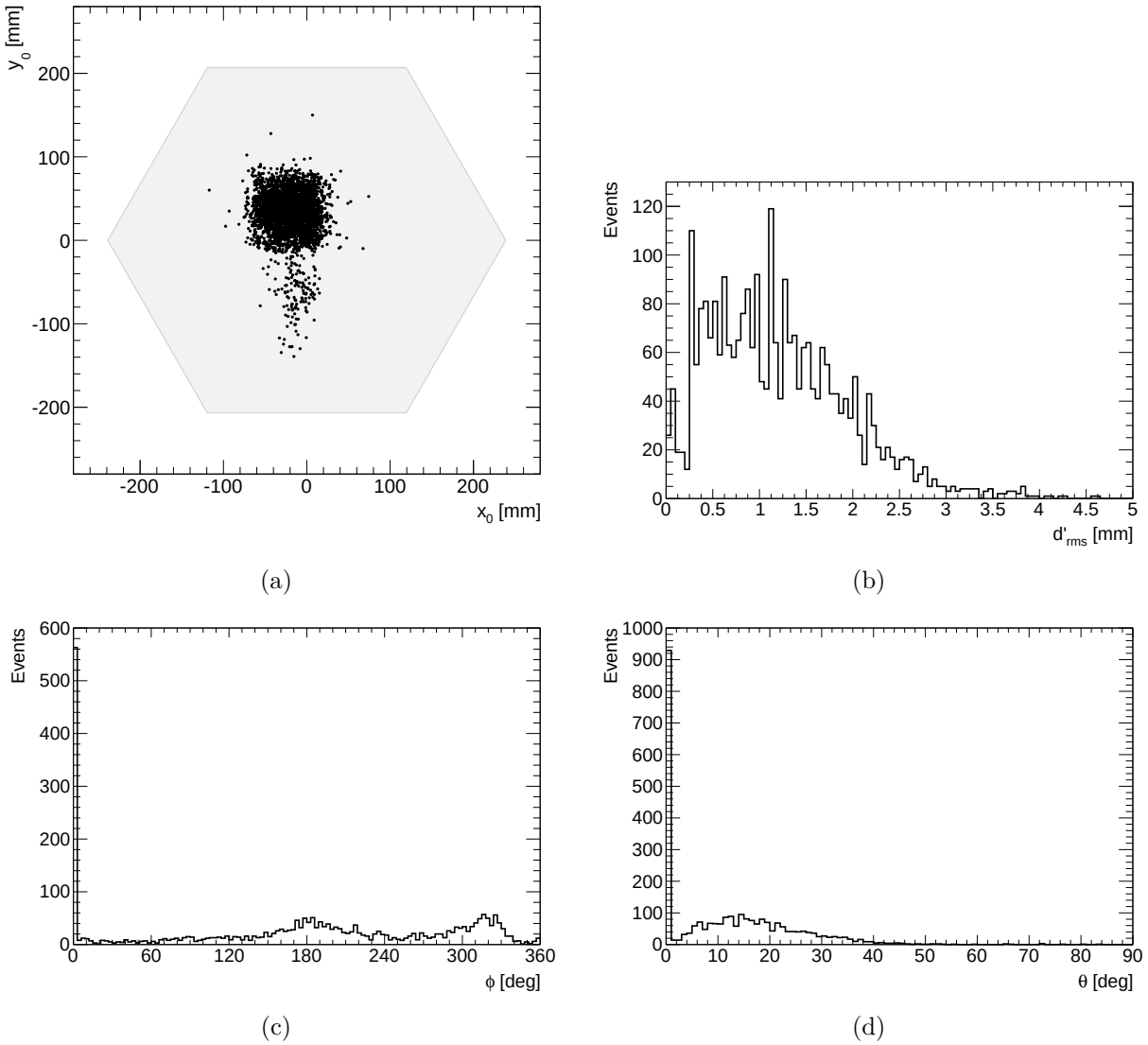
Although the ϕ-range constraining procedure may seem unjustified at the first glance and it may be accused of generating the non-linearities in the angular distributions, it is indeed crucial for a successful track reconstruction. This is clearly proven in Figure 6, which presents the same distributions as Figure 5, but obtained with $\phi \in [0;\ 2\pi]$ range used in the minimiser for all events. The vast majority of events were reconstructed with $\theta = 0$ and/or $\phi = 0$, which is far from true values, and leads the d'_{rms} distribution to smear and shift towards higher values.



(a) (b)

(c) (d)

Figure 5: (a) Location of the (x_0, y_0) point in all good events of the tested dataset. (b, c, d) Distributions of d'_{rms}, ϕ and θ in these events.



(a) (b)

(c) (d)

Figure 6: Analogous plots as presented in Figure 5, but after reconstruction without constraining the ϕ range.

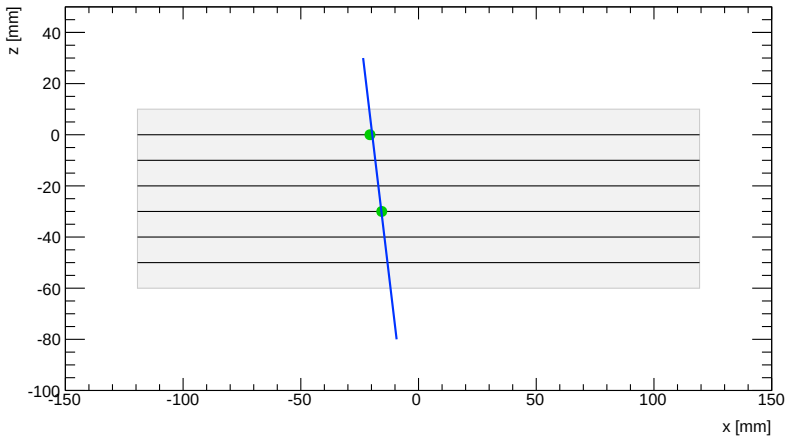
9. Example event displays

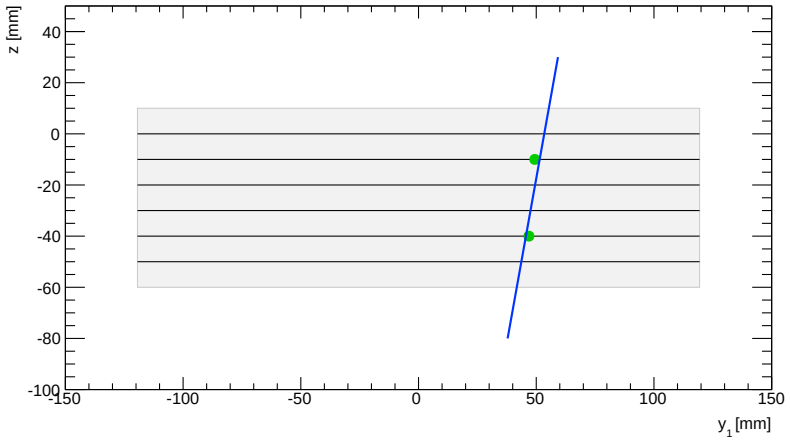
On the next few pages, displays of typical good and bad events are presented along with the information printed to the standard output at the reconstruction.

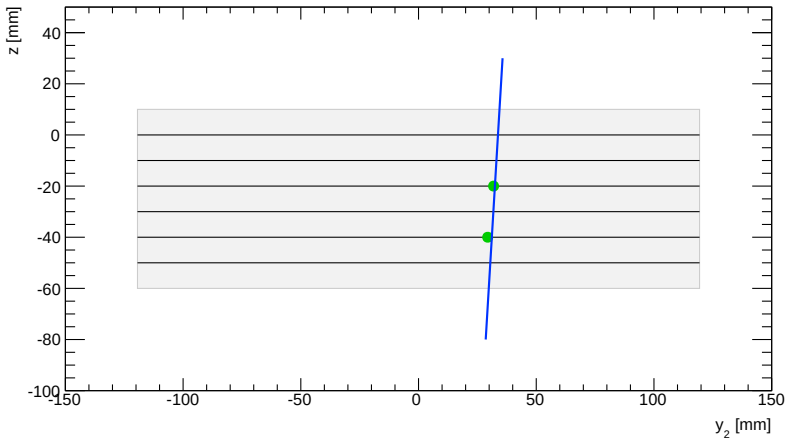
EVENT 21. A nearly-perfect event, with exactly one hit in each plane and a well-fitted track.

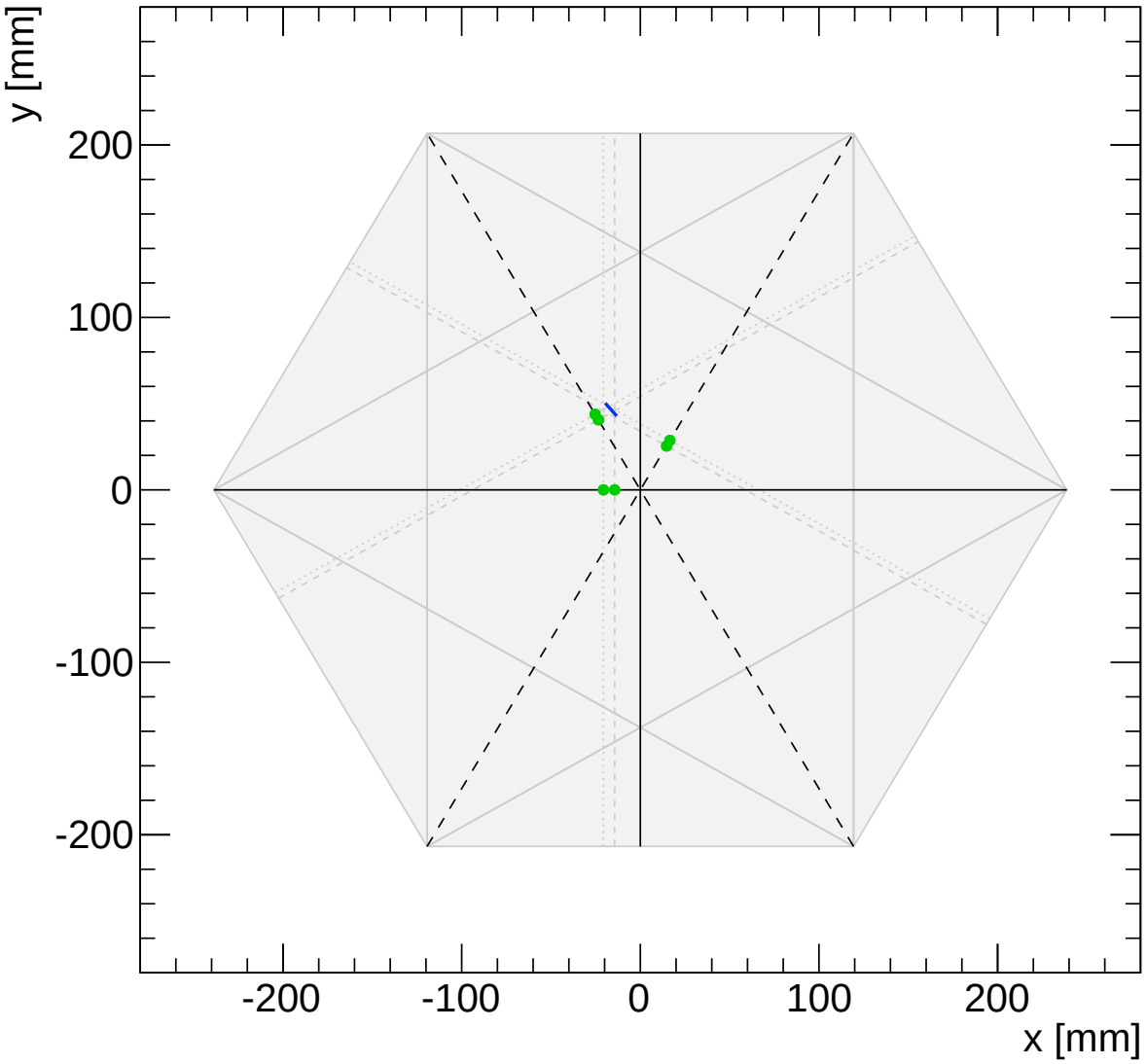
```
Info    : HrbEvent::HrbEvent() : In Event 21 : Plane 0 has 1 good hit
Info    : HrbEvent::HrbEvent() : In Event 21 : Plane 1 has 1 good hit
Info    : HrbEvent::HrbEvent() : In Event 21 : Plane 2 has 1 good hit
Info    : HrbEvent::HrbEvent() : In Event 21 : Plane 3 has 1 good hit
Info    : HrbEvent::HrbEvent() : In Event 21 : Plane 4 has 1 good hit
Info    : HrbEvent::HrbEvent() : In Event 21 : Plane 5 has 1 good hit
```

```
HIT LIST:
[GOOD]
plane 0 hit channels:  55
plane 0 hit samples:   20

[GOOD]
plane 1 hit channels:  67
plane 1 hit samples:   21

[GOOD]
plane 2 hit channels:  60
plane 2 hit samples:   25

[GOOD]
plane 3 hit channels:  41
plane 3 hit samples:   26

[GOOD]
plane 4 hit channels:  28
plane 4 hit samples:   27

[GOOD]
plane 5 hit channels:  35
plane 5 hit samples:   26
```







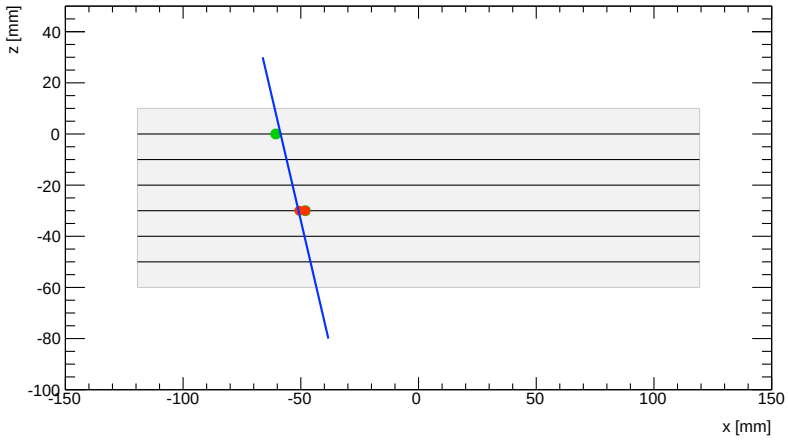


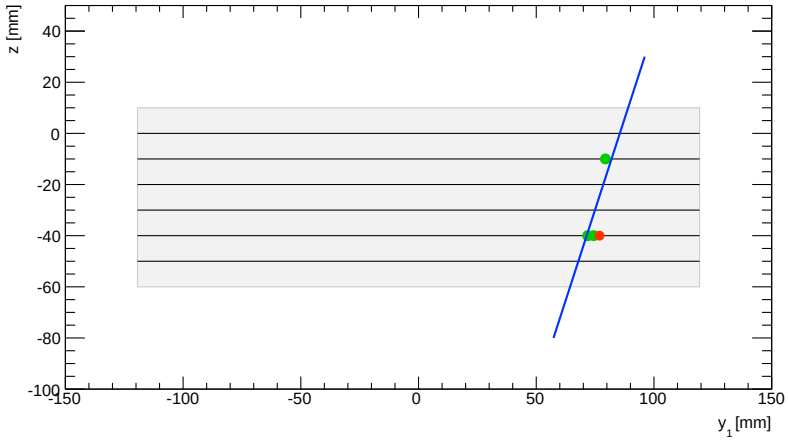
Example YX projection of a good event. Green dots represent averaged location of a good hit in each plane. The dotted grey lines represent averaged locations of good hit wires in the top three planes, whereas the dashed grey lines correspond to the three bottom planes. The fitted track is drawn in blue. The event 21, which is shown in this figure, has only one hit in each plane, thus the hit locations are exact.

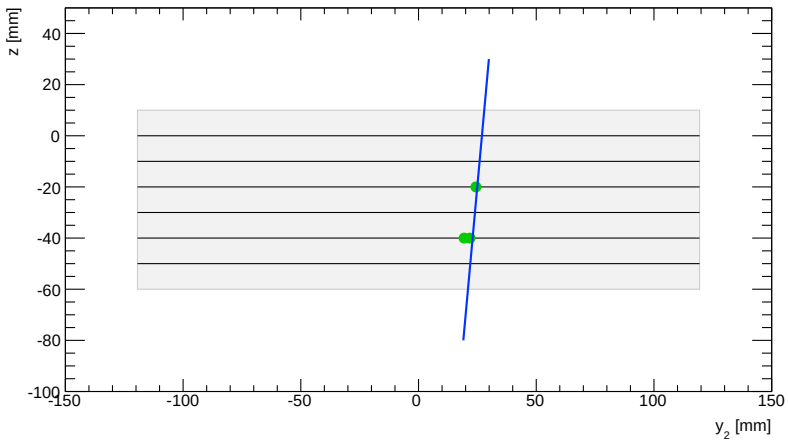
EVENT 54. A typical event with some good and some bad hits, six good planes and a well-fitted track. Bad hits are marked with `x` in the log and with red dots in the display.

```
Info    : HrbEvent::HrbEvent() : In Event 54 : Plane 0 has 1 good hit
Info    : HrbEvent::HrbEvent() : In Event 54 : Plane 1 has 1 good hit
Info    : HrbEvent::HrbEvent() : In Event 54 : Plane 2 has 1 good hit
Info    : HrbEvent::HrbEvent() : In Event 54 : Plane 3 has 1 good hit
Info    : HrbEvent::HrbEvent() : In Event 54 : Plane 4 has 2 good hits
Info    : HrbEvent::HrbEvent() : In Event 54 : Plane 5 has 2 good hits
```

```
HIT LIST:
[GOOD]
plane 0 hit channels:  71
plane 0 hit samples:    9

[GOOD]
plane 1 hit channels:  79
plane 1 hit samples:   10

[GOOD]
plane 2 hit channels:  57
plane 2 hit samples:   11

[GOOD]
plane 3 hit channels:  28   27x  28x
plane 3 hit samples:   16   19x  22x

[GOOD]
plane 4 hit channels:  17   18   16x
plane 4 hit samples:   15   17   21x

[GOOD]
plane 5 hit channels:  39   38
plane 5 hit samples:   19   21
```



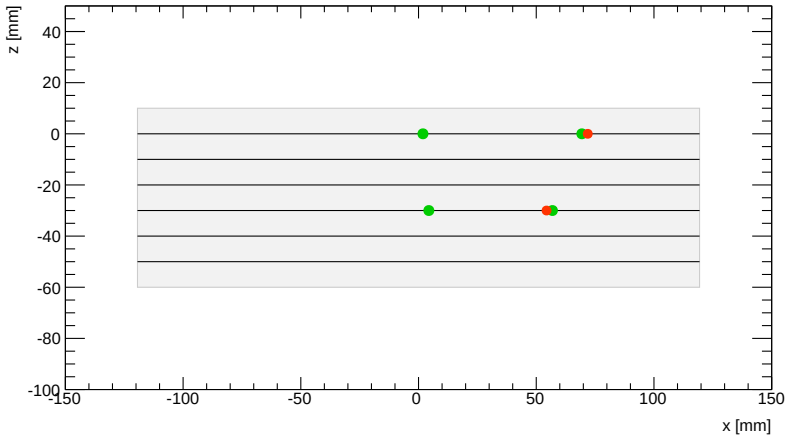


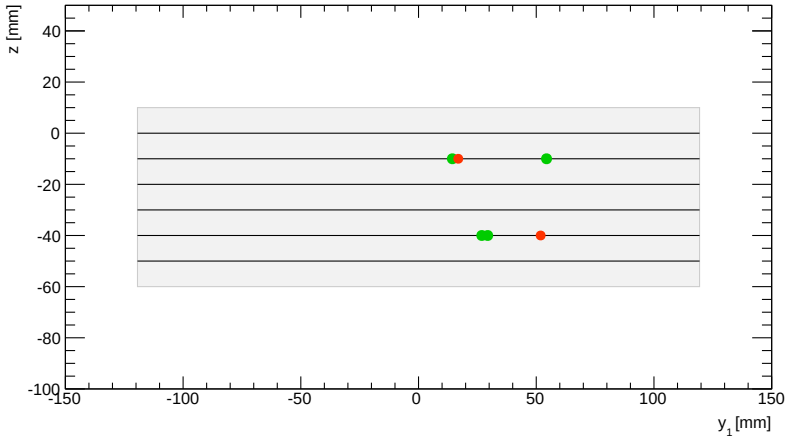


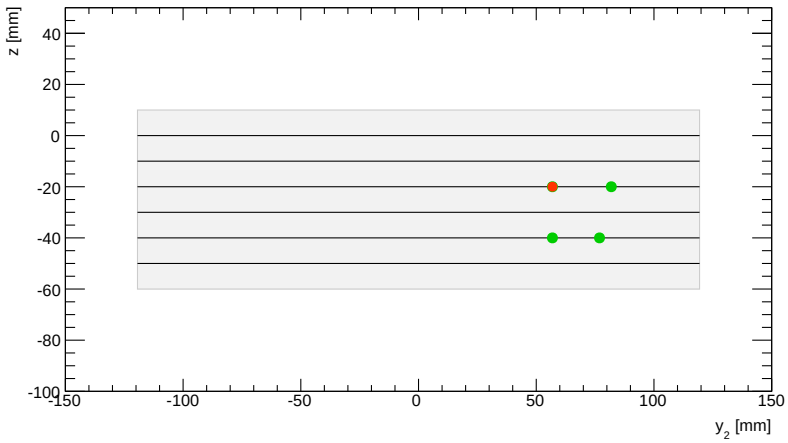
EVENT 156. An event with clear signatures of two particles crossing the detector at the same time.

```
Warning : HrbEvent::HrbEvent() : In Event 156 : Bad plane 0 - two hits too far from each other.
Warning : HrbEvent::HrbEvent() : In Event 156 : Bad plane 1 - two hits too far from each other.
Warning : HrbEvent::HrbEvent() : In Event 156 : Bad plane 2 - two hits too far from each other.
Warning : HrbEvent::HrbEvent() : In Event 156 : Bad plane 3 - two hits too far from each other.
Info    : HrbEvent::HrbEvent() : In Event 156 : Plane 4 has 2 good hits
Warning : HrbEvent::HrbEvent() : In Event 156 : Bad plane 5 - two hits too far from each other.
Warning : HrbEvent::HrbEvent() : Bad Event 156. Not enough good planes (1).
```

```
HIT LIST:
[BAD]
plane 0 hit channels:  46   19   18x
plane 0 hit samples:   15   17   21x

[BAD]
plane 1 hit channels:  53   69   54x
plane 1 hit samples:   16   16   19x

[BAD]
plane 2 hit channels:  80   70   70x
plane 2 hit samples:   16   18   22x

[BAD]
plane 3 hit channels:  70   49   69x
plane 3 hit samples:   22   24   29x

[GOOD]
plane 4 hit channels:  35   36   26x
plane 4 hit samples:   22   24   26x

[BAD]
plane 5 hit channels:  16   24
plane 5 hit samples:   22   23
```



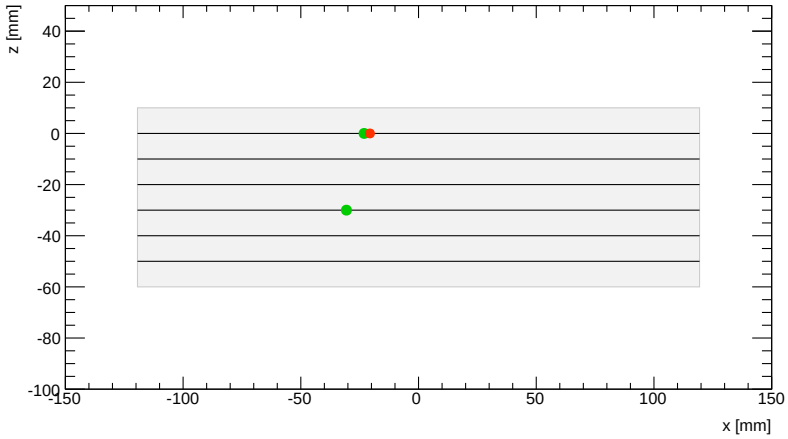


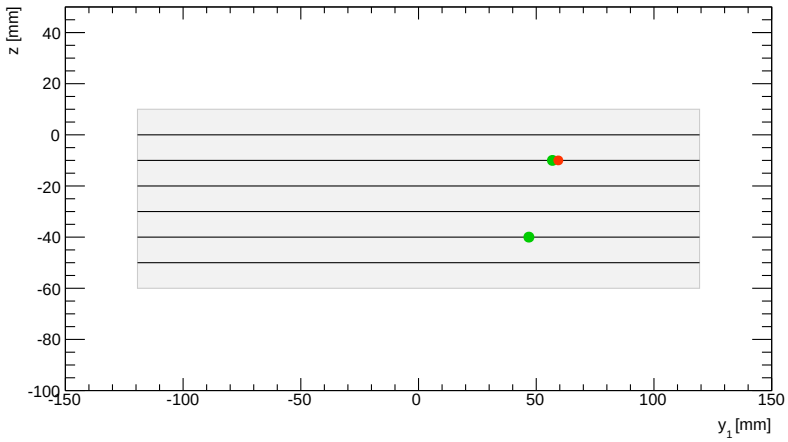


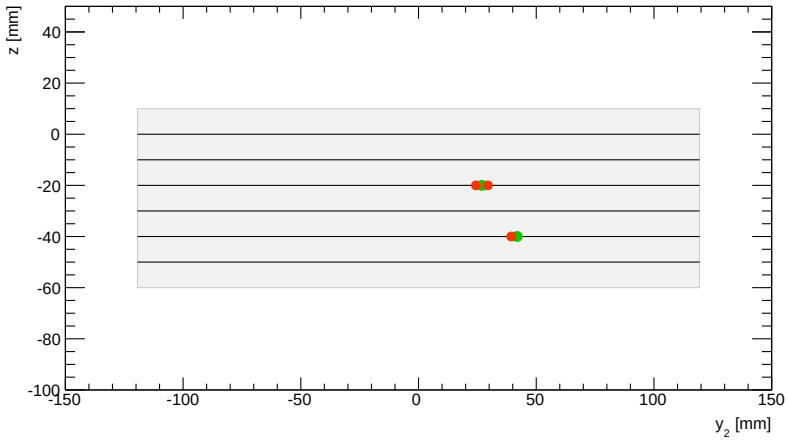
EVENT 195. An event with an impossible combination of Δi values. A three-dimensional straight line cannot be fitted well to these hits.

```
Info    : HrbEvent::HrbEvent() : In Event 195 : Plane 0 has 1 good hit
Info    : HrbEvent::HrbEvent() : In Event 195 : Plane 1 has 1 good hit
Info    : HrbEvent::HrbEvent() : In Event 195 : Plane 2 has 1 good hit
Info    : HrbEvent::HrbEvent() : In Event 195 : Plane 3 has 1 good hit
Info    : HrbEvent::HrbEvent() : In Event 195 : Plane 4 has 1 good hit
Info    : HrbEvent::HrbEvent() : In Event 195 : Plane 5 has 1 good hit
Warning : HrbEvent::PhiRange() : Impossible combination --+
Warning : HrbEvent::HrbEvent() : Bad Event 195. Wrong result of PhiRange().
```

```
HIT LIST:
[GOOD]
plane 0 hit channels:  56   55x
plane 0 hit samples:   10   17x

[GOOD]
plane 1 hit channels:  70   71x
plane 1 hit samples:   11   17x

[GOOD]
plane 2 hit channels:  58   59x  57x
plane 2 hit samples:   12   15x  17x

[GOOD]
plane 3 hit channels:  35
plane 3 hit samples:   18

[GOOD]
plane 4 hit channels:  28
plane 4 hit samples:   17

[GOOD]
plane 5 hit channels:  30   31x
plane 5 hit samples:   18   24x
```







Appendix A. HrbDecoder class reference

Public methods

`HrbDecoder ()`

Empty constructor (needed for ROOT ClassDef compatibility)

`HrbDecoder (Int_t nfiles, char** filenames)`

Standard constructor
`nfiles` - number of input files
`filenames` - array of file names, having size of `nfiles`

`~HrbDecoder ()`

Default destructor.

`Bool_t FilesGood ()`

Returns true if all input file streams are good for readout (no eofbit, failbit or badbit)

`UInt_t ReadWord32 (UInt_t ifile)`

Reads a 32-bit word from a file with index `ifile`

`ULong64_t ReadWord64 (UInt_t ifile)`

Reads a 64-bit word from a file with index `ifile`

`bitset<128> ReadWord128 (UInt_t ifile)`

Reads a 128-bit word from a file with index `ifile`

`void CheckSyncType (UInt_t data)`

Checks and prints the Sync word and the frame type. If bad Sync word encountered, it sets `m_badsync = kTRUE`. The argument `data` is a 32-bit word read from a file.

`void CheckLengthSeq (UInt_t data)`

Prints the frame length and sequence number (counter). The argument `data` is a 32-bit word read from a file.

`void CheckDstSrc (UInt_t data)`

Prints the destination and source address. The argument `data` is a 32-bit word read from a file.

`void CheckDataType (UInt_t data)`

Prints the data type, flags and fragment lenght. The argument `data` is a 32-bit word read from a file.

`void CheckFragmentID (UInt_t data)`

Prints the fragment ID and offset. The argument `data` is a 32-bit word read from a file.

`void CheckCRC (UInt_t data)`

Prints the CRC (cyclic redundancy check). The argument `data` is a 32-bit word read from a file.

`Int_t GetPlaneID (UInt_t data)`

Checks the device serial number and returns the corresponding plane number. The argument `data` is a 32-bit word read from a file.

`Int_t GetEventNumber (UInt_t data)`

Prints the channel number (trigger position) and the event number, and returns the latter. The argument `data` is a 32-bit word read from a file.

`void CheckTimestamp (ULong64_t data)`

Prints the timestamp. The argument `data` is a 64-bit word read from a file.

`void SetIDmap (vector<UInt_t> devID, vector<TString> planeName)`

Sets the map of plane numbers vs device IDs, and the plane names array

`list<vector<Int_t> > ReadHits (UInt_t ifile, Int_t iplane)`

Reads nsample of 128-bit words containg hit information and creates a list of (plane, sample, channel) hit vectors
`ifile` - the index of file to be read
`iplane` - the corresponding plane index

`Int_t GetNfiles ()`

Returns the number of data files

`TString GetFilename (UInt_t ifile)`

Returns the name of i-th file

`Bool_t BadSync ()`

Returns true if bad Sync word was encountered

Private members

Int_t	m_nfiles	Number of input files
vector<TString>	m_filename	Input file names array
vector<ifstream*>	m_filestr	Input file streams array
Bool_t	m_badsync	True if bad sync word encountered
map<UInt_t,UInt_t>	m_devIDmap	Map of the device IDs and the corresponding plane numbers
vector<TString>	m_planeName	Array of plane names. The name index corresponds to the plane number
Int_t	m_nsamples	Number of samples in a single frame
Int_t	m_vb	Verbosity level, as in TError.h: kInfo, kWarning, kError, etc.

Appendix B. HrbEvent class reference

Public methods

`HrbEvent ()`

Empty constructor

`HrbEvent (Long64_t timestamp, UInt_t evNumber, list<vector<Int_t> > rawHitsArray, Int_t nplanes)`

Standard constructor
`timestamp` - event timestamp (number of miliseconds since Unix 'epoch')
`evNumber` - the event number
`rawHitsArray` - a list of hit vectors (plane, sample, channel) - output from `HrbDecoder::ReadHits()`
`nplanes` - number of planes of the detector

`~HrbEvent ()`

Default destructor

`void Print ()`

Prints the event information (track and fit parameters)

`TString GetTimeString ()`

Returns a string representing time of the event in the format yyyy-mm-dd HH:MM:SS

`Bool_t IsGoodEvent ()`

Returns true if the event has exactly one hit in each plane

`Double_t GetX (Double_t z)`

Returns a value of the x-coordinate of a point on the track, at a given value of the z-coordinate

`Double_t GetY (Double_t z)`

Returns a value of the y-coordinate of a point on the track, at a given value of the z-coordinate

`Double_t GetY1 (Double_t z)`

Returns a value of the y1-coordinate of a point on the track, at a given value of the z-coordinate

`Double_t GetY2 (Double_t z)`

Returns a value of the y2-coordinate of a point on the track, at a given value of the z-coordinate

`Double_t GetUncX (Double_t z)`

Returns the uncertainty of x(z)

`Double_t GetUncY (Double_t z)`

Returns the uncertainty of y(z)

`Double_t GetUncY1 (Double_t z)`

Returns the uncertainty of y1(z)

`Double_t GetUncY2 (Double_t z)`

Returns the uncertainty of y2(z)

`Double_t GetPx ()`

Returns the x-coordinate of the track momentum (px)

`Double_t GetPy ()`

Returns the y-coordinate of the track momentum (py)

`Double_t GetPz ()`

Returns the z-coordinate of the track momentum (pz)

`Double_t GetPhi ()`

Returns the azimuthal angle of the track (phi)

`Double_t GetTheta ()`

Returns the polar angle of the track (theta)

`Double_t GetUncPx ()`

Returns the uncertainty of the x-coordinate of the track momentum (px)

`Double_t GetUncPy ()`

Returns the uncertainty of the y-coordinate of the track momentum (py)

`Double_t GetUncPz ()`

Returns the uncertainty of the z-coordinate of the track momentum (pz)

`Double_t GetUncPh ()`

Returns the uncertainty of the azimuthal angle of the track (phi)

`Double_t GetUncTheta ()`

Returns the uncertainty of the polar angle of the track (theta)

`Double_t GetAvDhit (Int_t iplane)`

Returns the hit distance in a given plane (averaged number of channel * distance between wires + offset)

`Double_t GetSumOfSquares ()`

Returns the sum of squared track-hit distances

`UInt_t GetNhits ()`

Returns the total number of hits

`UInt_t GetNhitsGood ()`

Returns the number of good hits in good planes (all which are used in track fitting)

`Double_t GetHitPlanes ()`

Returns the number of planes in which hits were recorded

`Double_t GetHitPlanesGood ()`

Returns the number of planes used in track fitting

`void SetPlaneNames (vector<TString> planeName)`

Sets the vector of plane names

`Long64_t GetTimestamp ()`

Returns the event timestamp

`UInt_t GetEventNumber ()`

Returns the event number

`list<vector<Int_t> > GetRawHits ()`

Returns the raw hit list - a list of 3-element vectors (plane, sample, channel)

`vector<vector<Bool_t> > GetRawGoodHitFlag ()`

Returns the nplanes-element vector of hit vectors containing good hit flags

`vector<vector<Int_t> > GetRawChannels ()`

Returns the nplanes-element vector of hit vectors containing channel numbers

Private methods

`void SortHits ()`

Sorts the hits by plane number, sample number, channel number. The implementation of this function uses the function `HitsComparison(vector<Int_t>,vector<Int_t>)` defined locally in the file `HrbEvent.cpp`.

`vector<Double_t> PhiRange (Double_t dx, Double_t dy1, Double_t dy2)`

Returns a two-element vector with values defining the range of possible phi angle values with given deltaX, deltaY1, deltaY2

`vector<Double_t> PhiRange5 (Double_t dx, Double_t dy1, Double_t dy2)`

Same as `PhiRange()` but for the case of 5 planes with recorded hits

`double SumOfSquares (const double* vv)`

The sum of squared distances function, for the least squares method

Private members

Long64_t	m_timestamp	Event timestamp
UInt_t	m_evNumber	Event number
list<vector<Int_t> >	m_hitsArrayRaw	List of 3-element vectors (plane,sample,channel)
vector<vector<Int_t> >	m_ichanRaw	nplanes-element vector of hit vectors (channel numbers)
vector<vector<Int_t> >	m_sampleRaw	nplanes-element vector of hit vectors (sample numbers)
vector<vector<Bool_t> >	m_goodHit	nplanes-element vector of hit vectors (good hit flag)
vector<vector<Double_t> >	m_dhit	nplanes-element vector of hit vectors (hit in-plane coordinates)
Int_t	m_vb	Verbosity level, as in TError.h: kInfo, kWarning, kError, etc.
Bool_t	m_sorted	True if raw hit list, `m_hitsArrayRaw`, is sorted
Int_t	m_nplanes	Total number of planes
vector<TString>	m_planeName	Vector of plane names
Double_t*	m_avDhit	Vector of averaged hit distances x1, yl1, yl2, x2, yr1, yr2
Double_t	m_dw	The distance between wires (horizontal)
Double_t	m_hw	Width of one plane ($95.5 \cdot d_w$)
Double_t	m_dz	The distance between planes (vertical)
Double_t	m_px	x-coordinate of the track momentum vector (p_x)
Double_t	m_py	y-coordinate of the track momentum vector (p_y)
Double_t	m_pz	z-coordinate of the track momentum vector (p_z)
Double_t	m_x0	x-coordinate of track at z=0 (x_0)
Double_t	m_y0	y-coordinate of track at z=0 (y_0)
Double_t	m_phi	Azimuthal angle ϕ (in the xy-plane)
Double_t	m_theta	Polar angle θ (in the rz-plane)
Double_t	m_uncPx	Uncertainty of p_x
Double_t	m_uncPy	Uncertainty of p_y
Double_t	m_uncPz	Uncertainty of p_z
Double_t	m_uncX0	Uncertainty of x_0
Double_t	m_uncY0	Uncertainty of y_0
Double_t	m_uncPhi	Uncertainty of ϕ
Double_t	m_uncTheta	Uncertainty of θ
Bool_t	m_isGoodEvent	True if requirements of a good event are met
Double_t	m_sumOfSquares	Sum of squared track-hit distances
Int_t	m_nhits	Number of hits in the event
Int_t	m_nhitsGood	Number of hits in good planes
Int_t	m_hitPlanes	Number of planes in which hits were recorded
Int_t	m_hitPlanesGood	Number of planes with good hits
Bool_t*	m_goodPlane	Vector of boolean values indicating good/bad planes
Double_t	m_bigDist	Distance much larger than detector dimensions – this number is used in case of no-hit planes