



AGH

AGH University of Science and Technology

FACULTY OF COMPUTER SCIENCE, ELECTRONICS AND TELECOMMUNICATIONS

DEPARTMENT OF COMPUTER SCIENCE

Engineering thesis

*Proton reconstruction algorithm for track pattern recognition and fitting in
the TOTEM experiment at CERN*

Authors:

Jakub Sawicki

Jan Wójcik

Degree programme:

Computer Science

Supervisor:

Leszek Grzanka, PhD

CERN Supervisor:

Jan Kašpar

Kraków, January 2015



We hereby declare that we have written this thesis completely by ourselves, that we have used no other sources or resources than the ones mentioned, and that we are aware of the rules of the Penal Code referring to criminal liability for providing false evidence.

We would like to appreciate all the support received from the people working for the TOTEM experiment at CERN. Especially to Jan Kašpar, who has provided indispensable guidance and support. But also to Valentina Avati, who wouldn't hesitate to extend a helping hand.

Special thanks should also go to Leszek Grzanka for helping out to organise the internship and supplying us with comments on the thesis.

I would also like to include thanks to all the summer interns who made the time spent at the Swiss-French border an unforgettable experience.

Contents

1. Problem overview	7
1.1. CERN	7
1.2. The TOTEM experiment	7
1.3. The Roman Pots	8
1.4. Track reconstruction in RPs	10
2. The detectors upgrade	11
3. Software upgrade.....	13
3.1. Functional requirements.....	13
3.2. Non-functional requirements	13
4. Algorithms	14
4.1. Terminology	14
4.2. Algorithm outline.....	15
4.2.1. Cluster building	15
4.2.2. Selection and filtering	16
4.2.3. Overlaps removal	16
4.2.4. Fitting	17
5. Testing and performance	18
5.1. Track distributions	18
5.1.1. Uniform distribution	18
5.1.2. 4 TeV, $\beta^* = 90$ m.....	19
5.1.3. 4 TeV, $\beta^* = 0.6$ m.....	19
5.2. Debugging and verbose output analysis.....	20
5.3. Final performance results.....	20
5.3.1. Comparison with the previous module	20
5.3.2. Multi-track performance	24
6. Package architecture	26
6.1. CMSSW - Event Data Model	26
6.2. Modules.....	27
6.2.1. RPStationMultiTrackFinderFitter module	27
6.2.2. Helper modules	27
6.3. CMSSW configurations	28

6.4. Visualisation	28
6.5. Code management.....	29
7. Project environment and usage	30
7.1. Building SCRAM project.....	30
7.1.1. Patching the code (dirty hack)	31
7.2. Simulations	31
7.2.1. Components of a simulation	31
7.2.2. Adjustable reconstruction parameters	32
7.3. Modules for additional tasks	35
7.3.1. Performance testing.....	35
7.3.2. Analysis of a single event.....	36
7.4. Generate track distribution.....	36
7.5. Documentation.....	37
8. Modules' inputs and outputs.....	38
8.1. Input, output and Event Setup.....	38
8.1.1. Input events.....	38
8.1.2. Output events.....	39
8.1.3. Event Setups.....	39
8.2. Cooperating packages	39
9. Source code	41
9.1. Non-module classes	41
9.2. Structure of module classes.....	45
9.3. Framework classes.....	48
9.3.1. Roman Pot data formats	48
9.3.2. CMSSW	49
9.3.3. ROOT.....	49

1. Problem overview

1.1. CERN

The European Centre for Particle Physics (CERN) is an international organization focused on providing means for scientists to research the particle physics. It is located on the Swiss-French border on the outskirts of Geneva.

The well established way to study the behaviour of particles are particle accelerators combined with colliders. Usually, particles are accelerated to given energy and smashed against each other or against a target. In the result of such event new particles can be created, due to the mass-energy equivalence, and observed in detectors placed near the collision.

In contemporary high energy physics, in order to achieve as high energy at interaction point as possible, two beams are used. In this way, all the energy accumulated in particles coming from both beams can be used to create new particles. Fixed target with single beam hitting it introduces a loss of energy available for particle creation.

The Large Hadron Collider (LHC) is a synchrotron, what means that it is keeping the particles on an approximately circular path bending their trajectory with magnets. Two opposing beams are accelerated and kept in the LHC and cross with each other at the interaction points (IP) to induce collisions which can be observed by detectors placed around.

At CERN, the LHC has recently been built to achieve collision energies of up to 14 TeV. Up to date, only collisions with energies of up to 8 TeV has occurred due to the Incident in 2008 [4]. After the failure it has been decided that in the first run only lower energies will be used and to reach the designed energies and higher luminosities during the following runs. Currently (beginning of 2015), the Long Shutdown 1 is almost over and the machine should resume operations in the spring 2015 reaching its design energy.

The Long Shutdown 1 is an operation which is aimed at maintenance and upgrade of the LHC and the experiments. [2] Since its beginning in February 2013 planned upgrades were performed. Currently, the LHC sections has already been cooled down and powering tests are underway (or has been performed).

1.2. The TOTEM experiment

TOTEM is an experiment based at CERN. Its main aim is to measure the total proton-proton cross-section, elastic scattering and diffractive processes at the LHC. Its goal is complementary to most of other experiments at CERN. While they are focused on what happens if two beams collide with each other head-on (with particles produced to large angles), TOTEM is more interested glancing collisions yielding particles in forward direction, i.e. at very small scattering

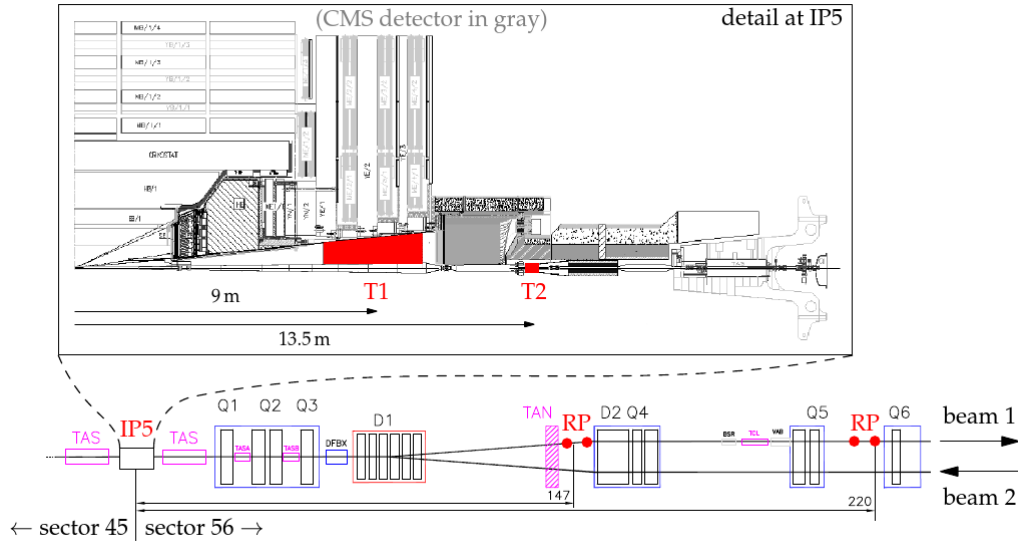


Figure 1.1: Overview of the TOTEM detectors. They are installed symmetrically around the interaction point. Trackers T1 and T2 are visible at the enlarged part of the diagram. On the diagram, old positions of the RPs are shown, currently station at 147 m is moved further away from IP to 200 m. [3, p. 33]

angles. Because of that its detectors are placed not around the IP (as most of other experiments do) but quite far from the IP and close to the beam itself.

TOTEM is placed at the IP5 sharing space with CMS (another experiment). TOTEM's machinery comprises of three groups of detectors: T1, T2 and Roman Pots (RPs). T1 and T2 are trackers which are placed close to the end caps of the CMS detector and measure particles escaping the IP at medium angles. They are centered at 9 m and 13.5 m from the IP accordingly, see Figure 1.1. RPs (Roman Pots) are in turn placed in a distance of about 200 m from IP. They were designed with the requirements of the TOTEM experiment in mind. They host silicon devices with the specific objective of reducing the insensitive region at the edge facing the beam. [5]

1.3. The Roman Pots

The Roman Pots are placed symmetrically around the IP. In total, there are 4 stations. Each station consists of two units, see Figure 1.2. Each unit includes horizontal, top and bottom RPs. To sum up, there are 8 RP units in total with 24 individual RPs.

Each RP has 10 silicon microstrip detectors which were developed specially for the experiment with active volume already at around $50 \mu\text{m}$ from the edge. 5 sensors have the strips oriented at an angle of $+45^\circ$ with respect to the edge and the other 5 at the angle of -45° . To get a better understanding of the design see Figure 1.3. Each microstrip detector consists of 512 strips with a pitch, i.e. the distance between the centres of two adjacent strips, of $66 \mu\text{m}$.

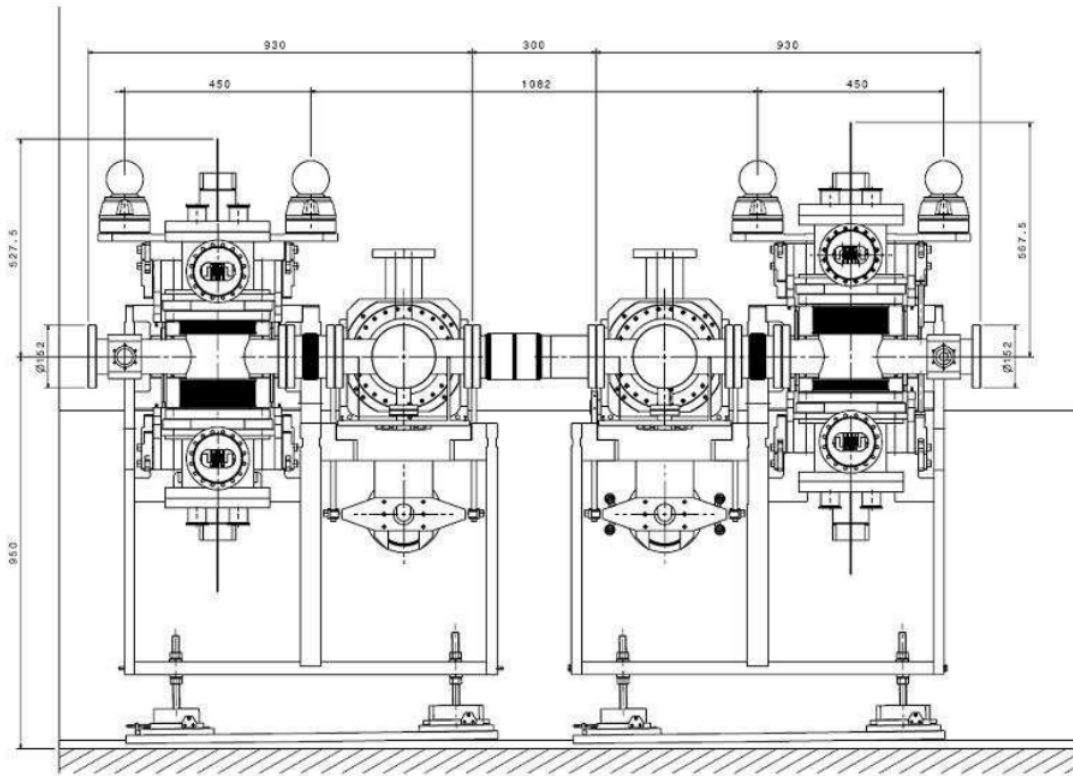


Figure 1.2: Design drawing of a RP station, i.e. an assembly of two RP units. [5, p. 51]

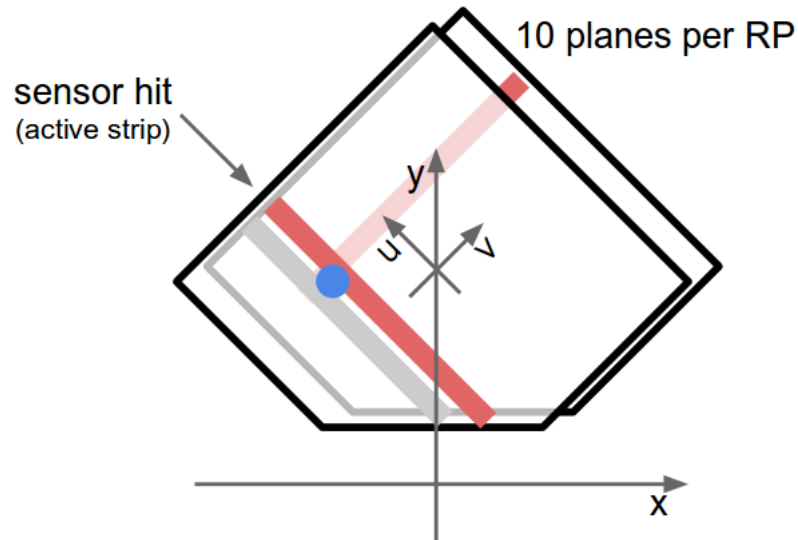
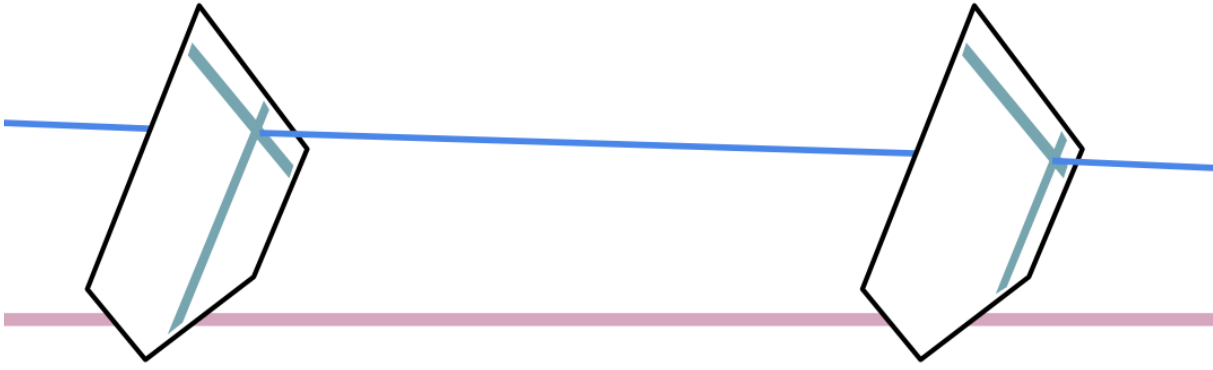
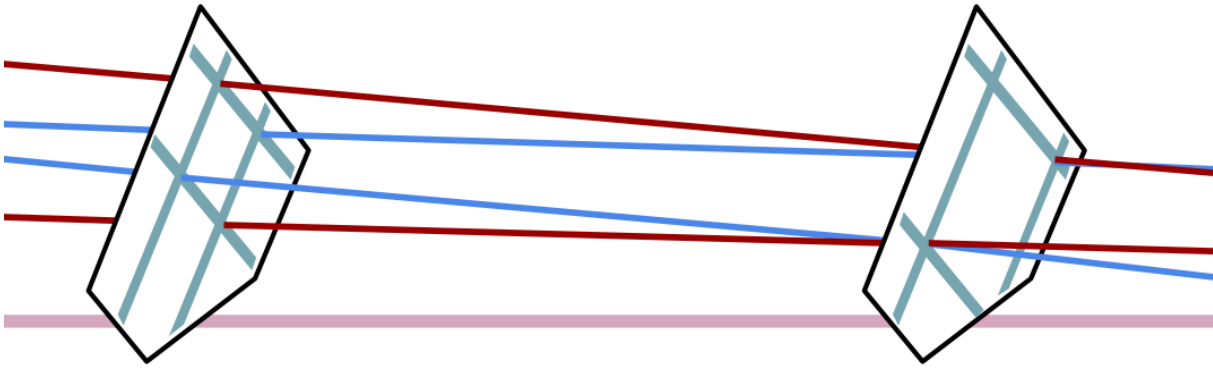


Figure 1.3: Drawing of the two silicon detectors with direction of their strips shown. U and V directions follow the readout directions of the perpendicular strips. They are shown in respect to the global XY frame. The view is oriented along the beam. A particle hit is shown as a blue dot and an activated strip is presented as a red line.



(a) No problem arises if only single track is to be reconstructed.



(b) If multiple particles has been detected then the basic reconstruction fails to unambiguously determine the pair of correct tracks.

Figure 1.4: Track reconstruction based on U/V patterns. Blue lines represent the actual particles passing through the detectors. Red lines represent fake tracks that are equally possible as the true tracks. U/V patterns are indicated in light green. Beam has light pink colour.

1.4. Track reconstruction in RPs

During the reconstruction process, the sensor hits from each RP are processed so as to reduce the volume of data (instead of 5 in each U/V direction, only single *U/V pattern*) and suppress noise (if hits don't form a line). The reconstruction module is looking for line patterns formed by sensor hits, hence the name *U/V patterns*. These patterns can be seen at Figure 1.4.

Using these data if only a single particle has been observed is straightforward. With data from only two RPs it is possible to reconstruct the particle track correctly as can be seen at Figure 1.4a.

If more than a single particle has passed through the detectors than it is impossible to make out the correct track combination, see Figure 1.4b. There are two patterns in each (U and V) direction forming 4 possible places through which the particles might have passed. Combining it with the data from the other RP there are at least 4 possible tracks (if one assumes only two particles were present). It is possible to rule out some combinations, if one U/V pattern intersection lies outside the cut edge of the detector, but such cases are rare.

2. The detectors upgrade

Learning from the experiences of the run 1 of LHC which provided energies of up to 4 TeV per beam, it has been proposed to introduce a hardware modification which would make possible to reconstruct events with multiple particles correctly. The modification has been described in the Upgrade Proposal [1]. The station RP220 (with units at 215 m and 220 m from IP5) has remained unchanged. Yet the former station RP147 has been removed from its position (around 147 m from IP5) and placed at 202 m (210-Near) and 213 m (210-Far). Unit 210-F which is moved from 147 m has been reinstalled with a tilt of 8° around the beam axis, see Figure 2.1. This particular change would enable reconstruction of events with multiple particles present. The rotated RP will provide an additional frame of reference which will be used to distinguish between correct and false track candidates as can be seen on Figure 2.2.

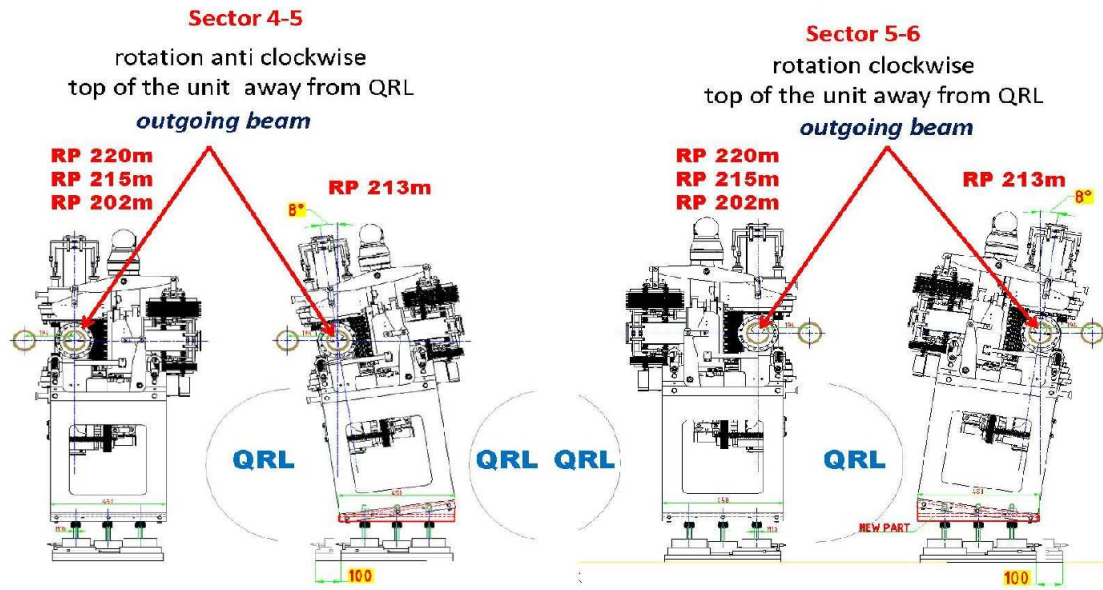


Figure 2.1: Drawings of the rotated RP units compared to non-rotated ones. [1, p. 12]

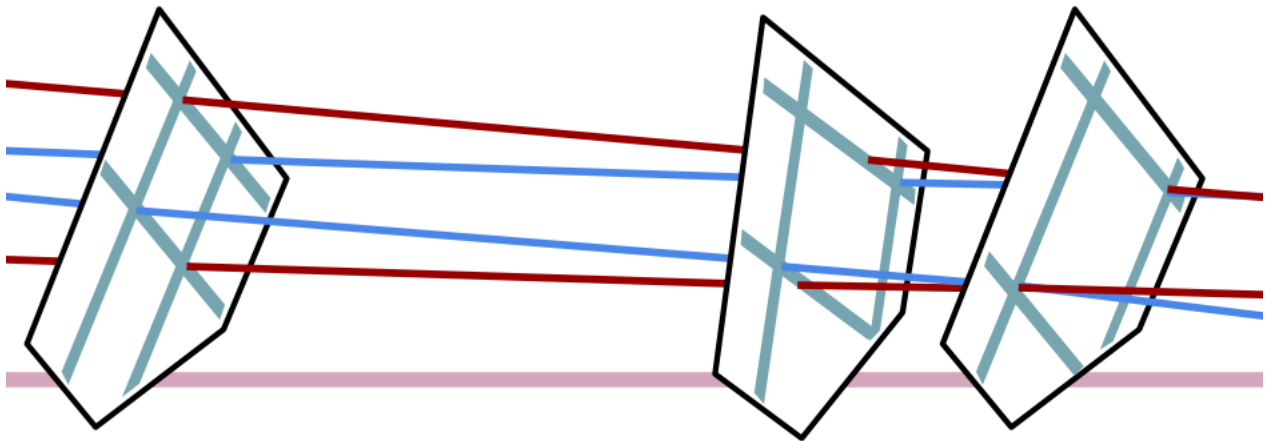


Figure 2.2: Reconstruction of an event shown on 1.4b with additional rotated RP present. It is possible to tell the false track candidates (shown in red) from the correct ones (shown in blue).

3. Software upgrade

For the measurements and analysis of the events collected during the previous run of the LHC modules were developed, which handled the reconstruction process using the information from two RPs only. In order to be able to make use of the hardware modifications the reconstruction software needs to be upgraded as well. The module which has been developed for the thesis handles this transition.

3.1. Functional requirements

The goal of the thesis was to create a module as follows:

- Is a CMSSW platform module.
- Is capable of reconstructing events with one as well as with a few tracks.
- Is configurable from files written in the Python language.
- Its efficiency is well determined by simulations.

3.2. Non-functional requirements

- Computational complexity should be acceptable for the problem domain.
- Code should be documented.
- Code should operate on data types from within the experiment.
- The project should use the SVN version control system used at CERN.

4. Algorithms

In order to describe the implemented algorithm in a clear and understandable way this chapter is divided into parts.

Section 4.1 defines the terminology. A more precise description of the algorithm follows in Section 4.2.

4.1. Terminology

In what follows all the spatial coordinates are kept in millimetres, unless stated otherwise.

Interaction point, abbreviated IP, is the place where collisions and scatterings take place. It is assumed that it is placed in the geometrical centre of the CMS detector.

Sensor hit is information about an activated sensor strip. It contains the ordinal number of strip, which is translated into the spatial coordinate in the readout direction of the microstrip detector. It is oriented in U or V direction. Each `RPrecoHit` contains detector ID, position and its uncertainty. Its collection is stored in `edm::DetSetVector<RPrecoHit>`.

U/V pattern is a result of 1-RP pattern reconstruction. It is a process in which all the sensor hits per RP are considered to yield only patterns in U and V directions which are expected to come from physics tracks. Each pattern contains its position (in U or V direction), slope and corresponding sensor hits. Patterns are kept in collections per RP, `RPrecoPatterns`. In this package such reconstruction is performed by `RPNonParallelTrackCandidateFinder`. In order to suppress noise and reconstruct strongly inclined tracks it uses Hough transform. [3, Section 3.4]

Track is a set of values defining a linear path in space. It contains the line's intercept values b_x and b_y and slopes a_x and a_y at a specific z defined in the configuration. Where z is a distance from the IP (interaction point) along the beam.

$$\mathbf{t} = \begin{bmatrix} a_x \\ a_y \\ b_x \\ b_y \end{bmatrix} \quad (4.1)$$

Track description might also contain a covariance matrix reflecting the uncertainty of the parameters.

True track is a track which has been generated in a simulation. Usually, the reconstructed tracks are compared to it, to check whether are they compatible with it.

Hit candidate represents a possible hit, see Section 4.2.1 for more details on how it is created. It is created as a combination of U and V pattern coming from a single RP. Using

information from these two directions and the RP position it is effectively a point in 3D (x, y, z) space.

Track candidate is a line interpolating hit candidates from two different RPs. Covariance matrix is already a part of it.

Compatibility graph is a construct used to optimise the cluster building process. Vertices are track candidates and edges are created between candidates compatible with each other (χ^2 distance between candidates is considered).

(Track candidate) clique is a subset of track candidates, that contains only track candidates which are compatible with each other, i.e. connected in the compatibility graph.

Cluster is track candidate representing the average over the elements of one clique. It contains the covariance matrix computed as an average of covariance matrices of all contained track candidates.

Cluster – U/V pattern hit table is used during overlaps removal, see Section 4.2.3. It is a mapping between clusters and associated U/V patterns, i.e. a cluster and a pattern which lie closest to each other.

Hit count table is also used during overlaps removal and stores the total number of hits in each RP plane (U and V for each RP).

Overlap is a case, when two clusters are associated with the same U/V pattern (from the hit table).

4.2. Algorithm outline

The `Algorithm` class contains the main flow of the algorithm. Being given a collection of U/V patterns from the 1-RP reconstruction module, firstly it uses a cluster builder `ClusterBuilderSubsets` to create a list of clusters. This list is then passed to selection and filtering processes. Clusters which survived this far are fed to the overlaps removal algorithm `OverlapsRemoval` which is meant to reject the remaining false clusters. After that the clusters can be refined using information directly from sensor hits and not U/V patterns by applying the least squares method.

4.2.1. Cluster building

All combinations of U and V patterns from all possible pairs of RPs are considered forming track candidates. Track candidates are kept as vectors of their parameters a_x, a_y, b_x, b_y so that instead of looking for similar lines one can look for clusters of closely lying points in 4D space, which is the essence of *Hough transform*.

Some simple filtering is applied at this step, rejecting track candidates which are too inclined to be physical tracks. Remaining track candidates will form the set of vertices V_G of graph G (stored as an ordered vector).

Next, all pairs of vertices are analysed to create the edges E_G (a set of pairs of candidate ids). Edge is connecting only the vertices with distance between them below a limit set in configuration, i.e. parameter `edge_create_th`. The distance is calculated as:

$$d(\mathbf{t}_1, \mathbf{t}_2) = (\mathbf{t}_2 - \mathbf{t}_1)^T (\mathbf{V}_1 + \mathbf{V}_2)^{-1} (\mathbf{t}_2 - \mathbf{t}_1) \quad (4.2)$$

where $\mathbf{t}_1$ and $\mathbf{t}_2$ are vectors of track parameters, $\mathbf{V}_1$ and $\mathbf{V}_2$ the corresponding covariance matrices.

It is assumed that if a subset of vertices is to have been caused by a physical track then all of them must be compatible with each other. Assuming, that there is 100% acceptance (all detected tracks will be present in all active RPs) one can look for cliques of the size of $\frac{N-1}{2}N$, where N is the number of active RPs, which usually is equal to 3.

The module is performing the full lookup of the solution space. The cliques have too few vertices or don't include the required number of RPs (usually the number of active ones) are rejected. The remaining cliques are then converted into clusters and returned.

Its time complexity is $O(M!)$ where M is the number of track candidates. However, the average complexity is only $O\left(\left(\frac{N-1}{2}N\right)!\right)$ as the worst case with unacceptable complexity arises if all candidates forming the E_G are members of a single clique whereas physical tracks only form cliques smaller than $\frac{N-1}{2}N$. The number N is also limited, as only the horizontal, only the top or only the bottom RPs are likely to be active at the same time. The complexity issue must be taken into account though if some further modifications were to be implemented.

The algorithm only returns non-inclusive cliques, i.e. no clique is contained in any other clique. It happens though, that some cliques can overlap causing conflicts later on. There is no obvious way to deal with such issue though and no solution has been devised at the time.

4.2.2. Selection and filtering

Selection is dealing with the RPs associated with a cluster. If a cluster doesn't involve all the active RPs it is rejected. It is a redundant check, but as the older version of the cluster building algorithm didn't perform checks present in the current algorithm it is kept in place as it isn't time consuming.

Filtering makes sure that each cluster is actually passing through U/V patterns in all planes, except those it should miss due to limited acceptance. If only 3 RPs are working and the cluster search is tuned to search for cliques of size 3 it is redundant as well. If 4 RPs are active and one doesn't want to reject cliques of size 3 only, it is important, as it checks whether these cliques can be physical. The same applies for other configurations with the number of active RPs not corresponding to the size of cliques looked for.

Their implementation is included in the `Algorithm` class because of their simplicity.

4.2.3. Overlaps removal

The algorithm is contained in the `OverlapsRemoval` class. Apart from cluster collection it must be provided with the cluster – U/V pattern hit table and hit count table.

Firstly, it verifies if the hit counts for each plane can actually be satisfied by the clusters present. If not, the number of actually possible hits per plane is stored.

Then, the algorithm tries to find a combination of clusters which will contain the expected number of tracks, i.e. deduced from the numbers of U/V patterns in each plane, computed as the maximum number among the possible hits count. These tracks may not create overlaps where they aren't allowed to do so. Overlaps can only happen, if the number of U/V patterns in the plane is smaller than the expected number of tracks.

Based on these assumptions a full lookup of the remaining solution space is performed. As the number of allowed overlaps is of the order of 1–2 and the number of U/V patterns per

plane is bound by the number of particles per event then the complexity is $O(\binom{N}{M})$, where N is the number of clusters and M is the number of particles. M is rarely more than 2–3 and N is most often in the order of M . It happens rarely, but if N surges then the check described in the following paragraph stops the execution as it doesn't make sense to perform anyway.

Another restriction on the solution is that it must be unique. If more than one possible cluster combination is found, both are rejected and no clusters are returned.

4.2.4. Fitting

The track parameters that are in effect after the overlaps removal are still only based on the U/V patterns which themselves are an approximation of actual sensor hits. In order to improve the accuracy of the track reconstruction the least squares method is used.

The fitted track parameters are given by

$$\mathbf{P} = (\mathbf{G}^T \mathbf{V}^{-1} \mathbf{G})^{-1} \mathbf{G}^T \mathbf{V}^{-1} \mathbf{M} \quad (4.3)$$

where $\mathbf{V}^{-1}$ is an inverse of the covariance matrix. In what follows d_x^n and d_y^n are projections of readout direction (more or less U or V) of detector which detected the n^{th} hit to x and y directions of the global coordinate system, c_x^n , c_y^n and c_z^n define position of the centre of that detector in the global coordinate system, z_h is position of the plane perpendicular to the beam direction at which intercepts are measured, N is the number of hits associated with the track, σ_i is the uncertainty of i^{th} hit position p_i .

$$\begin{aligned} \mathbf{M} &= \begin{bmatrix} p_1 + c_x^1 d_x^1 + c_y^1 d_y^1 \\ \vdots \\ p_N + c_x^N d_x^N + c_y^N d_y^N \end{bmatrix}, \\ \mathbf{G} &= \begin{bmatrix} d_x^1(c_z^1 - z_h) & d_y^1(c_z^1 - z_h) & d_x^1 & d_y^1 \\ \vdots & \vdots & \vdots & \vdots \\ d_x^N(c_z^N - z_h) & d_y^N(c_z^N - z_h) & d_x^N & d_y^N \end{bmatrix}, \\ \mathbf{V}_{i,j}^{-1} &= \begin{cases} \frac{1}{\sigma_i^2} & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}, \\ \mathbf{P} &= \begin{bmatrix} a_x \\ a_y \\ b_x \\ b_y \end{bmatrix}. \end{aligned}$$

5. Testing and performance

Although there weren't any unit or other automatic tests created the module must have been verified in some way during the development process.

As the module's main role was to yield tracks based on data from detectors (simulated or not) additional module was created to compare the set of tracks simulated with the ones returned from the reconstruction module, it is called `RPStationMultiTrackFinderFitterAnalyzer`. Basically, it takes these two sets of tracks and tries to match them with each other analysing each event and aggregating the data from the single run and giving statistics as well. There are two basic failures distinguished: fake and missing tracks.

Fake tracks These are tracks which were returned from the reconstruction process, yet they cannot be matched with tracks generated. It means that some mostly random information is present in the output. The number of fakes should be minimised as much as possible.

Missing tracks These are the tracks which are present in the simulated tracks set, yet haven't been reconstructed.

The reconstruction process, as described in Chapter 4 has two main phases. Firstly, clusters are created without much filtering, i.e. not paying attention if there are too many of them or if they induce overlaps. Of course there are some cuts introduced, not to consider utterly nonsensical tracks, that is mainly the ones which are just too inclined. Secondly, this set of track candidates (or clusters) is reduced to match actual U/V patterns found. This phase must be performed so that the result is unambiguous while preserving the correct tracks. Making sure that it is the case is the very reason why the hardware modification, see Chapter 2 was introduced.

5.1. Track distributions

The module is planned to process data coming from the detectors. As the protons will be reaching the detectors in different circumstances several scenarios must be taken into account separately. One can see the exemplary distributions of hits in Figure 5.1. The densities of hits vary greatly between different distributions, so the difficulty of reconstruction is also different in each case.

5.1.1. Uniform distribution

In the beginning only a simple distribution of tracks was used. It doesn't resemble the physical distributions but it was simple enough to understand main issues with the reconstruction process

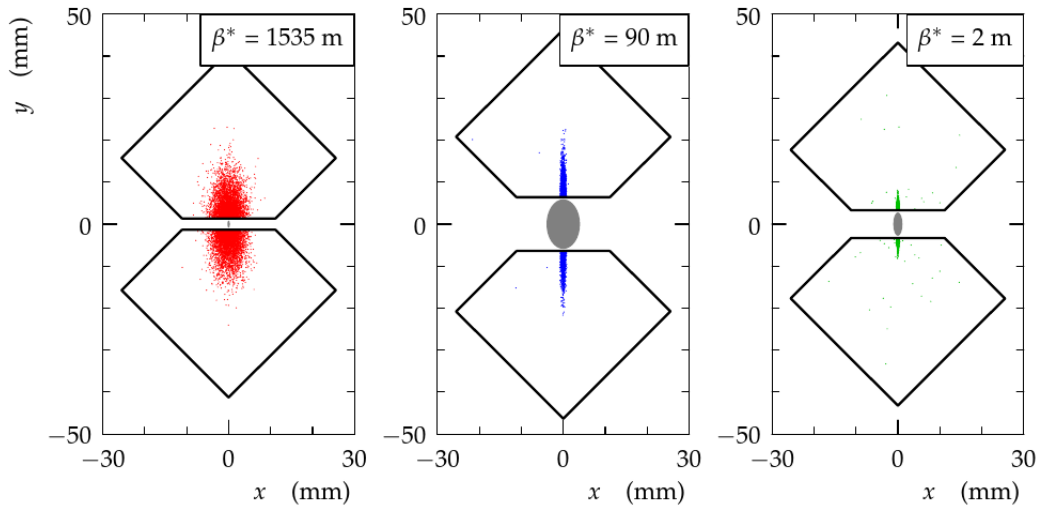


Figure 5.1: Simulated distributions of elastic-scattering protons in the 220-Far unit, each colour corresponds to an optics type. The grey ellipses show 10σ beam contours. The RP sensors are placed at $10\sigma + 0.5$ mm. [3, p.40]

in the early phase. It is assumed that track intercepts are distributed uniformly throughout a specified rectangle and track angles distributed according to Gaussian distribution with $3 \mu\text{rad}$ RMS. See Figure 5.2. This distribution will be referred to as *uniform*, in what follows.

5.1.2. 4 TeV, $\beta^* = 90$ m

This distribution was simulated in order to reflect a distribution of tracks for the energy of 4 TeV per beam with $\beta^* = 90$ m optics, see [3, Section 2.2] for definition of β^* . This distribution consists of two parts, which are later mixed together (see `CachedMultiTrackSource`). There are protons which come from elastic scattering and single diffraction, their distributions can be seen at Figure 5.3 along with both of them combined and constrained to tracks passing through detectors. The proportion in which they were mixed together is 25:7 (elastic scattering:single diffraction).

Most protons are detected in the top and bottom RP (IDs: 100, 104, 120 and 124) so only these RPs were used for simulation and reconstruction. The distribution will be referred to as *medium beta*, in what follows.

5.1.3. 4 TeV, $\beta^* = 0.6$ m

Another physics distributions corresponds to the $\beta^* = 0.6$ m, see Figure 5.4.

This distribution consists only of protons coming from the single diffraction process. As they lose energy during the interaction they tend to drift horizontally. Most protons are visible in the horizontal RPs (IDs: 102, 103, 122 and 123), so only these RPs are active for this distribution. The distribution will be referred to as *low beta*, in what follows.

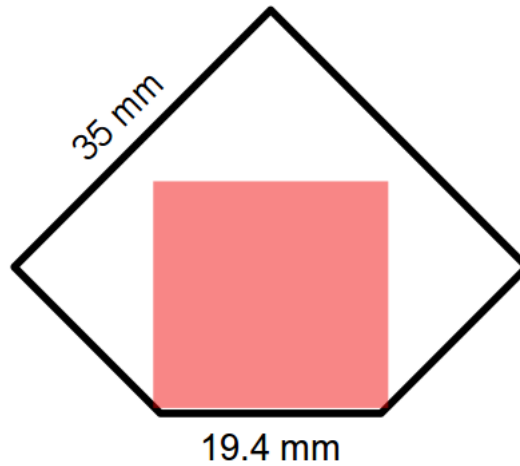


Figure 5.2: Uniform track distribution as seen in a RP. The red region is where track intercepts are distributed uniformly.

5.2. Debugging and verbose output analysis

In order to put RP IDs which are presented in further graphs in perspective, Figure 5.5 shows how the RPs are placed around the beam.

Verbose output can be adjusted using the `Verbose` parameter which is passed to the module. It is set to a high number by the `test/selected_event_analysis_cfg.py` configuration so that errors can be spotted right where they appear.

The output is rich and impossible to grasp for events with more protons present. Even two protons can produce events with hefty logs. Thorough analysis is usually done for simple events. Only when everything works well more complex cases are analysed.

In order to avoid doing the analysis by reading through the logs, it is possible to use visualisation tool placed at `utils/verbose_to_asy.py` and written in Python. It processes a fully verbose output from the reconstruction returning asymptote code, which should be rendered to see the analysis plots. An example of such visualisation is given in Figure 5.6.

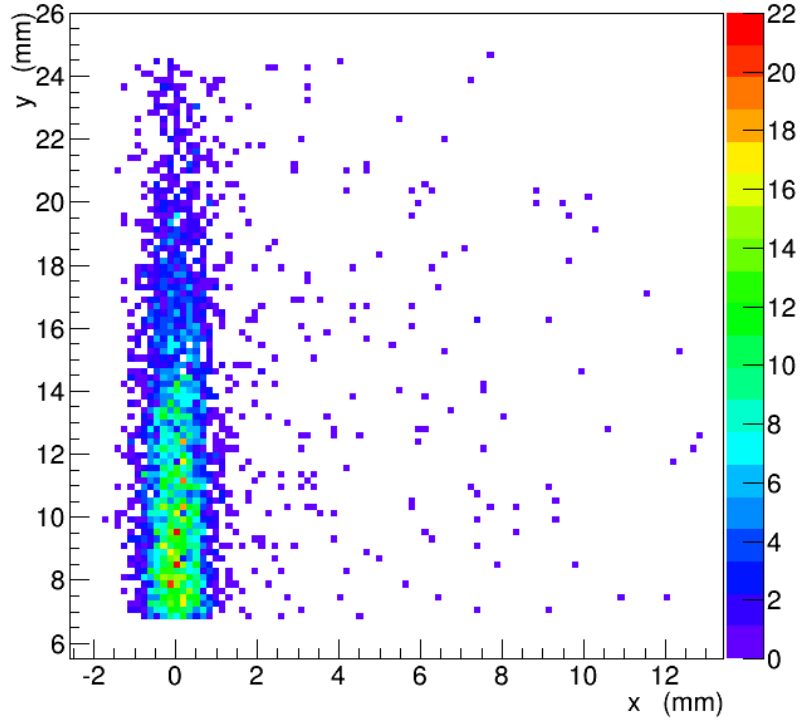
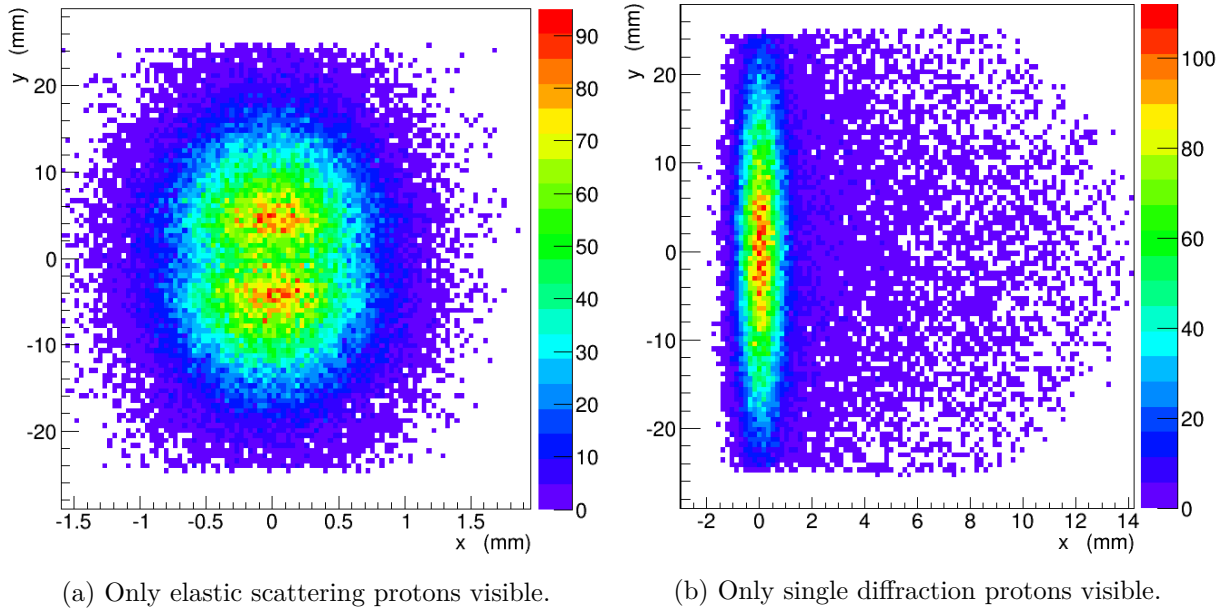
5.3. Final performance results

The performance assessments has been done all along the development process. In order not to litter this document with all of them only the most substantial results will be presented.

5.3.1. Comparison with the previous module

Before the multi-tracking capabilities were present due to the hardware modification the modules were used only with 2 RPs active. The module which was used for reconstruction was `RPNonParallelTrackCandidateFinder`, which is still used by this module to obtain U/V patterns.

The performance of both modules was assessed using the *medium* and *low beta* distributions. In order for the current module to be able to reconstruct such events correctly the clustering part of the algorithm had to be deactivated – with only two RPs active there is no need for clustering.



(c) Both distributions combined with only protons passing through at least one RP visible. Distributions were mixed in 25:7 proportion accordingly.

Figure 5.3: Scatter plots of proton tracks for the 4 TeV, $\beta^* = 90$ m scenario. They are projected on the plane perpendicular to the beam line at z close to the RPs positions. Notice the differences in scale between axes.

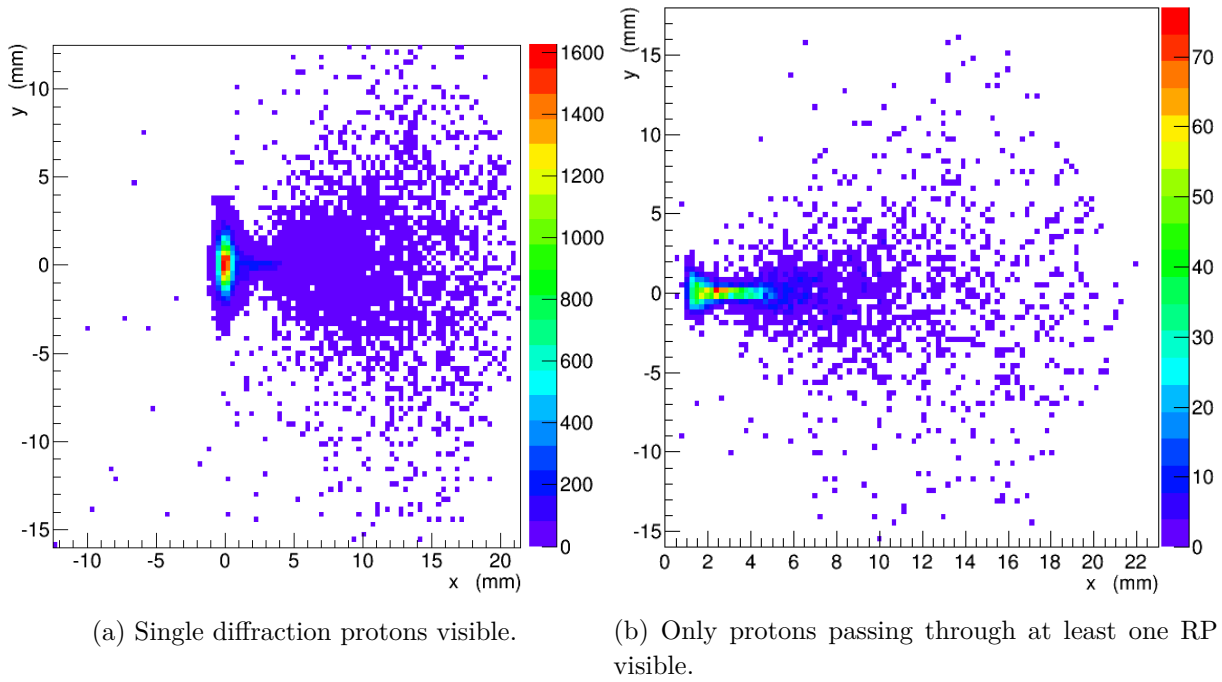


Figure 5.4: Scatter plots of proton tracks for the 4 TeV, $\beta^* = 0.6$ m scenario. They are projected on the plane perpendicular to the beam line at z close to the RPs positions. Notice the differences in scale between axes.

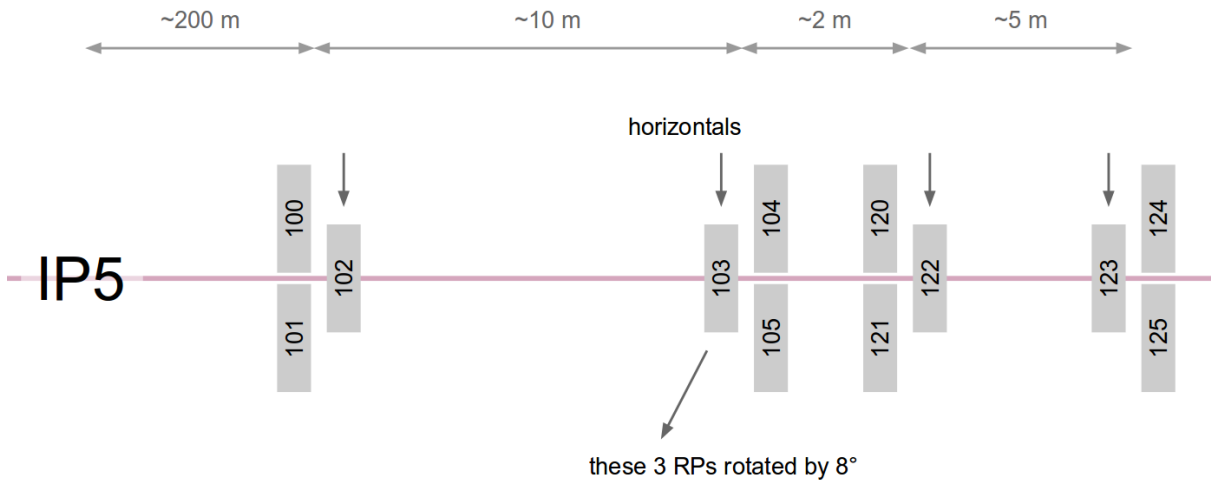


Figure 5.5: An overview of the RPs configuration around the beam, shown in light red. The distances are not to scale and are only rough approximations. The view is from outside the LHC ring, meaning that horizontal RPs are located on the outside.

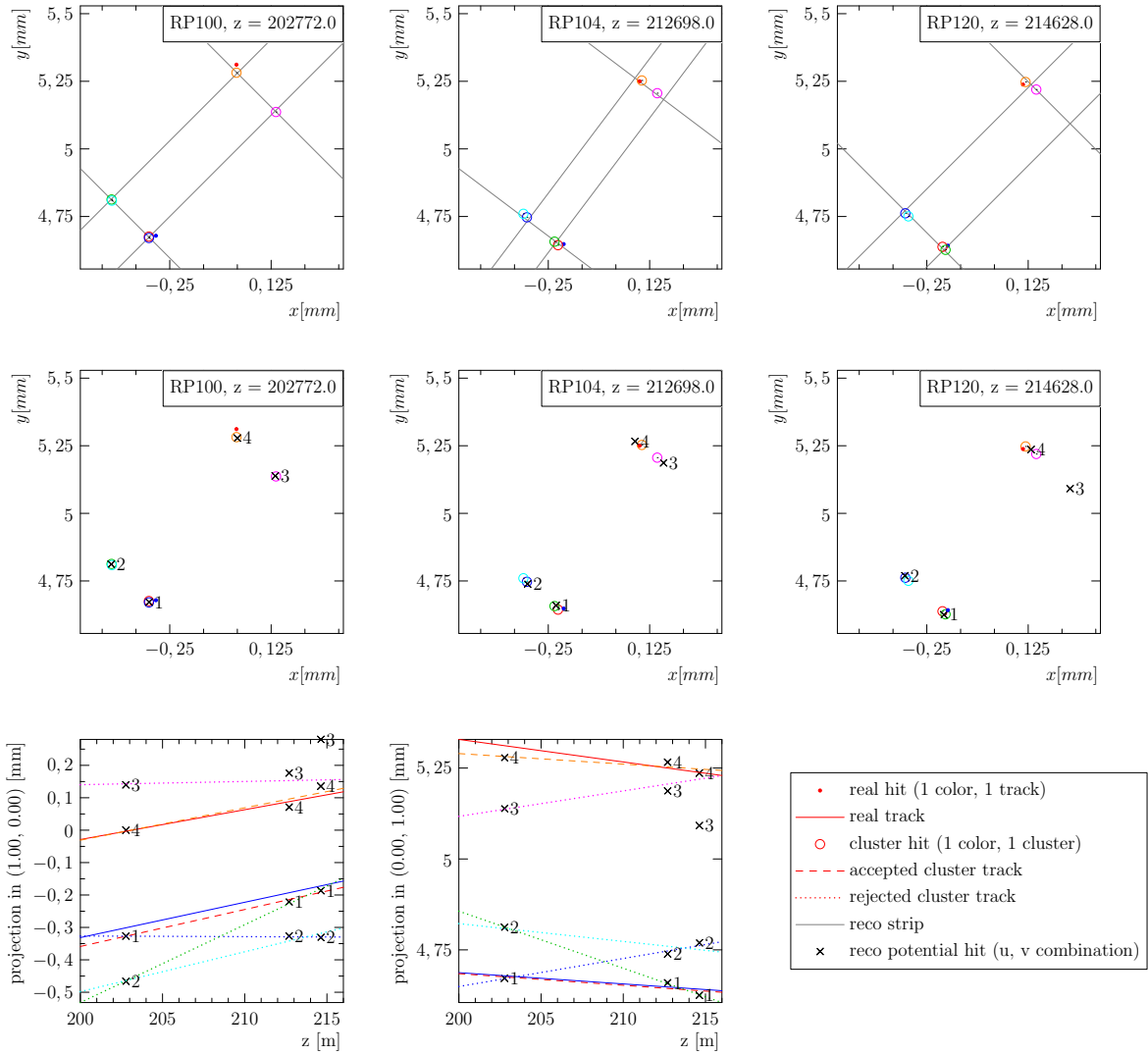


Figure 5.6: An example of the visualisation output. First two rows are showing XY plane at z of a specified RP. In the first row U/V patterns are shown as grey lines (RP 104 is rotated slightly). Sensor hits are dots and track candidate (cluster) hits are circles. In the second row numbered U/V pattern intersections (potential hits) are shown as crosses. Third row are the ZX and ZY projections. Potential hits are visible, as well as simulated tracks (solid lines), clusters rejected during overlaps removal (dotted lines) and clusters accepted (dashed lines). In this particular event correct tracks are reconstructed, the dashed lines are close to the solid ones.

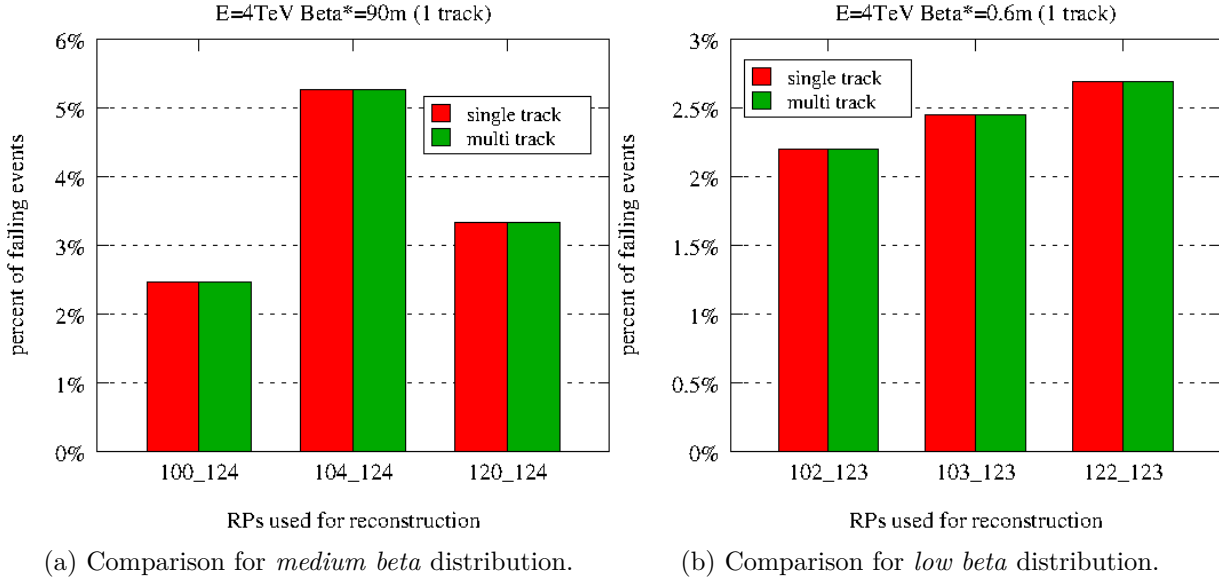


Figure 5.7: Graphs comparing the performance of the old module (RPNonParallelTrackCandidateFinder, referred to as *single track*) with the new one (RPStationMultiTrackFinderFitter, referred to as *multi track*). The tests were performed for a single track per event only. The numbers in labels refer to RPs active during reconstruction.

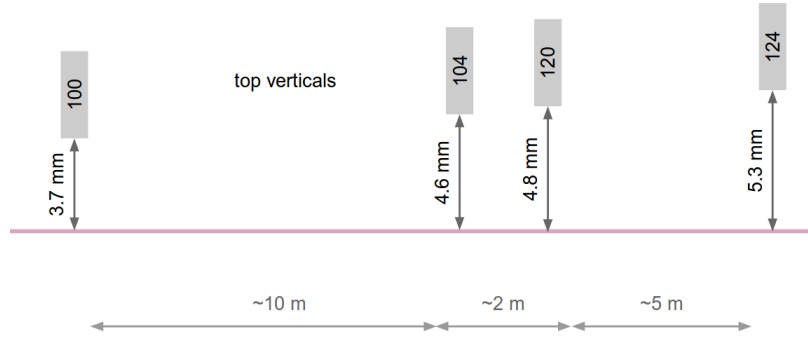
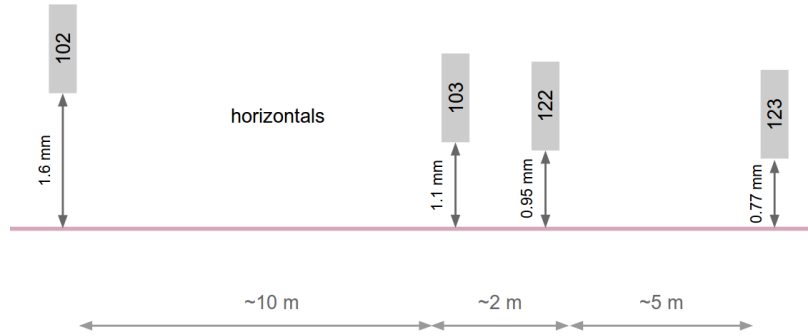
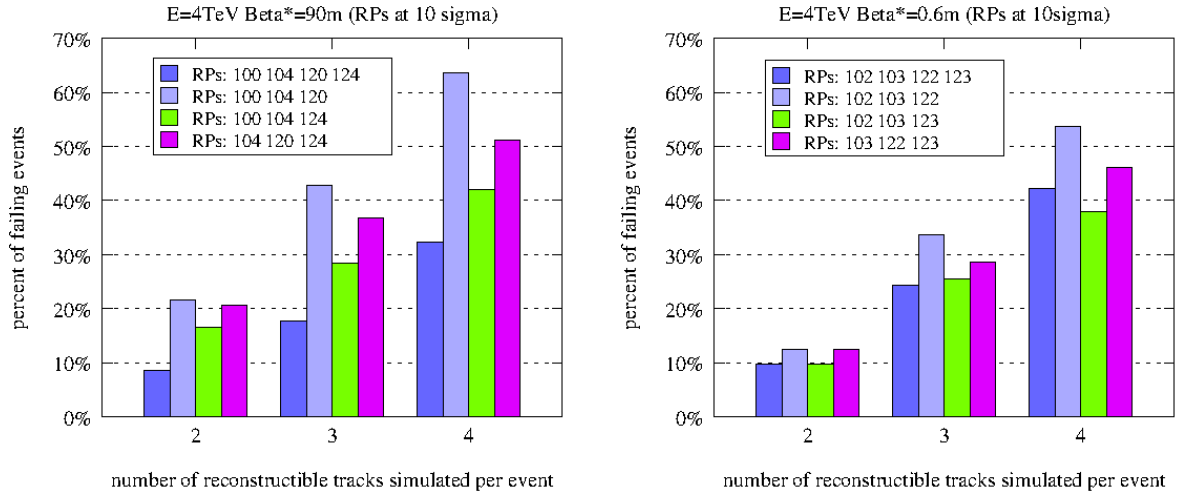
The previous module lacked the fitting part so it has been provided by RPStationFitter. The performance hasn't been hindered by the new algorithm as seen in Figure 5.7.

5.3.2. Multi-track performance

In order to avoid issues due to limited acceptance (that is tracks outside sensor surface) the RPs have been aligned to have their edges at 10σ from the beam, as shown in Figure 5.8.

Figure 5.9 shows the final results of the multi-track reconstruction performance. They were obtained for different combinations of RPs active, as is indicated in the legend.

The results indicate that the performance is worse than for single track events. The failing events are mostly coming from missing tracks, i.e. very few fake tracks are returned. It means that there is high probability that the reconstructed tracks are true physical tracks.

(a) Distances of RPs from the beam for the *medium beta* distribution.(b) Distances of RPs from the beam for the *low beta* distribution.Figure 5.8: Final RP positions for both the *medium* and *low beta* distributions.(a) Performance for the *medium beta* distribution. (b) Performance for the *low beta* distribution. Only top RPs are considered.Figure 5.9: Final performance for both *medium* and *low beta* distributions. Each colour corresponds to a different set of RPs active, as indicated in the legend. The columns give the number of particles per event in each test.

6. Package architecture

The aim of this chapter is to describe the structure of the package and its contribution to the pre-existing computing model.

Directories in this chapter, if not indicated otherwise, are relative to the main directory of the module, i.e. `TotemRecoRP/RPStationMultiTrackFinderFitter`.

6.1. CMSSW - Event Data Model

TOTEM experiment uses software which extends the capabilities of the CMSSW framework which is used by the CMS collaboration.

The CMSSW works by grouping together a set of plugin modules and connecting them in the order of execution and setting their parameters to adjust their behaviour. Data are passed between modules in the form of *Events*. This way of arranging modules together with data is called the Event Data Model (EDM). [6, WorkbookCMSSWFramework]

There are six types of modules available in the CMSSW framework, from which only the following are used in the project.

`EDProducer` performs specified operation on the data stored inside an Event and puts the result back to the Event.

`EDAnalyzer` doesn't put anything back to the Event, it analyses the data and gives the results in some other way, for example by writing them to output.

`EDFilter` returns a boolean for every Event to know if it fits some conditions and filters out Events which don't match them.

A module code consists of C++ source code and python configuration files which initialize it with parameters. A C++ class that represents a module must inherit from an abstract module type from EDM C++ libraries.

For instance, a module `RPMultiTrackFilter` which is of type `EDProducer` has a following definition

```
#include "FWCore/Framework/interface/EDFilter.h"
//(...)
#include "FWCore/Framework/interface/MakerMacros.h"
//(...)
class RPMultiTrackFilter : public edm::EDFilter {
//definitions of fields and methods
}
DEFINE_FWK_MODULE(RPMultiTrackFilter);
```

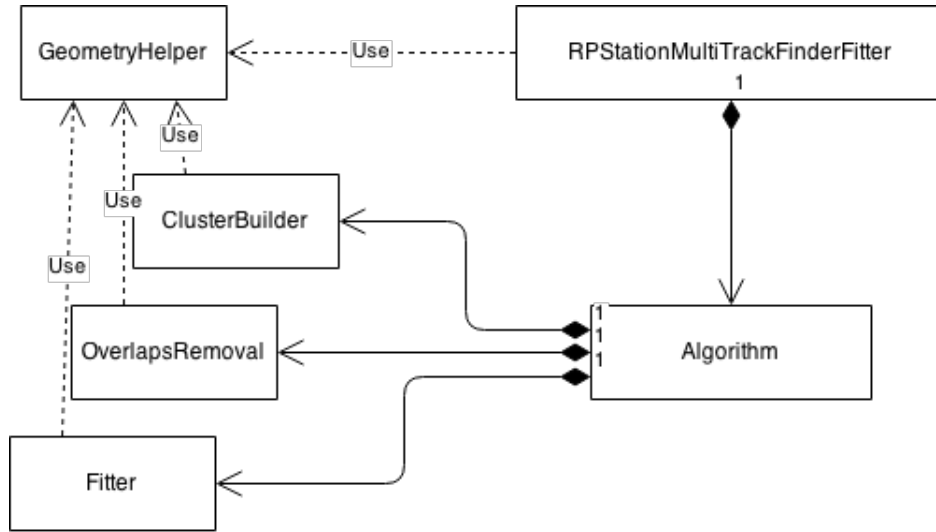


Figure 6.1: Architecture of the `RPStationMultiTrackFinderFitter` module.

Inside a configuration file, a module sequence, through which the Event will be passed by the framework is specified. In case of simulated events, which were used in this project, the path consists of geometry setup, simulation, reconstruction and analysis.

6.2. Modules

The package is developed as a module of the CMSSW in order to be able to use it with the CMSSW's EDM.

The package consists of the main fitter module called `RPStationMultiTrackFinderFitter` and helper modules.

6.2.1. `RPStationMultiTrackFinderFitter` module

It is the main module of the package which performs the reconstruction process. It is an adapter which provides a compatibility layer between the CMSSW and the reconstruction algorithm. It extends the type `EDProducer`. Its structure is shown in Figure 6.1. The building blocks are described in Chapter 4.

6.2.2. Helper modules

`RPStationMultiTrackFinderFitterAnalyzer`

A module of type `EDAnalyzer`. It is designed to compare the simulated tracks with those which are reconstructed. It takes parameters, such as names of modules responsible for reconstruction and simulation, maximal number of missing and faked tracks, and the names of files where histograms and statistics will be kept.

`CachedMultiTrackSource`

A module of type `EDProducer` that reads tracks from a file (or possibly files) and produces Events containing desired track mixtures.

RPMultiTrackFilter

A module of type `EDFilter` which takes a collection of fits or recognized patterns as input. It is constructed with a set of criteria, such as number of tracks or distance between patterns.

RPStationFitter

A module of type `EDProducer`. It takes input in form of tracks candidates, performs only fitting and returns a collection of track fits.

6.3. CMSSW configurations

Configurations which were used for testing are placed in the `test` directory. They are designed to make it possible to extend the analysis to different track distributions.

Processing path is created as indicated in Section 6.1. The simulation is performed using modules from `/TotemAlignment/RPFastSimulation`. The reconstruction part consists of 1-RP analysis (`RPNonParallelTrackCandidateFinder`) and multi-track fitting (`RPStationMultiTrackFinderFitter`). Analysis is performed by `RPStationMultiTrackFinderFitterAnalyzer`. In case filtering is needed `RPMultiTrackFilter` is used to limit number of reconstructed events.

Each scenario imports the specific configuration of modules from the python module with configuration specified as an execution parameter. Such configuration specifies the track distribution to be used. The scenario only modifies the parameters which are crucial for its context of operation, i.e. number of events processed, event track multiplicity, active RPs and verbosity levels to ensure output readability. Thanks to this it is possible to use different track distributions, defined in subdirectories, with configurations specifying the mode of operation.

There are two modes of operation are available:

set_particle_count

It is meant to analyse a bulk of events. Typically, it runs the reconstruction for a large number of events to provide meaningful statistics. The output is written to files `stats_<track count>` and `hists_<track count>.root`. Verbose output is suppressed.

selected_event_analysis

It is meant to thoroughly analyse a single event. Verbose output is set to a high level for the single event to be analysed. No files are written.

In order to analyse the same event which was analysed automatically during bulk analysis (events are generated pseudo-randomly) the simulation is performed for all the events preceding the chosen event, but no reconstruction is performed for them.

6.4. Visualisation

It is possible to visualise a selected event. A python script has been written which parses the output from the `selected_event_analysis` and outputs an asymptote code which will generate the visualisation when rendered using `asy`. The script called `verbose_to_asy.py` is placed in `utils` directory alongside with `pad_layout.asy`, which is written by Jan Kašpar and organises the asymptote output better than the default.

6.5. Code management

The TOTEM is using an extended version of the CMSSW framework which is available as an svn repository under ssh://svn.cern.ch/repos/totem/branches/CMSSW_7_0_4/offline/cmssw/src. It is also possible to inspect the repository from a browser under https://svnweb.cern.ch/cern/wsvn/totem/branches/CMSSW_7_0_4/offline/cmssw/src/. During summer 2014, a migration has been performed from version 6.2.0 of the CMSSW to version 7.0.4, hence the branch name.

The plugin which is the matter of this document is placed at the path `RecoTotemRP/RPStationMultiTrackFinderFitter`. If simulation module which might be mentioned later is placed at `TotemAlignment/RPFastSimulation`.

All classes except for the ones exposed to the CMSSW framework are defined in the `RPStationMultiTrackFinderFitter` namespace to avoid global name collisions.

7. Project environment and usage

The chapter attempts to describe the process of setting the project up as well how to run it. Usage patterns used during its development and testing are presented.

Relative path in this chapter, if not indicated otherwise, are relative to the main directory of the module, i.e. `TotemRecoRP/RPStationMultiTrackFinderFitter`. Absolute paths, beginning with a slash / are relative to the SCRAM project's `src` directory.

7.1. Building SCRAM project

First, one has to set up a scram project. It uses the pre-compiled files of a basic project in desired version and architecture from afs. As TOTEM uses a modified version of the framework, CMSSW TOTEM files must be placed in the `src` directory of the project in order to perform the required tasks.

The desired architecture (with information about compiler) has to be set. It can be done using the command:

```
export SCRAM_ARCH=slc6_amd64_gcc481
```

In order to place scram on one's PATH variable and set up the environment:

```
source /afs/cern.ch/cms/cmsset_default.sh
```

Then, it is possible to review all the available CMSSW versions available for the architecture.

```
scram list
```

For our purposes version 7.0.4 will be used. The project for this version is created by:

```
scram project CMSSW CMSSW_7_0_4
```

The project files will be placed in the directory `CMSSW_7_0_4`, relative to the working directory when running the command. Following commands will be executed from this directory.

It is possible to run the framework already, but to use the additional TOTEM functionality one has to checkout the code from the TOTEM code repository to the `src` directory of the project.

```
svn co svn+ssh://svn.cern.ch/repos/totem/branches/CMSSW_7_0_4/  
offline/cmssw/src/ src
```

Current code of the TOTEM CMSSW will be placed in `src` directory.

Now all is set to build the project. Remember about the patch which is presented in the following subsection if you wish to use the `test/set_particle_count_cfg.py` configuration.

```
scram build -j <number of threads>
```

First build can take some time, following builds should finish much quicker.

In case of some additional trouble refer to <https://twiki.cern.ch/twiki/bin/view/TOTEM/CompOfflineGettingStartedCMSSW704>.

7.1.1. Patching the code (dirty hack)

There are changes in `TotemRawData/Filters/plugins/EventNumberTimeFilter.cc` (see 7.3.2 for more explanation) that need to be applied before configuration file `test/set_particle_count_cfg.py` can work properly. They are included in `filter_fix.diff`. This is because they are useful only when working with modules described in this guide. In order to apply them move to the root of the branch (one with directory `TotemRawData`) and run

```
patch -p0 -i RecoTotemRP/RPStationMultiTrackFinderFitter/
    filter_fix.diff
```

7.2. Simulations

Simulation is to be run from the `test` directory, relative to the module's directory. The configuration files which provide different modes of operation are present there.

Geometry, track distribution are defined in configurations (usually named `simulate_cfg.py`) placed in `test/**scenario_name**` directory. Each configuration file is an ordinary Python module.

7.2.1. Components of a simulation

Because input is took from `CachedMultiTrackSource` module, which reads it from files like `sample_ES` and `sample_SD` (to be found under `test/**scenario_name**`), there is no need to get input from standard EDM source, so a source is set to `EmptySource`. `EmptySource` passes empty output to `GeometryInfoModule`, a module printing info about geometry, that is about RPs alignment, readout directions, etc. After `CachedMultiTrackSource` finishes its activity, it passes information tracks to `RPFastFullSimulation200`, or some other simulation module, which can be found under `/TotemAlignment/RPFastSimulation`. This is where tracks are converted to sensor hits. A collection of hits gets to `RPNonParallelTrackCandidateFinder`. In this case it is needed only to produce RP patterns. `RPStationMultiTrackFinderFitter` uses them as input, does reconstruction and puts computed tracks back to the event. `RPStationMultiTrackFinderFitterAnalyzer` fetches tracks generated by `RPFastFullSimulation200`, RP patterns computed by `RPNonParallelTrackCandidateFinder` and tracks reconstructed by `RPStationMultiTrackFinderFitter` from the event. Then it performs final analysis and writes statistics to text file (called `stats` by default) and histograms to file called `results.root` by default.

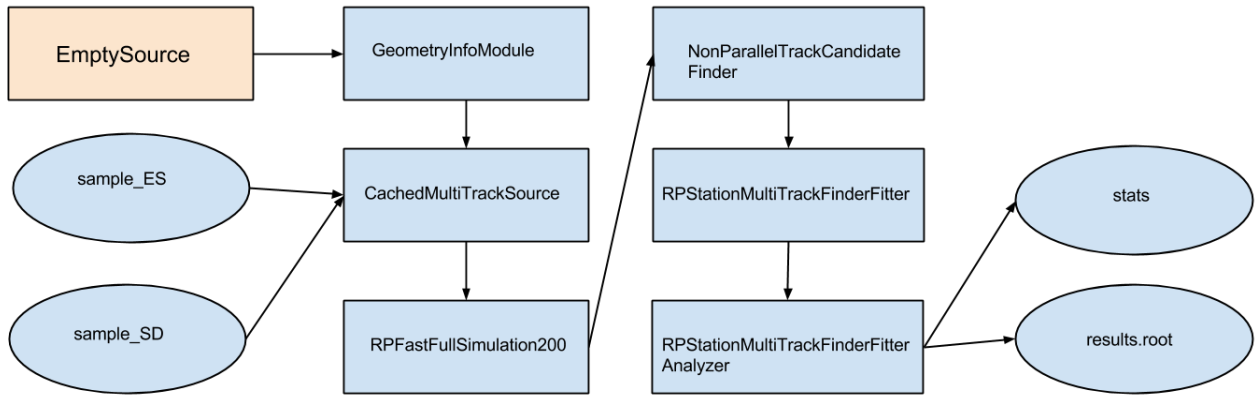


Figure 7.1: Modules and files in simulation scenarios. Source is shown in light orange, files in ovals.

7.2.2. Adjustable reconstruction parameters

There are a few geometrical reconstruction parameters which can be configured from `simulate_cfg.py` by assigning desired value to `process.RPStationMultiTrackFinderFitter.[parameter_name]` after this module is loaded. For instance,

```

process.load("RecoTotemRP.RPStationMultiTrackFinderFitter.
             RPStationMultiTrackFinderFitter_cfi")
#(...)
process.RPStationMultiTrackFinderFitter.ay_mean = 261.5e-6

```

sets `ay_mean` to $261.5e - 6$. Such configuration overwrites the settings in `**module_directory**/python/RPStationMultiTrackFinderFitter_cfi.py`. Each of this parameters is meaningful in track candidate rejection. Setting them properly can lead to reduction of false and missed tracks.

1. `ax_cut`, `ax_mean`, `ay_cut`, `ay_mean`

The algorithm assumes that a track is a three-dimensional straight line defined by equations $x = a_x \cdot z + b_x$ and $y = a_y \cdot z + b_y$. As tracks with unreasonable slope values need to be eliminated, we only consider tracks with a_x parameter such that $|a_x - a_x^{mean}| < a_x^{cut}$. `ay_cut` and `ay_mean` work analogically for a_y . It gives a tester control over track's slope both in vertical and horizontal direction. `ax_cut` and `ay_cut` both default to $800\mu rad$ while `ax_mean` and `ay_mean` take the default value of 0.

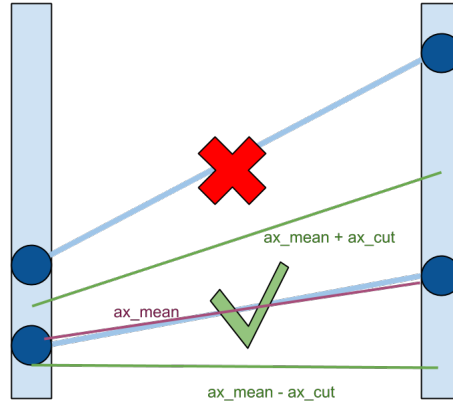


Figure 7.2: Two tracks, first one's a_x parameter is out of range

2. z_h

This parameter's default value is $212698mm$. It represents the distance between IP5 and a plane on which track intercepts are projected. In other words, $-z_h$ is IP5's z coordinate in global coordinate system.

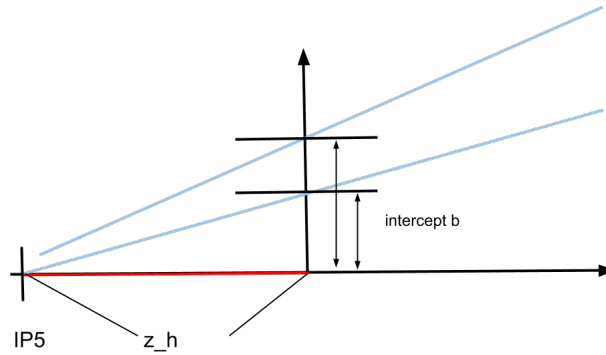


Figure 7.3: z_h shown in red

3. $edge_create_th$

Knowing the parameters of two tracks it is possible to calculate distance between them. As mentioned in description of ax_cut , ax_mean , etc. parameters, it is assumed that a track is a straight line given by equations $x = a_x \cdot z + b_x$ and $y = a_y \cdot z + b_y$. Distance between two tracks can be computed as

$$d = (t_1 - t_2)^T (V_1 - V_2)^{-1} (t_1 - t_2)$$

where

$$t_i = \begin{pmatrix} a_x^i \\ a_y^i \\ b_x^i \\ b_y^i \end{pmatrix}$$

for $a_x^i, a_y^i, \dots$ being the parameters of i 'th track. V is a matrix of Roman Pot measurement uncertainty. It's computed from sensor strip width, which is given in the same configuration file as `uv_unc`. The closer two possible track candidates are, the more likely is that they actually represent single track. Clusters are formed by grouping together lines such that every two of them are close enough to each other. If the distance is less than `edge_create_th` they can belong to common cluster. Default value is $7mm$.

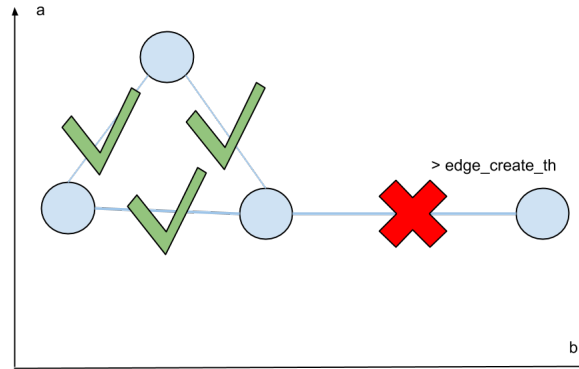
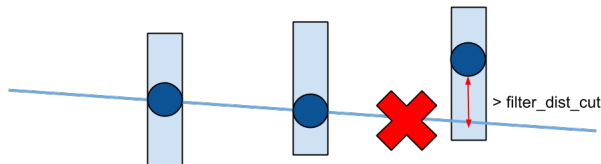


Figure 7.4: Simplified model of four track candidates in two dimensions (a represents a_x and a_y , b is for b_x and b_y). Candidate on the right is too far from one of possible tracks, hence it can't be included in the cluster.

4. `filter_dist_cut`

An U/V pattern is a hit candidate from a single Roman Pot described in this pot's own two dimensional coordinate system. Any cluster that has a smallest distance to any RP's U/V pattern greater than `filter_dist_cut` is rejected. The distance is computed from the center of a Roman Pot to a place where the plane parallel to the strips and containing that point meets the cluster. Defaults to $4mm$.

Figure 7.5: Cluster rejected because it's distance to last RP's U/V is too big.



5. `track_pattern_assoc_cut`

This variable is similar to `filter_dist_cut`, but serves different purpose. It is used to show the overlap removing algorithm whether a track is close enough to U/V pattern to indicate that this track triggered the pot at U/V coordinates. It is computed in the same way as `filter_dist_cut`. Defaults to *4mm*.

6. `enableFitting`

This should be set to either true or false. It tells the algorithm whether to use fitting, i.e. to perform the least squares fitting with the information from all sensor strips. By default fitting is turned on

7.3. Modules for additional tasks

There are also modules at test directory level which do not modify distribution or reconstruction parameters but adjust the way simulations are run and what output is generated.

They operate on a track distribution which is passed to them as a name of a Python module.

That track distribution module is assumed to be of structure similar to the one described in 7.2.1. Original parameters of modules are changed by loading process from distribution module by instructions

```
module = __import__(module_name, globals(), locals(), ['process',
])
process = module.process
```

, selectively changing parameters of modules it consists of and adding filters to control the simulation process.

7.3.1. Performance testing

To test performance for a large number of events, run

```
cmsRun set_particle_count_cfg.py <module> <track count> <event
count> [<RP> ...]
```

For example, in order to run a 4000GeV Beta*=90m track distribution with two tracks present and 1000 events generated using only RPs with IDs: 100, 104 and 120 one should run

```
cmsRun set_particle_count_cfg.py 4000GeV_90m.simulate_cfg 2 1000
100 104 120
```

Verbose output is suppressed. Output is generated in `stats_<track count>` and `hists_<track count>`. In `stats_<track count>` lines have the following format:

```
<run>:<event> <missing>/<fake>
```

6 last lines contain statistical analysis of the former lines, they should be self explanatory.

An example output looks like this:

```
\ldots
1:998 0/0
1:999 2/0
1:1000 0/0
Summary:
total: 839
failed: 88:10.49%
fake: 3:0.36%
missing: 88:10.49%
both: 3:0.36%
```

7.3.2. Analysis of a single event

In order to be able to select a single event for analysis a module from another package is used – `EventNumberTimeFilter`. It is aimed at real data, not simulation so it checks if there is a proper raw event associated. In case of simulation it is not true. In order to use it only for event number filtering the part validating aforementioned condition must be commented out. It can be done manually or by following instruction from 7.1.1, so in case of following instructions from the beginning it is already done.

A thorough analysis of a single event may be performed by running

```
cmsRun selected_event_analysis_cfg.py <module> <track count> <
event number> [<RP> ...]
```

No statistics are generated in such run. All verbose output is enabled which makes it possible to analyse what went wrong during the reconstruction of the selected event.

To understand the output reading the docs and the code is advised.

Should the need arise it is also possible to pipe the output to `utils/verbose_to_asy.py [RPid ...]` in order to obtain visualisation of event. The output of the tool is in asymptote code so it should be rendered with `asy` in order to obtain readable graphs.

7.4. Generate track distribution

As generating track distributions may be more time consuming than reconstruction itself, generation configurations are available in directory `test/**scenario_name**` directory under the name `generate_ES_cfg.py` or `generate_SD_cfg.py` for elastic scattering and single

diffraction respectively, depending on which of them is or are used. This information can be obtained from `test/**scenario_name**/README` file or by reading the assignment of variable `process.generator` in file `simulate_cfg.py`. For instance, if it looks as follows

```
process.generator = cms.EDProducer("CachedMultiTrackSource",
    verbosity = cms.untracked.uint32(0),

    tracksPerEvent = cms.uint32(5),

    input = cms.VPSet(
        cms.PSet(
            weight = cms.double(7),
            fileName = cms.string(working_dir+"/sample_SD"),
        )
    )
)
```

it is clear that only single diffraction is used.

7.5. Documentation

To generate documentation and put it into `doc` directory run:

```
doxygen Doxyfile
```

`doxygen` is a tool for documentation generation for a few programming languages, including C++. It's easily obtainable due to being open source. `Doxyfile` is a configuration file located directly in `RPStationMultiTrackFinderFitter` directory. Each option inside the file is thoroughly described in comments so it's relatively easy to suit the configuration to one's needs. One of the more interesting options is enabling xml output generation. It produces files in xml format containing information about classes and directories which can be useful for example in generating UML diagrams.

If for some reason `Doxyfile` is not present, template of another one can be generated by running

```
doxygen -g [configFileName]
```

8. Modules' inputs and outputs

This chapter is intended to show the general rules to apply in order to connect modules from package `RPStationMultiTrackFinderFitter` to each other and their interaction with other modules.

8.1. Input, output and Event Setup

Every module needs to fetch certain data from events which are put into it and produces certain types of data, of desired class or with desired label. This is crucial in communication with other modules. In case of absence of such data program throws an exception at the point of dereferencing data from an event. It is also important to consider Event Setup. It describes some circumstances occurring in a period of time when an event was created (for example in a detector). More description, including how setups are provided can be found at <https://twiki.cern.ch/twiki/bin/view/CMSPublic/WorkBookMoreOnCMSSWFramework>. Technical details, including using Event Setups in a module C++ code can be found at <https://twiki.cern.ch/twiki/bin/view/CMSPublic/SWGuideEventSetupHowTos>.

8.1.1. Input events

Modules' input objects are indicated below. If only class name is given, they are retrieved from Event by class. It is also possible to provide a string name to look for.

CachedMultiTrackSource

Doesn't fetch anything from input event.

RPMultiTrackFilter

- `RPRemovedCandidatesCollection`, by module label passed as parameter "generatorLabel", when any of parameters "trackCount.active", "reconstructableTrackCount.active" or "minDistBetweenTracks.active" is set to True
- `RPRemovedCandidatesCollection`, by module label passed as parameter "stationRecoLabel", when parameter "recoTrackCount.active" is set to True
- `RPRemovedPatternsCollection`, by module label passed as parameter "oneR-PRemovedLabel", when parameter "minDistBetweenPatterns" is set to True

RPStationFitter

- `RPTrackCandidateCollection`, by module label passed as parameter `"RPTrackCandCollProducer"`

RPStationMultiTrackFinderFitter

- `RPSRecognizedPatternsCollection`, by module label passed as parameter `"RPTrackCandCollProducer"`

RPStationMultiTrackFinderFitterAnalyzer

- `RPStationTrackFitCollection`, by module label passed as parameter `"simTrack-sProducer"`
- `RPStationTrackFitCollection`, by module label passed as parameter `"recoTrack-sProducer"`
- `RPSRecognizedPatternsCollection`, by module label passed as parameter `"oneRP-PatternsProducer"`

8.1.2. Output events

Below are data objects produced by modules in `RPStationMultiTrackFinderFitter` and put into events. Obviously, this applies only to modules extending `EDProducer`.

CachedMultiTrackSource

- `HepMCProduct`

RPStationFitter

- `RPStationTrackFitCollection`

RPStationMultiTrackFinderFitter

- `RPStationTrackFitCollection`

8.1.3. Event Setups

`RPStationMultiTrackFinderFitter` and `RPStationFitter` use `RealGeometryRecord` retrieved from Event Setup for information about geometry.

8.2. Cooperating packages

This section describes modules from other packages used in simulations and thus being possibly useful in cooperation with modules from `RPStationMultiTrackFinderFitter`.

RPNonParallelTrackCandidateFinder

Located in `src/RecoTotemRP`. Gets the collection of RP hits and produces one RP hit patterns (U/V patterns) and track candidates. Gets `edm::DetSetVector<RPRecoHit>`, returns `RPTTrackCandidateCollection` and `RPTTrackCandidateCollection`. Uses Event Setup for `RealGeometryRecord`.

RPFastFullSimulation200

Simulation module which takes the `HepMCProduct` and returns `RPStationTrackFitCollection`. Gets `MisalignedGeometryRecord`, `BeamOpticsParamsRcd` and `ProtonTransportRcd` from Event Setup.

GeometryInfoModule

Located in `src/Geometry/TotemRPGeometryBuilder`. `EDAnalyzer`, which prints information about geometry based on Event Setup, from which it takes a few records, depending on settings.

9. Source code

This chapter consists of a description of the source code of the module.

9.1. Non-module classes

Each of `RPStationMultiTrackFinderFitter` and other modules mentioned in Section 6.2.2 has its class representation, deriving from one of the EDM base modules. All of their member definitions are placed in the `plugins` directory. There are also subsidiary classes, the files containing definitions of their members are kept inside the `src` directory. The `interface` directory contains header files for both aforementioned types of classes.

Those subsidiary classes are:

Algorithm

It is responsible for running the stages of reconstruction. Associated with `RPStationMultiTrackFinderFitter`, it combines the Strategy and Template Method design patterns. It can be derived from to provide reconstruction with another reconstruction algorithm. It has a public method, called `reconstruct`, which is a template method and uses field of type `IClusterBuilder` to build clusters (Strategy pattern) and then protected methods `selectClusters`, `filterClusters` and `removeOverlaps`.

```
void Algorithm::reconstruct(
    vector<Cluster> *output,
    const RPRemovedPatternsCollection *input) {
    // clusters
    vector<Cluster> clusters;
    clusterBuilder.build(clusters, *input);
    // rough selection of clusters
    vector<Cluster> selectedClusters;
    selectClusters(selectedClusters, clusters);
    // filter clusters
    vector<Cluster> filteredClusters;
    filterClusters(filteredClusters, selectedClusters, *input);
    ;
    // remove overlaps
    removeOverlaps(*output, filteredClusters, *input);
}
```

Cluster

A structure containing hits associated with a cluster along with uncertainties. Public members:

```
map<RPId, unsigned int> rpIdCount
    Map between roman pot ids and

set<RPRecoHit> hits
    Hits associated with the cluster

TMatrixD sum_W
    Inversion of uncertainty matrix for the center of cluster -  $W = V^{-1}$ 

TMatrixD sum_W_i
    Uncertainty matrix  $V$  of center of the cluster (mentioned as variance matrix in Section 4.2.1)

TVectorD sum_Wv
    Sum of  $W * v = V^{-1}$  for all tracks inside a cluster

TVectorD center
    The track being the result of  $\text{sum\_W\_i} * \text{sum\_WV}$ 
```

Public methods:

```
inline Track GetCenterTrack()const
    Converts the cluster to a track by returning center track's coordinates.

double Distance(const TVectorD &v, const TMatrixD &V_v, const TMatrixD &
    V_v_i)const
    Calculates the distance between the cluster and a track.  $v$  is the vector representing the track,  $V_v$  is an uncertainty matrix for that track,  $V_v_i$  is inverted  $V_v$ . Returns the distance between the cluster and the track.

void Add(RPId rpId1, RPId rpId2, const TVectorD &v, const TMatrixD &W)
    Adds a track represented by the vector  $v$  and uncertainty matrix  $W$  to the cluster and increments  $\text{rpIdCount}[\text{rpId1}]$  and  $\text{rpIdCount}[\text{rpId2}]$ .

Cluster& Copy(Cluster &copy)
    copies the cluster into  $\text{copy}$ 
```

IClusterBuilder

Abstract class with one public purely virtual method

```
virtual void build(
    vector<Cluster> &clusters,
    const RPRecognizedPatternsCollection &rpPatColl) = 0;
```

designed to take recognized patterns as `rpPatColl` and return clusters into parameter `clusters`, which serves as output.

ClusterBuilderIterable

A cluster builder which implements `IClusterBuilder` and selects clusters out

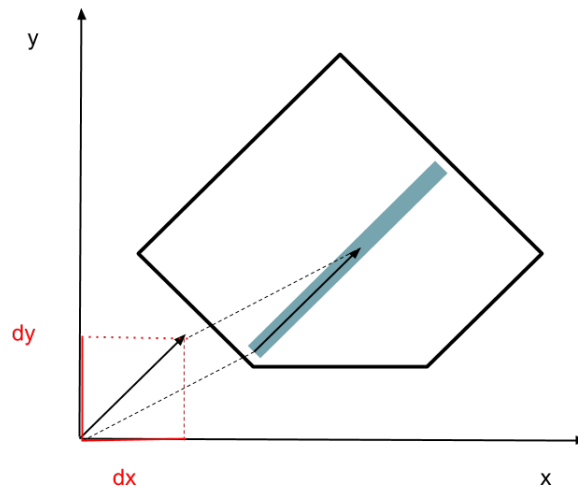


Figure 9.1: dx and dy taken from sensor readout direction.

of `RPrecognizedPatternsCollection` in a following way: it iterates through track candidates and decides whether to merge them with existing clusters. It is inherited from previous algorithm which dealt with only two Roman Pots and therefore deprecated.

ClusterBuilderSubsets

Also derives from `IClusterBuilder`, but takes another approach. At the beginning it creates a graph using method `populateGraph`, then finds the largest non inclusive cliques and finally transforms subset of graph vertices into clusters. It uses its own private structures to represent edges and track candidates.

DetInfo

A structure describing a single detector (sensor strip) of a Roman Pot. It has the following members fields:

`RPDetId detId`

ID of the detector

`double cx, cy, cz`

position of the Roman Pot's center

`double dx, dy`

Projection of the readout direction of the sensor onto x and y axes.

Fitter

Encapsulates fitting, i.e. refining the results with the least squares method, described in Section 4.2.4. It has three public methods:

`virtual void fitCollection(RPStationTrackFitCollection *rpTrColl, const vector<Cluster> *input)`

Takes a collections of clusters as `input`, computes the best fits using `fit` method and returns a collection of fits in `rpTrColl`.

```
virtual void fitCollection(RPStationTrackFitCollection *rpTrColl, const
    RPTrackCandidateCollection *input)
    Does the same as the method above, but taking RPTrackCandidateCollection as
    input instead of vector<Cluster>.
```

```
virtual void fit(RPStationTrackFit &out, const vector<RPRecoHit> &hits)
    Takes a vector of hits from Roman Pots, determines the best possible track fit using
    the least squares method and puts the result to out.
```

GeometryHelper

A helper for the geometry. Provides convenient view of the geometry from the perspective of track finding and fitting. It is written in a way that enables caching the returned information for later use. RPInfos and RPDetIds are created on demand and put inside maps. Public methods are:

```
inline void setGeometry(const TotemRPGeometry *geometry)
    Updates the geometry information, resets the cache of RPInfo objects.
```

```
virtual bool isPointOutside(RPId rpID, CLHEP::Hep3Vector point)
    Informs if a point represented by point (in global coordinates) lies within rpID's
    sensor.
```

```
virtual bool isPointOutsideEdge(RPId rpID, CLHEP::Hep3Vector point)
    Informs if a point represented by point (in global coordinates) lies outside rpID's
    sensor edge.
```

```
RPInfo getRPInfo(RPId RPId)
    Creates the proper RPInfo if it is not cached and returns it.
```

```
DetInfo getDetInfo(RPDetId detId)
    Creates the DetInfo based on detId if it is not cached and returns it.
```

OverlapsRemoval

Removes non-compatible clusters. Searches for a best combination of clusters within the full problem space. If there is more than a single solution the algorithm is unable to distinguish between the solutions and returns empty result. Solution is considered to be good if:

- each U/V pattern is considered with at least one chosen track
- number of overlaps (in terms of U/V patterns) between patterns is acceptable

The problem of overlaps is described more thoroughly in Section 4.2.3. The class contains a single public method:

```
void run(
    vector<Cluster> &out,
    const vector<Cluster> &in,
    const vector<unsigned int> &hitCount,
    const vector<vector<unsigned int> > &cluHits
);
```

These are the parameters:

out
 output clusters
in
 input clusters
hitCount
 number of U/V patterns in each plane (U and V in each Roman Pot)
cluHits
 for each cluster a vector of associated U/V patterns

RPInfo

A structure describing single Roman Pot. Its member fields are:

RPId **RPId**
 ID of the Roman Pot
double **cx, cy, cz**
 Position of the Roman Pot's center
double **u_dx, u_dy, v_dx, v_dy**
 Projection of the readout directions of U and V sensor directions onto x and y axis.
 Analogical to DetInfo's dx and dy.
CLHEP::Hep3Vector **edge_p, edge_n**
 Representation of the cut edge position, consisting of global position of the Roman Pot and normal vector pointing outwards of the edge.

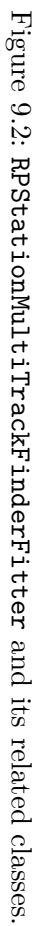
Track

Structure representing a single track. Keeps the track parameters and provides a public method **void Print(const string &prefix = "")const** for printing. A track given by functions $x(z) = a_x * (z - z_0) + b_x$ and $y(z) = a_y * (z - z_0) + b_y$ is described by its parameters, a_x, a_y, b_x, b_y and z_0 . This class keeps them as **ax, ay, bx, by, z0**, each of them of type **double**.

9.2. Structure of module classes

The classes which inherit from **EDProducer** (in this case **CachedMultiTrackSource**, **RPStationFitter** and **RPStationMultiTrackFinderFitter**) implement method **void produce(edm::Event&, const edm::EventSetup&);**, which takes events from the first argument and puts them back to it. The classes extending **EDAnalyzer**, like **RPStationMultiTrackFinderFitterAnalyzer** implement method **void analyze(const edm::Event&, const edm::EventSetup&);**. The difference is that it doesn't modify the incoming event, but instead just performs the analysis and passes the outcome in some other way. **RPMultiTrackFilter**, which inherits from **EDFilter**, must implement **virtual bool filter(edm::Event&, const edm::EventSetup &)** which returns a value indicating if an event can be passed further.

Although **RPStationMultiTrackFinderFitter** is not a complicated class in terms of volume, it interacts with the largest number of helper classes, which can be seen on Figure 9.2.



RPStationMultiTrackFinderFitterAnalyzer has a wide variety of private methods and fields in order to provide complex analysis using one- and two-dimensional histograms and files to write output into.

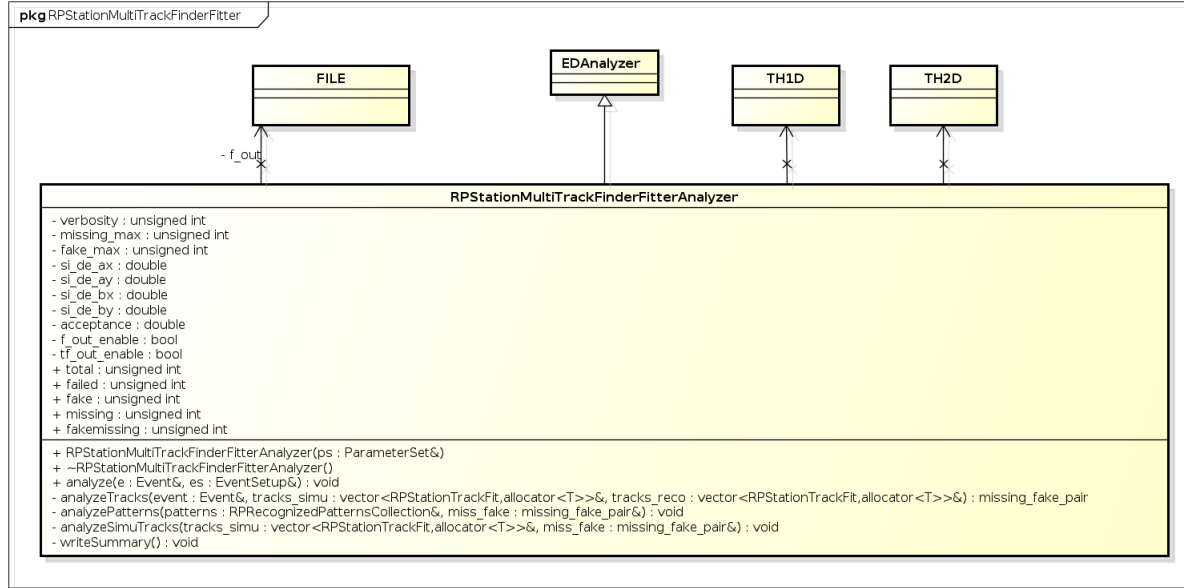


Figure 9.3: **RPMultiTrackFinderFitterAnalyzer** and its related classes.

CachedMultiTrackSource operates on files and makes use of **GSLRandomEngine**, a random number engine with the purpose of selecting a track from random source with a given probability.

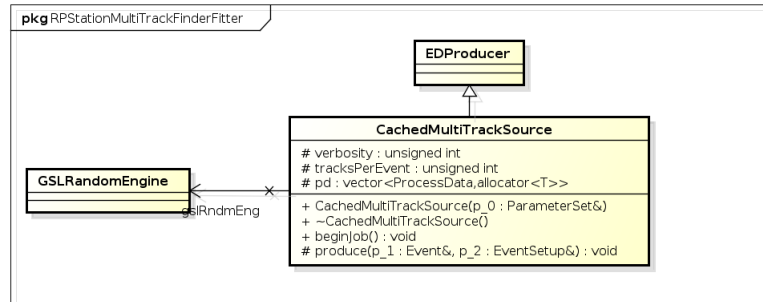


Figure 9.4: **CachedMultiTrackSource** and its related classes.

RPMultiTrackFilter uses only the help of **InputTag**, a class which helps to fetch data from incoming event by a label.

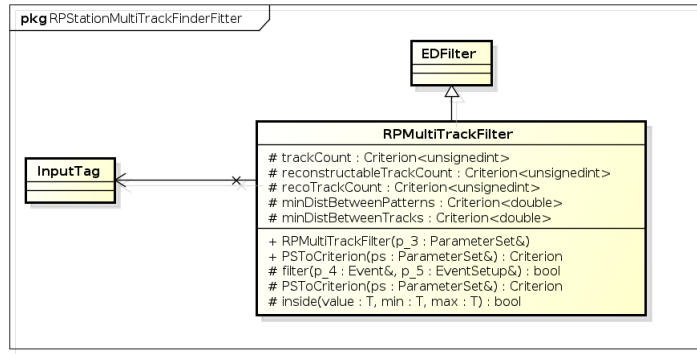


Figure 9.5: RPSMultiTrackFilter and its related classes.

Since among all the parts of algorithm, `RPStationFitter`'s responsibility is only fitting, it is associated only with `Fitter` and `GeometryHelper`.

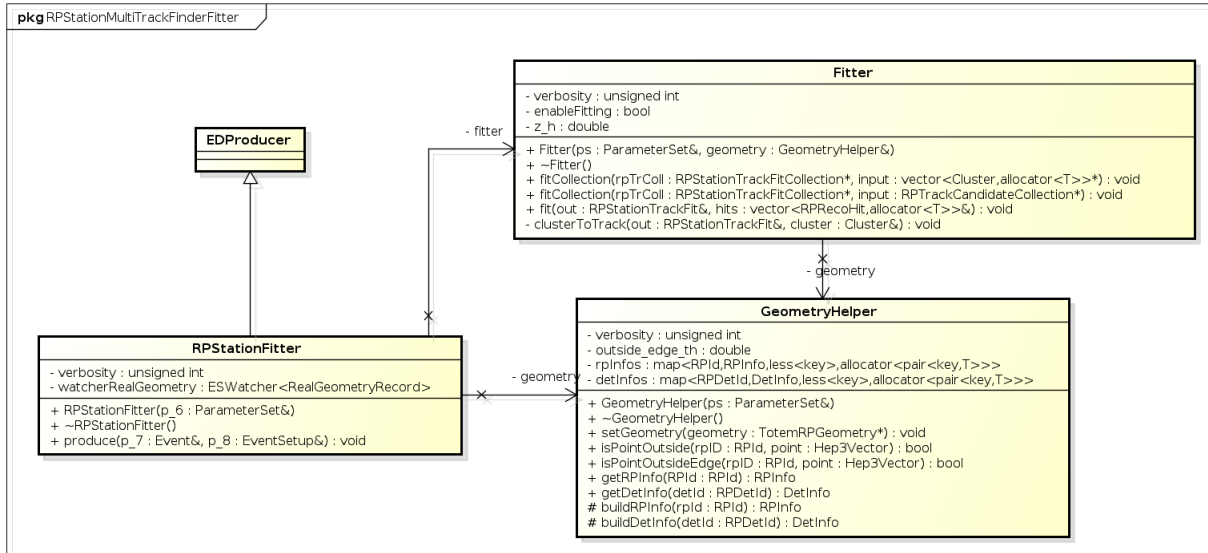


Figure 9.6: RPStationFitter and its related classes.

9.3. Framework classes

This section describes associations between classes from `RPStationMultiTrackFinderFitter` package to classes outside of it.

9.3.1. Roman Pot data formats

Data about tracks, fits, recognized patterns, etc. are kept inside classes from `RecoTotemRP/RPRecoDataFromats` and `DataFormats/TotemData`. This includes:

- `RPId`
- `RPStationId`
- `RPDetId`

- `RPSRecognizedPatternsCollection`
- `RPSStationTrackFitCollection`

There is also `RealGeometryRecord` from `Geometry/TotemRecords` which stores information about Roman Pot geometry. The given file paths refer to a location inside extended Totem version of CMSSW framework, which was mentioned in Chapter 6.

9.3.2. CMSSW

All classes connected with Event Data Model, such as events, module parameters are placed in original (not extended) CMSSW framework, mainly at location `FWCore/Framework`.

9.3.3. ROOT

ROOT is an analysis framework commonly used in high-energy physics. This project uses several of its C++ classes for storing data like matrices and vectors (classes `TMatrixD` and `TVectorD`), outputting data to special type of files with extension '.root' (class `TFile`) and preparing histograms (`TH1D` and `TH2D`) which later can be analyzed with other ROOT tools (<https://root.cern.ch/>).

Bibliography

- [1] The TOTEM Collaboration. Totem upgrade proposal, cern-lhcc-2013-009 / lhcc-p-007. Technical report, TOTEM, CERN, 2013.
- [2] Katy Foraz Frédérick Bordry. Long shutdown 1 2013-2014. Technical report, CERN, 2012.
- [3] Jan Kašpar. *Elastic scattering at the LHC*. PhD thesis, Institute of Particle and Nuclear Physics, Charles University in Prague, 2011.
- [4] Philippe Lebrun. Interim summary report on the analysis of the 19 september 2008 incident at the lh. Technical report, CERN, 2008.
- [5] Hubert Niewiadomski. *Reconstruction of protons in the TOTEM Roman Pot detectors at the LHC*. PhD thesis, The University of Manchester, 2008.
- [6] Cmssw 7.0.4. <https://twiki.cern.ch/twiki/bin/view/CMSPublic/WorkBook>.