LHCb Note 2006-057
TRACKING

Dealing with Clones in the Tracking

E. Rodrigues

NIKHEF, Amsterdam, The Netherlands

November 2006

Abstract

The note describes the way clone tracks are found and eliminated in the LHCb tracking. Both the "clone killer" algorithm and the related "clone finder" tool are presented. The performance of the algorithm as it is used at present in Brunel is also discussed.

Contents

List of Tables

List of Figures

1 Introduction

In LHCb the tracking pattern recognition sequence comprises a series of algorithms. At the end, we are left with several sets of different types of tracks: `Velo` tracks, `Long` tracks, etc. Some of these pattern recognition algorithms use information output by others in order to "build" more complete tracks. For example, the matching algorithm tries to match in phase space the `Velo` tracks with the `Ttrack` (seed) tracks from the downstream tracking stations. Other algorithms use different techniques for finding similar tracks - e.g. the forward and matching algorithms. As a result, it may happen that the set of tracks made at the end of the pattern recognition sequence contain in fact clone tracks. The main purpose of the "clone killer" algorithm described in this note is to first find all the clone tracks and to flag them.

This note is divided as follows: in the next paragraph the clone killer algorithm is presented, as well as the possible options it possesses and the output it produces. Then the heart of the finding strategy, the "clone finder" tool, is described. Next it is shown how to use the algorithm and the tool. At last, a thorough discussion of how clones are actually dealt with at present, in Brunel, is given, as well as the present performance.

1.1 LHCb Definition of Clone Tracks

Broadly speaking, a track is a clone track if it is a sub-track or a copy of another track. More precisely, two tracks are considered clones of each other if they share at least 70% of the hits in the VELO and at least 70% of the hits in the T-stations seeding region. Note that the present LHCb definition does not take into consideration hits in the Trigger Tracker (TT) stations.

2 The Clone Killer Algorithm

The clone killer algorithm is run at the end of the tracking sequence. Its purpose is two-fold:

- identify clone tracks among the different containers of tracks output by the several pattern recognition algorithms, and flag those found as clones;
- provide at the end of the tracking sequence a set of "best" tracks that are to be used for physics analyses. These unique tracks are consequently stored in the so-called "best" container.

The actual algorithm is called `TrackEventCloneKiller`; it is part of the `Tr/TrackUtils` package [1].

The default logic for finding and flagging the clones proceeds as follows:

1. collect all the valid input tracks;
2. remove all ancestors of tracks from the list of input tracks;
3. loop over the remaining tracks, to find and flag the clones among them;
4. store the unique tracks into the "best" container.

This logic corresponds to the default options as used in Brunel. The algorithm is in fact rather configurable. We postpone to the next section, section 2.1, a discussion of the variants in which `TrackEventCloneKiller` can be run.

All valid tracks are accepted as input, the selection being based on the track's general flag `Track::Invalid` [1]. As a consequence, all tracks that were not fitted successfully are discarded. Still, it is possible to consider, for some special reason, non-fitted tracks, but this requires the flag `Track::Invalid` to be set to `false` "manually" after the fit.

As a "pre-cleaning", all input tracks are searched for ancestor tracks [2], since ancestors are, by construction a "sub-track" and therefore a clone. Ancestors are flagged as clones and discarded thereafter. This is particularly useful for removing at the earliest stage those tracks that will anyway be found as clones of their "parent track". It also significantly reduces the number of combinatorics to be considered in the track-by-track comparison.

Finally, all remaining input tracks are looped over. The algorithm calls internally the helper tool `TrackCloneFinder` to actually determine the clone tracks, setting the clones with the `Track::Clone` flag to `true`. All capabilities of this tool are described in section 3.

2.1 Options

The `TrackEventCloneKiller` algorithm possesses a certain number of options that control the way the algorithm is run and in particular what is put into the output container. The list of options is given in table 1 together with their default values.

The `TracksInContainers` option specifies the list of (paths for the) input tracks. The algorithm is configured by default to take all the tracks containers output by the pattern recognition algorithms.

At the end the algorithm outputs the unique tracks to the "best" container, the standard container of tracks as given by the tracking to the rest of the reconstruction. One can also change this path via the `TracksOutContainer` option.

[1] All track flags are defined in the `Track` class in the `Event/TrackEvent` package [2].

[2] Ancestors are tracks that have been used in a pattern recognition algorithm to make a new track, typically adding hits. As an example, the matching algorithm links `Velo` and `Ttrack` tracks to make `Long` tracks; and it also adds compatible hits from the TT stations.

TrackEventCloneKiller options		
Option name	Description	Default value
TracksInContainers	Paths to input tracks containers	{ TrackLocation::VeloTT TrackLocation::Forward, TrackLocation::Match, TrackLocation::Tsa, TrackLocation::KsTrack}
TracksOutContainer	Path to output tracks container	TrackLocation::default
IgnoredTrackTypes	List of track types to ignore. Can be used so that certain tracks are considered for finding clones but are not output to the `TracksOutContainer`	{ }
StoreCloneTracks	Decide whether clone tracks are also output	false
SkipSameContainerTracks	Decide whether to skip clone comparison between tracks of the same container	true
CloneFinderTool	Name of the clone finder tool	"TrackCloneFinder"

Table 1: Set of user-definable options of the TrackEventCloneKiller algorithm. The three columns give the name of the option, a short description, and the default value.

One can also decide not to output some track types giving a list to the `IgnoredTrackTypes` property. At first this possibility may seem redundant with `TracksInContainers`, and therefore irrelevant. But, in fact, it is handy: even when the user is not interested in outputting tracks of a certain type, he probably still needs to input those tracks so that the clone finder tool considers all possibilities of clones.

For special studies it may be needed to output the unique as well as the clone tracks; this is done setting the `StoreCloneTracks` property to `true`.

The algorithm assumes, understandably, that the container of tracks produced by a pattern recognition contains no clones. It then skips such a clones check. But the user still has the possibility to check for clones in tracks belonging to the same container setting the `SkipSameContainerTracks` property to false.

The last option in table 1, `CloneFinderTool`, specifies the tool for comparing pairs of tracks and flagging possible clones. This makes it trivial – via a simple job option – to test another clone finder tool.

The properties just described are the ones specific to the `TrackEventCloneKiller` algorithm. To these are added the properties inherited from the base class algorithm. In debug mode, when one sets

```
TrackEventCloneKiller.OutputLevel = 2;
```

some debugging information is given: size of all the input containers, total number of tracks to be considered, and number of clones found and flagged.

3 The Clone Finder Tool

As said above in section 2, the clone killer algorithm `TrackEventCloneKiller` internally calls a helper tool, effectively delegating the finding of clones. The actual search for clone tracks is a pair-wise comparison done by the clone finder tool `TrackCloneFinder`; we hereafter describe it.

The clone finder tool implements the interface of the `ITrackCloneFinder` interface, defined in `Tr/TrackInterfaces` [3]. The tool can be found in the `Tr/TrackTools` package [4].

As far as the pattern recognition goes, a track is characterized in particular by its "contents", that is by the set of detector hits that composes it. A realistic pattern recognition is in general not perfect and fully efficient; it may happen that several tracks share one or several hits. This fact is exploited in the clone finder tool `TrackCloneFinder` for finding clone tracks: put simply, the clone finder tool compares the shared hits on pairs of tracks.

The tool implements at present a single method:

```
/** Compare two input Tracks and find whether one is a clone
 *  of the other based on some "overlap criteria".
 *  The corresponding flag may be set accordingly (NOT DONE BY DEFAULT)
 *  depending on the value of the "setFlag" argument.
 *  Note: the method ignores whether the Tracks themselves have been
 *        previously flagged as clones! It merely does a comparison.
 *  @return bool: True if one Track is a clone of the other.
 *                False otherwise.
 *  @param  track1 input 1st track
 *  @param  track2 input 2nd track
 *  @param  setFlag input parameter indicates whether the clone track
 *          is to be set as such (default = false)
 */
virtual bool areClones( LHCb::Track& track1,
                        LHCb::Track& track2,
                        bool setFlag = false ) const;
```

The heart of the clone finding logic can be best summarised in pseudo-code:

```
IF     ( tracks share hits in the VELO )
   AND ( number of common hits < matching-fraction ): tracks are not clones

IF     ( tracks share hits in the T-seeding stations )
   AND ( number of common hits < matching-fraction ): tracks are not clones

IF ( tracks have no common region in VELO and T-seeding stations ):
   tracks are not clones

ELSE: tracks are clones
```

The matching-fraction above refers to the standard 70% hits-sharing criteria widely used in LHCb (*c.f.* the definition in section 1.1).

When two tracks are found to be clones of each other, one still has to decide which one will be flagged as the clone [3]. The track with less hits will always be flagged as the clone track; this is the most common case. When the situation is such that the number of hits is the same for both tracks, then the first input track of the pair being considered is always flagged as the clone. Note that a same number of hits does not imply that all hits are the same, of course, and that the two tracks are an exact copy of each other.

This may seem rather far from optimal, but is also the simplest way to choose given that the overlap criteria is anyway solely based on the number of hits. One possible improvement would be to then flag using extra information on the tracks, in particular the χ^2 from the fit. Such studies are foreseen, and a brief discussion is given at the end, in section 6.

This logic takes particular care that, for instance, the e^+e^- pairs from photon conversions are not seen as clones when the conversion happens in the VELO detector: in such a situation, the electron and positron may share hits in the VELO region, but their other hits in the seeding stations make the two tracks distinct. Also `Velo` and `Ttrack` tracks are, by definition, not clones, etc.

Hits were here taken as a general term that can actually mean `LHCbIDs` or `Measurements` in the context of the LHCb track event model. In the default behaviour, the `TrackCloneFinder` tool is set to compare the `Measurements` on the tracks and calculate their overlap (section 3.1 further discusses how to instead compare the overlap of the `LHCbIDs`).

Comparing `Measurements` or `LHCbIDs` can make a (expectedly very small) difference, and it is important to realise this. The difference stems from the fact that `LHCbIDs` are a non-mutable property of a track as set by a pattern recognition algorithm when the track is

[3] Note that it is also possible to call the `TrackCloneFinder` tool to merely check whether two tracks are clones of each other, without the `Track::Clone` being set on the clone. Please refer to the methods of the tool for usage, see [4].

first created, whereas Measurements aren't: when the track is fitted, the Measurements are loaded into the track, starting from the list of LHCbIDs, and so-called outlier hits may be removed during the fit. At the end, a track can be found – it is in fact often the case – with less Measurements than LHCbIDs, the difference corresponding precisely to the outlier Measurements removed. In other words, the search for clones is done at the level of "fitted tracks" (with Measurements) rather than at the level of "pattern recognition tracks" (with only LHCbIDs).

TrackCloneFinder options		
Option name	Description	Default value
MatchingFraction	Percentage of matching hits for clone tracks	70%
CompareAtLHCbIDsLevel	Compare LHCbIDs or Measurements	false

Table 2: Set of user-definable options of the TrackCloneFinder tool. The three columns give the name of the option, a short description, and the default value.

3.1 Options

The behaviour of the TrackCloneFinder tool can be steered with two properties that control what the matching criteria is. Table 2 defines these two options and their default values.

As stated in 1.1, two tracks are considered clones of each other only if they share in both the VELO and T-seeding regions at least 70% of hits. It is important to note that the matching is done independently for VELO and seeding hits. This matching percentage is widely used and agreed upon in LHCb. For dedicated studies, it is possible to change this value with the MatchingFraction job option.

The TrackCloneFinder tool compares by default the number of shared Measurements, therefore considering fitted tracks (see section 3). This behaviour can be changed via the CompareAtLHCbIDsLevel property, and may be handy to investigate, for example, if track fitting has an influence on the flagging of clones (since the matching will then consider LHCbIDs instead of Measurements).

In debug mode, when one sets, for instance,

```
ToolSvc.TrackCloneFinder.OutputLevel = 2;
```

a long output is produced; it contains information on the containers and keys of both input tracks, the list of the LHCbIDs and whether they were found to be clones of each other.

4 Using the Clone Finder and Killer

In this section we now focus on the possible applications of both the clone finder tool and the clone killer algorithm, illustrating a couple of handy examples.

4.1 Clone Finding and Killing in Brunel

The `TrackEventCloneKiller` algorithm runs with all default options at the end of the tracking sequence in Brunel. The actual sequence is defined in the `RecoTracking.opts` file in the `Tr/TrackSys` package [5]. The relevant part is simply (copied from `RecoTracking.opts`)

```
RecoTrSeq.Members += { "TrackEventCloneKiller" };
```

The clone killer runs over all track types produced by the various algorithms except the `Velo` tracks (*c.f.* the `TracksInContainers` flag defined in table 1). A sub-sample of these `Velo` tracks is then fitted [4], after removal of those that have been "used" by other pattern recognition algorithms and are therefore, and by construction, ancestors. This concerns the `Long` tracks produced by `PatForward` and `TrackMatching`. The sub-sample of unused and fitted `Velo` tracks is finally added to the "best" container. The tracking sequence ends.

4.2 Standalone Usage

We give below a few examples of what can be done with the "clone killing set" for user-specific studies.

4.2.1 Usage via Options

We've shown in the previous sections how flexible the `TrackEventCloneKiller` algorithm and the related `TrackCloneFinder` tool are. Assume, for the sake of example, that we are only interested in producing a container of unique `Long` tracks, and want to check for clones even in the tracks from the same container. Since the latter are produced only by the `PatForward` and `TrackMatching` algorithms, the following options suffice:

[4] All `Velo` tracks are fitted with the same momentum hypothesis – $p_T = 400$ MeV.

```
RecoTrSeq.Members += { "TrackEventCloneKiller/LongTracksCloneKiller" };

LongTracksCloneKiller.TracksInContainers      = { "Rec/Track/Forward"
                                                  , "Rec/Track/Match" };
LongTracksCloneKiller.TracksOutContainer      = "Rec/Track/MyLong";
LongTracksCloneKiller.SkipSameContainerTracks = false;
```

4.2.2 Usage in Python

All the tracking classes and tools have been exposed to Python with the help of the `Tr/TrackPython` package [6], and are accessible via the `gtracktools.py` Python module. Full details are to be found in the note [7]. We here restrict the examples to the functionality in `gtracktools.py` relevant to this note.

Let us assume that we have two tracks to be compared, and that we are running a Python job with the necessary setup already done (libraries and dictionaries have been loaded, at least one event has been run, etc.). The `TrackCloneFinder` tool can be retrieved very simply with [5]

```
>>> from gtracktools import TrackCloneFinder
```

Checking whether they are clones is as simple as

```
>>> TrackCloneFinder.areClones( track0, track1 ) == True
False
```

This elementary manipulation can be exploited in more sophisticated studies. Supposing we want to look interactively at the clone tracks produced by the two `Long` tracks pattern recognition algorithms. Best is then to define a function that can be called for each event:

[5] This syntax is simpler compared to the one described in [7], and reflects improvements and simplifications made recently to `Tr/TrackPython`.

```
>>> EVT = appMgr.evtSvc()
>>> from gtracktools import TrackCloneFinder
>>> def check_long_clones():
>>>   forwards = EVT[ '/Event/Rec/Track/Forward' ]
>>>   matches  = EVT[ '/Event/Rec/Track/Match' ]
>>>   nclones  = 0
>>>   for t1 in forwards:
>>>     for t2 in matches:
>>>       areClones = TrackCloneFinder.areClones( t1, t2 )
>>>       if areClones:
>>>         nclones += 1
>>>       print 'Forward, Match: keys', t1.key(), ',', t2.key(),\
            '\t areClones =', areClones
>>>   print 'found', nclones, 'clones in', forwards.size(),\
          ' Forward tracks and', matches.size(), 'Match tracks'
```

5 Study of Clones in Brunel

In this final section we collect the results of studying the present content of clone tracks as seen at the end of the tracking in Brunel. We have used Brunel v30r12 together with the corresponding tracking packages. All distributions presented hereafter in the note relate to $\mathrm{B}^0_\mathrm{d} \to \mathrm{J}/\psi(\mu^+\mu^-)\mathrm{K}^0_\mathrm{S}(\pi^+\pi^-)$ signal events.

Figure 1 shows the distribution of the total number of tracks produced by all the pattern recognition algorithms run in Brunel. From an average number of about 235 tracks we are left, after identification and removal of clones, with an average of roughly 105 "unique" tracks (`c.f.` figure 2) in the "best" container.

The next figures, from 3 to 8, present the distributions of the number of tracks produced by the existing pattern recognition algorithms. The corresponding figures 9 to 14 show the same distributions for only those tracks flagged as clones by the clone killer algorithm. All the average values have been collected in table 3, for an easy read and comparison.

The distributions of the fraction of `Long` tracks flagged as clones (figures 11 and 12) show a particular peak at 0.5; this simply indicates that there is a large probability of half of the tracks produced by `PatForward` and `TrackMatching` to be found as clones; which in turn happens mostly when the number of `Long` clones is equal to the number of tracks produced by either `PatForward` or `TrackMatching`.

As pointed out frequently, the percentage of clones of all types hints at some redundancy in our tracking strategy; nevertheless, one should not mis-interpret the average numbers

Pattern Recognition	Output Container	Number of Tracks	Percentage of Clones
PatVelo	TrackLocation::Velo	70	55%
PatVeloTT	TrackLocation::VeloTT	27.5	75%
PatForward	TrackLocation::Forward	30	36%
TrackMatching	TrackLocation::Match	26	47%
TsaAlgorithms	TrackLocation::Tsa	59	63%
PatKShort	TrackLocation::KsTrack	34	60%
Sum over all	-	235	-
After Clone Killing	TrackLocation::Best	105	0

Table 3: Average numbers of tracks produced by the pattern recognition algorithms and average number of tracks flagged as clones by the clone killer, as taken from figures 1 to 8 and figures 9 to 14, respectively.

from table 3. As an example, consider the `Upstream` tracks from `PatVeloTT`: it may seem at first that this pattern recognition is largely unnecessary, since about 75% of the tracks produced are in fact clones, but one needs to realise that a large fraction of these are indeed "sub-tracks" of `Long` tracks found by `PatForward` and/or `TrackMatching`. The remaining are most probably low-momentum tracks that do not make it to the T-seeding stations – and that is the very reason for an upstream tracking.

A similar argument can be given for `Ttrack` tracks produced by `TsaAlgorithms`: 63% of them are flagged as clones by the clone killer. A straightforward check indicates that these correspond to "sub-tracks" of either `Long` or `Downstream` (output from `PatKShort`) tracks. In fact, both the downstream and the matching algorithms have as ancestors `Ttrack` tracks. Also `Downstream` tracks can be "sub-tracks" of `Long` tracks.

It is worth discussing further the special case of `Long` tracks. Being found by two different algorithms – `PatForward` and `TrackMatching` –, it is natural to expect some redundancy. In fact, it turns out, as can be concluded from figure 15, that on average we produce approximately 88% of clone `Long` tracks; note that we define this ratio as
pairs of `Long` clones / MIN(# `PatForward`, # `TrackMatching`). This number cannot be directly compared with the average numbers of flagged clones in the `Long` tracks pattern recognition algorithms: as given in table 3, on average 36% of "`PatForward`" tracks are flagged as clones, whereas the percentage is roughly half for "`TrackMatching`" tracks. The different numbers have to do with the way we flag clones – based on the number of `Measurements`. This means that there is a mix of "`PatForward`" and "`TrackMatching`" tracks being flagged on an event-by-event basis. The higher percentage for "`TrackMatching`" clone tracks is easily understood given that the "`PatForward`" track found as clone of a "`TrackMatching`" track has more often a higher number of `Measurements`, and so the

"TrackMatching" is the one flagged as clone of the "PatForward" track (figure 16). One can also conclude from figure 16 that most of the pairs of Long clones have the same number of Measurements.

6 Outlook and Final Remarks

The LHCb tracking strategy consists of several pattern recognition algorithms, some of which have by construction some degree of redundancy. Whereas this approach improves the track finding efficiency, it also produces some clone tracks. This note described how clones are dealt with in our tracking after the pattern recognition. It presented the software tools at hand, and their functionality, with useful examples. We have also proved that the identification of clones is performing well.

It was pointed out the need to maybe go beyong the simple way in which it is at the moment decided which is the clone track of a pair. At least two issues are at hand:

1. is it indeed "best" to flag as the clone the one track that has less hits on it?
2. when both clone tracks have the same number of hits, what would be the "best" decision criteria?

This second point is most relevant for the case of Long track clones: on average some 30% of the pairs of Long clones ("PatForward", "TrackMatching") have the same number of Measurements (figure 17).

In section 3 we have already mentioned that a possible alternative would be to decide based on extra information on the tracks, in particular based on the χ^2 per degree-of-freedom from the fit. But other quality criteria could be studied as well.

Clearly several scenarios are possible, and need to be investigated and compared. Obvious quantities to be looked at are:

- hit purity and efficiency;
- resolutions;
- ghost rate.

This just translates the fact that one should try and have a clone flagging strategy that leaves us with non-ghost tracks with good resolutions and a minimal number of wrong hits and outliers [6].

[6] Clone killing can also remove ghosts, either clones of ghost tracks or genuine ghosts. The latter situation may seem strange, but can happen: say, two tracks are clones of each other. The one with more hits has more that 70% of its hits associated with a Monte Carlo particle, and the other doesn't. Then the second track, with less hits, is flagged as clone and removed, and is actually a ghost.

Any future study towards a new clone flagging strategy should take these points into account.

Finally, it would also be important to consider clone killing from a larger perspective, in the framework of the whole tracking sequence in Brunel, taking into account the CPU budget. Given the studies presented in the previous section, I would tend to conclude, just as an example, that it would be more efficient to eliminate, at the earliest stage possible, the clones among the samples of `Long` tracks produced by "`PatForward`" and "`TrackMatching`". This could in fact be done before fitting these tracks, and would roughly reduce the `Long` tracks fitting time by a factor of almost two.

References

[1] *http://isscvs.cern.ch/cgi-bin/cvsweb.cgi/Tr/TrackUtils/?cvsroot=lhcb*

[2] *http://isscvs.cern.ch/cgi-bin/cvsweb.cgi/Event/TrackEvent/?cvsroot=lhcb*

[3] *http://isscvs.cern.ch/cgi-bin/cvsweb.cgi/Tr/TrackInterfaces/?cvsroot=lhcb*

[4] *http://isscvs.cern.ch/cgi-bin/cvsweb.cgi/Tr/TrackTools/?cvsroot=lhcb*

[5] *http://isscvs.cern.ch/cgi-bin/cvsweb.cgi/Tr/TrackSys/?cvsroot=lhcb*

[6] *http://isscvs.cern.ch/cgi-bin/cvsweb.cgi/Tr/TrackPython/?cvsroot=lhcb*

[7] E. Rodrigues, *Tracking in Python*, LHCb Tracking/Software Note LHCb 2006-014

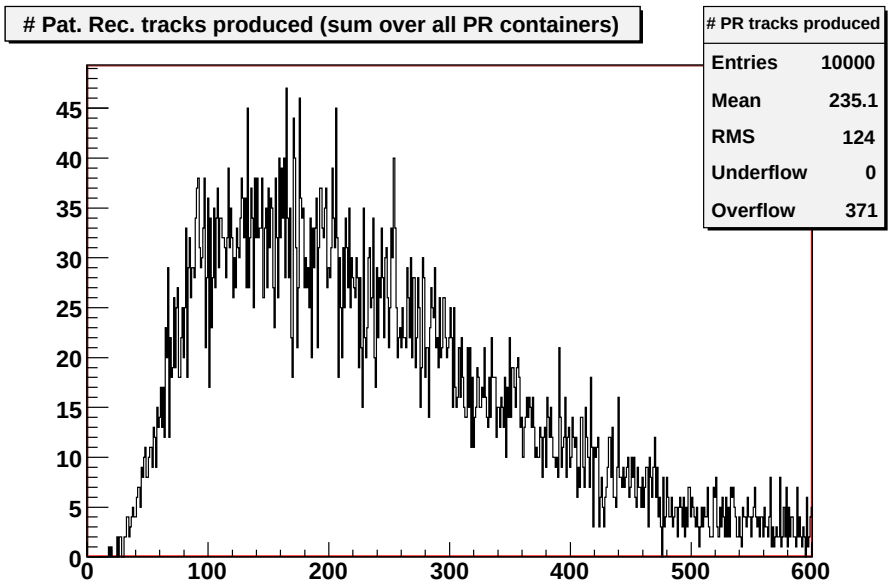

Figure 1: Distribution of the total number of tracks produced by all the pattern recognition algorithms run in Brunel.

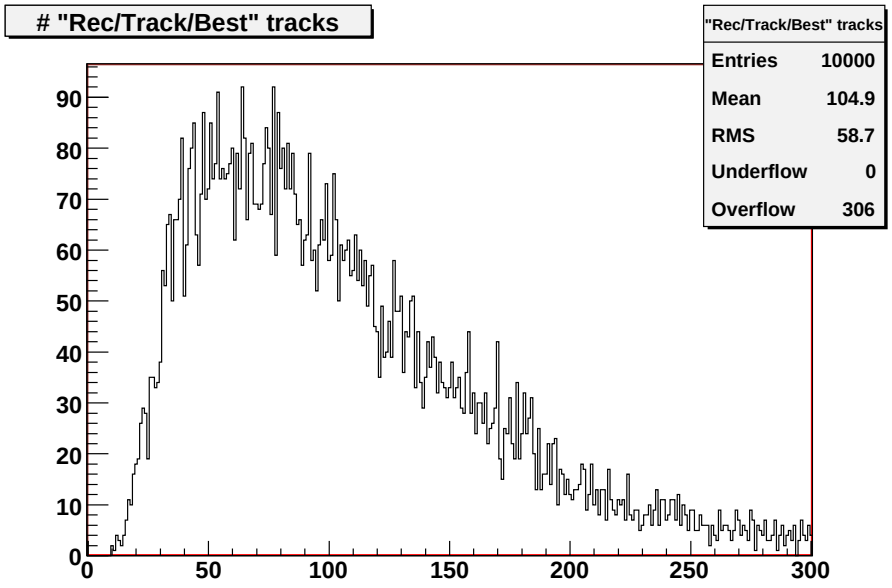

Figure 2: Distribution of the number of tracks in the so-called "best" container, *i.e.*, output by the tracking sequence of Brunel after the removal of clones and fit-failed tracks.

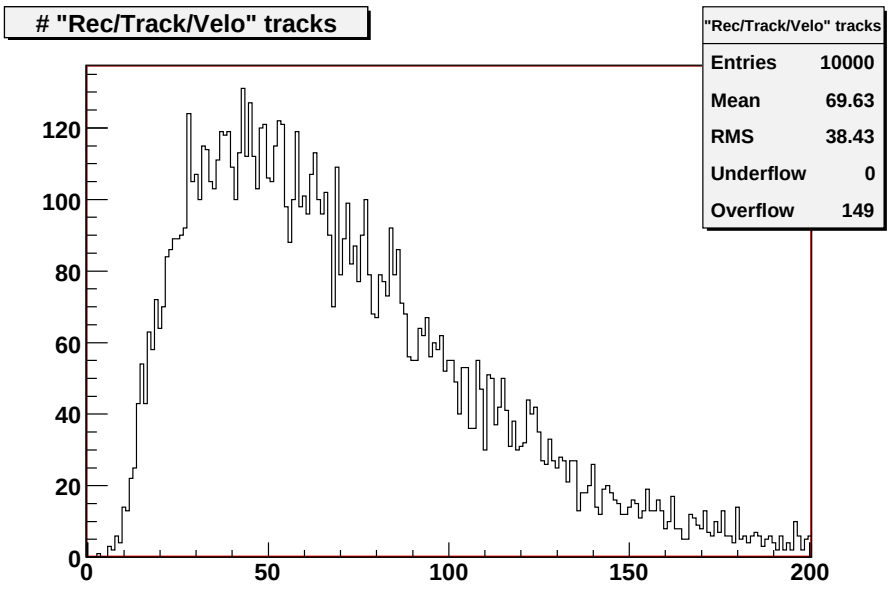

Figure 3: Distribution of the number of `Velo` tracks produced by the `PatVeloSpaceTracking` pattern recognition algorithm (`Pat/PatVelo` package).

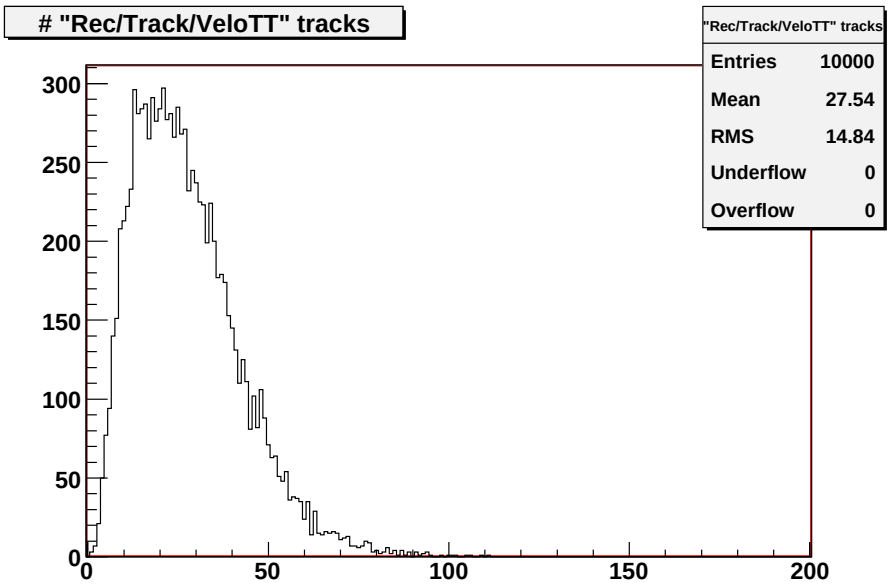

Figure 4: Distribution of the number of `Upstream` tracks produced by the `PatVeloTT` pattern recognition algorithm (`Pat/PatVeloTT` package).

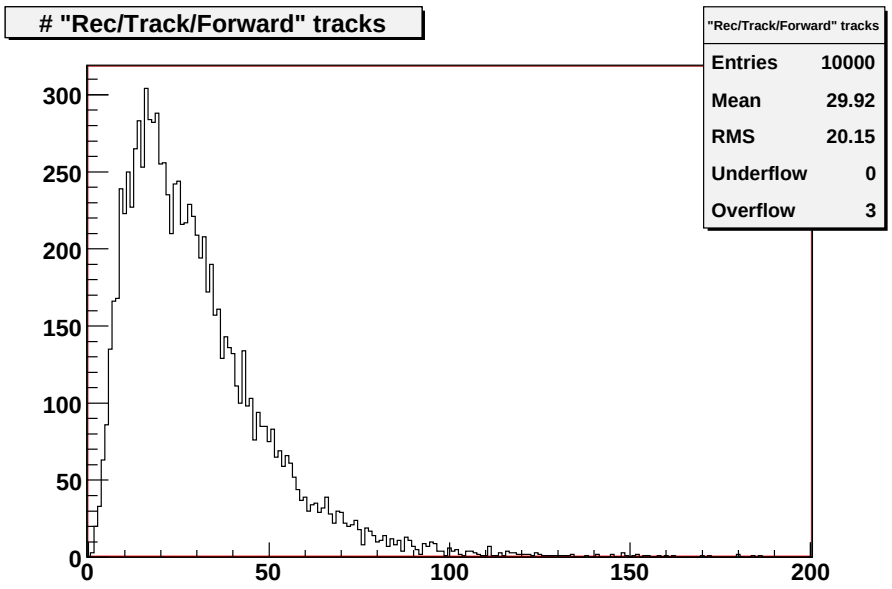

Figure 5: Distribution of the number of `Long` tracks produced by the `Patforward` patternumber of n recognition algorithm (`Pat/PatForward` package).

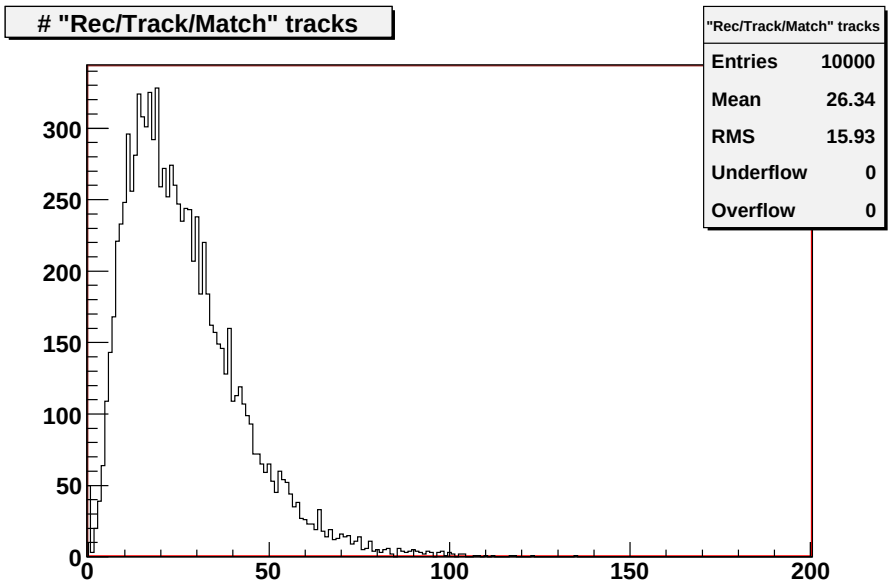

Figure 6: Distribution of the number of `Long` tracks produced by the `TrackMatckVeloSeed` pattern recognition algorithm (`Tr/TrackMatching` package).

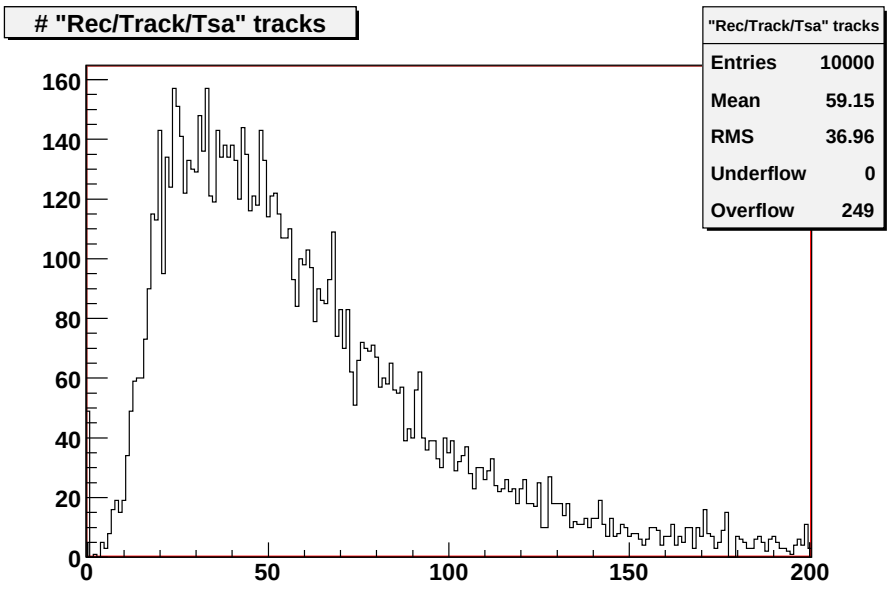

Figure 7: Distribution of the number of `Ttrack` tracks produced by the `TsaSeed` pattern recognition algorithm (`Tr/TsaAlgorithms` package).

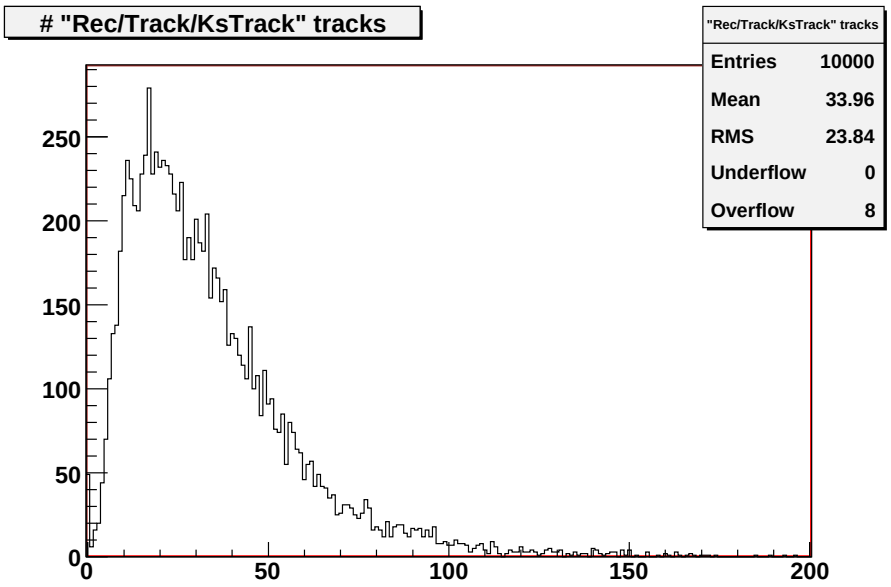

Figure 8: Distribution of the number of `Downstream` tracks produced by the `PatKShort` pattern recognition algorithm (`Pat/PatKShort` package).

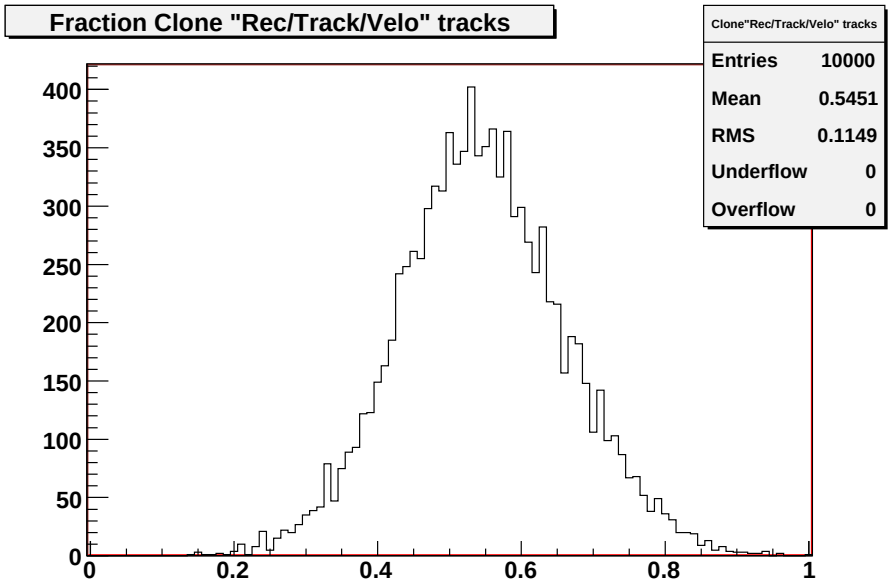

Figure 9: Distribution of the fraction of `Velo` tracks produced by the `PatVeloSpaceTracking` pattern recognition algorithm (`Pat/PatVelo` package) and flagged as clones by the clone killer algorithm.

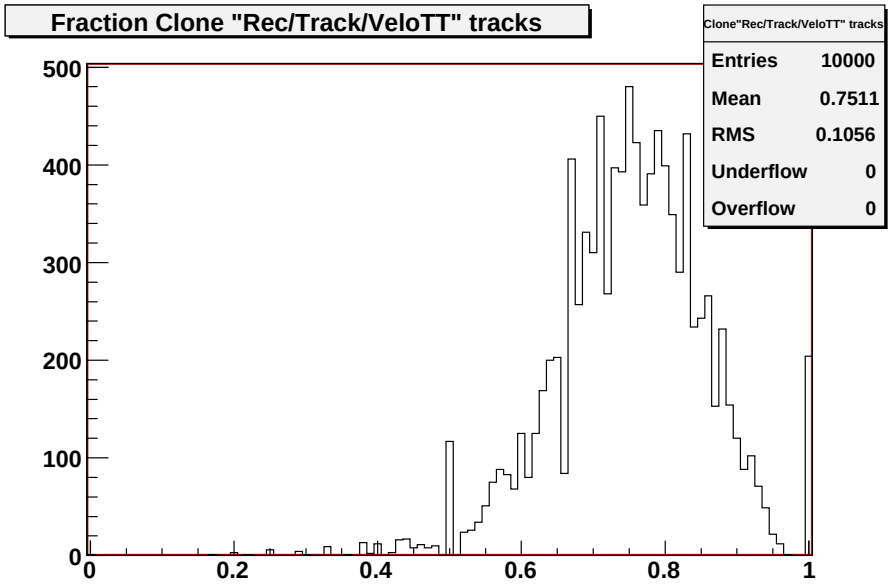

Figure 10: Distribution of the fraction of `Upstream` tracks produced by the `PatVeloTT` pattern recognition algorithm (`Pat/PatVeloTT` package) and flagged as clones by the clone killer algorithm.

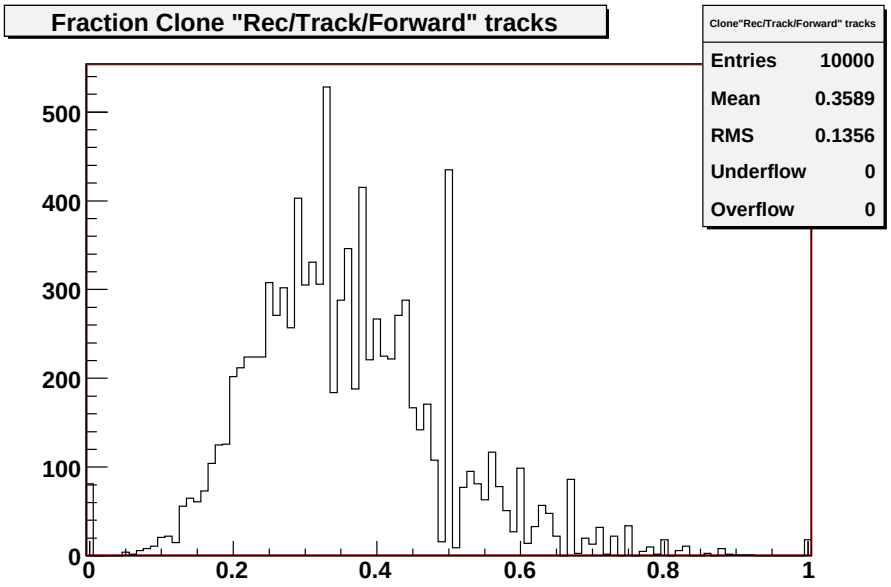

Figure 11: Distribution of the fraction of `Long` tracks produced by the `Patforward` pattern recognition algorithm (`Pat/PatForward` package) and flagged as clones by the clone killer algorithm.

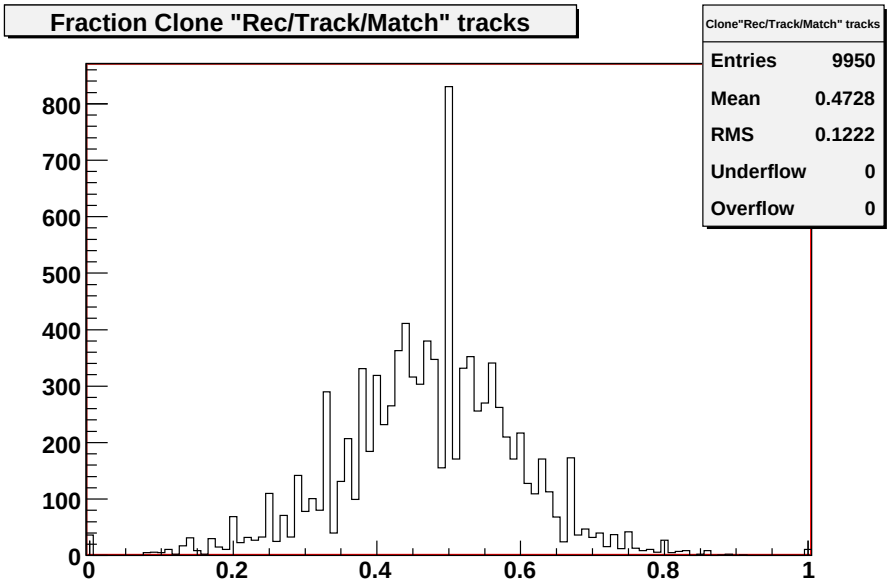

Figure 12: Distribution of the fraction of `Long` tracks produced by the `TrackMatckVeloSeed` pattern recognition algorithm (`Tr/TrackMatching` package) and flagged as clones by the clone killer algorithm.

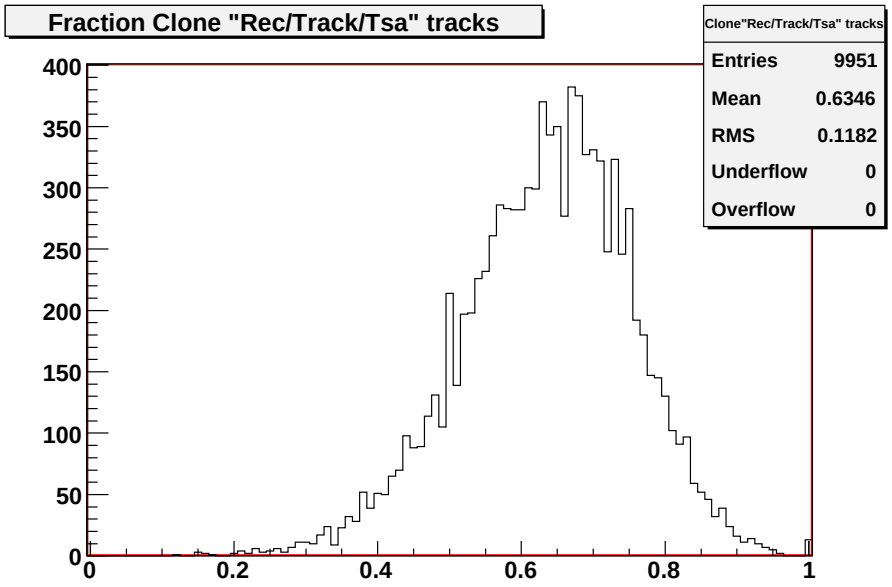

Figure 13: Distribution of the fraction of `Ttrack` tracks produced by the `TsaSeed` pattern recognition algorithm (`Tr/TsaAlgorithms` package) and flagged as clones by the clone killer algorithm.

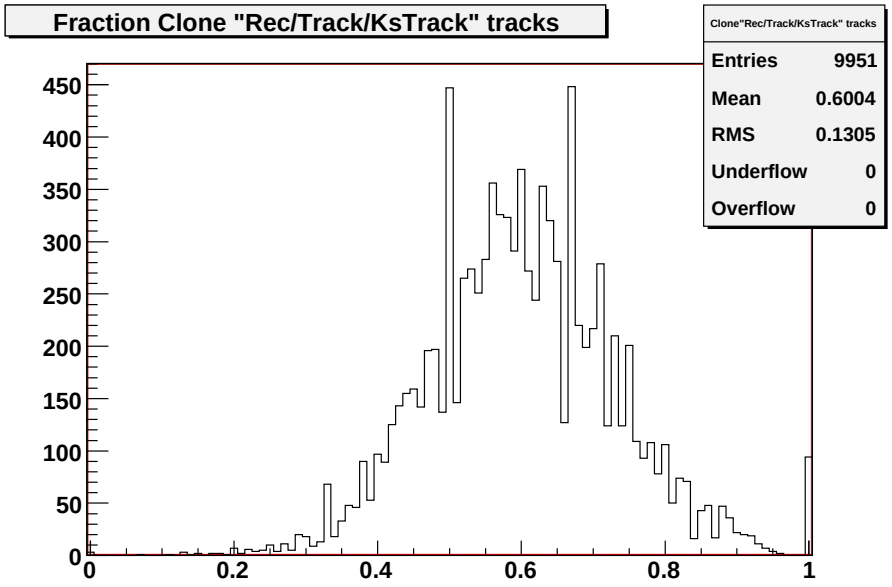

Figure 14: Distribution of the fraction of `Downstream` tracks produced by the `PatKShort` pattern recognition algorithm (`Pat/PatKShort` package) and flagged as clones by the clone killer algorithm.

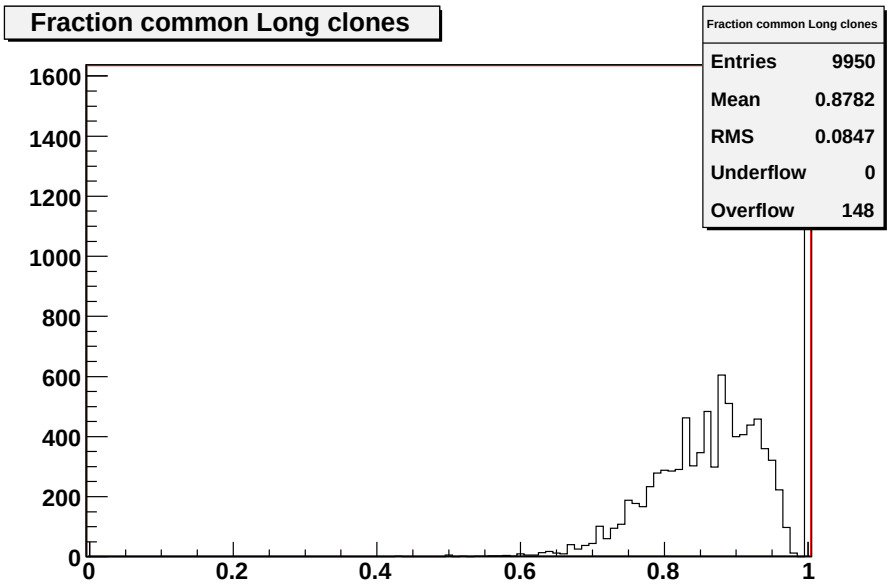

Figure 15: Fraction of common `Long` clones, `i.e.` the percentage of clone `Long` tracks among those found by `PatForward` and `TrackMatching`. This ratio has been defined as: # pairs of clones / MIN(# `PatForward`, # `TrackMatching`).

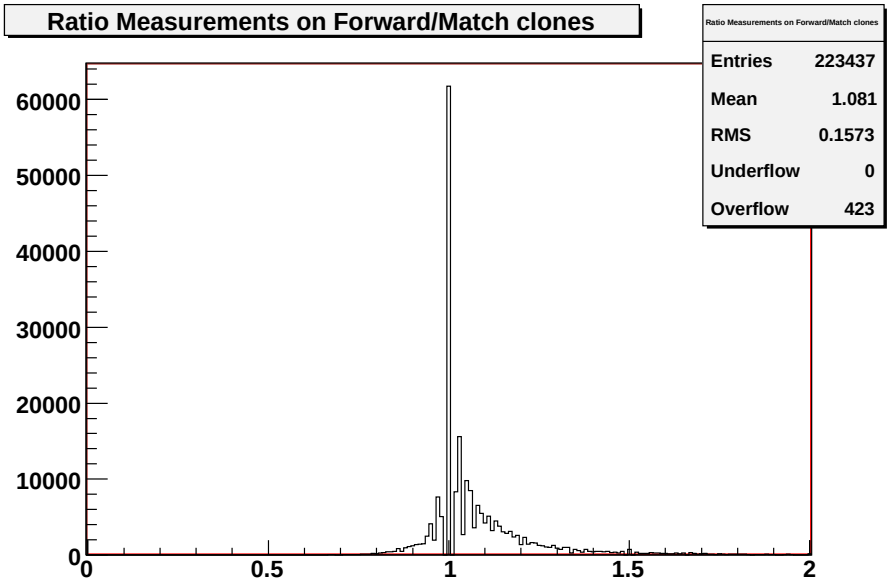

Figure 16: Ratio of the number of `Measurements` between `Long` track clones found by `PatForward` and `TrackMatching`.

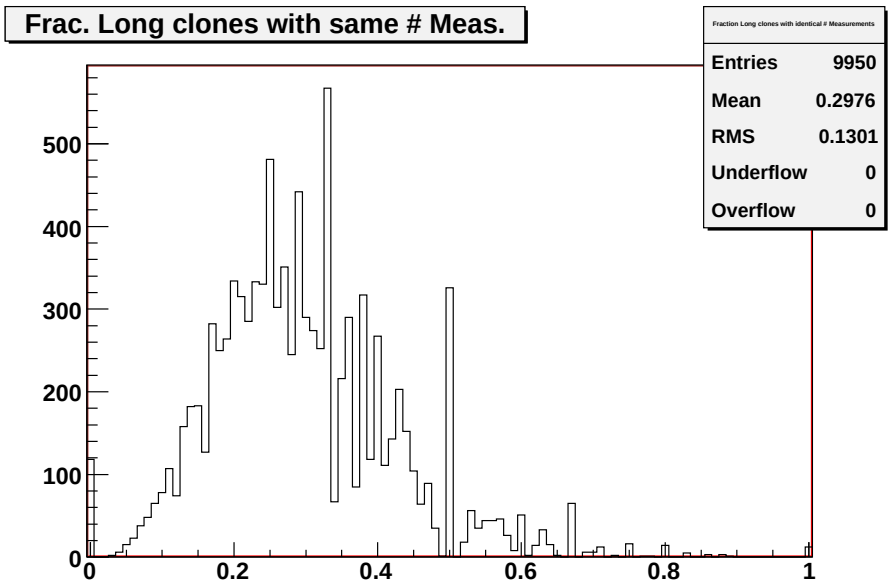

Figure 17: Fraction of `Long` clones among the tracks found by `PatForward` and `TrackMatching` with the same number of `Measurements`.