St_est_Maker

User Guide and Reference Manual

Revision: 1.00

Date: 2000/11/08

Contents

Part I

User Guide

1 Introduction

Quite a long time ago, two tracking methods have been developed to perform independent track reconstruction in the SVT. The so-called grouper or SGR [1] is using a global mapping technique to group SVT hits into segments of three hits. Since the method assumes that tracks come the main vertex, the grouper is quite efficient in finding primary tracks. The second method STK[2] is reconstructing SVT tracks using a follow your nose technique. STK is generally used after a first pass using SGR and in a mode devoted to the reconstruction of secondary particles. Once the three hit segments are formed in the SVT, an other method SVM[3]

is used to try to associate them individually to the tracks reconstructed in the TPC.

The Silicon Strip Detector (SSD) is an additional layer which will be install in between the outermost layer of the SVT and the inner radius of the TPC. Two main advantages were foreseen when proposing the SSD to enhance the STAR tracking capabilities : to add an extra hit to each track passing through the SVT and recover the tracks lost due to geometrical inneficiency, to extend the short lived particle acceptance by reconstructing the track of daughter particles produced at a radius bigger than the innermost SVT layer. Several attempts have been made in order to extend the SVT tracking methods to include the SSD. Performances obtained with STK adapted to a four layer geometry were quite poor especially for secondary tracks for which the impact of the SSD is supposed to be the more important.

In order to fully benefit from the SSD a new tracking method called EST has been developed. The main idea of the method is to start from the reconstructed TPC tracks and to iteratively try to associate hits from the SSD and the SVT layers. The method allows to find a priori with the same efficiency primary and secondary tracks since no main vertex constrain is imposed. Moreover, no lower limit being set on the number of hits forming the segments in the vertex detector, the method does not preclude the reconstruction of daughter tracks coming from short-lived particle decaying far from the main vertex. In order to maximize the efficiency of the method, the easiest tracks to find are searched first : the tracks with the higher momentum and the tracks leaving 4 hits in the vertex detector (by requiring four hits to be associated during the first steps).

This method has been implemented and used to demonstrate the impact of the SSD on the STAR capabilities[4]. Whereas the obtained performances were quite good, simple modifications of the method were foreseen in order to improve these results. The original version of the code was written in FORTRAN and in a form which did not allow a strait forward implementation of these new ideas. Consequently a new version of EST has been written in C++ which includes the improvement of the method. This document is referring to this version of the method. Following this introduction, the first main part of this manual contains a presentation of the method and the code organization followed by some results illustrating the

[1] See for details : STAR Note #0153 or http://www.star.bnl.gov/STAR/html/svt_l/soft_l/svt_software/sgr/SvtSgr.html
[2] See for details : STAR Note #0145 or http://www.star.bnl.gov/STAR/html/svt_l/soft_l/svt_software/stk/SvtStk.html
[3] See for details : http://www.star.bnl.gov/STAR/html/svt_l/soft_l/svt_software/svm/SvtSvm.html
[4] See for details : STAR Note #0400

performances obtained with EST. The second part of the document is devoted to a more detailed description of the main objects and methods implemented in the code completed by a full description of the classes developed in the code.

2 Description of the tracking method

2.1 The main ingredients of the method

As already mentioned above, the basic idea of this method is to reconstruct the particle trajectories by doing track to hit associations. Since the track reconstruction is done in the TPC in a first stage, the parameters of the track (assumed to be perfect helices) reconstructed in the TPC are known. From these parameters, fro each track a prediction is done on the intercept of the track with the SSD layer. If a hit in the SSD satisfies some geometrical criteria, it is attached to the track. The track parameters are then updated and a new prediction is made for the next outermost layer of the silicon detector. These steps are repeated for the other layers until the first SVT layer. If in a given layer, no hit satisfies the association criteria, the track parameters are not updated and a prediction is calculated for the next layer.

New features have been added to the method which are illustrated on the figure 1.Contrary to the original version of the code, in a given layer several hits can be attached to the same track which results in the formation of branches for the track. These branches are extended in parallel to the first SVT layer at which point a choice has to be made to decide which branch to finally keep. This idea is based on two observations :

- quite often the correct hit to be associated to a track is not the nearest to the track projection. This is especially true when projecting the track on the SSD. Allowing only one branch to be formed per track would strongly compromise the chances to correctly associated the hits to those tracks.

- usually a wrong hit association in a given layer strongly reduces the chance to prolongate this branches in the innermost layers. Thanks to the good position resolution in the silicon wafers, the update of the track parameters with a hit associated significantly improves the quality of the projection in the next layer. Once again this is especially true in the SSD layer. This allows us to use stronger geometrical criteria for the innermost layers and leads to prediction for the wrong branch in an area where no hit satisfy these criteria.

An other new improvement which is used a combine manner with the track branching is the possibility to allow hit sharing for tracks (the sharing of the hits by different branches belonging to the same track is implicitly implemented in the branching method). In a similar fashion of the excluding the track branching, without hit sharing between tracks, a incorrect hit to track association would ruin the chances to associate a given hit to its correct track.

After an detailed analysis of the quality of the tracks sharing one or several hits, one can found out that most of the time the chi square (of the refitted track) of the wrong tracks is higher than the one of the correct one. As a consequence at a given stage of the tracking phase, the hit sharing is suppressed. More precisely, the hit shared by several tracks are identified then all the tracks using a given hit are sorted by increasing chi

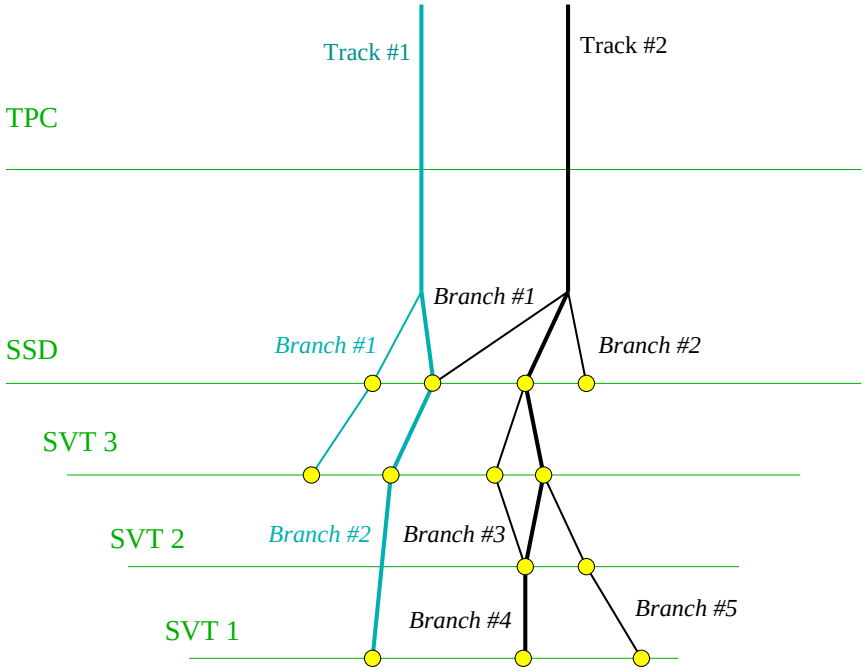

Figure 1: Illustration of the track branching and hit sharing implemented in the new version of the code. Several branches can be formed for each track (2 branches for the Track #1 and 5 branches for the Track #2). Hits are shared between branches (the first hit from the left of the second SVT layer) and between tracks (the second SSD hit from the left).

square. The track with the lowest value is kept whereas the other tracks are deleted and their hits (including those which are not shared) released for future associations.

The hit to track association in consecutive layers is the basic step used in the tracking part of EST. In order to maximize the tracking performances, this step is applied successively to several defined samples of tracks. This iterative procedure is motivated by the obvious idea of first searching the tracks which are supposed to be the easiest to find. The contribution of the multiple scattering increasing with a decreasing momentum leads us to starts with the tracks with the highest momentum. The iterative search is thus organized into passes. Each passe is dedicated to the tracking of tracks with a transverse momentum within a given range.

Taking into account the geometrical unefficiency, the slightly different rapidity coverage of the layers, the main vertex spread along the beam axis and the radial distribution of the secondary vertices, one should expect to look for tracks with various patterns and sizes for the hit segments in the vertex detector. Although, the tracking method is required to be able to reconstruct all these kind of tracks, is seems reasonable to iteratively search for a given type at a time. As already stated, the association of a hit in a given layer strongly constrains the possible associations in the inner layers. This idea is implemented in the code by the mean of superpasses. In the first superpass, one requires that the branch formed possess a hit in all the

active layers. During the last superpass, loose cuts are applied and can be for example devoted to the search of one hit segment without a specific location required for this hit.

The various passes dedicated to the search of tracks with a given transverse momentum are integrated into the superpass iterations. At the end of a given pass, a unique branch is selected for each track considered during this pass using the branch chi squares and hit pattern and segment size requirements set for the current superpass. More precisly, those requirements are not applied once all the branches are completly formed but during the branch formation time (ie. during the loops over the layer). After the tracking in a given layer, knowing the hits already attached and the remaining layers to scan, the branches which have no chances to fulfill the superpass requirements are supressed. This particular location of the segment selection in the tracking steps is improving the cpu time consumption but more importantly maximize the chances to form branches with the correct pattern.

Finally, the last new feature which has been introduced in the new version of this method is the possibility to use information on the main vertex location in order to improve the accuracy of the predictions when projecting the tracks on the vertex detector layers. The main vertex location is used as a guide to give an improved prediction when trying to associate the first hit to the tracks. In the current impementation the main vertex information can be used only for tracks which already point toward it (using a initial distance of closest approach to it) or for the full sample of tracks. Morever once a first hit association has been done, it can be removed when updating the track parameters or kept until the end of the branch formation phase.

2.2 Current implementation of the method

To be done...

3 Performances

3.1 Track projection accuracy versus the primary vertex

3.2 Efficiency vs hit sharing

3.3 Efficiency vs track branching

3.4 General performances for the full tracking steps

3.5 Detailled analysis of the performances

4 Current code limitations and future improvements

Part II

Reference Manual

5 Description of the main objects and methods

5.1 St_est_Maker

5.1.1 mParams

The `mParams` objects contain the parameters which are set for each pass over pt. The key parameters stored in these objects are :

- `onoff[i]` : Determines if the layer i (0 to 2 are the SVT layers and 3 is the SSD) is active or not. By setting `onoff[3]` to 0 allow to switch from a year-3 to a year-2 geometry.
- `nbranch[i]` : Sets to the allowed number of branches which can be formed in a given layer i. It corresponds to the number of new branches allowed to be formed in a layer per mother branch constructed in the previous layer or from the TPC tracks in the case of the SSD layer.
  By setting all the `nbranch` to 1, the branching is forbitten in the pass.
- `ntotbranch[i]` : Sets the maximum number of branches per track kept after the tracking in a given layer. The combination of the `nbranch` parameters can lead to a value bigger than `ntotbranch`. The ChooseBestNBranches method limits the number of branches to `ntotbranch` by selecting those with the lower chisquares after re-fitting.
- `share[i]` : Determines the number of times that a hit can be shared by different tracks. There is no limit on the number of branches from the same track which can share a given hit.
- `geomcutl[4]` and `geomcutw[4]` : Set the upper limits on the distance along the z axis and in the x-y plane between the branch projection and the candidate hits.

The optimization of the tracking performances in term of efficiency and purity is a balance between these parameters. Values bigger than one for the branching and sharing parameters tends to support the correct association although the correct hits are not at the first rank in term of distances. The ranking of the correct association can be checked using the root histograms named “LocationX” where X is a layer index and built during the tracking phase.

On the other hand, the parameters controlling the geometrical cuts in each layer tend to limit the number of hit candidates which can be associated to a given branch in a given layer. Setting those cuts to a large value is not reasonable since the number of wrong branches will strongly increase and affect the efficiency of the branch selection performed later on. They have to be set by knowing the accuracy of the projection one can expect in a given layer and taking into account the hits already attached from the previous layers. Several root histograms are filled named “disthitipX”,”disthitipXl” and “disthitipXw” repectively filled with the

hit-projection 3d distances, distances along the z axis and distances in the transverse plane assuming a perfect tracking.

The histograms mentionned above are currently filled for the tracks having 4 hit segment and in a year-3 geometry. They can be easily adapted for various type of segments.

Pass	ptmax	ptmin
0	100.	1.0
1	1.0	0.7
2	0.7	0.4
3	0.4	0.2
4	0.2	0.1

Table 1: Upper and lower limits in transverse momemtum for the passes

Pass	cutl[3]	cutl[2]	cutl[1]	cutl[0]
0	1.0	0.5	0.2	0.2
1	1.0	0.5	0.2	0.2
2	1.5	0.5	0.2	0.2
3	3.0	0.5	0.2	0.2
4	5.0	0.5	0.2	0.2

Table 2: Geometrical cuts along the beam axis for five passes and the four layers.

Layer	onoff	share	nbranch	ntotbranch
0	1	5	1	6
1	1	5	1	6
2	1	5	2	6
3	1	5	3	6

Table 3: Track branching and hit sharing parameters for a given pass and the four layers

Currently 5 passes are defined with the `ptmin` and `ptmax` values (in GeV/c) summarised in the table 5.1.1. As an example, the tables 2 and 3 contains the parameter values currently used in a superpass which looks for 4 hit segments. The geometrical cuts in the transverse plane are equal to those in along the z axis. One can observe that the cuts in the SSD are increasing with the pass number in order to compensate for the decreasing accuracy of the TPC track projection. Morever the cut values decrease when going from the SSD to the innermost SVT layer which relies on the expected improvement of the track projection accuracy when attaching more hits to the tracks. Based on the same expectation the branching in the SSD is set to 3 while on the innermost layers only 1 branch is allowed.

5.1.2 mSegments

The `mSegments` parameters allow to set and control the requirements for each super-pass. Two types of parameters are stored in these objects which are respectively related to the caracteristics of the TPC tracks that one tries to prolongate and to the hit pattern of the segments which are formed during the super-passes.

Some of the TPC tracks are excluded for the tracking using the parameters `minTPChits` which set a lower limit on the number of hits in the TPC attached to the track and the parameters `rminTPC` which set a upper limit on the distance in the transverse plane between the STAR geometrical center and the first TPC hit. These parameters allow to temporarily flag the short TPC tracks and the TPC tracks reconstructed in the outer part of the TPC for which a relatively poor prediction of the track projection is expected. These parameters are used in the `FlagTPCTracksSP` method (see 5.1.8).

The hit pattern criteria for the segments formed during a given super-pass are determined by the `minhits` parameter which sets a lower limit on the number of SVT/SSD hits and the /verb+slay[i]+ parameters. In a given layer `i` the association of a hit to a track can be forbidden `slay[i]=0`, allowed `slay[i]=1` or required `slay[i]=2`. The table 4 shows three examples of settings for these parameters. The `ChooseSegment` method uses these parameters to select branches formed during the tracking phase.

	A	B	C
slay[0]	2	1	1
slay[1]	2	1	1
slay[2]	2	1	1
slay[3]	2	2	1
minhits	4	3	1

Table 4: Examples of segment criteria: In the case A four hits are required with obviously a hit in every layer; in the case B a minimun number of three hits are requiered with a mandatory hit in the SSD; in the case C at least one hit is required without any specific layer origin.

The last parameter `chisqcut` of the `mSegment` object can be set to impose a higher limit on the chisq value of the fitted branches. This parameter is not used at this moment but can be easily implemented in the `ChooseBestNBranches` and `ChooseBestBranch` for instance.

5.1.3 Preprojection

The aim of this method is to quickly determine for each branch formed by the tracker which wafers should be considered as containing possible hit candidates to be associated to the branch.

The closest intercept of the helix track with a cylinder of radius equal to the current layer radius is determined. The (`phi`,`z`) position of the intercept is then compared with the wafer map `mIndexGeom` . The Wafers in the corresponding (`phi`,`z`) cell are stored in the `mPreProjTable`. This list of wafer to be scanned is then completed by considering the neighbors of the wafers already in the list. The `mPreprojNumber` data member contains the final number of wafers in the list.

The variable mPreprojection is locally used to avoid double counting the wafers. The size of the output

table mPreProjTable is limited to MAXFINDWAF usually set to 70.

5.1.4 Projection

This method is called immediately after the Preprojection method and thus applied to each branch.

For each wafer stored in mPreProjTable , the intercept of the track helix with the wafer plane is calculated. Every hit contained in the current wafer is scanned and is added to a list if :

- the hit is available (GetFlag()>0),
- the distances in z and in the bending plane between the hit and the track projection are smaller than mParams[mPass]->geomcutl[slay] and mParams[mPass]->geomcutw[slay] ,
- the number of hits already considered is smaller than MAXHITPROJ

The list mProjOut containing the accepted hits (possibly from different wafers) is sorted by increasing 3D distance between the hit and the track projection. The maximum number of hits in this list MAXHITPROJ is usually set to 400.

5.1.5 Init

5.1.6 SVTInit

5.1.7 TPCInit

This method is called by the Init method to initialize the objects related to the TPC tracks. The St_est_TPCTrack objects and the mTPCTrack table pointing to these objects are built. The TPC tracks with a negative flag are not loaded. The mTptIndex translation table from the track id to its row in the mTPCTrack table is constructed.

For each track declared in mTPCTrack its hit table mR is built. After the hit table has been built, the hits are sorted according to increasing order.

5.1.8 FlagTPCTracksSP

This method is called at the beginning of each super-pass to flag some of the TPC tracks (of type St_est_TPCTrack) which will not be considered for tracking in that super-pass. Currently two types of tracks are rejected by using this special flag :

- TPC tracks with a number of hits smaller than minTPChits (from mSegments) have a mFlagSP sets to -1,
- TPC tracks with a distance (in the x-y plane) from the detector center to the innermost hit bigger than rminTPC have a mFlagSP sets to -2.

5.1.9 ChooseSegment

This method is called at each tracking step to prevent from building branches which at the end of the loop on the layers will not satisfy the pattern criteria required for the current super-pass. These criteria are sets in the `mSegments` object and combine the `minhits` and the `slay[nlay]` conditions. Branches are deleted if :

- the number of already attached hits plus the number of layers which remain to be scanned is smaller than `minhits`.
- the condition `slay[i]`=2 requiering a hit in the layer `i` already scanned is not fulfilled.

5.1.10 ChooseBestNBranches

This method is called at each tracking step to prevent from building a too large number of branches for each track. For each track in a given layer, its branches are sorted by increasing chisq. In order to emphasise on the SVT/SSD hit contribution to the chisq, the chisq of the track obtained only with the TPC hits is first subtracted to the total chisq.

Once sorted, the `mParams[mPass]->ntotbranch[slay]` branches with the best chisq are selected. The remaining branches are deleted.

5.1.11 ChooseBestBranch

This method is called for each pass at the end of the tracking in all the layers in order to select for each track a unique branch. For the time being, the branch with the smaller chisq is selected. No upper limit on the chisq value is applied.

5.1.12 RemoveHitSharing

Depending on the value of `share[nlay]` in the `mParams` structure, hit sharing can be allowed. Hit sharing is quite usefull in order to preserve the chances for a hit which is already connected to a wrong branch to be in parallele attached to the correct one. By studying the chisq of the branches formed after a given pass which share some hits with some other branches, one can find that frequently the correct branch has the smallest chisq. In addition, in some cases hits are shared by two wrong branches. This method has been developed in order to supress the number of bad branches without affecting too much the number of correct branches. The method is called at the end of each pass and do the following operations :

- Loop in the the hit object list and look for hit shared (`mNShare`),
- For each shared hit, select the branch with the smallest chisq among all the branches sharing the current hit,
- For all branches with bigger chisq, detache all the hits (including the one which are not shared).

5.1.13 Eval

The tracking evaluation can be performed at a first level analysis corresponding to the usual counting of possible/good/bad tracks.

Three track status are required to calculate the efficiency and purity of the tracking procedure.

the number of ideal tracks : all ideal (possible) TPC tracks with at least hit in the SVT/SSD.

the number of good tracks : all tracks which have been reconstructed with correct hits.

the number of bad tracks : all reconstructed tracks with at least one wrong hit.

One should note that in this evaluation function, the definition of the ideal tracks is somehow more severe than the above definition. Tracks which are not considered (trackable) by the tracking method are excluded. As an example, a tracking pass requiering a hit in every layers will lead to exclude tracks with less that 4 hits from the ideal track list.

The efficiency and purity are defined as follow :

$$\text{Efficiency} = \frac{\text{good}}{\text{possible}} \qquad \text{Purity} = \frac{\text{good}}{\text{good} + \text{bad}}$$

A detailed analysis of the tracking evaluation can be also made in order to take into account the fact that a given TPC track can be splitted into several segments. The idea is to count the number of TPC tracks according their `mcid` instead of their `id`. In other words, and in the view of future physics analyses, each produced particle should be counted once independently from the fact that the TPC tracking can reconstruct a track in one or several segments. Consequently, a TPC track formed with, for example 2 segments, will be counted once as a possible track. With the former evaluation, 2 different `id` would have been assigned to the segments and thus 2 possible tracks would have been counted .

In summary, if several segments belonging to one TPC track are correctly associated, only one association will be counted, the other ones will be considered as ghosts.

The efficiency and the purity are defined as :

$$\text{Efficiency} = \frac{\text{correct}}{\text{input}} \qquad \text{Purity} = \frac{\text{found}}{\text{found}}$$

The following example illustrates the two evaluation methods applied to a TPC track splitted into two segments. The efficiency and purity are expressed in pourcentage.

	Detailled analysis					First Level analysis				
Association	Input	Found	Correct	Effi.	Purity	Possible	Good	Bad	Effi.	Purity
2 bad	1	2	0	0	0	2	0	2	0	0
1 bad 1 good	1	2	1	100	50	2	1	1	50	50
2 good	1	2	1	100	50	2	2	0	100	100
1 good	1	1	1	100	100	2	1	0	50	100
1 bad	1	1	0	0	0	2	0	1	0	0

Several classes of tracks are distinguished in the eval method and output on the screen :

Among all ideal tracks :

• Ideal not found but unique : ideal track not reconstructed. Its mcid is unique, and not shared by an other TPC segment.

• Ideal not found but copy in good : ideal track not reconstructed but with a copy (i.e. another TPC segment with the same mcid) which has been correctly reconstructed.

• Ideal not found but copy in bad : ideal track not reconstructed but with a copy which has been badly reconstructed.

• Ideal not found but copy in ideal: ideal track not reconstructed and with a copy which is not reconstructed either.

$\Longrightarrow$ Real ideal tracks : all the ideal tracks minus all the possible copies.

Among all good tracks :

• Good found twice in good : a track correctly reconstructed but with a copy which is also correctly reconstructed. One of the 2 correct segments is considered as a bad track.

$\Longrightarrow$ Real good tracks : all the initial number of good tracks minus the good tracks found twice.

Among all bad tracks:

• Bad in ideal : track badly reconstructed which is in the ideal track list.

• Bad not in ideal : track badly reconstructed and which was not in the ideal set of tracks. They are for example the TPC tracks which should not have any SVT/SSD hit attached.

• Bad found twice in bad : track badly reconstructed which has already a copy in the bad track list.

$\Longrightarrow$ Real bad tracks : all the initial number of bad tracks since it is used only in the purity estimate.

6 Class Reference

6.1 St_est_Maker

Summary

Synopsis

```
#include "St_est_Maker.hh"
class St_est_Maker;
```

Description

Persistence None

Related Classes

Protected Data Member

`St_DataSet* mEvent;`
DataSet containing the EST output ???

`St_DataSetIter* mEventIter;`
Iterator on the Event DataSet.

`St_est_Wafer* mPreprojTable[MAXFINDWAF];`
List of wafers build by the `Preprojection` method to be scanned by the `Projection` method.

`St_est_Params** mParams;`
Table containing the tracking parameters for the passes (over pt).

`St_est_Segments** mSegments;`
Table containing the tracking parameters for the superpasses (on the track pattern).

`St_est_IndexGeom* mIndexGeom;`
Coarse description of the SVT/SSD geometry.

`St_est_Wafer** mIndexWaf;`
List of wafer objects.

`St_est_Hit** mSvtHit;`
List of SVT/SSD hits.

`St_est_Hit* mVertex;`
Main vertex.

`St_est_TPCTrack**  mTPCTrack;`
List of TPC tracks (tracks with `flag<0` are not loaded).

`St_est_Track** mTrack;`
List of global tracks.

`svg_shape_st* mSvgShape;`
Table containing the SVT/SSD wafer shapes.

`St_est_ProjOut mProjOut;`
List of candidate hits to be considered by the tracking.

`long* Eval_id_mctrk2est_Track;`
To be explained.

`St_est_Hit*** Eval_mchits;`
List of SVT/SSD hits for each tracks.

`long mWafId2IndexWaf[9000];`
Wafer id (1101,1102,..) to Wafer index (0,1,..) translation table.
`long* mTptIndex;`
Translation table between the TPC track id and its row number in the `mTPCTrack` table.
`long mNTPCTrack;`
Number of TPC tracks (TPC tracks with negative flag are excluded).
`long mNTrack;`
Number of global tracks.
`long mNSvtHit;`
Number of SVT/SSD hits.
`int mPass;`
Current pass number.
`int mNPass;`
Number of passes.
`int mSuperPass;`
Current superpass number.
`int mNSuperPass;`
Number of superpasses.
`int mStartLayer;`
Index of the outermost layer used in the tracking.
`int mEndLayer;`
Index of the innermost layer used in the tracking.
`int mPreprojNumber;`
Number of wafers to be scanned by the `Projection` method.
`int mIdealTracking;`
Flag for the perfect tracking.
`long mFindable;`
Number of findable tracks.
`long mFoundOK;`
Number of good tracks found by the tracker.
`long mFoundFake;`
Number of ghost tracks generated by the tracker.

Public Constructors

`St_est_Maker(const char* name = "est");`

Protected Functions

`int Preprojection(St_est_Branch&, int);`
Build a list of wafers to be scanned for a branch in a given layer.
`int Projection(St_est_Branch* branch, long slay);`
Returns a list of hits which satisfy some distance criteria to the branch projection.
`int Tracking(int);`
Main tracking method.
`int RefitBranch(St_est_Branch *br, St_est_Hit* hit=NULL, int exclhit=-1, int usevertex=0);`
Method used to refit the branch `br` with the additional hit `hit` possibly excluding

some hits and using the main vertex.
`int SVTInit(St_DataSetIter* dataiter);`
Initialises and fills the St_est_Wafer objects.
Initialises and fills the St_est_Hit objects.
`int TPCInit(St_DataSetIter* dataiter);`
Initialises and fills the St_est_TPCTrack objects.
`void ChooseBestNBranches(St_est_Track *tr, int slay);`
Choose a given number of branches after tracking in the layer `slay` for the track `tr`.

6.2 St_est_Params

Summary Structure containing the parameters for the passes.

Public Data Member

`int debug;`
Debugging level (0 = only critical errors are mentioned).
`int onoff[4];`
Tracking status for a given super-layer (1 = tracking is allowed for the considered layer; 0 = no tracking is performed within this layer).
`double ptmin;`
Lowest pt value for a pass.
`double ptmax;`
Maximum pt value for a pass.
`int nbranch[4];`
Number of daughter branches allowed to be formed in a layer per the mother branch.
`int ntotbranch[4];`
Maximum number of branches per track kept after the tracking in a given layer.
`int maxbranches;`
Maximum number of branches that can be attached to a track. This is used temporarily to initialize the branch list of each Track to a large value. Once the parameters `nbranch` and `ntotbranch` will be defined consistently, then `maxbranches` will become obsolete.
`int share[4];`
Number of times that a hit can be shared by different tracks.
`int maxtpchits;`
Maximum number of hits per TPC track. It is used for the initialization of the St_TPCTrack object and corresponds to the maximum number of TPC hits taken to this object.
`int maxsvthits;`
Maximum number of SVT hits in the branches. It is used also to initialize the size of the hit list of each branch.
`double lrad[4][2];`
Radii of the silicon detector layers. 2 values are set for the SVT layers (the sublay-

ers) and one for the SSD.
`int nneighbours[4];`
Number of rings of wafers which are neighbors of the wafer containing the track projection and whose space points will be candidate for the track to hit association.
`double phibin;double zbin;`
Size of the bins in phi and along the z direction.
`int nphibin;int nzbin;`
Number of bins in phi or z.
These last 4 parameters are used for searching the `nneighbourgs` wafers. Since this step is performed during the preprojection phase, they should be normally defined once, and not like it is done now, for each pt pass.
`double geomcutl[4];`
Geometrical cut for length (along z axis beam) used to define hit searching areas.
`double geomcutw[4];`
Geometrical cut for width (in the x-y plane) used to define hit searching areas.

6.3 St_est_Segment

Summary Structure containing the parameters for the super-passes.

Public Data Member
`int slay[nlay];`
Hit requirements defined for each super-layer (2 = required; 1 = allowed; 0 = forbidden). `int minhits;`
Minimum number of hits in a segment.
`int minTPChits;`
Minimum number of hits that a TPC track has to contain for being considered.
`double rminTPC;`
Minimal radius at which the first hit (the nearest from the geometrical center) of the TPC track has to be located.
`double chisqut;`
Maximum allowed chisq value after the branch refitting (not used anymore).

6.4 St_est_ProjOut

Summary Structure containing the output of the Projection Method for a given branch.

Public Data Member
`int nhit;`
Number of hits in hit array.
`St_est_Hit* hit[MAXHITPROJ];`
List of pointer to hits found in the projection.
`double dist[MAXHITPROJ];`
3D distance between the hit and the track projection.
`double distw[MAXHITPROJ];`

Distance in the transverse plane between the hit and the track projection.
`double distl[MAXHITPROJ];`
Distance along the z axis between the hit and the track projection.

6.5 St_est_Gtrk

Summary Structure containing the inputs for the egr fitting routines.

Public Data Member
`long nhit;`
Number of hits in the track (SVT,SSD,TPC and main vertex).
`long ipnt[220];`
Pointers to the hits belonging to the track.
`long pos[220];;`
Positions of the hits on the tracks.
`long det[220];;`
Detector index of the hits (SVT/SSD,TPC,main vertex).
`float p[9];`
Track parameters output from the fitting routines.
`long nfit;`
Number of hits included in the fit.
`long flag;`
Status flag of the fit.
`long ntpc;`
Number of hits belonging to the TPC.
`long nmax;`
Maximum number of hits considered in the fit.

6.6 St_est_Wafer

Summary Class describing the SVT and SSD wafers.

Synopsis
```
#include "St_est_Maker.hh"
class St_est_Maker;
```

Description

Persistence None

Related Classes

Protected Data Member

`int mShape;`
Geometrical shape of the wafer (taken from `svg_geom.id_shape`).
`int mLayer;`
Layer the wafer belong to.
`long int mNHits;`
Number of hits in the wafer.
`long int mMaxHits;`
Maximum number of hits in the wafer (usually 200).
`St_est_Hit **mHits;`
List of hits within the wafer.
`StThreeVector<double>* x;`
Position of the center of the wafer within the STAR global reference frame.
`StThreeVector<double>* n;`
Unitary vector normal to the wafer plane.

Public Data Member

`char mPreprojection;`
Flag for the preprojection method.
`long int mId;`
Id of the wafer (according to the `svg_geom` table. Should be protected member
`St_est_Wafer* neighbour[8];`
Table of neighbouring wafers.

Public Constructors

```
St_est_Wafer(long int nr, long int mh, StThreeVector<double>* xx,
StThreeVector<double>* nn, int shape);
```

Public Member Functions

`StThreeVector<double>* St_est_Wafer::GetX();`
Returns the position of the center of the wafer.
`StThreeVector<double>* St_est_Wafer::GetN();`
Returns the orientation vector of the wafer.
`St_est_Hit* GetHit(long nr);`
Returns a pointer to the nr hit of the wafer.
`int AddHit(St_est_Hit *hit);`
Adds a hit to the wafer hit list.
`int GetLayer();`
Returns the wafer layer number.
`int GetShape();`
Returns the wafer shape number.
`long GetNHits();`
Returns the number of hits in the wafer.
`long int GetId();`
Returns the id of the wafer.

6.7 St_est_IndexGeom

Summary Class for the preprojection method.

Synopsis
```
#include "St_est_Maker.hh"
class St_est_Maker;
```

Description This class contains a simple SVT/SSD geometry description which allows a fast access to the wafer object located in a given z-phi area.

Related Classes

Protected Data Member

`int*** nWaf;`
Table containing the number of wafers per z bin and phi bin for each layer.
`St_est_Wafer***** pWaf;`
Table pointing to the wafers in each bin in z and phi and in each layer.
`int nphibins;`
Number of bins in phi to map the SVT/SSD geometry.
`int nzbins;`
Number of bin in z to map the SVT/SSD geometry.

Public Data Member

Public Constructors

`St_est_IndexGeom::St_est_IndexGeom(int np, int nz)`

Public Member Functions

`int setWafTab(int phi, int z, int slay, St_est_Wafer* waf)`
Stores the wafer `waf` in the tables.
`int getNWaf(int phi, int z, int slay)`
Returns the number of wafers in a given (z,phi) cell and a layer.
`St_est_Wafer** getWafTab(int phi, int z, int slay)`
Returns a pointer list of the wafers contained in a given (z,phi) cell and a layer.

6.8 St_est_Hit

Summary Class describing the SVT and SSD hits.

Synopsis
```
#include "St_est_Branch.hh"
class St_est_Wafer;
class St_est_Branch;
class St_est_Maker;
```

Description

Related Classes

Protected Data Member

`St_est_Wafer* mDetector;`
Id of the wafer containing the hit.
`St_est_Branch** mBranch;`
List of branches the hit is attached to.
`StThreeVector<double>* mXL;`
Local coordinates of the hit in the wafer fref. frame.
`StThreeVector<double>* mXG;`
Global coordinates of the hit.
`double mDE;`
Energy loss measurement associated with the hit.
`long mNShare;`
Number of times the hit is shared by various tracks.
`long mMaxShare;`
Maximum number of sharing allowed for different tracks (equal to `mParams[0]->share[lay]`).
`long mNBranch;`
Number of times the hit is shared by various branches (possibly from different tracks).
`long mMaxBranches;`
Maximum number of sharing allowed for different branches.
`int mDebugLevel;`
Variable used to control the flow of debugging info in the methods. Most of the information is printed out if the variable is bigger than zero.
`long mId;`
Id of the hit (taken from `scs_spt` table).
`int mFlag;`
Hit status flag : 0 = hit available, 1 = hit already used, -1 = problems.
`long mEvalTrack;`
Number of ideal tracks ???

Public Constructors

```
St_est_Hit(long id, StThreeVector<double> *xg, StThree-
Vector<double> *xl, double dE, long maxsh, St_est_Wafer *det);
```

Public Member Functions

`GetNBranch();`
Returns the number of branches using the hit.
`int JoinBranch(St_est_Branch *br);`
The hit is attached to the branch `br` (returns 1 if success)
`void LeaveBranch(St_est_Branch *br);`
The hit is removed from the branch `br`
`void SetDebugLevel(int deb);`
Set the debugging level.

`int CheckAvailability();`
Returns 1 (0) if the hit can (not) be shared.
`StThreeVector<double>* GetGlobX();`
Returns the hit global coordinates.
`StThreeVector<double>* GetLocX();`
Returns the hit local coordinates.
`St_est_Wafer* GetWafer();`
Returns a pointer to the wafer containing the hit.
`St_est_Branch* GetBranch(int i);`
Returns a pointer to the branch `i` using the hit.
`int GetFlag();`
Returns the hit status flag.
`void SetFlag(int fl);`
Sets the hit status flag.
`long GetId();`
Returns the hit id.
`void SetId(long i);`
Sets the hit id.

6.9 St_est_Track

Summary Class describing the global tracks.

Synopsis
```
#include "St_est_Branch.hh"
class St_est_Branch;
class St_est_Hit;
class St_est_TPCTrack;
class St_est_Maker;
```

Description This is the main object describing the tracks formed during the global tracking. It contains a pointer to the TPC track and a list of pointers to the branches formed with the SVT/SSD hits.

Protected Data Member
`St_est_TPCTrack *mTPCTrack;`
Pointer to the associated TPC track.
`St_est_Branch **mBranch;`
List of branches attached to the track. A track has always at least one branch (possibly empty) to allow branch duplication.
`long mNBranch;`
Number of branches for this track.
`long mMaxBranch;`
Maximum number of branches (taken from `mParams[0]->maxbranches`).

`int mFlag;`
Flag describing the track status : 0 = track is available, 1 = track is already used, -1 = problem with the track ??

Public Constructors

`St_est_Track(long int maxbranch, St_est_TPCTrack *tr)`

Public Member Functions

`int AddBranch(St_est_Branch *branch);`
Adds the new branch `branch` to the track.
`int SetBranch(long i, St_est_Branch *branch);`
Sets `branch` to the i^{th} location in the branch list.Usefulness ??
`int RemoveBranch(long int nbr);`
Remove the branch `nbr` to the list. The hits attached to the branch are released.
`inline long int GetNBranches();`
Returns the number of branches.
`St_est_Branch* GetBranch(long int nbr);`
Returns a pointer to the branch `nbr`.
`int CheckAvailability();`
Returns 1 (0) if the current number of branches is lower (or not) than the number of branches allowed.
`void SetFlag(int flag);`
Sets the track status flag.
`int GetFlag()`
Returns the track status flag.

6.10 St_est_TPCTrack

Summary

Class describing the TPC tracks.

Synopsis

```
#include "StThreeVector.hh"
#include "StHelix.hh"
class St_est_Maker;
```

Description

This class describes the TPC tracks.

Protected Data Member

`StThreeVector<double>** mR;`
List of TPC hit positions belonging to this track.
`StThreeVector<double>** mdR;`
List of TPC hit position errors.
`StHelix* mHelix;`
Helix object used to store the track parameters.
`long int *mHitId;`
List of tpc hit id attached to the track.

`long int mId;`
Id of the TPC track (equal to the row number from the tpc track table).
`long int mMcId;`
Id of the Monte Carlo track associated to the reconstructed tpc track (taken from tpc eval table).
`long int mNHits;`
Number of tpc hits attached to the track.
`long int mMaxHits;`
Maximum number of tpc hits attached to the track. Sets to `mParams[0]->maxtpchits`.
`long int *mHitIndex;`
List of tpc hit indexes ???
`int mFlag;`
Status flag of the TPC track equal the the flag variable in the tpc track table.
`int mFlagSP;`
TPC Flag set/used in superpasses. Allow to consider tpc tracks depending on their number of hits and/or their innermost hit radius.
See the `St_est_Maker::FlagTPCTracksSP` method.
`int* row;`
List of tpc hit row numbers (equal to `row` from the tphit table).
`double mPt;`
Transverse momentum of the tpc tracks.
`mChiSq;`
Total chisq of the tpc track.
`double mChiSqCir;`
Chisq in the bending plane of the tpc track.
`double mChiSqLin;`
Linear chisq of the tpc track.
`int mNFitHits;`
???
`double mr;`
Radius in the bending plane of the innermost tpc hit.

Public Constructors

`St_est_TPCTrack(long int id, long int maxhits, StHelix *hel, double pt)`

Protected Member Functions

`void Sort(long left, long right)`
???
`void Swap(long i, long j)`
???

Public Member Functions

`void SetR(double rr)`
Sets the lowest radius of the tpc hit. Necessary after the hit sorting.
`double GetR()`
Returns the lower radius of the tpc hit. `SetFlagSP(int fl)`

Sets the superpass Flag to `fl`.
`GetFlagSP()`
Returns the TPC track superpass flag.
`int AddHit(long int id, StThreeVector <double> *x, StThree-Vector <double> *dx, int nrow)`
Adds a hit to the tpc hit list.
`long int GetNHits()`
Returns the number of hits.
`StHelix* GetHelix()`
Returns the helix object.
`void SetFlag(int fl)`
Sets the TPC status flag.
`int GetFlag()`
Returns the TPC status flag.
`void SortHits()`
Sorts the tpc hits according to ...
`long int GetMcId()`
Returns the tpc tracks mc id.

6.11 St_est_Branch

Summary This class describes the branches of hits formed during the tracking.

Synopsis

```
#include "StHelix.hh"
#include "St_est_Track.hh"
#include "St_est_hit.hh"
#include "St_est_Wafer.hh"

class St_est_Track;
class St_est_Hit;
class St_est_Maker;
```

Description

Protected Data Member

`St_est_Track* mTrack;`
Global track that possesses the branch.
`St_est_Hit**  mHits;`
List of SVT/SSD hits which form the branch.
`StHelix* mHelix;`
Helix object used to store the branch parameters.
`long int mNHits;`
Number of hits which form the branch.
`long int mMaxHits;`
Maximum number of hits allowed in a branch (given by `mParams[0]->maxsvthits`).

`double mChisq;`
Total chisq of the branch.
`double mChisqCir;`
Circular chisq (in the bending plane) of the branch.
`double mChisqLin;`
Linear chisq of the branch.
`double *mDist;`
List of distances between the branch projection and the hits.
`int mLastLay;`
Layer number of the latest hit attached to the branch.
`int mDebugLevel;`
Debug level used to control the printed information flow.
`int mStep;`
Variable recording the step when the branch was formed (equal to 10*Superpass+pass).

Public Constructors

`St_est_Branch(St_est_Track* tr=NULL, long int maxhits=1, long int nh=0, St_est_Hit** hit=NULL, double *dist=NULL);`

Protected Member Functions

`int AddHit(St_est_Hit *hit, double dist);`
Adds the hit `hit` with its distance `dist` to the branch.
`int RemoveHit(long int nr);`
Removes the hit number `nr` from the branch.
`int RemoveHit(St_est_Hit* hit);`
Removes the hit `hit` from the branch.
`long int GetNHits();`
Returns the number of hits.
`St_est_Hit* GetHit(long int nr);`
Returns a pointer the hit number `nr`.
`int CheckAvailability();`
Returns 1 (0) if more hits can (not) be attached to the branch.
`St_est_Branch* Duplicate();`
Duplicates the branch. Calls the constructor to copy the branch. The hits from the copied branch are also in the duplicated branch.
`int JoinTrack(St_est_Track *tr);`
Attaches the branch to the track `tr`. If no additional branch can be attached to the track, the considered branch is inserted in the sorted (on chisq) list of branches. As a consequence, the branch with the worst chisq is lost.
`void LeaveTrack();`
Removes the branch from its track. Calls the method `RemoveBranch` from the `St_est_Track` object.
`St_est_Track* GetTrack();`
Returns a pointer to the branch parent track.
`void SetHelix(StHelix *hel);`
Sets the helix `hel` to be the branch helix.
`StHelix* GetHelix();`

Returns the branch helix.

`void SetChiSq(double chi);`

Sets the total chisq to `chi`.

`void SetChiSqCir(double chi);`

Sets the circular chisq to `chi`.

`void SetChiSqLin(double chi);`

Sets the linear chisq to `chi`.

`double GetChiSq();`

Returns the total chisq.

`double GetChiCir();`

Returns the circular chisq.

`double GetChiLin();`

Returns the linear chisq.

`GetDist(long nr);`

Returns the hit-projection distance of the hit number `nr`.

`double GetDist(St_est_Hit* hit);`

Returns the hit-projection distance of the hit `hit`.

`int GetLastLayer();`

Returns the layer number of the latest hit attached to the branch.

`void SetDebugLevel(int deb);`

Sets the debugging level.

`void SetStep(int step);`

Sets the step status.

`int GetStep();`

Returns the step status.

Index